CS 4104: Data and Algorithm Analysis

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Fall 2010

Title page

CS4014: What You Need to Already Know

- Discrete Math
  - Proof by contradiction and induction
  - Summations
  - Set theory, relations
- The basics of Asymptotic Analysis
  - Big-oh, Big-Ω, Θ
- Most of what was covered in CS2606
  - Basic data structures
  - Algorithms for searching and sorting

Basic data structures: Lists, search trees, heaps, graphs

Basic sort algorithms; search techniques such as binary search, hashing

CS4104: What We Will Do

- Finally understand upper/lower bounds
- Lower bounds proofs
- Analysis techniques (no hand waving!)
  - Recurrance Relations
- Reductions, $\mathcal{NP}$-completeness theory, and a little computability theory

Process:

- Weekly homework sets (they are hard!)
- Work in pairs

The first homework has been posted. You should get your partner decided ASAP and get started.

Introduction to Problem Solving (1)

Principle of Intimate Engagement

- This is the most important consideration
- Actively engaging the problem, getting involved
- Need to build up “mental muscles” for problem solving

For more details, see the “PSintro.pdf” notes posted at the website.

Introduction to Problem Solving (2)

Effective vs. Ineffective problem solvers (Engagers vs. Dismissers)

- Engagers have a history of success
- Dismissers have a history of failure
- You probably engage some problems and dismiss others
- You could solve more problems if you overcame the mental hurdles that lead to dismissing
- Transfer successful problem solving in some parts of your life to other areas.

Mental hurdles: That is, you have the knowledge and ability necessary to solve the problem, if you had sufficient motivation.

Introduction to Problem Solving (2)

Getting your hands dirty

- Example: Repairing a wobbly table
  - Get underneath and look
- Example: Repairing a dryer
  - Open up back panel and look

We will see examples of this concept, initially with doing summations

Investigation and Argument

Problem solving has two parts: the investigation and the argument.

- Students are used to seeing only the argument in their textbooks and lectures.
- To be successful in school and in life, one needs to be good at both
- To solve the problem, you must investigate successfully.
- Then, to give the answer to your client, you need to be able to make the argument in a way that gets the solution across clearly and succinctly.
- Writing skills. Proof Skills
- Methods of argument: Deduction (direct proof), contradiction, induction

Unfortunately, while seeing lots of examples of argument (proof), too many students don't recognize the importance of being good at **doing** it.

Heuristics for Problem Solving (1)

These heuristics most appropriate for problem solving "in the small."

- Puzzles
- Math and CS test or homework problems

A list of standard Heuristics:

1. Externalize: write it down
   - After motivation and mental attitude, the most important limitation on your ability to solve problems is biological
   - For active manipulation, you can only store 7 ± 2 pieces of information
   - Take advantage of your environment to get around this
   - Write things down
   - Manipulate problem (good representation)

no notes

Heuristics for Problem Solving (2)

2. Get your hands dirty
   - “Play around” with the problem to get some initial insight.
3. Look for special features
   - Example: Cryptogram addition problems.

$$\begin{array}{ccc} & A & D \\ + & D & I \\ \hline D & I & D \end{array}$$

4. Go to the extremes
   - Study problem boundary conditions
5. Simplify
   - This might give a partial solution that can be extended to the original problem.

Extremes: We will use this often

Heuristics for Problem Solving (3)

6. Penultimate step
   - What precondition must take place before the final solution step is possible?
   - Solving the penultimate step might be easier than the original problem.
7. Lateral thinking
   - Don’t be lead into a blind alley.
   - Using an inappropriate problem solving strategy might blind you to the solution.
8. Wishful thinking
   - A version of simplifying the problem
   - Transform problem into something easy; take start position to something that you “wish” was the solution
   - That might be a smaller step to the actual solution

Rush Hour is an excellent example. We will see another example next week: TOH

Heuristics for Problem Solving (4)

9. Sleep on it
10. Symmetry & Invariants
    - Symmetries in the problem might give clues to the solution

no notes

Pairs Problem Solving

An effective way to work in pairs to solve problems:

- Partner roles: problem solver and listener

Responsibilities of the problem solver

- Constant vocalization
- Spell out all the assumptions
- Carefully detail all steps taken

Responsibilities of the listener

- Continually check for accuracy
- Demand constant vocalization

See paper on pairs programming.

Errors in Reasoning

Getting the wrong answer on a test or homework usually results from a “breakdown” in problem solving. Typical breakdowns:

- Failing to observe and use all relevant facts of a problem.
- Failing to approach the problem in a systematic manner. Instead, making leaps in logic without checking steps.
- Failing to spell out relationships fully.
- Being sloppy and inaccurate in collecting information and carrying out mental activities.

In pairs problem solving (such as the homework in this class) there had to be a serious breakdown if the answer is wrong since the partner (the listener) should never have let it happen.

Program Efficiency

Our primary concern is EFFICIENCY.

We want efficient programs. How do we measure the efficiency of a program? (Assume we are concerned primarily with time.)

- On what input?
- How do we speed it up?
- When do we stop speeding it up?
- Should we bother with writing the program in the first place?

no notes

Algorithm Efficiency (1)

Since we don’t want to write worthless programs, we will focus on **algorithm** efficiency.

We need a yardstick.

- It should measure something we care about.
- It should by quantitative, allowing comparisons.
- It should be easy to compute (the measure, not the program).
- It should be a good predictor.

Remember that we are discussing an analytic *model*. We do not want to do performance analysis on a real program.

Algorithm Efficiency (2)

We need:

- A measure for problem size.
- A measure for solution effort.
- Use key operations as a measure of solution effort.
- Total cost is a function of problem size and key operations.

no notes

Cost Model (1)

To get a measurement, we need a model.

Example:

- Assigning to a variable takes fixed time.
- All other operations take no time.

```
sum = n*n;
```

One assignment was made, so the cost is 1.

```
sum = 0;
for (i=1; i<=n; i++)
  sum = sum + n;
```

Assignments made are $1 + \sum_{i=1}^{n} 1 = n + 1$. (Depending on how you want to deal with loop variables, you might want to say it is $2n + 1$.)

Example of a *model* for cost measure. It might or might not be a *good* model.

$n + 1$ vs $2n + 1$: Does it matter?
Not so much. We didn't know the exact amount of time for an operation to begin with, so the factor of 2 doesn't seem to mean much.
What is important is that the growth rates of these two are the same.

Cost Model (2)

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum = sum + 1;
```

Assignments made are $1 + \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n^2 + 1$.

What makes a model "good"?

- Consider string assignment (done by copying). Is this a good model?

In our example with for loops, $n + 1$ and $2n + 1$ are both linear, so they are both equally predictive of growth rate.

Big Issues

How do we create an efficient algorithm?

Q: How do we recognize a "good" algorithm?
A: By the relationship of its performance to the intrinsic difficulty of the problem.

How "hard" is a problem?

Problem solving and algorithm design. We will see some standard algorithm design techniques. Example: Dynamic programming.

A key issue, because we don't know whether to stop with trying to create a "good" algorithm unless we can recognize one. This

is where lower bounds come in.

Big Issues (2)

General Plan:

- Define a PROBLEM.
- Build MODEL to measure cost of solution to problem.
- Design an ALGORITHM to solve the problem.
- ANALYZE both the problem and the algorithm under the model.
  - Analyze an algorithm to get an UPPER BOUND.
  - Analyze a problem to get a LOWER BOUND.
- COMPARE the bounds to see if our solution is "good enough".
  - Redesign the algorithm.
  - Tighten the lower bound.
  - Change the model.
  - Change the problem.

If not, here are some options:

Problems (1)

Our problems must be well-defined enough to be solved on computers.

A **problem** is a function (i.e., a mapping of inputs to outputs).

We have different **instances** (inputs) for the problem, where each instance has a **size**.

To **solve** a problem, we must provide an algorithm, a coding of problem instances into inputs for the algorithm, and a coding for outputs into solutions.

Actually, to solve a problem we need more than just a clear definition. By the end of the semester, we will discuss problems that are not computable (i.e., cannot be solved) even though their definition is clear.

Problems (2)

An **algorithm** executes the mapping.

- A proposed algorithm must work for ALL instances (give the correct mapping to the output for that input instance).

GOAL: Solve problems with as little computational effort per instance as possible.

Actually, we will relax this restriction later... Approximation and Probabilistic algorithms.

We are most often interested in solutions to “large” instances of the problem (asymptotic Analysis).
Occasionally we are concerned with small instances. Then, constants matter.

Categories of Hard Problems (1)

- A conceptually hard problem.
  - If we understood the problem, the algorithm might be easy. [Natural Language Processing]
  - Artificial Intelligence.
- An analytically hard problem.
  - We have an algorithm, but can’t analyze its cost. [Collatz sequence]
  - Complexity Theory.

Or maybe not, but it still might run fast.Important to realize: Difficulty of analyzing the cost is a different issue from what the cost **is**!

Categories of Hard Problems (2)

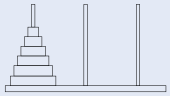
- A computationally hard problem.
  - The algorithm is expensive.
  - Class 1: No inexpensive algorithm is possible. [TOH]
  - Class 2: We don’t know if an inexpensive algorithm is possible. [Traveling Salesman]
  - Complexity Theory
- A computationally unsolvable problem. [Halting problem]
  - Computability Theory.

$\mathcal{NP}$-complete problems.
A major focus for this course: Determining if a problem is computationally hard.

No such algorithm can possibly exist.

Towers of Hanoi

Given: 3 pegs and n disks of different sizes placed in order of size on Peg 1.



Problem: Move the disks to Peg 3, given the following constraints:

- A "move" takes the topmost disk from one peg and places it on another peg (the only action allowed).
- A disk may never be on top of a smaller disk.

Model: We will measure the cost of this problem by the number of moves required.

no notes

TOH Algorithm

(This is an exercise in the process of problem solving. Pretend that you have never seen this problem before, and that you are approaching it for the first time.)

Start by trying to solve the problem for small instances.

- 0 disks, 1 disk, 2 disks...
- When we get to 3 disks, it starts to get harder.
- Can we generalize the insight from solving for 3 disks? 4 disks?

Observation: The largest disk has no effect on the movements of the other disks. Why?

Think about all the possible choices for a 3-disk series of moves.

Because it is always below the other disks, so they can move around as though it did not exist.

Problem solving often relies on a "key insight" that lets you "crack" the problem.

Similarly, *analysis* of the problem might rely on a "key insight" on how to view the analysis. Often a simplification for the "states" or progess of the algorithm, or a recognition of the key input classes for the problem.

Recursive Solutions (1)

When we generalize the TOH problem to more disks, we end up with something like:

- Move all but the bottom disk to Peg 2.
- Move the bottom disk from Peg 1 to Peg 3.
- Move the remaining disks from Peg 2 to Peg 3.

Problem-solving heuristics used:

- Get our hands dirty: Try playing with some simple examples
- Go to the extremes: Check the small cases first
- Penultimate step: Key insight is that we can't solve the problem until we move the bottom disk.

How do we deal with the $n-1$ disks (twice)?

Use recursion.

Recursive Solutions (2)

Forward-backward strategy: Solve simple special cases and generalize their solution, then test the generalization on other special cases.

```
void TOH(int n, POLE start, POLE goal, POLE temp) {
  if (n == 0) return;           // Base case
  TOH(n-1, start, temp, goal); // Recurse: n-1 rings
  move(start, goal);           // Move one disk
  TOH(n-1, temp, goal, start); // Recurse: n-1 rings
}
```


no notes

Algorithm Upper Bounds (1)

Worst case cost (for size n): Maximum cost for the algorithm over all problem instances of size n.

Best case cost (for size n): Minimum cost for the algorithm over all problem instances of size n.

$\mathcal{A}$: The algorithm.
I_n: The set of all possible inputs to $\mathcal{A}$ of size n.
$f_{\mathcal{A}}$: Function expressing the resource cost of $\mathcal{A}$.
I is an input in I_n.

$$\text{worst cost}(\mathcal{A}) = \max_{I \in I_n} f_{\mathcal{A}}(I).$$

$$\text{best cost}(\mathcal{A}) = \min_{I \in I_n} f_{\mathcal{A}}(I).$$

It is possible that the {best, worst} case cost changes radically with n. That is, even n might have a very different cost from odd n.

This point that we are considering all of the inputs of size n is crucial. In other words, we don't pick the n for which the best (or worst) case occurs. So it is wrong to say something like "The best case is when $n = 1$."

Algorithm Upper Bounds (2)

Examples:

- Factorial: One input of size n, one cost
- Find: Various models for number of inputs, n different costs
- Findmax: Various models for number of inputs, all cases have same cost

The input is just a value, n. Model choices:

- All numbers: Infinite number of inputs.
- Permutation of 1 to n: $n!$ inputs.
- Focus only on position of x: n inputs.

Same model choices as for find.

Show graphs of cost vs I_n for factorial, find (3rd model) and findmax (3rd model).

Average Case

We may want the **average case** cost. For each input of size n, we need:

- Its frequency.
- Its cost.

Given this information, we can calculate the weighted average.

Q: Can the average cost be worse than the worst cost? Or better than the best cost?

Frequency can be hard to determine!
Example: Average cost of sequential search is $(n+1)/2$, but *only* if the frequency of occurence for each case is equal.

$$\sum_{I \in I_n} \mathit{freq}(I) * \mathit{cost}(I)$$

No, because that would require at least one case with greater cost than the worst case.
No, for the same reason.

Analysis of TOH

There is only one input instance of size n.

How does this affect the decision to measure worst, best, or average case cost?

We want to count the number of moves required as a function of n.

Some facts:

- $f(1) = 1$.
- $f(2) = 3$.
- $f(3) = 7$.
- $f(n) = f(n-1) + 1 + f(n-1) = 2f(n-1) + 1, \forall n \geq 4$.

(Actually, we can simplify our list of facts.)

Worst/best/average cost are the same, so it doesn't matter which you do.

We only need f(1) and f(n), facts f(2) and f(3) are redundant information.

Recurrence Relation

The following is a **recurrence relation**:

$$f(n) = \begin{cases} 1 & n = 1 \\ 2f(n-1) + 1 & n > 1 \end{cases}$$

How can we find a closed form solution for the recurrence?

It looks like each time we add a disk, we roughly double the cost – something like 2^n.

If we examine some simple cases, we see that they appear to fit the equation $f(n) = 2^n - 1$.

How do we prove that this ALWAYS works?

In practice, this is a common way to start: look for a pattern.
It is so common, it has its own name: Guess and Test.

Proof for Recurrence

Let's ASSUME that $f(n-1) = 2^{n-1} - 1$, and see what happens.

From the recurrence,

$$f(n) = 2f(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1.$$

Implication: if there is EVER an n for which $f(n) = 2^n - 1$, then for all greater values of n, f conforms to this rule.

This is the essence of **proof by induction**.

no notes

Proof by Induction

To prove by induction, we need to show two things:

- We can get started (**base case**).
- Being true for k implies that it is true also for $k + 1$.

Here again is the proof for TOH:

- For $n = 1$, $f(1) = 1$, so $f(1) = 2^1 - 1$.
- Assume $f(k) = 2^k - 1$, for $k < n$.
  - Then, from the recurrence we have

$$\begin{aligned} f(n) &= 2f(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 = 2^n - 1 \end{aligned}$$

  - Thus, being true for $k - 1$ implies that it is also true for k.
- Thus, we conclude that formula is correct for all $n \geq 1$.

Is this a good algorithm?

That would depend on what? On the intrinsic difficulty of the problem!

Lower Bound of a Problem (1)

To decide if the algorithm is good, we need a lower bound on the cost of the PROBLEM.

We can measure the lower bound (over all possible algorithms) for the {worst case, best case, or average case}.

Consider a graph of cost for each possible algorithm.

- For a given problem size n, the graph shows the costs for all problem instances of size n.

The worst case lower bound is the LEAST of all the HIGHEST points on all the graphs.

no notes

Lower Bound of a Problem (2)

$\mathcal{A}_M$ is the set of algorithms within model M that solve the problem. Lower Bound on Problem P

$$= \min_{\mathcal{A} \in \mathcal{A}_M} \{\max_{I \in I_n} f_{\mathcal{A}}(I)\}$$

We need the model to define:

- What problem
- What cost metric

Lower Bound on Problem P (for instance of size n).See Rawlins Figure 1.7.

Growth Rate vs. I_n

Note the important difference between a growth rate graph for a given problem, and a graph showing all the I_n's (for a given n) of that problem.

Examples: Consider the graphs for each of these

- Find: Best, average, and worst cases as n grows
- Find: Cost for all inputs of a given size n
- Findmax: Cost as n grows (same for best, average, worst cases)
- Findmax: Cost for all inputs of a given size n

The fact that (for some problems) different Is in I_n can have different costs is the reason why we must use the qualifier of "best" "worst" or "average" cases.

Show graphs for each of the cases.

Lower Bound (cont.)

- Lower bounds (of problems) are harder than upper bounds (of algorithms) because we must consider ALL of the possible algorithms – including the ones we don't know!
  - Upper bound: How bad is the algorithm?
  - Lower bound: How hard is the problem?
- Lower bounds don't give you a good algorithm. They only help you know when to stop looking.
- If the lower bound for the problem matches the upper bound for the algorithm (within a constant factor), then we know that we can find an algorithm that is better only by a constant factor.
- Can a lower bound tell us if an algorithm is NOT optimal?

Since we cannot even enumerate all the algorithms and check all the bounds, we need a different approach!

No, sorry!
Why not? Because we might not have the tightest possible lower bound!

Lower Bounds for TOH

- Try #1: We must move each disk at least twice, except for the largest we move once.
  - $f(n) = 2n - 1$.
- Is this a good match to the cost of our algorithm?
- Where is the problem: the lower bound or the algorithm?
- Insight #1: $f(n) > f(n-1)$.
  - Seems obvious, but why?
  - Is this true for all problems?
- Try #2: To move the bottom disk to Peg 3, we MUST move $n-1$ disks to Peg 2. Then, we MUST move $n-1$ disks back to Peg 3.

$$f(n) \geq 2f(n-1) + 1.$$

- Thus, TOH is optimal (for our model).

No! $\Omega(n)$ isn't close to $O(2^n)$.

We must move $n-1$ disks off the bottom disk first.
No! For example, sorting cost depends on particular problem instances.Since it does nothing more than the minimum required by the observation.

Warning: Normally we cannot "prove" anything about a problem in general with this sort of behavioristic argument. Usually, we cannot say so much about *how* an algorithm *must* work.

New Models

New model #1: We can move a stack of disks in one move.

New model #2: Not all disks start on Peg 1.

New model #3: Different numbers of pegs.

New model #4: We want to know what the kth move is.

Model #1: A big help! $O(n)$ or even $O(1)$.

Model #2: Doesn't seem to change the cost of the problem.

Combining these two things: Looks to be $O(n)$.

Problem Solving Algorithm

If the upper and lower bounds match,
then stop,
else if close or problem isn't important,
  then stop,
  else if model focuses on wrong thing,
    then restate it,
    else if the algorithm is too fat,
      then generate slimmer algorithm,
      else if lower bound is too weak,
        then generate stronger bound.

Repeat until done.

Does this "algorithm" always terminate?
No – you might get stuck in a look if you go through and make no progress.

Factorial Growth (1)

Which function grows faster? $f(n) = 2^n$ or $g(n) = n!$

How about $h(n) = 2^{2n}$?

	n	1	2	3	4	5	6	7	8
$g(n)$	$n!$	1	2	6	24	120	720	5040	40320
$f(n)$	2^n	2	4	8	16	32	64	128	256
$h(n)$	2^{2n}	4	16	64	256	1024	4096	16384	65536

Hopefully your intuition tells you that $n!$ grows much faster than 2^n.

This one is probably not as obvious. Of course, this is 4^n, so if your intuition is good, you will realize that $n!$ is much faster growing (since most numbers are bigger than 4).

It just so happens that $n!$ will be become bigger than 2^{2n} for $n = 9$.

Factorial Growth (1)

Consider the recurrences:

$$h(n) = \begin{cases} 4 & n = 1 \\ 4h(n-1) & n > 1 \end{cases}$$

$$g(n) = \begin{cases} 1 & n = 1 \\ ng(n-1) & n > 1 \end{cases}$$

I hope your intuition tells you the right thing.

But, how do you PROVE it?

Induction? What is the base case?

The $n > 1$ clause is the important part of the recurrence for growth.
The second recurrence is just $n!$ in recurrence form.

Sorry, we don't know the base case. It must be something bigger than 8. So, we can't use induction!

Induction is great for verifying a hypothesis. It is not so good for generating candidate formulae!

Using Logarithms (1)

$n! \geq 2^{2n}$ iff $\log n! \geq \log 2^{2n} = 2n$. Why?

$$\begin{aligned} n! &= n \times (n-1) \times \cdots \times \frac{n}{2} \times (\frac{n}{2} - 1) \times \cdots \times 2 \times 1 \\ &\geq \frac{n}{2} \times \frac{n}{2} \times \cdots \times \frac{n}{2} \times 1 \times \cdots \times 1 \times 1 \\ &= (\frac{n}{2})^{n/2} \end{aligned}$$

Therefore

$$\log n! \geq \log(\frac{n}{2})^{n/2} = (\frac{n}{2})\log(\frac{n}{2}).$$

Need only show that this grows to be bigger than 2n.

Take log of both sides.
Note that log always means $\log_2$ unless explicitly stated otherwise.

We have $\frac{n}{2}$ $n/2$ times and we have 1 also $n/2$ times. This isn't quite perfect. What if n is odd?

Since we noted earlier that $\log n! > 2n$ if $n! > 2^{2n}$.

Using Logarithms (2)

$$\begin{aligned} (\tfrac{n}{2})\log(\tfrac{n}{2}) &\geq 2n \\ \Longleftrightarrow \quad \log(\tfrac{n}{2}) &\geq 4 \\ \Longleftrightarrow \quad n &\geq 32 \end{aligned}$$

So, $n! \geq 2^{2n}$ once $n \geq 32$.

Now we could prove this with induction, using 32 for the base case.

- What is the tightest base case?
- How did we get such a big over-estimate?

Multiply by $2/n2 \cdot 2^4 = 32$. Take antilog and multiply by 2.

9

We grossly overestimated when going from $n!$ to $(\frac{n}{2})^{n/2}$.

Logs and Factorials

We have proved that $n! \in \Omega(2^{2n})$.

We have also proved that $\log n! \in \Omega(n \log n)$.

From here, its easy to prove that $\log n! \in O(n \log n)$, so $\log n! = \Theta(n \log n)$.

This does **not** mean that $n! = \Theta(n^n)$.

- Note that $\log n = \Theta(\log n^2)$ but $n \neq \Theta(n^2)$.
- The log function is a "flattener" when dealing with asymptotics.

Graphically, we can see a curve for $n!$ that is above the curve for 2^{2n}. But we dn't know how big the gap is (if any).

Why? Because $n! < n^n$.

Note from a previous slide that we claimed

$$n! \geq 2^{2n} \text{ iff } \log n! \geq \log 2^{2n} = 2n.$$

But while $A \geq B$ iff $\log A \geq \log B$, it is NOT TRUE that $A > B$ iff $\log A > \log B$.

A Simple Sum (1)

```
sum = 0; inc = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++) {
    sum = sum + inc;
    inc++;
  }
```

Use summations to analyze this code fragment. The number of assignments is:

$$2 + \sum_{i=1}^{n}(\sum_{j=1}^{i} 2) = 2 + \sum_{i=1}^{n} 2i = 2 + 2\sum_{i=1}^{n} i$$

no notes

A Simple Sum (2)

Give a good estimate.

- Observe that the biggest term is $2 + 2n$ and there are n terms, so its at most: $2n + 2n^2$
- Actually, most terms are much less, and its a linear ramp, so a better estimate is: about n^2.

Give the exact solution.

- Of course, we all know the closed form solution for $\sum_{i=1}^{n} i$.
- And we should all know how to prove it using induction.
- But where did it come from?

$2n + 2n^2$

About half of this, so about n^2.

A Problem-Specific Approach

Observe that we can "pair up" the first and last terms, the 2nd and $(n-1)$th terms, and so on. Each pair sums to: $n+1$.

The number of pairs is: $n/2$.

Thus, the solution is: $(n+1)(n/2)$.

Each pair sums to $n + 1$.
# of pairs is $n/2$.
The solution is $(n + 1)(n/2)$.

This is pretty! But it is not useful for solving any *other* summation!
Note that there is no question about its being correct.

A Little More General

Since the largest term is n and there are n terms, the summation is less than n^2.

If we are lucky, the solution is a polynomial.

Guess: $f(n) = c_1 n^2 + c_2 n + c_3$.
$f(0) = 0$ so $c_3 = 0$.
For $f(1)$, we get $c_1 + c_2 = 1$.
For $f(2)$, we get $4c_1 + 2c_2 = 3$.
Setting this up as a system of 2 equations on 2 variables, we can solve to find that $c_1 = 1/2$ and $c_2 = 1/2$.

Being polynomial is an *assumption*.

More General (2)

So, if it truely is a polynomial, it **must** be

$$f(n) = n^2/2 + n/2 + 0 = \frac{n(n+1)}{2}.$$

Use induction to prove. Why is this step necessary?

Why is this not a universal approach to solving summations?

Because we merely guessed that it is a polynomial and then fit some points. For all we know, it could be something like $c_1 n^2 + c_2 n \log n$.

Because lots of summations do not have polynomial closed-form solutions.

An Even More General Approach

Subtract-and-Guess or **Divide-and-Guess** strategies.

To solve sum f, pick a known function g and find a pattern in terms of $f(n) - g(n)$ or $f(n)/g(n)$.

Find the closed form solution for

$$f(n) = \sum_{i=1}^{n} i.$$

no notes

Guessing (cont.)

Examples: Try $g_1(n) = n$; $g_2(n) = f(n-1)$.

n	1	2	3	4	5	6	7	8
$f(n)$	1	3	6	10	15	21	28	36
$g_1(n)$	1	2	3	4	5	6	7	8
$f(n)/g_1(n)$	2/2	3/2	4/2	5/2	6/2	7/2	8/2	9/2
$g_2(n)$	0	1	3	6	10	15	21	28
$f(n)/g_2(n)$		3/1	4/2	5/3	6/4	7/5	8/6	9/7

What are the patterns?

$\frac{f(n)}{g_1(n)} =$

$\frac{f(n)}{g_2(n)} =$

$(n+1)/2$
$(n+1)/(n-1)$

Of couse, lots of other approachs do NOT work.

- $f(n) - g_1(n) = f(n-1)$. Knowing that $f(n) = f(n-1) + n$ is not useful.
- $f(n) - g_2(n) = n$. Knowing that $f(n) = f(n-1) + n$ is not useful.

It can be like finding a needle in a haystack.

Solving Summations (cont.)

Use algebra to rearrange and solve for $f(n)$

$$\frac{f(n)}{n} = \frac{n+1}{2}$$

$$\frac{f(n)}{f(n-1)} = \frac{n+1}{n-1}$$

(1) is pretty direct. So $f(n) = (n+1)(n)/2$.

(2) is not so direct, but useful as an example.

Solving Summations (cont.)

$$\begin{aligned}
\frac{f(n)}{f(n-1)} &= \frac{n+1}{n-1} \\
f(n)(n-1) &= (n+1)f(n-1) \\
f(n)(n-1) &= (n+1)(f(n)-n) \\
nf(n) - f(n) &= nf(n) + f(n) - n^2 - n \\
2f(n) &= n^2 + n = n(n+1) \\
f(n) &= \frac{n(n+1)}{2}
\end{aligned}$$

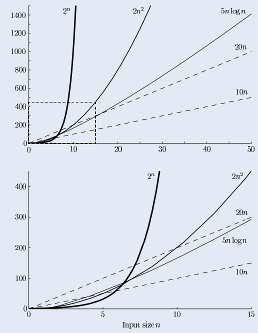
Important Note: This is **not a proof** that $f(n) = n(n+1)/2$. Why?

So long as we have both $f(n)$ and $f(n-1)$ in the equation, we are stuck. So, how can we get rid of $f(n-1)$? What can we replace it with? Something in terms of $f(n)$. Replacing $f(n-1)$ with $f(n) - n$ is the key step.

Because we did not prove either (1) or (2). We merely detected a pattern from looking at a few terms. Now we have a *hypothesis*. Fortunately, its easy to check a hypothesis with induction.

Growth Rates

Two functions of n have different **growth rates** if as n goes to infinity their ratio either goes to infinity or goes to zero.



Where does $(1.618)^n$ go on here?

Estimating Growth Rates

Exact equations relating program operations to running time require machine-dependent constants.

Sometimes, the equation for exact running time is complicated to compute.

Usually, we are satisfied with knowing an approximate growth rate.

Example: Given two algorithms with growth rate $c_1 n$ and $c_2 2^{n!}$, do we need to know the values of c_1 and c_2?

Consider n^2 and $3n$. PROVE that n^2 must eventually become (and remain) bigger.

no notes

Proof by Contradiction

Assume there are some values for constants r and s such that, for all values of n,

$$n^2 < rn + s.$$

Then, $n < r + s/n$.

But, as n grows, what happens to s/n?

Since n grows toward infinity, the assumption must be false.

It goes to zero.

Conclusion: In the limit, as $n \to \infty$, constants don't matter. Limits are the typical way to prove that one function grows faster than another.

Some Growth Rates (1)

Since n^2 grows faster than n,

- 2^{n^2} grows faster than 2^n.
- n^4 grows faster than n^2.
- n grows faster than $\sqrt{n}$.
- $2 \log n$ grows <u>no slower</u> than $\log n$.

Took antilog of both sides.
We squared boths sides.
$n = (\sqrt{n})^2$. We replaced n with $\sqrt{n}$.
Took log of both sides. Log "flattens" growth rates.

Some Growth Rates (2)

Since $n!$ grows faster than 2^n,

- $n!!$ grows faster than $2^n!$.
- $2^{n!}$ grows faster than 2^{2^n}.
- $n!^2$ grows faster than 2^{2n}.
- $\sqrt{n!}$ grows faster than $\sqrt{2^n}$.
- $\log n!$ grows no slower than n.

Apply factorial to both sides.
Take antilog of both sides.
Squared both sides.
Took square root of both sides.
Took log of both sides. Actually, it grows faster since $\log n! = \Theta(n \log n)$.

Some Growth Rates (3)

If f grows faster than g, then

- Must $\sqrt{f}$ grow faster than $\sqrt{g}$?
- Must $\log f$ grow faster than $\log g$?

$\log n$ is related to n in exactly the same way that n is related to 2^n.

- $2^{\log n} = n$

Yes.
No.

$\log n \approx \log n^2$ within a constant factor, that is, the growth *rate* is the same!

Fibonacci Numbers (Iterative)

$f(n) = f(n-1) + f(n-2)$ for $n \geq 2$; $f(0) = f(1) = 1$.

```
long Fibi(int n) {
  long past, prev, curr;
  past = prev = curr = 1;     // curr holds Fib(i)
  for (int i=2; i<=n; i++) {  // Compute next value
    past = prev; prev = curr; // past holds Fib(i-2)
    curr = past + prev;       // prev holds Fib(i-1)
  }
  return curr;
}
```

The cost of Fibi is easy to compute:

$3n$ assignments.

Fibonacci Numbers (Recursive)

```
int Fibr(int n) {
  if ((n <= 1) return 1;          // Base case
  return Fibr(n-1) + Fibr(n-2);   // Recursive call
}
```

What is the cost of Fibr?

It is a recursive function.
So we use a recurrence relation to describe its cost.
Basically, the number of function calls (the cost, since each function does constant work aside from calling other functions) is the same as the size of the Fibonacci number itself!

Analysis of Fibr

Use divide-and-guess with $f(n-1)$.

n	1	2	3	4	5	6	7	8
$f(n)$	1	2	3	5	8	13	21	28
$f(n)/f(n-1)$	1	2	1.5	1.666	1.6	1.625	1.615	1.619

Following this out, it appears to settle to a ratio of 1.618.

Assuming $f(n)/f(n-1)$ really tends to a fixed value x, let's verify what x must be.

$$\frac{f(n)}{f(n-2)} = \frac{f(n-1)}{f(n-2)} + \frac{f(n-2)}{f(n-2)} \rightarrow x + 1$$

From $f(n) = f(n-1) + f(n-2)$.
We divide by $f(n-2)$ to make the second term go away – and we also get something useful in the first term. Remember that the goal of such manipulations is to give us an equation that relates $f(n)$ to something *without* recursive subcalls.

Analysis of Fibr (cont.)

For large n,

$$\frac{f(n)}{f(n-2)} = \frac{f(n)}{f(n-1)}\frac{f(n-1)}{f(n-2)} \rightarrow x^2$$

If x exists, then $x^2 - x - 1 \rightarrow 0$.

Using the quadratic equation, the only solution greater than one is

$$x = \frac{1+\sqrt{5}}{2} \approx 1.618.$$

What does this say about the growth rate of f?

We get this by muliplying and rearranging:

$$\frac{f(n)}{f(n-2)}\frac{f(n-1)}{f(n-1)}$$

As n gets big, the two ratios go to x.

The growth rate is exponential. $f(n) \approx (1.618)^n$.

n	1	2	3	4	5	6	7
$f(n)$	1	2	3	5	8	13	21
1.62^n	1.62	2.62	4.24	6.9	11.09	17.94	29.03

Note that the value is always in the right range, even if the scale is off a bit.

Order Notation

little oh	$f(n) \in o(g(n))$	$<$	$\lim f(n)/g(n) = 0$
big oh	$f(n) \in O(g(n))$	$\leq$	
Theta	$f(n) = \Theta(g(n))$	$=$	$f = O(g)$ and $g = O(f)$
Big Omega	$f(n) \in \Omega(g(n))$	$\geq$	
Little Omega	$f(n) \in \omega(g(n))$	$>$	$\lim g(n)/f(n) = 0$

I prefer "$f \in O(n^2)$" to "$f = O(n^2)$"

- While $n \in O(n^2)$ and $n^2 \in O(n^2)$, $O(n) \neq O(n^2)$.

Note: Big oh does not say how good an algorithm is – only how bad it CAN be.

If $\mathcal{A} \in O(n)$ and $\mathcal{B} \in O(n^2)$, is $\mathcal{A}$ better than $\mathcal{B}$?

Perhaps... but perhaps better analysis will show that $\mathcal{A} = \Theta(n)$ while $\mathcal{B} = \Theta(\log n)$.

no notes

Limitations on Order Notation

Statement: Algorithm $\mathcal{A}$'s resource requirements grow slower than Algorithm $\mathcal{B}$'s resource requirements.

Is $\mathcal{A}$ better than $\mathcal{B}$?

Potential problems:

- How big must the input be?
- Some growth rate differences are trivial
  - Example: $\Theta(\log^2 n)$ vs. $\Theta(n^{1/10})$.
- It is not always practical to reduce an algorithm's growth rate
  - Shaving a factor of n reduces cost by a factor of a million for input size of a million.
  - Shaving a factor of $\log\log n$ saves only a factor of 4-5.

Notation: $\log n^2 (= 2\log n)$ vs. $\log^2 n (= (\log n)^2)$ vs. $\log\log n$.
$\log 16^2 = 2\log 16 = 8$. $\log^2 16 = 4^2 = 16$.
$\log\log 16 = \log 4 = 2$.

If n is $10^{12} (\approx 2^{40})$ then $\log^2 n \approx 1600$, $n^{1/10} = 16$ even though $n^{1/10}$ grows faster than $\log^2 n$.
n must be enormous (like 2^{150}) for $n^{1/10}$ to be bigger than $\log^2 n$.

"Practical" here means that the constants might become too much higher when we shave off the minor asymptotic growth.

Practicality Window

In general:

- We have limited time to solve a problem.
- We have a limited input size.

Fortunately, algorithm growth rates are USUALLY well behaved, so that Order Notation gives practical indications.

Input can only get so big before the computer chokes.

“Practical” is the keyword. We use asymptotics because they provide a simple *model* that *usually* mirrors reality. This is *useful* to simplify our thinking.

Searching

Assumptions for search problems:

- Target is well defined.
- Target is fixed.
- Search domain is finite.
- We (can) remember all information gathered during search.

We search for a record with a **key**.

Well defined: We recognize a hit or miss.

Fixed: The target doesn’t move during the life of the search.

We often choose not to remember information. For example, sequential search does not remember the values seen already.

A Search Model (1)

Problem:
Given:

- A list L, of n elements
- A search key X

Solve: Identify one element in L which has key value X, if any exist.

Model:

- The key values for elements in L are unique.
- One comparison determines $<, =, >$.
- Comparison is our only way to find ordering information.
- Every comparison costs the same.

What if the key values are not unique? Probably the cost goes down, not up. This is an assumption for *analysis*, not for implementation.

We would have a slightly different model (though no asymptotic change in cost) if our only comparison test was $<$. We would have a very different model if our only comparison was $= / \neq$.

A comparison-based model.

String data might require comparisons with very different costs.

A Search Model (2)

Goal: Solve the problem using the minimum number of comparisons.

- Cost model: Number of comparisons.
- (Implication) Access to every item in L costs the same (array).

Is this a reasonable model and goal?

- We are assuming that the # of comparisons is proportional to runtime.
- Might not always share an array (assumption that all accesses are equal). For example, linked lists.
- We assume there is no relationship between value X and its position.

Linear Search

General algorithm strategy: Reduce the problem.

- Compare X to the first element.
- If not done, then solve the problem for $n-1$ elements.

```
Position linear_search(L, lower, upper, X) {
  if L[lower] = X then
    return lower;
  else if lower = upper then
    return -1;
  else
    return linear_search(L, lower+1, upper, X);
}
```

What equation represents the worst case cost?

Warning: We are using this simple, familiar algorithm as an illustration of how to do full, formal analysis. This includes some recurrence solving techniques, and attention to lower bounds.

Cost given on next slide.

Worst Cost Upper Bound

$$f(n) = \begin{cases} 1 & n = 1 \\ f(n-1)+1 & n > 1 \end{cases}$$

Reasonable to guess that $f(n) = n$.
Prove by induction:
Basis step: $f(1) = 1$, so $f(n) = n$ when $n = 1$.
Induction hypothesis: For $k < n$, $f(k) = k$.
Induction step: From recurrence,

$$\begin{aligned} f(n) &= f(n-1)+1 \\ &= (n-1)+1 \\ &= n \end{aligned}$$

Thus, the worst case cost for n elements is linear.
Induction is great for verifying a hypothesis.

no notes

Approach #2

- What if we couldn't guess a solution?
- Try: Substitute and Guess.
  - Iterate a few steps of the recurrence, and look for a summation.

$$\begin{aligned} f(n) &= f(n-1)+1 \\ &= \{f(n-2)+1\}+1 \\ &= \{\{f(n-3)+1\}+1\}+1\} \end{aligned}$$

- Now what? Guess $f(n) = f(n-i)+i$.
- When do we stop? When we reach a value for f that we know.

$$f(n) = f(n-(n-1)) + n - 1 = f(1) + n - 1 = n$$

- Now, go back and test the guess using induction.

Replace i with $n-1$.

Alternative: Recognize $f(n) = f(1 + \sum_{i=2}^{n} 1$.

Approach #3

Guess and Test: Guess the form of the solution, then solve the resulting equations.

Guess: $f(n)$ is linear.
$f(n) = rn + s$ for some r, s.

What do we know?

- $f(1) = r \times 1 + s = r + s = 1$.
- $f(n) = r \times n + s = r \times (n-1) + s + 1$.

Solving these two simultaneous equations, $r = 1$, $s = 0$.

Final form of guess: $f(n) = n$.

Now, prove using induction.

Often, $f(0)$ is easier. Or maybe $f(2)$.

By definition, $f(n) = f(n-1)+1$, so $r \times n = r \times (n-1)+1$.
So $rn + s = rn - r + s + 1$.
$s = s - r + 1$
$r - 1 = 0$

Since $f(n) = f(n-1)+1$.

Why is this a guess and not a proof? Because all we did is show that our model passes through two points that the “real” curve also passes through. If the curve really is linear, 2 points is all that we need. But, we need to prove that it is linear.

Lower Bound on Problem

Theorem: Lower bound (in the worst case) for the problem is n comparisons.

Proof: By contradiction.

- Assume an algorithm A exists that requires only $n - 1$ (or less) comparisons of X with elements of L.
- Since there are n elements of L, A must have avoided comparing X with $L[i]$ for some value i.
- We can feed the algorithm an input with X in position i.
- Such an input is legal in our model, so the algorithm is incorrect.

Is this proof correct?

Be careful about assumptions on how an algorithm might (must) behave.
After all, where do new, clever algorithms come from? From different behavior than was previously assumed!

Fixing the Proof (1)

Error #1: An algorithm need not consistently skip position i.
Fix:

- On any given run of the algorithm, *some* element i gets skipped.
- It is possible that X is in position i at that time.

no notes

Fixing the Proof (2)

Error #2: Must allow comparisons between elements of L.
Fix:

- Include the ability to “preprocess” L.
- View L as initially consisting of n “pieces.”
- A comparison can join two pieces (without involving X).
- The total of these comparisons is k.
- We must have at least $n - k$ pieces.
- A comparison of X against a piece can reject the whole piece.
- This requires $n - k$ comparisons.
- The total is still at least n comparisons.

no notes

Average Cost

How many comparisons does linear search do on average?

We must know the probability of occurrence for each possible input.

(Must X be in L?)

Ignore everything except the position of X in L. Why?

What are the $n + 1$ events?

$$\mathbf{P}(X \notin L) = 1 - \sum_{i=1}^{n} \mathbf{P}(X = L[i]).$$

No, X might not be in L! What is this probability?

The actual values of other elements is irrelevent to the search routine.

$L[1], L[2], ..., L[n]$ and *not found*.

Assume that array bounds are $1..n$.

Average Cost Equation

Let $k_i = i$ be the number of comparisons when $X = L[i]$.
Let $k_0 = n$ be the number of comparisons when $X \notin L$.

Let p_i be the probability that $X = L[i]$.
Let p_0 be the probability that $X \notin L[i]$ for any i.

$$\begin{aligned} f(n) &= k_0 p_0 + \sum_{i=1}^{n} k_i p_i \\ &= n p_0 + \sum_{i=1}^{n} i p_i \end{aligned}$$

What happens to the equation if we assume all p_i's are equal (except p_0)?

no notes

Computation

$$\begin{aligned} f(n) &= p_0 n + \sum_{i=1}^{n} ip \\ &= p_0 n + p \sum_{i=1}^{n} i \\ &= p_0 n + p \frac{n(n+1)}{2} \\ &= p_0 n + \frac{1-p_0}{n} \frac{n(n+1)}{2} \\ &= \frac{n+1+p_0(n-1)}{2} \end{aligned}$$

Depending on the value of p_0, $\frac{n+1}{2} \leq f(n) \leq n$.

$$p = \frac{1-p_0}{n}$$

.
Show a graph of p_0 vs. cost for $0 \leq p_0 \leq 1$, with y axis going from 0 to n.

Problems with Average Cost

- Average cost is usually harder to determine than worst cost.
- We really need also to know the variance around the average.
- Our computation is only as good as our knowledge (guess) on distribution.

Example: Quicksort variance is rather low. For this linear search, the variances is higher (normal curve).

Sorted List

Change the model: Assume that the elements are in ascending order.

Is linear search still optimal? Why not?

Optimization: Use linear search, but test if the element is greater than X. Why?

Observation: If we look at $L[5]$ and find that X is bigger, then we rule out $L[1]$ to $L[4]$ as well.

More is Better: If we look at $L[n]$ and find that X is bigger, then we know in one test that X is not in L. Great!

- What is wrong here?

We have more information a priori.

Can quit early.
What is best, worst, average cost? 1, n, $n/2$, respectively.
Effectively eliminates case of x not on list.

If we find that x is smaller, we only rule out one element.
Cost is 1 either way, but we don't get much information in worst case.
Small probability for big information, but big probability for small information.

Jump Search

Algorithm:

- From the beginning of the array, start making jumps of size k, checking $L[k]$ then $L[2k]$, and so on.
- So long as X is greater, keep jumping by k.
- If X is less, then use linear search on the last sublist of k elements.

This is called Jump Search.

What is the right amount to jump?

no notes

Analysis of Jump Search

- If $mk \leq n < (m+1)k$, then the total cost is at most $m + k - 1$ 3-way comparisons.

$$f(n,k) = m + k - 1 = \left\lfloor \frac{n}{k} \right\rfloor + k - 1.$$

- What should k be?

$$\min_{1 \leq k \leq n} \left\{ \left\lfloor \frac{n}{k} \right\rfloor + k - 1 \right\}$$

- Take the derivative and solve for $f'(x) = 0$ to find the minimum.
- This is a minimum when $k = \sqrt{n}$.
- What is the worst case cost?
  - Roughly $2\sqrt{n}$.

m is number of big steps, k is size of big step.

Rawlins has a discussion about some technicalities related to how to take derivative since k is an integer. Essentially, the real-valued equivalent cannot be off by more than 1.

Lessons

We want to balance the work done while selecting a sublist with the work done while searching a sublist.

In general, make subproblems of equal effort.

This is an example of **divide and conquer**

What if we extend this to three levels?

- We'd jump to get a sublist, then jump to get a sub-sublist, then do sequential search
- While it might make sense to do a two-level algorithm (like jump search), it almost never makes sense to do a three-level algorithm
- Instead, we resort to recursion

This takes us to binary search.

Binary Search

```
int binary(int K, int* array, int left, int right) {
  // Return position of element (if any) with value K
  int l = left-1;
  int r = right+1;    // l and r beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Middle of remaining subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```


no notes

Worst Case for Binary Search (1)

$$f(n) = \begin{cases} 1 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 & n > 1 \end{cases}$$

Since $n/2 \geq \lfloor n/2 \rfloor$, and since $f(n)$ is assumed to be non-decreasing (why?), we can use

$$f(n) = f(n/2) + 1.$$

Alternatively, assume n is a power of 2.
Expand the recurrence:

$$\begin{aligned} f(n) &= f(n/2) + 1 \\ &= \{f(n/4) + 1\} + 1 \\ &= \{\{f(n/8) + 1\} + 1\} + 1 \end{aligned}$$

We get rid of at least $\lceil n/2 \rceil$ elements.

Adding more elements won't decrease the work.

Worst Case for Binary Search (2)

Collapse to

$$f(n) = f(n/2^i) + i = f(1) + \log n = \log n + 1$$

Now, prove it with induction.

$$\begin{aligned} f(n/2) + 1 &= (\log(n/2) + 1) + 1 \\ &= (\log n - 1 + 1) + 1 \\ &= \log n + 1 = f(n). \end{aligned}$$

By the IH,

$$f(n/2) = \log(n/2) + 1.$$

Lower Bound (for Problem Worst Case)

How does n compare to $\sqrt{n}$ compare to $\log n$?

Can we do better?

Model an algorithm for the problem using a decision tree.

- Consider only comparisons with X.
- Branch depending on the result of comparing X with $L[i]$.
- There must be at least n leaf nodes in the tree. (Why?)
- Some path must be at least $\log n$ deep. (Why?)

Thus, binary search has optimal worst cost under this model.

Assumption: A deterministic algorithm: For a given input, the algorithm always does the same comparisons.

Since L is sorted, we already know the outcome of any comparisons between elements in L, so such comparisons are useless.

There must be some point in the algorithm, for each position in the array, where only that position remains as the possible outcome. Each such place corresponds to a (leaf) node.

Because a tree of n nodes requires at least this depth.Show decision tree illustration.

Average Cost of Binary Search (1)

An estimate given these assumptions:

- X is in L.
- X is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer k.

Cost?

- One chance to hit in one probe.
- Two chances to hit in two probes.
- 2^{i-1} to hit in i probes.
- $i \leq k$.

What is the equation?

no notes

Average Cost (2)

$$\frac{1 \times 1 + 2 \times 2 + 3 \times 4 + ... + \log n 2^{\log n - 1}}{n} = \frac{1}{n}\sum_{i=1}^{\log n} i2^{i-1}$$

$$\begin{aligned}\sum_{i=1}^{k} i2^{i-1} &= \sum_{i=0}^{k-1}(i+1)2^i = \sum_{i=0}^{k-1} i2^i + \sum_{i=0}^{k-1} 2^i \\ &= 2\sum_{i=0}^{k-1} i2^{i-1} + 2^k - 1 \\ &= 2\sum_{i=1}^{k} i2^{i-1} - k2^k + 2^k - 1\end{aligned}$$

$2^{\log n - 1} = n/2$.

From the second line, and through the next slide, works on solving the summation in its own right. We'll come back to solving the original equation after we have the summation.

Change variables: $i \rightarrow i + 1$.

0th term contributed nothing. Take out the kth term.

Now we have $f(n) = 2f(n) - \text{stuff}$ so $f(n) = \text{stuff}$.

Form: $x = 2x - y$ so $x = y$.

Average Cost (3)

Now what? Subtract from the original!

$$\sum_{i=1}^{k} i2^{i-1} = k2^k - 2^k + 1 = (k-1)2^k + 1.$$

$$\sum_{i=1}^{k} i2^{i-1} = 2\sum_{i=1}^{k} i2^{i-1} - k2^k + 2^k - 1$$

So,

$$\begin{aligned}\sum_{i=1}^{k} i2^{i-1} &= k2^k - 2^k + 1 \\ &= (k-1)2^k + 1\end{aligned}$$

Result (1)

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{\log n} i2^{i-1} &= \frac{(\log n - 1)2^{\log n} + 1}{n} \\ &= \frac{n(\log n - 1) + 1}{n} \\ &\approx \log n - 1\end{aligned}$$

So the average cost is only about one or two comparisons less than the worst cost.

Now we come back to solving the original equation. Since we have a closed-form solution for the summation in hand, we can restate the equation with the appropriate variable substitutions.

$2^{\log n} = n$.

Result (2)

If we want to relax the assumption that $n = 2^k$, we get:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \frac{\lceil \frac{n}{2} \rceil - 1}{n} f(\lceil \frac{n}{2} \rceil - 1) + \frac{1}{n}0 + & \\ \frac{\lfloor \frac{n}{2} \rfloor}{n} f(\lfloor \frac{n}{2} \rfloor) + 1 & n > 1 \end{cases}$$

Identify each of the components of this equation.

Left branch ($X < L[i]$)
$L(i) == X$ (no cost, $1/n$ chance)
Right branch ($X > L[i]$)

Average Cost Lower Bound

- Use decision trees again.
- **Total Path Length**: Sum of the level for each node.
- The cost of an outcome is the level of the corresponding node plus 1.
- The average cost of the algorithm is the average cost of the outcomes (total path length/n).
- What is the tree with the least average depth?
- This is equivalent to the tree that corresponds to binary search.
- Thus, binary search is optimal.

(In worst case.)

Fill in tree row by row, left to right. So node i is at depth $\lfloor \log i \rfloor$.

Changing the Model

What are factors that might make binary search either unusable or not optimal?

- We know something about the distribution.
- Data are not sorted. (Preprocessing?)
- Data sorted, but probes not all the same cost (not an array).
- Data are static, know all search requests in advance.

Or otherwise know more about the data.

Do more preprocessing than sorting?

Linked list.

Could order data to optimize the total series of requests (e.g., by frequency).

Interpolation Search

(Also known as Dictionary Search)

Search L at a position that is appropriate to the value of X.

$$p = \frac{X - L[1]}{L[n] - L[1]}$$

Repeat as necessary to recalculate p for future searches.

That is, readjust for new array bounds.

Note that p is a fraction, so $\lfloor pn \rfloor$ is an index position between 0 and $n - 1$.

Quadratic Binary Search

This is easier to analyze:

- Compute p and examine $L[\lceil pn \rceil]$.
- If $X < L[\lceil pn \rceil]$ then sequentially probe
$$L[\lceil pn - i\sqrt{n} \rceil], i = 1, 2, 3, ...$$
until we reach a value less than or equal to X.
- Similar for $X > L[\lceil pn \rceil]$.
- We are now within $\sqrt{n}$ positions of X.
- ASSUME (for now) that this takes a constant number of comparisons.
- Now we have a sublist of size $\sqrt{n}$.
- Repeat the process recursively.
- What is the cost?

We will come back and examine this assumption.

How many times can we take the square root of n?
Keep dividing the exponent by 2 until we reach 1 – that is, take the log of the *exponent*.
What is the exponent? It is $\log n$.
$\log \log n$ is the number of times that we can take the square root.

QBS Probe Count (1)

Cost is $\Theta(\log \log n)$ IF the number of probes on jump search is constant.

Number of comparisons needed is:

$$\sum_{i=1}^{\sqrt{n}} i\mathbf{P}(\text{need exactly } i \text{ probes})$$
$$= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}$$

This is equal to:

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

no notes

QBS Probe Count (2)

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

$$\begin{aligned} &= 1 + (1 - \mathbf{P}_1) + (1 - \mathbf{P}_1 - \mathbf{P}_2) + \cdots + \mathbf{P}_{\sqrt{n}} \\ &= (\mathbf{P}_1 + ... + \mathbf{P}_{\sqrt{n}}) + (\mathbf{P}_2 + ... + \mathbf{P}_{\sqrt{n}}) + \\ &\qquad (\mathbf{P}_3 + ... + \mathbf{P}_{\sqrt{n}}) + \cdots \\ &= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}} \end{aligned}$$

no notes

QBS Probe Count (3)

We require at least two probes to set the bounds, so cost is:

$$2 + \sum_{i=3}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

Useful fact (Čebyšev's Inequality):
The probability that we need probe i times ($\mathbf{P}_i$) is:

$$\mathbf{P}_i \leq \frac{p(1-p)n}{(i-2)^2 n} \leq \frac{1}{4(i-2)^2}$$

since $p(1-p) \leq 1/4$.

This assumes uniformly distributed data.

Original C's Inequality $\leq$ the result of recognizing that $p(1-p) \leq 1/4$.

Important assumption!

QBS Probe Count (4)

Final result:

$$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} \approx 2.4112$$

Is this better than binary search?

What happened to our proof that binary search is optimal?

The assumption of uniform distribution (resulting in constant number of probes on average) is much stronger than the assumptions used by the lower bounds proof.

Comparison (1)

Let's compare $\log \log n$ to $\log n$.

n	$\log n$	$\log \log n$	Diff
16	4	2	2
256	8	3	2.7
$64K$	16	4	4
2^{32}	32	5	6.4

Now look at the actual comparisons used.

- Binary search $\approx \log n - 1$
- Interpolation search $\approx 2.4 \log \log n$

n	$\log n - 1$	$2.4 \log \log n$	Diff
16	3	4.8	worse
256	7	7.2	$\approx$ same
$64K$	15	9.6	1.6
2^{32}	31	12	2.6

no notes

Comparison (2)

Not done yet! This is only a count of comparisons!

- Which is more expensive: calculating the midpoint or calculating the interpolation point?

Which algorithm is dependent on good behavior by the input?

Taking an interpolation point.

QBS

Hashing

Assume we can preprocess the data.

- How should we do it to minimize search?

Put record with key value X in $L[X]$.

If the range is too big, then use hashing.

How much can we get from this?

Simplifying assumptions:

- We hash to each slot with equal probability
- We probe to each (new) slot with equal probability
- This is called uniform hashing

This is the theoritical "ideal" for hashing. True hash functions and probe functions can't do quite this well.

Perfect hashing is an even more extreme case. In perfect hashing, we must know all records in advance (no dynamic update of the database). We then *construct* a hash function for *that* set of records. Constructing the hash function takes time roughly equivalent to sorting. After that, the search cost is constant.

Hashing Insertion Analysis (1)

Define $\alpha = N/M$ (Records stored/Table size)

Insertion cost: sum of costs times probabilities for looking at 1, 2, ..., $N+1$ slots

- Probability of collision on insertion? $\alpha = N/M$
- Probability of initial collision and another collision when probing? α^2

$$\sum_{i=0}^{i=N} i\left(\frac{N}{M}\right)^i \frac{M-N}{M} = \sum_{i=0}^{i=N} i\alpha^i(1-\alpha)$$

no notes

Hashing Insertion Analysis (2)

Simpler formulation: Always look at least once, look at least twice with probability α, look at least three times with probability α^2, etc.

$$\sum_{i=0}^{\infty} \alpha^i = 1 + \alpha + \alpha^2 \cdots = \frac{1}{1 - \alpha}$$

How does this grow?

Similar to analysis of QBS.

This grows super-linearly on α.

Need to show graph of alpha vs. cost.

Searching Linked Lists

Assume the list is sorted, but is stored in a linked list.

Can we use binary search?

- Comparisons?
- “Work?”

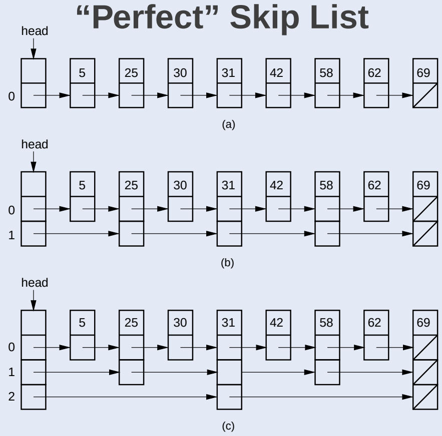
What if we add additional pointers?

Same. Is this a good model? No.

Much higher since we must move around a lot (without comparisons) to get to the same position.

Might get to desired position faster.

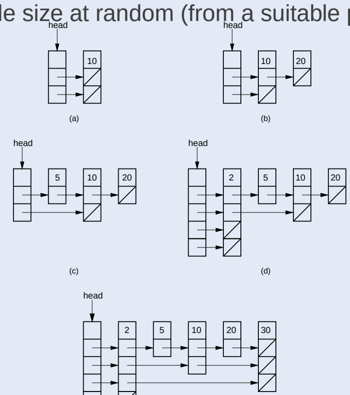
“Perfect” Skip List



What is the access time? $\log n$.
We can insert/delete in $\log n$ time as well.

Building a Skip List

Pick the node size at random (from a suitable probability distribution).



no notes

Skip List Analysis (1)

What distribution do we want for the node depths?

```
int randomLevel(void) { // Exponential distrib
  for (int level=0; Random(2) == 0; level++);
  return level;
}
```

What is the worst cost to search in the “perfect” Skip List?

What is the average cost to search in the “perfect” Skip List?

What is the cost to insert?

What is the average cost in the “typical” Skip List?

Exponential decay. 1 link half of the time, 2 links one quarter, 3 links one eighth, and so on.

$\log n$.

Close to $\log n$.

$\log n$.

$\log n$.

Skip List Analysis (2)

How does this differ from a BST?

- Simpler or more complex?
- More or less efficient?
- Which relies on data distribution, which on basic laws of probability?

About the same.

On average, about the same *if* data are well distributed.

BST relies on data distribution, while skiplist merely relies on chance.

Other Types of Search

- Nearest neighbor (if X not in L).
- Exact Match Query.
- Range query.
- Multi-dimensional search.
- Is L static?

Is linear search on a sorted list ever better than binary search?

Use a minor variant on binary search.
This is what we have been talking about.
This really changes the rules, need to think about amortization.
Example: 2D or 3D points.
What if L can change (how much?) after each comparison?

Lots of cases:

- Linked list
- Small list
- High probability of search key near front

Selection

How can we find the ith largest value

- in a sorted list?
- in an unsorted list?

Can we do better with an unsorted list than to sort it?

Assumption: Elements can be **ranked**.

Constant – go to position i.

Sorting costs $n \log n$ time.

Properties of Relationships (1)

Partial Order: Given a set S and a binary operator R, R defines a partial order on S if R is:

- Antisymmetric: Whenever aRb and bRa, then $a = b$, for all $a, b \in \mathbf{S}$.
- Transitive: Whenever aRb and bRc, then aRc, for all $a, b, c \in \mathbf{S}$.

Think of a relationship as a set of tuples.

- A tuple is in the set (in the relation) iff the relation holds on that tuple.

Example: S is Integers, R is $<$.

Example: S is the power set of $\{1, 2, 3\}$, R is subset.

It is "anti" symmetric because it says that if aRb then it is NOT bRa unless $a = b$. Consider for example $\leq$ relation.

Not all authors use the same definitions.

$<$ is vacuously antisymmetric.

Properties of Relationships (2)

A partial order is also called a **poset**.

If every pair of elements in S is relatable by R, then we have a **linear order**.

We cannot relate $\{1, 2\}$ with $\{1, 3\}$. Which is "bigger? Neither!

Why are we interested in partial orders? Can we find the ith biggest in a partial order? Maybe, but often not.

However, posets are useful to represent *current* knowledge, and also weaker relationships such as **max**.

General Model

For all of our problems on Selection and Sorting:

- The poset has a linear ordering. (Usually natural numbers and a relationship of $\leq$.)
- Cost measure is the number of 3-way element-element comparisons.

Selection problems:

- Find the max or min.
- Find the second largest.
- Find the median.
- Find the ith largest.
- Find several ranks simultaneously.

no notes

Finding the Maximum

```
int Find_max(int *L, int low, int high) {
  max = low;
  for(i=low+1; i<= high; i++)
    if(L[i] > L[max])
      max = i;
  return max;
}
```

What is the cost?

Is this optimal?

$n - 1 = \Theta(n)$ comparisons.

What is the lower bound for this problem?

Proof of Lower Bound (1)

Try #1:

- The winner must compare against all other elements, so there must be $n-1$ comparisons.

Try #2:

- Only the winner does not lose.
- There are $n-1$ losers.
- A single comparison generates (at most) one (new) loser.
- Therefore, there must be $n-1$ comparisons.

Try #1 is flawed: There is no reason why the winner needs to directly compare against each other element. (Note that it does not in our algorithm!)

Proof of Lower Bound (2)

Alternative proof:

- To find the max, we must build a poset having one max and $n-1$ losers, starting from a poset of n singletons.
- We wish to connect the elements of the poset with the minimum number of links.
- This requires at least $n-1$ links.
- A comparison provides at most one new link.

This proof is not simpler than try #2! But it is a model for proofs that will be useful later.

Average Cost

- What is the average cost for `Find_max`?
  - Since it always does the same number of comparisons, clearly $n-1$ comparisons.
- How many assignments to `max` does it do?
- Ignoring the actual values in L, there are $n!$ permutations for the input.
- `Find_max` does an assignment on the ith iteration iff $L[i]$ is the biggest of the first i elements.
- Since this event does happen, or does not happen:
  - Given no information about distribution, the probability of an assignment after each comparison is 50%.

Warning: For the next few problems, we are not going to be looking at asymptotic growth rate as we usually do. Instead, we will look at the exact number of operations of interest (comparisons, or whatever), and try to minimize the number.

If all values are unique.

Wrong! As i grows, the probability that the next element is bigger than any of those already seen reduces.

Average Number of Assignments

`Find_max` does an assignment on the ith iteration iff $L[i]$ is the biggest of the first i elements.

Assuming all permutations are equally likely, the probability of this being true is $1/i$.

$$1 + \sum_{i=2}^{n} \frac{1}{i} \times 1 = \sum_{i=1}^{n} \frac{1}{i}.$$

This sum generates the nth **harmonic number**: $\mathcal{H}_n$.

$\sum_{i=2}^{n} \frac{1}{i}$ is the probability, and 1 is the cost.

Technique (1)

Since $i \leq 2^{\lceil \log i \rceil}$, $1/i \geq 1/2^{\lceil \log i \rceil}$.

Thus, if $n = 2^k$

$$\begin{aligned}
\mathcal{H}_{2^k} &= 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{2^k} \\
&\geq 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} \\
&\qquad + \ldots + \frac{1}{2^k} \\
&= 1 + \frac{1}{2} + \frac{2}{4} + \frac{4}{8} + \ldots \frac{2^{k-1}}{2^k} \\
&= 1 + \frac{k}{2}.
\end{aligned}$$

no notes

Technique (2)

Using similar logic, $\mathcal{H}_{2^k} \leq k + \frac{1}{2^k}$. Thus, $\mathcal{H}_n = \Theta(\log n)$.

More exactly, $\mathcal{H}_n$ is close to $\ln n$.

$k = \log n$

ln means natural log of n ($\log_e n$).

Conclusion: The number of assignments is about $\log n$ in the *average* case.

Variance (1)

How "reliable" is the average?

- How much will a given run of the program deviate from the average?

Variance: For runs of the program, average square of differences.

Standard deviation: Square root of variance.

From Čebyšev's Inequality, 75% of the observations fall within 2 standard deviations of the average.

For `Find_max`, the variance is

$$\mathcal{H}_n - \frac{\pi^2}{6} = \ln n - \frac{\pi^2}{6}$$

Čebyšev's Inequality applies to a normal distribution.

Variance (2)

The standard deviation is thus about $\sqrt{\ln n}$.

- So, 75% of the observations are between $\ln n - 2\sqrt{\ln n}$ and $\ln n + 2\sqrt{\ln n}$.
- Is this a narrow spread or a wide spread?

A wide spread. Example:

- $n = 16$. $\ln n \approx 4, \pm 2\sqrt{4} = 4$, so 4 ± 4.
- $n = 64k$. $\ln n \approx 16, \pm 2\sqrt{16} = 8$, so 16 ± 8.

Finding the Second Best

In a single-elimination tournament, is the second best the one who loses in the finals? Simple algorithm:

- Find the best.
- Discard it.
- Now, find the second best of the $n-1$ remaining elements.

Cost? Is this optimal?

As we discuss this problem, we consider *exact* counts, not asymptotics.

Not necessarily – the best 2 could compete in the first round! Note that we ignore variations in performance, the outcome between two players will always be the same.

$2n-3$.

To know, need a lower bound on the problem.
Naive: $\approx n$ might work. Clearly not optimal here! But, tighten lower bound.

Lower Bound for Second (1)

Lower bound:

- Anyone who lost to anyone who is not the max cannot be second.
- So, the only candidates are those who lost to max.
- `Find_max` might compare max to $n-1$ others.
- Thus, we might need $n-2$ additional comparisons to find second.
- Wrong!

What is wrong with this argument?

Lower Bound for Second (2)

The previous argument exhibits the **necessity fallacy**:

- Our algorithm does something, therefore all algorithms solving the problem must do the same.

Alternative: Divide and conquer

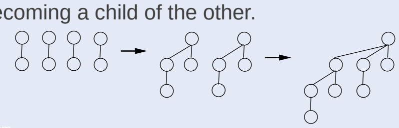
- Break the list into two halves.
- Run `Find_max` on each half.
- Compare the winners.
- Run `Find_max` on the winner's half for second.
- Compare that second to second winner.

Cost: $\lceil 3n/2 \rceil - 2$.
Is this optimal?
What if we break the list into four pieces? Eight?

In particular, it is not necessary that the max element compare with $n-1$ others, even in the worst case.
$\lfloor n/2 \rfloor - 1 + \lceil n/2 \rceil - 1 \ldots +1 = n-1$.
Worst case: $\lceil n/2 \rceil - 1$ elements, since winner need not compete again.
$+1$.
Cost of $\lceil 3n/2 \rceil - 2$ just closed half of the gap between our old lower bound and our old algorithm – pretty good progress!
4: about $5/4$.
8: $n-1+\lceil n/8 \rceil - 1 = \lceil 9n/8 \rceil - 2$.
What if we do this recursively?
$f(n) = 2f(n/2) + 2; f(1) = 0$ which is $3n/2 - 2$, which is no better than halves. So recursive divide & conquer (in a naive way) does not work! Quarters would be better!

Binomial Trees (1)

- Pushing this idea to its extreme, we want each comparison to be between winners of equal numbers of comparisons.
- The only candidates for second are losers to the eventual winner.
- A **binomial tree** of height m has 2^m nodes organized as:
  - a single node, if $m = 0$, or
  - two height $m-1$ binomial trees with one tree's root becoming a child of the other.



but, we want as *few* of these as possible.

Binomial Trees (2)

Algorithm:

- Build the tree.
- Compare the $\lceil \log n \rceil$ children of the root for second.

Cost?

Binomial Trees (2)
Algorithm:
Build the tree.
Compare the $\lceil \log n \rceil$ children of the root for second.
Cost?

$n + \lceil \log n \rceil - 2$.

Binomial Tree Representation

- We could store the binomial tree as an explicit tree structure.
- Can also store binomial tree implicitly: In array.
- Assume two trees, each with 2^k nodes, are in array as:
  - First tree in positions 1 to 2^k.
  - Second tree in positions $2^k + 1$ to 2^{k+1}.
  - The root of a subtree is in the final array position for that subtree.
- To join:
  - Compare the roots of the subtrees.
  - If necessary, swap subtrees so larger root element is second subtree.
- Trades space for time.

Need more time to swap the trees, but less space.
But all the swaps add up to a total of $\Theta(n \log n)$ time in the worst case.
Not really practical to add $\Theta(n \log n)$ swaps to the cost.

Adversarial Lower Bounds Proof (1)

Many lower bounds proofs use the concept of an **adversary**.

The adversary's job is to make an algorithm's cost as high as possible.

The algorithm asks the adversary for information about the input.

The adversary may never lie.

no notes

Adversarial Lower Bounds Proof (2)

Imagine that the adversary keeps a list of all possible inputs.

- When the algorithm asks a question, the adversary answers, and crosses out all remaining inputs inconsistent with that answer.
- The adversary is permitted to give any answer that is consistent with at least one remaining input.

Examples:

- Hangman.
- Search an unordered list.

Adversary maintains dictionary, and can give any answer that conforms with at least one entry in the dictionary.

Adversary always says "not found" until last element.

Lower Bound for Second Best

At least $n-1$ values must lose at least once.

- At least $n-1$ compares.

In addition, at least $k-1$ values must lose to the second best.

- I.e., k direct losers to the winner must be compared.

There must be at least $n+k-2$ comparisons.

How low can we make k?

Lower Bound for Second Best
At least $n-1$ values must lose at least once.
At least $n-1$ compares.
In addition, at least $k-1$ values must lose to the second best.
I.e., k direct losers to the winner must be compared.
There must be at least $n+k-2$ comparisons.
How low can we make k?

What does your intuition tell you as a lower bound for k? $\Omega(n)$? $\Omega(\log n)$? $\Omega(c)$?

Adversarial Lower Bound

Call the **strength** of element $L[i]$ the number of elements $L[i]$ is (known to be) bigger than.

If $L[i]$ has strength a, and $L[j]$ has strength b, then the winner has strength $a+b+1$.

What should the adversary do?

- Minimize the rate at which any element improves.
- Do this by making the stronger element always win.
- Is this legal?

Adversarial Lower Bound
Call the strength of element $L[i]$ the number of elements $L[i]$ is (known to be) bigger than.
If $L[i]$ has strength a, and $L[j]$ has strength b, then the winner has strength $a+b+1$.
What should the adversary do?
Minimize the rate at which any element improves.
Do this by making the stronger element always win.
Is this legal?

The winner has now proved stronger than $a+b+$ the one who just lost.

Yes. The adversary cannot "fix" the fight to give contradictory answers. But, it *can* give answers consistent with *some* legal input.

Lower Bound (Cont.)

What should the algorithm do?

If $a \geq b$, then $2a \geq a+b$.

- From the algorithm's point of view, the best outcome is that an element doubles in strength.
- This happens when $a=b$.
- All strengths begin at zero, so the winner must make at least k comparisons for $2^{k-1} < n \leq 2^k$.

Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons.

Lower Bound (Cont.)
What should the algorithm do?
If $a \geq b$, then $2a \geq a+b$.
From the algorithm's point of view, the best outcome is that an element doubles in strength.
This happens when $a=b$.
All strengths begin at zero, so the winner must make at least k comparisons for $2^{k-1} < n \leq 2^k$.
Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons.

Need to get the final strength up to $n-1$.
These k losers are candidates for 2nd place.

Find Min and Max (1)

Find them independantly: $2n-2$.

- Can easily modify to get $2n-3$.

Should be able to do better(?)

Try divide and conquer.

Find Min and Max (1)
Find them independantly: $2n-2$.
Can easily modify to get $2n-3$.
Should be able to do better(?)
Try divide and conquer.

A slightly different problem.
Question: Which is the tougher problem? Find first and second? Or find first and last?
The intuition is not obvious.
On the one hand, it seems that in the process of finding the maximum, you will learn more about the second than you will about the min.
On the other hand, a given comparison tells you something about a candidate for max, and a candidate for min.

Find Min and Max (2)

```
Find_Max_Min(ELEM *L, int lower, int upper) {
  if (upper == lower) return lower, lower; // n=1
  if (upper == lower+1)                    // n=2
    return max(L[upper], L[lower]),
           min(L[upper], L[lower]); // 1 compare
  mid = (lower + upper)/2;                 // n>2
  max1, min1 = Find_Max_Min(L, lower, mid);
  max2, min2 = Find_Max_Min(L, mid+1, upper);
  return max(L[max1], L[max2]),
         min(L[min1], L[min2]);
}
```

Recurrence:

$$f(n) = \begin{cases} 2f(n/2) + 2 & n > 2 \\ 1 & n = 2 \end{cases}$$

no notes

Solving the Recurrence (1)

Assume $n = 2^k$.
Let's expand the recurrence a bit.

$$\begin{aligned} f(n) &= 2f(n/2) + 2 \\ &= 2[2f(n/4) + 2] + 2 \\ &= 4f(n/4) + 4 + 2 \\ &= 4[2f(n/8) + 2] + 4 + 2 \\ &= 8f(n/8) + 8 + 4 + 2 \\ &= 2^i f(n/2^i) + \sum_{j=1}^{i} 2^j \end{aligned}$$

no notes

Solving the Recurrence (2)

$$\begin{aligned} f(n) &= 2^{k-1} f(n/2^{k-1}) + \sum_{j=1}^{k-1} 2^j \\ &= 2^{k-1} f(2) + \sum_{j=1}^{k-1} 2^j \\ &= 2^{k-1} + \sum_{j=1}^{k-1} 2^j \\ &= n/2 + 2^k - 2 \\ &= 3n/2 - 2 \end{aligned}$$

no notes

Looking Closer (1)

But its not always true that $n = 2^k$.
The true cost recurrence is:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 2 & n > 2 \end{cases}$$

Here is what really happens:

n	2	3	4	5	6	7	8	9	10	11
$f(n)$	1	2	4	6	8	9	10	12	14	16
$3n/2 - 2$	1	2.5	4	5.5	7	8.5	10	11.5	13	14.5

The true cost for $f(n)$ ranges between $3n/2 - 2$ and $5n/3 - 2$.

- For what sort of input does the algorithm work best?

no notes

Finding a Better Algorithm

What is the cost with six values?

What if we divide into a group of 4 and a group of 2?

With divide and conquer, we seek to minimize the work, not necessarily balance the input sizes.

When does the algorithm do its best?

What about 12? 24?

Lesson: For divide and conquer, pay attention to what happens for small *n*.

8

Only need 7.

When each part is a power of 2.

8 vs. 4. 16 vs. 8.

Algorithms from Recurrences (1)

What does this model?

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \min_{1 \le k \le n-1}\{f(k) + f(n-k)\} + 2 & n > 2 \end{cases}$$

n	1	2	3	4	5	6	7	8
3	$\underline{3}$	$\underline{3}$						
4	5	$\underline{4}$	5					
5	7	$\underline{6}$	$\underline{6}$	7				
6	9	$\underline{7}$	8	$\underline{7}$	9			
7	11	$\underline{9}$	$\underline{9}$	$\underline{9}$	$\underline{9}$	11		
8	13	$\underline{10}$	11	$\underline{10}$	11	$\underline{10}$		13
9	15	$\underline{12}$	$\underline{12}$	$\underline{12}$	$\underline{12}$	$\underline{12}$	$\underline{12}$	15

$k = 2$ looks promising.

no notes

Algorithms from Recurrences (2)

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ f(2) + f(n-2) + 2 & n > 2 \end{cases}$$

Cost: What is the corresponding algorithm?

$f(n) = 3/2n - 2$.

The Lower Bound (1)

Is $\lceil 3n/2 \rceil - 2$ optimal?

Consider all states that a successful algorithm must go through: The **state space** lower bound.

At any given instant, track the following four categories:

- Novices: not tested.
- Winners: Won at least once, never lost.
- Losers: Lost at least once, never won.
- Moderates: Both won and lost at least once.

no notes

The Lower Bound (2)

Who can get ignored?

What is the initial state?

What is the final state?

How is this relevant?

The Lower Bound (2)

Moderates – Can't be min or max.

Initial: (n, 0, 0, 0).

Final: (0, 1, 1, n-2).

We must go from the initial state to the final state to solve the problem.
So, we can analyze how this gets done.

Lower Bound (3)

Every algorithm must go from $(n, 0, 0, 0)$ to $(0, 1, 1, n-2)$.

There are 10 types of comparison.

Comparing with a moderate cannot be more efficient than other comparisons, so ignore them.

Lower Bound (3)

That gets rid of 4 types of comparisons.

Lower Bound (3)

If we are in state (i, j, k, l) and we have a comparison, then:

$$
\begin{array}{lllll}
N:N & (i-2, & j+1, & k+1, & l) \\
W:W & (i, & j-1, & k, & l+1) \\
L:L & (i, & j, & k-1, & l+1) \\
L:N & (i-1, & j+1, & k, & l) \\
\quad or & (i-1, & j, & k, & l+1) \\
W:N & (i-1, & j, & k+1, & l) \\
\quad or & (i-1, & j, & k, & l+1) \\
W:L & (i, & j, & k, & l) \\
\quad or & (i, & j-1, & k-1, & l+2)
\end{array}
$$

Lower Bound (3)

no notes

Adversarial Argument

What should an adversary do?

- Comparing a winner to a loser is of no value.

Only the following five transitions are of interest:

$$
\begin{array}{lllll}
N:N & (i-2, & j+1, & k+1, & l) \\
L:N & (i-1, & j+1, & k, & l) \\
W:N & (i-1, & j, & k+1, & l) \\
\hline
W:W & (i, & j-1, & k, & l+1) \\
L:L & (i, & j, & k-1, & l+1)
\end{array}
$$

Only the last two types increase the number of moderates, so there must be $n-2$ of these.

The number of novices must go to 0, and the first is the most efficient way to do this: $\lceil n/2 \rceil$ are required.

Adversarial Argument

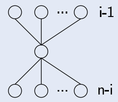
Minimize information gained.

Adversary will just make the winner win – No new information is provided.

This provides an algorithm. Think about it and you will see "MinMax" program.

Finding the ith Best

- We need to find the following poset:



- We don't care about the relative order within the upper and lower groups.
- Can we do better than sorting? ($\Theta(n \log n)$)
- Can we tighten the lower bound beyond n?
- What if we want to find the median element?

Hopefully, since less information is required.

No – the ith element could be *any* of the inputs.

This is probably the hardest.

Splitting a List

Given an arbitrary element, partition the list into those elements less and those elements greater.

```
// Initially, l and r are one position to left and
//   right of the subarray, respectively
int partition(Elem A[], int l, int r, Elem pivot) {
  do {             // Move bounds inward to meet
    while (A[++l] < pivot); // Move l right and
    while ((l < r) && (A[--r] > pivot)); // r left
    swap(A, l, r);          // Swap values
  } while (l < r);          // Stop when they cross
  return l;      // Return first position on right
}
```

If the pivot position is ith best, we are done.
If not, solve the subproblem recursively.

no notes

Cost (1)

What is the worst case cost of this algorithm?
Under what circumstances?

What is average case cost if we pick pivots at random?

- Let $f(n, i)$ be average time to find ith best of n elements.
- Array bounds go from 1 to n
- Call j the position of the pivot

$$f(n,i) = (n-1) + \frac{1}{n}\sum_{j=i+1}^{n} f(j-1,i) + \frac{1}{n}0 + \frac{1}{n}\sum_{j=1}^{i-1} f(n-j,i-j).$$

$\Theta(n^2)$ for bad pivots.
We will find average case cost by summing all the costs for all the cases, and divide by number of cases.

First part is partion cost, next is when $i < j$, then when $i = j$, and finally, the case when $i > j$.

Cost (2)

Let $f(n)$ be the cost averaged over all i.

$$f(n) = \frac{1}{n}\sum_{i=1}^{n} f(n,i).$$

Note: Even if we just want to analyze for median-finding, still need to be able to solve for arbitrary i on recursive calls.

no notes

Technique (1)

$$
\begin{aligned}
nf(n) &= \sum_{i=1}^{n} f(n,i) \\
&= n^2 - n + \frac{1}{n}\sum_{i=1}^{n}\left\{\sum_{j=i+1}^{n} f(j-1,i) + \right. \\
&\qquad \left. \sum_{j=1}^{i-1} f(n-j, i-j)\right\}.
\end{aligned}
$$

It turns out that the two double sums are the same (just going from different directions).

Factor $n^2 - n$ out of $f(n,i)$ since there are n of them.
Swap columns for rows in the two inner sums, they are the same.

Technique (2)

$$
\begin{aligned}
nf(n) &= n^2 - n + \frac{2}{n}\sum_{j=1}^{n-1}\sum_{i=1}^{j} f(j,i) \\
&= n^2 - n + \frac{2}{n}\sum_{j=1}^{n-1} jf(j)
\end{aligned}
$$

Therefore,

$$n^2 f(n) = n^3 - n^2 + 2\sum_{j=1}^{n-1} jf(j).$$

This is an example of a **full history** recurrence.

The inner sum on the first line is the same as the two inner sums on the previous page... the diagonals are the first one's columns.
Note:

$$
\begin{aligned}
f(n) &= 1/n\sum_{i=1}^{n} f(n,i) \\
f(j) &= 1/j\sum_{i=1}^{j} f(j,i)
\end{aligned}
$$

So $jf(j) = \sum_{i=1}^{j} f(j,i)$.
Cancel out $1/n$.

Solving the Recurrence (1)

If we subtract the appropriate form of $f(n-1)$, most of the terms will cancel out.

$$
\begin{aligned}
&n^2 f(n) - (n-1)^2 f(n-1) \\
&\quad = n^3 - n^2 + 2\sum_{j=1}^{n-1} jf(j) \\
&\qquad -(n-1)^3 + (n-1)^2 - 2\sum_{j=1}^{n-2} jf(j) \\
&\quad = 3n^2 - 5n + 2 + 2(n-1)f(n-1) \\
\Rightarrow n^2 f(n) &= (n^2-1)f(n-1) + 3n^2 - 5n + 2.
\end{aligned}
$$

The two sums add up to $2(n-1)f(n-1)$.

Now add back $(n-1)^2 f(n-1)$ to get next line

Gather up $f(n-1)$ terms on both sides:
$n^2 - 2n + 1 + 2n - 2 = n^2 - 1$.

Solving the Recurrence (2)

Estimate:

$$
\begin{aligned}
n^2 f(n) &= (n^2-1)f(n-1) + 3n^2 - 5n + 2 \\
&< n^2 f(n-1) + 3n^2 \\
\Rightarrow f(n) &< f(n-1) + 3 \\
\Rightarrow f(n) &< 3n
\end{aligned}
$$

Therefore, $f(n)$ is in $O(n)$.

Does this mean that the worst case is linear?

No, we are just computing the average.

Improving the Worst Case

Want worst case linear algorithm.

Goal: Pick a pivot that guarentees discarding a fixed proportion of the elements.

Can't just choose a pivot at random.

Median would be ideal – too expensive.

Choose a constant c, pick the median of a sample of size n/c elements.

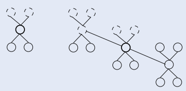
Will discard at least $n/2c$ elements.

no notes

Selecting an Approximate Median

Algorithm:

- Choose the $n/5$ medians for groups of 5 elements of L.
- Recursively, select the median of the $n/5$ elements.
- Use `partition` to partition the list into large and small elements around the "median."



- For 5, discard at least 2
- For 15, discard at least 5
- For 25, discard at least 8
- In general, discard at least $(3n+5)/10$

Can find median of 5 values in 6 compares.

Constructive Induction (1)

Is the following recurrence linear?

$$f(n) \leq f(\lceil n/5 \rceil) + f(\lceil (7n-5)/10 \rceil) + 6\lceil n/5 \rceil + n - 1.$$

To answer this, assume it is true for some constant r such that $f(n) \leq rn$ for all n greater than some bound.

$$\begin{aligned} f(n) &\leq f(\lceil \frac{n}{5} \rceil) + f(\lceil \frac{7n-5}{10} \rceil) + 6\lceil \frac{n}{5} \rceil + n - 1 \\ &\leq r(\frac{n}{5}+1) + r(\frac{7n-5}{10}+1) + 6(\frac{n}{5}+1) + n - 1 \\ &\leq (\frac{r}{5} + \frac{7r}{10} + \frac{11}{5})n + \frac{3r}{2} + 5 \\ &\leq \frac{9r+22}{10}n + \frac{3r+10}{2} \leq rn. \end{aligned}$$

Parts:

- Median of sample
- Largest possible fraction to find in recursive call – due to "select median of medians" process.
- Find median of 5 elements in 6 passes.
- Partition

Apply hypothesis

Constructive Induction (2)

Try $r = 1$: $3.1n + 7.5 \leq n$ which doesn't work.
Try $r = 23$: Get $22.9n + 39.5 \leq 23n$.
This is true for $n \geq 395$.

Thus, we can use induction to prove that,

$$\forall n \geq 395, f(n) \leq 23n.$$

This algorithm is not practical. Better to rely on "luck."

no notes

Changing the Model (1)

What if we settle for the “approximate best?”

Types of guarentees, given that the algorithm produces X and the best is Y:

1. $X = Y$. [Deterministic algorithm]
2. X’s rank is “close to” Y’s rank: [Approximation]

$$rank(X) \leq rank(Y) + \text{ “small”}.$$

3. X is “usually” Y. [Probabilistic]

$$\mathbf{P}(X = Y) \geq \text{ “large”}.$$

4. X’s rank is “usually” “close” to Y’s rank. [Heuristic]

no notes

Changing the Model (2)

We can also sacrifice reliability for speed:

1. We find the best, “usually” fast.
2. We find the best fast, or we don’t get an answer at all (but fast).

This is good if we can re-run with equal, *independent* probability of getting the correct answer.

Examples for Findmax

Choose m elements at random, and pick the best.

- For large n, if $m = \log n$, the answer is pretty good.
- Cost is $m - 1$.
- Rank is $\frac{mn}{m+1}$.

An approximation algorithm.

“Rank” meaning average best rank.
For $n = 1000$, that is $10/11n$ (top 100).
For $n = 1,000,000$, that is $20/21n$ (top 50k).

Probabilistic Algorithms

Probabilistic algorithms include steps that are affected by **random** events.

Problem: Pick one number in the upper half of the values in a set.

1. Pick maximum: $n - 1$ comparisons.
2. Pick maximum from just over 1/2 of the elements: $n/2$ comparisons.

Can we do better? Not if we want a **guarantee**.

no notes

Probabilistic Algorithm

Pick 2 numbers and choose the greater.

This will be in the upper half with probability 3/4.

Not good enough? Pick more numbers!

For k numbers, greatest is in upper half with probability $1 - 2^{-k}$.

Monte Carlo Algorithm: Good running time, result not guaranteed.

Las Vegas Algorithm: Result guaranteed, but not running time.

Pick k large enough, and the chance for failure becomes less than the chance that the machine will crash (e.g., probability that determinisitic algorithm will give no answer).

Think that you would rather have no answer than the wrong answer? If k is big enough, the probability of a wrong answer is less than that of any calamity (with non-zero probability) that you can think of – with this probability independent of n, and independent of the data.

An example would be finding a value in an array by guessing a position.

Sorting

Initial model:

- Sort key has a linear order (comparable).
- We have an array of elements.
- We wish to sort the elements in the array.
- We get information about elements only by comparison of two elements.
- We can preserve order information only by swapping a pair of elements.

To simplify analysis:

- Assume all elements are unique.
- For analysis purposes, consider the input to be a permutation of the values 1 to n.

What if the ALGORITHM could make this assumption?

With this assumption, the algorithm could just be simple binsort. The goal is to simplify our *analysis*, not our *problem*.

Swap Sorts (1)

Repeatedly scan input, swapping any out-of-order elements.

Bubble sort: $O(n^2)$ in worst case.

Inversions of an element: the number of smaller elements to the right of the element.

The sum of inversions for all elements is the number of swaps required by bubblesort.

ANY algorithm that removes one inversion per swap requires at least this many swaps.

no notes

Swap Sorts (2)

Worst number of inversions:

Best number of inversions:

Average number of inversions:

- Note that the sum of the total inversions for any permutation and its reverse is $\frac{n(n-1)}{2}$.
- Alternative view: every one of the $\frac{n(n-1)}{2}$ possible inversions occurs in a given permutation or its reverse.

Worst # inversions:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Best # inversions: 0

So, $n(n-1)/4$ on average.

Heap Sort (1)

Heap: complete binary tree with the value of any node at least as large as its two children.

Algorithm:

- Build the heap.
- Repeat n times:
  - Remove the root.
  - Repair the heap.

This gives us list in reverse sorted order.

Since the heap is a complete binary tree, it can be stored in an array.

no notes

Heap Sort (2)

To delete max element:

- Swap the last element in the heap with the first (root).
- Repeatedly swap the placeholder with larger of its two children until done.

no notes

Building the heap

To build a heap, first heapify the two subheaps, then push down the root to its proper position.

- Cost: $f(n) \leq 2f(n/2) + 2\log n$.

Alternatively: Start at first internal node and, moving up the array, siftdown each element.

- Cost:

$$\begin{aligned} f(n) &= \sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} \\ &= \frac{n}{2}\sum_{i=1}^{\log n - 1}\frac{i}{2^i} < 2\frac{n}{2} = n. \end{aligned}$$

Distance from bottom $\times$ # of nodes at that distance.

This is an example where exponential growth works in your favor. A *lot* of the elements are at the bottom, where they do not have much work to do.

Quicksort

Algorithm:

- Pick a pivot value.
- Split the array into elements less than the pivot and elements greater than the pivot.
- Recursively sort the sublists.

Worst case:

Pick the pivot at random, so that no particular input has bad performance.

n^2

Quicksort Average Cost (1)

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1)) & n > 1 \end{cases}$$

Since the two halves of the summation are identical,

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} f(i) & n > 1 \end{cases}$$

Multiplying both sides by n yields

$$nf(n) = n(n-1) + 2\sum_{i=0}^{n-1} f(i).$$

Why multiply by n? Because otherwise (when we subtract later) you get

$$f(n) - f(n-1) = (n-1) - (n-2) + \frac{2}{n}\sum_{i=0}^{n-1} f(i) - \frac{2}{n-1}\sum_{i=0}^{n-2} f(i)$$

which is no improvement!

Average Cost (2)

Get rid of the full history by subtracting $nf(n)$ from $(n+1)f(n+1)$

$$\begin{aligned} nf(n) &= n(n-1) + 2\sum_{i=1}^{n-1} f(i) \\ (n+1)f(n+1) &= (n+1)n + 2\sum_{i=1}^{n} f(i) \\ (n+1)f(n+1) - nf(n) &= 2n + 2f(n) \\ (n+1)f(n+1) &= 2n + (n+2)f(n) \\ f(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1}f(n). \end{aligned}$$

no notes

Average Cost (3)

Note that $\frac{2n}{n+1} \leq 2$ for $n \geq 1$. Expanding the recurrence, we get

$$\begin{aligned} f(n+1) &\leq 2 + \frac{n+2}{n+1}f(n) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}f(n-1)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1}f(n-2)\right)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2}f(1))\right) \end{aligned}$$

no notes

Average Cost (3)

$$\begin{aligned} &= 2\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots \right. \\ &\qquad \left. + \frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right) \\ &= 2\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right) \\ &= 2 + 2(n+2)(\mathcal{H}_{n+1} - 1) \\ &= \Theta(n \log n). \end{aligned}$$

$\mathcal{H}_n$ is the Harmonic series.

This actually just tells us $O(n \log n)$, but

$$\mathcal{H}_n = \sum_{i=1}^{n} 1/i = \Theta(\log n).$$

Lower Bound for Sorting (1)

What is the smallest number of comparisons needed to sort n values?

Clearly, sorting is as hard as finding the min and max element: $\lceil 3n/2 \rceil - 2$.

- Why?

<u>**Information theory**</u> says that, if an algorithm uses only binary decisions to distinguish between n possibilities, then it must use at least $\log n$ such decisions on average.

How is this relevant?

Because, if it weren't, we could sort and then get the min and max elements from the sorted list. This is an example of a *reduction*.

Comparisons are binary decisions. There are $n!$ possible inputs.

Lower Bound for Sorting (2)

There are $n!$ permutations to the input array.

So, by information theory, we need at least $\log n! = \Theta(n \log n)$ comparisons.

Using the decision tree model, what is the average depth of a node?

This is also $\Theta(\log n!)$.

$\log n -$ (1 or 2).

Linear Insert Sort

Put the element i into a sorted list of the first $i - 1$ elements.

Worst case cost:

Best case cost:

Average case cost:

What if we use binary search? (This is called binary insert sort.)

$\Theta(n^2)$: Each element does $i - 1$ comparisons.

n (1 comparison each).

$$\frac{n(n-1)}{4}$$

Cuts # of comparisons – does not change # of swaps.

Optimal Sorting (1)

If we count ONLY comparisons, binary insert sort is pretty good.

What is the absolute minimum number of comparisons needed to sort?

For $n = 5$, how many comparisons do we need for binary insert sort?

Binary search is best for what values of n?

Binary search is worst for what values of n?

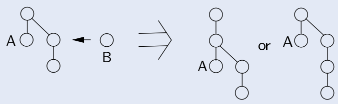
Binary insert sort: $1 + 2 + 2 + 3 = 8$ compares.

Best for $2^i - 1$

Worst for 2^i

Optimal Sorting (2)

Build the following poset:



Now, put in the fifth element (B) into the chain of 3.

Now, put in the off-element (A).

Total cost?

In two steps

2 compares

2 compare

7 compares

Ten Elements

Pair the elements: 5 comparisons.

Sort the winners of the pairings, using the previous algorithm: 7 comparisons.

Now, all we need to do is to deal with the original losers.

General algorithm:

- Pair up all the nodes with $\lfloor \frac{n}{2} \rfloor$ comparisons.
- Recursively sort the winners.
- Fold in the losers.

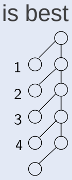
no notes

Finishing the Sort (1)

We will use binary insert to place the losers.

However, we are free to choose best ordering for inserting.

Recall that binary search is best for $2^k - 1$ items.



no notes

Finishing the Sort (2)

Pick the order of inserts to optimize the binary searches.

- 3 (2 compares: size 3)
- 4 (2 compares: size either 2 or 3, depending on where element 3 ends up)
- 1 (3 compares: size between 5 and 7)
- 2 (3 compares: size between 5 and 7)

We can form an algorithm: Binary Merge.

This sort is called **merge insert sort**

When we insert one of these numbers into the chain, we are concerned about everything on the chain below were that number comes in.

Total cost: $5 + 7 + 10 = 22$ compares.

Also called the Ford-Johnson sort.

Optimal Sort Algorithm?

- Merge insert sort is pretty good, but is it optimal?
- It does not match the information theoretic lower bound for $n = 12$.
  - Merge insert sort gives 30 instead of 29 comparison.
- BUT, exhaustive search shows the information theoretic bound is an underestimate for $n = 12$. 30 is best.
- Call the optimal worst cost for n elements $S(n)$.
  - $S(n+1) \leq S(n) + \lceil \log(n+1) \rceil$.
    Otherwise, we would sort n elements and binary insert the last.
  - For all n and m, $S(n+m) \leq S(n) + S(m) + M(m,n)$ for $M(m,n)$ the best time to merge two sorted lists.
  - For $n = 47$, we can do better by splitting into pieces of size 5 and 42, then merging.

$\lceil \log n! \rceil = 29$

Try *every* possible combination of comparison.

A Truly Optimal Algorithm

Pick the best set of comparisons for size 2.

Then for size 3, 4, 5, ...

Combine them together into one program with a big case statement.

Is this an algorithm?

No. Program size grows with size of n.
Algorithms must be of finite (fixed) length.

Note: There is no *particular* limit to the size of any partiulcar program. But, the program lenght must be fixed to *something*.

Numbers

Examples of problems:

- Raise a number to a power.
- Find common factors for two numbers.
- Tell whether a number is prime.
- Generate a random integer.
- Multiply two integers.

These operations use all the digits, and cannot use floating point approximation.

For large numbers, cannot rely on hardware (constant time) operations.

- Measure input size by number of binary digits.
- Multiply, divide become expensive.

n^2 for operations on numbers with n digits.

Analysis of Number Problems

Analysis problem: Cost may depend on properties of the number other than size.

- It is easy to check an even number for primeness.

Considering cost over all k-bit inputs, cost grows with k.

Features:

- Arithmetical operations are not cheap.
- There is only one instance of value n.
- There are 2^k instances of length k or less.
- The size (length) of value n is $\log n$.
- The cost may decrease when n increases in value, but generally increases when n increases in size (length).

So, we can go back to our normal intuition about cost growing with size (as opposed to special properties of value).

multiplication is *much* worse than add, divide is worse still.

Actually, 2^{k-1} have length exactly k.

Exponentiation (1)

How do we compute m^n?

We could multiply $n - 1$ times.
Can we do better?

Approaches to divide and conquer:

- Relate m^n to k^n for $k < m$.
- Relate m^n to m^k for $k < n$.

If n is even, then $m^n = m^{n/2}m^{n/2}$.

If n is odd, then $m^n = m^{\lfloor n/2 \rfloor}m^{\lfloor n/2 \rfloor}m$.

Why bother? Because the input size is $\Theta(\log n)$, so naive algorithm is exponential!

That is, take same power of a smaller number. $6^8 = 2^8 \cdot 3^8$.

That is, take smaller power of some number. $6^8 = 6^4 \cdot 6^4$.

Exponentiation (2)

```
int Power(int base, int exp) {
  int half, total;
  if exp = 0 return 1;
  half = Power(base, exp/2);
  total = half * half;
  if (odd(exp)) then total = total * base;
  return total;
}
```


no notes

Analysis of Power

$$f(n) = \begin{cases} 0 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 + n \bmod 2 & n > 1 \end{cases}$$

Solution: $f(n) = \lfloor \log n \rfloor + \beta(n) - 1$
where β is the number of 1's in binary representation of n.

How does this cost compare with the problem size?

Is this the best possible? What if $n = 15$?

What if n stays the same but m changes over many runs?

In general, finding the best set of multiplications is expensive (probably exponential).

$n \bmod 2$ is extra cost for odd.

Problem size is $\log n$, so linear.

Best to compute $n^5 \cdot n^5 \cdot n^5$. n^5 takes 3 multiplies, then 2 to combine, for 5 total. "Normal" algorithm takes 7 multiplies.

Compute and store the best multiplication ordering.

In fact, it is $\mathcal{NP}$-complete, but I've not defined this term yet. This is $O(2^n)$ work. Note that the "standard" exponential algorithm is $(O(\log n))$(cost to multiply) which is $(O(\log n))(\log m)^2$. So it isn't quite a direct comparison.

Largest Common Factor (1)

The largest common factor of two numbers is the largest integer that divides both evenly.

Observation: If k divides n and m, then k divides $n - m$.

So, $f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n)$.

Observation: There exists k and l such that

$$n = km + l \text{ where } m > l \geq 0.$$

$$n = \lfloor n/m \rfloor m + n \bmod m.$$

So, $f(n, m) = f(m, l) = f(m, n \bmod m)$.

Assuming $n > m$, then $n = ak$, $m = bk$, $n - m = (a - b)k$, for a, b integers.

For $n > m$. l is remainder.

From definition of mod.
LCF is of course a factor of n and km, so it is also a factor of l, since we just remove a multiple of it from n.
Example: $n = 35$, $m = 14$. Find 35, 14 $\Rightarrow$ find 14, 7 $\Rightarrow$ 7, 0.
Done.

Largest Common Factor (2)

$$f(n, m) = \begin{cases} n & m = 0 \\ f(m, n \bmod m) & m > 0 \end{cases}$$

```
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```


no notes

Analysis of LCF

How big is $n \bmod m$ relative to n?

$$\begin{aligned} n \geq m \quad &\Rightarrow \quad n/m \geq 1 \\ &\Rightarrow \quad 2\lfloor n/m \rfloor > n/m \\ &\Rightarrow \quad m\lfloor n/m \rfloor > n/2 \\ &\Rightarrow \quad n - n/2 > n - m\lfloor n/m \rfloor = n \bmod m \\ &\Rightarrow \quad n/2 > n \bmod m \end{aligned}$$

The first argument must be halved in no more than 2 iterations.

Total cost:

Depends in part on how big m is relative to n.

Multiply both sides by $m/2$.

By definition of $n \bmod m$.

Can split in half $\log n$ times. So $2 \log n$ is upper bound.
Note that this is *linear* on problem size, since problem *size* is $2 \log n$ (2 numbers).

Reminder: This upper bound is not necessarily tight!

Matrix Multiplication

Given: $n \times n$ matrices A and B.

Compute: $C = A \times B$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Straightforward algorithm:

- $\Theta(n^3)$ multiplications and additions.

Lower bound for any matrix multiplication algorithm: $\Omega(n^2)$.

Not quite as bad as it first looks, since input size is n^2.

Because we create n^2 outputs.

Another Approach

Compute:

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 &= (a_{11} + a_{12})b_{22} \\ m_5 &= a_{11}(b_{12} - b_{22}) \\ m_6 &= a_{22}(b_{21} - b_{11}) \\ m_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

Then:

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7 \end{aligned}$$

7 multiplications and 18 additions/subtractions.

Verify:

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ &= (a_{12} - a_{22})(b_{21} + b_{22}) + (a_{11} + a_{22})(b_{11} + b_{22}) \\ &\quad -(a_{11} + a_{12})b_{22} + a_{22}(b_{21} - b_{11}) \\ &= a_{12}b_{21} + a_{12}b_{22} - a_{22}b_{21} - a_{22}b_{22} + a_{11}b_{22} + a_{22}b_{11} \\ &\quad + a_{22}b_{22} - a_{11}b_{22} - a_{12}b_{22} + a_{22}b_{21} - a_{22}b_{11} \\ &= a_{11}b_{11} + a_{12}b_{21} \end{aligned}$$

Strassen's Algorithm (1)

(1) Trade more additions/subtractions for fewer multiplications in 2×2 case.

(2) Divide and conquer.

In the straightforward implementation, 2×2 case is:

$$
\begin{aligned}
c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\
c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\
c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\
c_{22} &= a_{21}b_{12} + a_{22}b_{22}
\end{aligned}
$$

Requires 8 multiplications and 4 additions.

no notes

Strassen's Algorithm (2)

Divide and conquer step:

Assume n is a power of 2.

Express $C = A \times B$ in terms of $\frac{n}{2} \times \frac{n}{2}$ matrices.

$$
\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}
$$

no notes

Strassen's Algorithm (3)

By Strassen's algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of $n/2 \times n/2$ matrices.

Recurrence:

$$
\begin{aligned}
T(n) &= 7T(n/2) + 18(n/2)^2 \\
T(n) &= \Theta(n^{\log_2 7}) = \Theta(n^{2.81}).
\end{aligned}
$$

Current "fastest" algorithm is $\Theta(n^{2.376})$

Open question: Can matrix multiplication be done in $O(n^2)$ time?

From recurrence Master Theorem. But this has a high constant due to the additions.

But is impractical due to overhead.

Divide and Conquer Recurrences (1)

These have the form:

$$
\begin{aligned}
T(n) &= aT(n/b) + cn^k \\
T(1) &= c
\end{aligned}
$$

... where a, b, c, k are constants.

A problem of size n is divided into a subproblems of size n/b, while cn^k is the amount of work needed to combine the solutions.

no notes

Divide and Conquer Recurrences (2)

Expand the sum; assume $n = b^m$.

$$\begin{aligned} T(n) &= a(aT(n/b^2) + c(n/b)^k) + cn^k \\ &= a^m T(1) + a^{m-1}c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\ &= ca^m \sum_{i=0}^{m} (b^k/a)^i \end{aligned}$$

$a^m = a^{\log_b n} = n^{\log_b a}$
The summation is a geometric series whose sum depends on the ratio

$$r = b^k/a.$$

There are 3 cases.

Set $a = b^{\log_b a}$. Switch order of logs, giving $(b^{\log_b n})^{\log_b a} = n^{\log_b a}$.

D & C Recurrences (3)

(1) $r < 1$

$$\sum_{i=0}^{m} r^i < 1/(1-r), \qquad \text{a constant.}$$

$$T(n) = \Theta(a^m) = \Theta(n^{\log_b a}).$$

(2) $r = 1$

$$\sum_{i=0}^{m} r^i = m + 1 = \log_b n + 1$$

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^k \log n)$$

$T(n) = 2T(n/2) + 1$.
$r = 2^0/2 = 1/2$.
$\Theta(n^{\log_2 2}) = n$

Since $r = b^k/a$, $a = b^k$, $k = \log_b a$.

$T(n) = n^0 \log n = \log n$.
$T(n) = 2T(n/2) + n$.
$r = 2^1/2 = 1$.
$T(n) = n^1 \log n = n \log n$.

D & C Recurrences (4)

(3) $r > 1$

$$\sum_{i=0}^{m} r^i = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$$

So, from $T(n) = ca^m \sum r^i$,

$$\begin{aligned} T(n) &= \Theta(a^m r^m) \\ &= \Theta(a^m (b^k/a)^m) \\ &= \Theta(b^{km}) \\ &= \Theta(n^k) \end{aligned}$$

$T(n) = 3T(n/4) + n$.
$r = 4^1/3$. So $T(n) = n^1 = \Theta(n)$.

$T(n) = T(n/2) + n^2$.
$r = 2^2/1$. So $T(n) = \Theta(n^2)$.

Strassen's Algorithm: $T(n) = 7T(n/2) + n^2$.
$r = 2^2/7$, so $r < 1$. $T(n) = \Theta(n^{\log_2 7})$.

Summary

Theorem 3.4:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

Apply the theorem:
$T(n) = 3T(n/5) + 8n^2$.
$a = 3, b = 5, c = 8, k = 2$.
$b^k/a = 25/3$.

Case (3) holds: $T(n) = \Theta(n^2)$.

no notes

Prime Numbers

How do we tell if a number is prime?

One approach is the prime sieve: Test all prime up to $\lfloor\sqrt{n}\rfloor$.

This requires up to $\lfloor\sqrt{n}\rfloor - 1$ divisions.

- How does this compare to the input size?

Note that it is easy to check the number of times 2 divides n for the binary representation

- What about 3?
- What if n is represented in trinary?

Is there a polynomial time algorithm?

Exponential, since problem size is $\log n$.

Not easy.

Now easy to check for 3.

We don't know of one. What if we are willing to settle for a probabilistic algorithm?

Facts about Primes

Some useful theorems from Number Theory:

- **Prime Number Theorem**: The number of primes less than n is (approximately)
$$\frac{n}{\ln n}$$
  - The average distance between primes is $\ln n$.
- **Prime Factors Distribution Theorem**: For large n, on average, n has about $\ln \ln n$ different prime factors with a standard deviation of $\sqrt{\ln \ln n}$.

To prove that a number is composite, need only one factor.
What does it take to prove that a number is prime?
Do we need to check all $\sqrt{n}$ candidates?

This is quite small. For 2^{32}, $\log \log n = 5$.Much harder than proving it is composite!

Depends on how safe you want to be. (Actually, only need to check primes $< \sqrt{n}$)

Probablistic Algorithms

Some probablistic algorithms:

- $\text{Prime}(n) = \text{FALSE}$.
- With probability $1/\ln n$, $\text{Prime}(n) = \text{TRUE}$.
- Pick a number m between 2 and $\sqrt{n}$. Say n is prime iff m does not divide n.

Using number theory, can create cheap test that determines a number to be composite (if it is) 50% of the time.

```
Prime(n) {
  for(i=0; i<COMFORT; i++)
    if !CHEAPTEST(n)
      return FALSE;
  return TRUE;
}
```

Of course, this does nothing to help you find the factors!

Works, except $1/\mathit{logn}$ times on average.
No improvement.
Not much help. Probably did *not* pick a factor!

One nice side effect: We actually use large primes for cryptography. The numbers used don't actually need to be prime. They only need to be hard to factor! And those numbers that continually pass the cheap 50/50 test tend to be hard to factor. So, even if a non-prime is used, it will still probably succeed in its intended use!

Random Numbers

Which sequences are random?

- 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
- 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
- 2, 7, 1, 8, 2, 8, 1, 8, 2, ...

Meanings of "random":

- Cannot predict the next item: **unpredictable**.
- Series cannot be described more briefly than to reproduce it: **equidistribution**.

There is no such thing as a random number sequence, only "random enough" sequences.

A sequence is **pseudorandom** if no future term can be predicted in polynomial time, given all past terms.

Which series of 9 digits is "most likely"? Answer: Every one is equally likely!

Most people are notoriously bad at "inventing" random sequences, or recognizing them. It stems from the fact that (a) most people don't have a gut-level understanding of probability, and (b) people expect that the global properties of randomness of the series will also apply locally. They tend to under-represent series of repeats.

A Good Random Number Generator

Most computer systems use a deterministic algorithm to select pseudorandom numbers.

Linear congruential method:

- Pick a **seed** $r(1)$. Then,

$$r(i) = (r(i-1) \times b) \bmod t.$$

Resulting numbers must be in what range?

What happens if $r(i) = r(j)$?

Must pick good values for b and t.

- t should be prime.

Numbers are in the range 0 to $t-1$.

Then $r(i+1) = r(j+1)$ and we get a repeating cycle.

Random Number examples

$r(i) = 6r(i-1) \bmod 13 =$
..., 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, ...

$r(i) = 7r(i-1) \bmod 13 =$
..., 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1, ...

$r(i) = 5r(i-1) \bmod 13 =$
..., 1, 5, 12, 8, 1, ...
..., 2, 10, 11, 3, 2, ...
..., 4, 7, 9, 6, 4, ...
..., 0, 0, ...

Suggested generator: $r(i) = 16807r(i-1) \bmod 2^{31} - 1$.

no notes

Introduction to the Sliderule

Compared to addition, multiplication is hard.

In the physical world, addition is merely concatenating two lengths.

Observation:

$$\log nm = \log n + \log m.$$

Therefore,

$$nm = \text{antilog}(\log n + \log m).$$

What if taking logs and antilogs were easy?

no notes

Introduction to the Sliderule (2)

The sliderule does exactly this!

- It is essentially two rulers in log scale.
- Slide the scales to add the lengths of the two numbers (in log form).
- The third scale shows the value for the total length.

This is an example of a transform. We do transforms to convert a hard problem into a (relatively) easy problem.

Representing Polynomials

A vector **a** of n values can uniquely represent a polynomial of degree $n-1$

$$P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} \mathbf{a}_i x^i.$$

Alternatively, a degree $n-1$ polynomial can be uniquely represented by a list of its values at n distinct points.

- Finding the value for a polynomial at a given point is called **evaluation**.
- Finding the coefficients for the polynomial given the values at n points is called **interpolation**.

That is, a polynomial can be represented by it coefficients.

Multiplication of Polynomials

To multiply two $n-1$-degree polynomials A and B normally takes $\Theta(n^2)$ coefficient multiplications.

However, if we evaluate both polynomials, we can simply multiply the corresponding pairs of values to get the values of polynomial AB.

Process:

- Evaluate polynomials A and B at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.

no notes

Multiplication of Polynomials (2)

This can be faster than $\Theta(n^2)$ IF a fast way can be found to do evaluation/interpolation of $2n-1$ points (normally this takes $\Theta(n^2)$ time).

Note that evaluating a polynomial at 0 is easy, and that if we evaluate at 1 and -1, we can share a lot of the work between the two evaluations.

Can we find enough such points to make the process cheap?

no notes

An Example

Polynomial A: $x^2 + 1$.
Polynomial B: $2x^2 - x + 1$.
Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

Notice:

$$\begin{aligned} AB(-1) &= (2)(4) = 8 \\ AB(0) &= (1)(1) = 1 \\ AB(1) &= (2)(2) = 4 \end{aligned}$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

	-1	0	1
A	2	1	2
B	4	1	2
AB	8	1	4

Nth Root of Unity

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient.

Complex number ω is a **primitive nth root of unity** if

1. $\omega^n = 1$ and
2. $\omega^k \neq 1$ for $0 < k < n$.

$\omega^0, \omega^1, ..., \omega^{n-1}$ are the **nth roots of unity**.

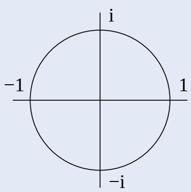
Example:

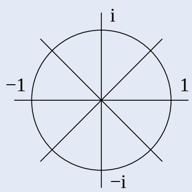
- For $n = 4$, $\omega = i$ or $\omega = -i$.

For the first circle, $n = 4, \omega = i$.

For the second circle, $n = 8, \omega = \sqrt{i}$.

Nth Root of Unity (cont)





$n = 4$, $\omega = i$.
$n = 8$, $\omega = \sqrt{i}$.

no notes

Discrete Fourier Transform

Define an $n \times n$ matrix $V(\omega)$ with row i and column j as

$$V(\omega) = (\omega^{ij}).$$

Example: $n = 4$, $\omega = i$:

$$V(\omega) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Let $\overline{a} = [a_0, a_1, ..., a_{n-1}]^T$ be a vector.
The **Discrete Fourier Transform** (DFT) of $\overline{a}$ is:

$$F_\omega = V(\omega)\overline{a} = \overline{v}.$$

This is equivalent to evaluating the polynomial at the nth roots of unity.

In the array, indexing begins with 0.

Example:
$1 + 2x + 3x^2 + 4x^3$
Values to evaluate at: $1, i, -1, -i$.

Array example

For $n = 8$, $\omega = \sqrt{i}$, $V(\omega) =$

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i} \end{matrix}$$

The key thing to note here is the symmetries in the array. This is what permits the fast algorithm to emerge. With suitable minor changes (like switching signs), we can easily share parts of the work through the recursion process.

Inverse Fourier Transform

The inverse Fourier Transform to recover $\overline{a}$ from $\overline{v}$ is:

$$F_{\omega}^{-1} = \overline{a} = [V(\omega)]^{-1} \cdot \overline{v}.$$

$$[V(\omega)]^{-1} = \frac{1}{n} V(\frac{1}{\omega}).$$

This is equivalent to interpolating the polynomial at the nth roots of unity.

An efficient divide and conquer algorithm can perform both the DFT and its inverse in $\Theta(n \lg n)$ time.

Just replace each ω with $1/\omega$

After substituting $1/\omega$ for ω.

Observe the sharable parts in the matrix.

Fast Polynomial Multiplication

Polynomial multiplication of A and B:

- Represent an $n-1$-degree polynomial as $2n-1$ coefficients:
$$[a_0, a_1, ..., a_{n-1}, 0, ..., 0]$$
- Perform DFT on representations for A and B.
- Pairwise multiply results to get $2n-1$ values.
- Perform inverse DFT on result to get $2n-1$ degree polynomial AB.

$\Theta(n \log n)$

$\Theta(n)$

$\Theta(n \log n)$

Total time: $\Theta(n \log n)$.

FFT Algorithm

```
FFT(n, a0, a1, ..., an-1, omega, var V);
Output: V[0..n-1] of output elements.
begin
  if n=1 then V[0] = a0;
  else
    FFT(n/2, a0, a2, ... an-2, omega^2, U);
    FFT(n/2, a1, a3, ... an-1, omega^2, W);
    for j=0 to n/2-1 do
      V[j] = U[j] + omega^j W[j];
      V[j+n/2] = U[j] - omega^j W[j];
end
```


no notes

Fibonacci Revisited (1)

Consider again the recursive function for computing the nth Fibonacci number.

```
int Fibr(int n) {
  if (n <= 1) return 1;         // Base case
  return Fibr(n-1) + Fibr(n-2); // Recursive call
}
```

Cost is Exponential. Why?

Lots of recomputation.

Fibonacci Revisited (2)

If we could eliminate redundancy, cost is greatly reduced.

- Keep a table

```
int Fibrt(int n, int* Values) {
  // Assume Values has at least n slots, and all
  // slots are initialized to 0
  if (n <= 1) return 1;      // Base case
  if (Values[n] == 0)        // Compute and store
    Values[n] = Fibrt(n-1, Values)
                 + Fibrt(n-2, Values);
  return Values[n];
}
```

Cost?
We don't need table, only last 2 values.

- Key is working bottom up.

Cost is only linear.
Of course, we can also do this iteratively.

Dynamic Programming (1)

The issue of avoiding recomputation of subproblems comes up frequently.

- General solution: Store a table to avoid recomputation.
- Can work bottom up (fill table from smallest to largest)
- Can work top down (recursively), remembering any subproblems that happen to be solved (check table first).

This approach is called **Dynamic Programming**

- Name comes from the field of dynamic control systems
- There, the act of storing precomputed values is referred to as "programming".

no notes

Dynamic Programming (2)

Dynamic Programming is an alternative to Divide and Conquer

- D&C: Split problem into subproblems, solve independently, and recombine.
- DP: Pay bookkeeping costs to remember solutions to shared subproblems.

no notes

A Knapsack Problem

Problem: Given an integer capacity K and n items such that item i has integer size k_i, find a subset of the n items whose sizes exactly sum to K, if possible.

Formally: Find $S \subset \{1, 2, ..., n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example:

- $K = 163$
- 10 items of sizes 4, 9, 15, 19, 27, 44, 54, 68, 73, 101.
- What if K is 164?

Instead of parameterizing problem just by n, parameterize with n and K.

- $P(n, K)$ is the problem with n items and capacity K.

9, 27, 54, 73.

9, 54,101.
The problem is that there is no necessary relationship between the answer for n and $n + 1$.

Solving the Knapsack Problem

Think about divide and conquer (alternatively, induction).

What if we know how to solve $P(n-1, K)$?

- If $P(n-1, K)$ has a solution, then it is a solution for $P(n, K)$.
- Otherwise, $P(n, K)$ has a solution $\Leftrightarrow P(n-1, K-k_n)$ has a solution.

What if we know how to solve $P(n-1, k)$ for $0 \leq k \leq K$?

Cost: $T(n) = 2T(n-1) + c$.

$T(n) = \Theta(2^n)$.

BUT... there are only $n(K+1)$ subproblems to solve!

There are two choices:
The nth item is in the solution OR
The nth item is not in the solution.

What does this mean? Drop the nth item.

Then we can solve $P(n-1, K-k_n)$.
Of course, we don't know if the nth item is in the solution or not, so...

$= 2(2T(n-2)+c)+c = 2(2(2T(n-3)+c)+c)+c$, etc.

Solution

Clearly, there are many subproblems being solved repeatedly.

Store a $n \times K + 1$ matrix to contain the solutions for all $P(i, k)$.

Fill in the rows from $i = 0$ to n, left to right.

If $P(n-1, K)$ has a solution,
Then $P(n, K)$ has a solution
Else If $P(n-1, K-k_n)$ has a solution
Then $P(n, K)$ has a solution
Else $P(n, K)$ has no solution.

Cost: $\Theta(nK)$.

no notes

Knapsack Example (1)

$K = 10$.

Five items: 9, 2, 7, 4, 1.

	0	1	2	3	4	5	6	7	8	9	10
$k_1=9$	O	–	–	–	–	–	–	–	–	I	–
$k_2=2$	O	–	I	–	–	–	–	–	–	O	–
$k_3=7$	O	–	O	–	–	–	–	I	–	I/O	–
$k_4=4$	O	–	O	–	I	–	I	O	–	O	–
$k_5=1$	O	I	O	I	O	I	O	I/O	I	O	I

no notes

Knapsack Example (2)

Key:

-: No solution for $P(i, k)$.
O: Solution(s) for $P(i, k)$ with i omitted.
I: Solution(s) for $P(i, k)$ with i included.
I/O: Solutions for $P(i, k)$ with i included AND omitted.

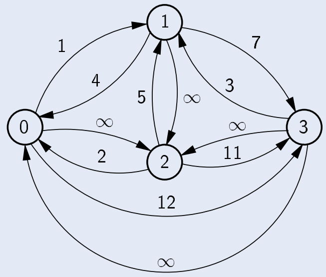
Example: $M(3, 9)$ contains O because $P(2, 9)$ has a solution.
It contains I because $P(2, 2) = P(2, 9-7)$ has a solution.

How can we find a solution to $P(5, 10)$?
How can we find ALL solutions to $P(5, 10)$?

no notes

All Pairs Shortest Paths (1)

For every vertex $u, v \in \mathbb{V}$, calculate d(u, v).
Define a **k-path** from u to v to be any path whose intermediate vertices all have indices less than k.



First, calculate all direct paths.
Then, calculate all 0 paths: For every i, j, look to see if i,0 + 0,j is less than i,j in the table.
Then, calculate all 1 paths: For every i, j, look to see if i,1 + 1,j is less than i,j in the table. And so on.

All Pairs Shortest Paths (3)

```
void Floyd(Graph& G) {    // All-pairs shortest paths
  int D[G.n()][G.n()];    // Store distances
  for (int i=0; i<G.n(); i++) // Initialize D
    for (int j=0; j<G.n(); j++)
      D[i][j] = G.weight(i, j);
  for (int k=0; k<G.n(); k++) // Compute all k paths
    for (int i=0; i<G.n(); i++)
      for (int j=0; j<G.n(); j++)
        if (D[i][j] > (D[i][k] + D[k][j]))
          D[i][j] = D[i][k] + D[k][j];
}
```


no notes

Reductions

A **reduction** is a transformation of one problem to another.

Purposes: To compare the difficulty of two problems.

- Use one algorithm to solve another problem (upper bound).
- Compare the relative difficulty of two problems (lower bound).

Notation: A problem is a mapping of inputs to outputs.
Format looks as follows:
SORTING:

- Input: A sequence of integers $x_0, x_1, ..., x_{n-1}$.
- Output: A permutation $y_0, y_1, ..., y_{n-1}$ of the sequence such that $y_i \leq y_j$ whenever $i < j$.

Its a particular type of transformation, done for a particular purpose.

“Code reuse.” Remember our transformation for FFT.

Reduction

PAIRING

PAIRING:

- Input: Two sequences of integers $X = (x_0, x_1, ..., x_{n-1})$ and $Y = (y_0, y_1, ..., y_{n-1})$.
- Output: A pairing of the elements in the two sequences such that the least value in X is paired with the least value in Y, and so on.

How can we solve this?

One algorithm:

- Sort X.
- Sort Y.
- Now, pair x_i with y_i for $0 \leq i < n$.

Terminology: We say that PAIRING is **reduced** to SORTING, since SORTING is used to solve PAIRING.

Reduce to the one being used.
Be careful: Most confusion comes with wich direction is meant on the reduction.

PAIRING Reduction Process

The reduction of PAIRING to SORTING requires 3 steps:

- Convert an instance of PAIRING to two instances of SORTING.
- Run SORTING (twice).
- CONVERT the output for the two instances of SORTING to an output for the original PAIRING instance.

What do we require about the transformations to make them useful?

What is the cost of this algorithm?

Transformation must be "fast."

$\Theta(n \log n)$.
The transformations are linear, so the cost is dominated by sorting.

PAIRING Lower Bound (1)

We have an upper bound for PAIRING equal to that of SORTING.

What is the lower bound for PAIRING?

Pretend that there is a $O(n)$ time algorithm for PAIRING. Consider this algorithm for SORTING:

- Transform SORTING to PAIRING with X being the input sequence for SORTING, and Y a sequence containing the values 0 through $n-1$
- Run the $O(n)$ time PAIRING algorithm.
- Take the pairs output by PAIRING and use a simple binsort to order them by the second value of the pair. The first items of the pair will be the sorted list.

Recall that lower bounds proofs are difficult.
Beware the "necessary fallacy:" There is no reason why a pairing algorithm *must* explicitly sort, nor that the resulting list be sorted.

PAIRING Lower Bound (2)

What is the cost of this algorithm?

What does this say about the existence of an $O(n)$ time algorithm for PAIRING?

$\Theta(n)$

It can't possibly exist, due to our known lower bound on sorting.

This is a proof by contradiction.
The only flaw in the process leading to the contradiction is the *assumption* of an $O(n)$ algorithm for PAIRING.

Reduction Process

Consider any two problems for which a suitable reduction from one to the other can be found.

The first problem $P1$ takes input instance $\mathbf{I}$ and transforms that to solution $\mathbf{S}$.

The second problem $P2$ takes input instance $\mathbf{I}'$ and transforms that to solution $\mathbf{S}'$.

A **reduction** is the following three-step process:

- Transform an arbitrary instance $\mathbf{I}$ of problem $P1$ and transform it to a (possibly special) instance $\mathbf{I}'$ of $P2$.
- Apply an algorithm for $P2$ to $\mathbf{I}'$, yielding $\mathbf{S}'$.
- Transform $\mathbf{S}'$ to a solution for $P1$ ($\mathbf{S}$). Note that $\mathbf{S}$ MUST BE THE CORRECT SOLUTION for $\mathbf{I}$!

It is important that the first transformation take an *arbitrary* instance of $\mathbf{I}$. We don't need to be able to produce every possible isntance of $\mathbf{I}'$. But we DO need to be able to handle every possible instance of $\mathbf{I}$.

Reduction Process (Cont.)

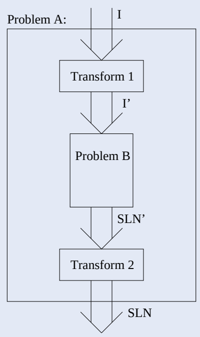
Note that reduction is NOT an algorithm for either problem.

It does mean, given “cheap” transformations, that:

- The upper bound for $P1$ is at most the upper bound for $P2$.
- The lower bound for $P2$ is at least the lower bound for $P1$.

no notes

General Black Box Diagram



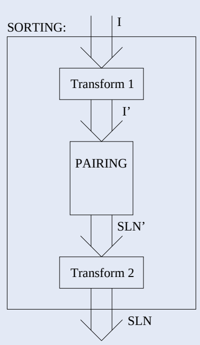
no notes

Notation Summary

- Problem A has input I, solution SLN
- Problem B has input I’, solution SLN’
- Problem A is reduced to Problem B
- Problem A is solved by reducing to Problem B (which has known upper bound)
- We prove a lower bound on B by a reduction from Problem A (which has known lower bound)
- Transformations 1 and 2 must be “cheap”
- We must be able to accept the full range of inputs I to Problem A.
- However, I’ may be a restricted subset of all possible inputs to B.

no notes

PAIRING Reduction Black Box



no notes

PAIRING Notation

- Transform 1 takes input I and produces output I'.
- I is a sequence S.
- I' is two sequences: S and the set of numbers from 0 to $n-1$.
- Transform 1 takes a sequence as input, and produces the two sequences as output.
- Transform 2 takes SLN' as input and produces output SLN.
- SLN' is a pairing.
- SLN is a sorted sequence
- Transform 2 takes the pairing and runs a binsort on it to generate the sorted sequence.

no notes

Another Reduction Example

How much does it cost to multiply two n-digit numbers?

- Naive algorithm requires $\Theta(n^2)$ single-digit multiplications.
- Faster (but more complicated) algorithms are known, but none so fast as to be $O(n)$.

Is it faster to square an n-digit number than it is to multiply two n-digit numbers?

- This is a special case, so might go faster.
- Answer: No, because

$$X \times Y = \frac{(X+Y)^2 - (X-Y)^2}{4}.$$

If a fast algorithm can be found for squaring, then it could be used to make a fast algorithm for multiplying.

no notes

Matrix Multiplication

Standard matrix multiplication for two $n \times n$ matrices requires $\Theta(n^3)$ multiplications.

Faster algorithms are known, but none so fast as to be $O(n^2)$.

A **symmetric** matrix is one in which $M_{ij} = M_{ji}$.

Can we multiply symmetric matrices faster than regular matrices?

$$\begin{bmatrix} 0 & \mathrm{A} \\ \mathrm{A}^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} 0 & \mathrm{B}^{\mathrm{T}} \\ \mathrm{B} & 0 \end{bmatrix} = \begin{bmatrix} \mathrm{AB} & 0 \\ 0 & \mathrm{A}^{\mathrm{T}}\mathrm{B}^{\mathrm{T}} \end{bmatrix}.$$

no notes

Some Puzzles

1. A hiker leaves at 8:00 AM and hikes over the mountain. The next day, she again leaves at 8:00 AM and returns to her starting point along the same path. Prove that there is a point on the path such that she was at that point at the same time on both days.

2. Take a chessboard and cover it with dominos (a domino covers two adjacent squares of the board). Now, remove the upper left and lower right corners of the board. Now, can it still be covered with dominos?

These puzzles have the quality that, while their answers may be hard to FIND, they are easy to CHECK.

3. Is 667 composite or prime?

Pretend that she is walking both ways on the same day. She must meet her self at some point (which means that she at the same place at the same time).

No. We lost two squares of the same color. A domino covers a square of each color. So it can only work when there are an equal number of squares of each color.

If I give you two factors, its easy to check. BUT if I claim the number is prime, how do you check? How do I prove to you that its prime? You have to do as much work verifying as I did solving.

Complexity and Computability (1)

Complexity:
- How cheaply can this be computed?
- How hard is this to compute?

Computability:
- When can this be computed?
- Can this be computed at all?

Upper bound, best algorithm.

Lower bound.

That is, what special cases or preconditions?

Some things are impossible.

Complexity and Computability (2)

Types of "hard" problems:
- Hard to understand (or specify) the problem
  - Software Engineering
- Hard to design a solution
  - Artificial Intelligence
- Hard to compute in reasonable time
  - Complexity Theory
- Hard (impossible) to do at all
  - Computability Theory

no notes

Hard Problems (1)

We say that a problem is computationally "hard" if the running time of the best known algorithm is exponential on the size of its input.

Support:
- Polynomials are closed under composition and addition.
  - Doing polynomial time operations in series is polynomial.
- All computers today are polynomially related.
  - If it takes polynomial time on one computer, it will take polynomial time on any other computer.
- Polynomial time is (generally) feasible, while exponential time is (generally) infeasible.
  - An empirical observation: For most polynomial-time algorithms, the polynomial is of low degree.

Conversely, polynomial-time algorithms are (relatively) "easy."

Hard Problems (2)

Note that for a faster machine, the size of problem that can be run in a fixed amount of time
- grows by a multiplicative factor for a polynomial-time algorithm.
- grows by an additive factor for an exponential-time algorithm.

no notes

Nondeterminism

- Imagine a computer that works by guessing the correct solution from among all possible solutions to a problem.
- Alternative: Super parallel machine that tests all possible solutions simultaneously.
- It might solve some problems more quickly than a regular computer.
- Consider a problem which, when given a proposed solution, we can check in polynomial time if the solution is correct.
- Even if the number of guesses is exponential, checking (in this case) is polynomial.
- Conversely: if you can't guess an answer and check in polynomial time, there can be no polynomial time algorithm!

This might appear to be irrelevent, but it turns out to be a practical classification tool!

This turns out to be a key question – one which we still don't know the answer to!

Nondeterministic Algorithm

An algorithm is **nondeterministic** if it works by guessing the right answer from among a finite number of choices.

Alternatively, imagine a tree of choices, polynomial levels deep.

- A super parallel machine follows all branches of the tree in parallel.
- If any single branch reaches a solution, the problem is solved.

A problem that can be solved in polynomial time by a nondeterministic machine is said to be "in $\mathcal{NP}$."

Is Towers of Hanoi in $\mathcal{NP}$?

Finite, but possibly large.

Nondeterministic Polynomial

No. Its too hard – can't verify answer in polynomial time.

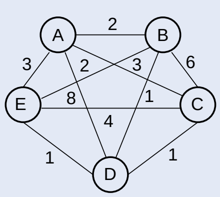
Problems solvable in polynomial time by a "normal" computer are said to be in $\mathcal{P}$.

Traveling Salesman Problem

TRAVELING SALESMAN (1):

- Input: A complete, directed graph G with distances assigned to each edge in the graph.
- Output: Shortest simple cycle that includes every vertex.

Problem: How to tell if a proposed solution is *shortest*?



You can't. You can verify that a proposed solution *is* a tour, and *is* of claimed cost. But, that's not necessarily *shortest*.

Traveling Salesman (Cont.)

Decision problem: A problem with a YES or NO answer.

TRAVELING SALESMAN (2):

- Input: A complete, directed graph G with distances assigned to each edge in the graph, and an integer K.
- Output: YES if there is a simple cycle with total distance $\leq K$ containing every vertex in G, and NO otherwise.

In $\mathcal{NP}$: We can guess a cycle, and quickly check if it meets the requirements.

no notes

$\mathcal{NP}$-complete Problems (1)

Many problems are like traveling salesman:

- They are in $\mathcal{NP}$.
- Nobody knows a polynomial time algorithm.
- Is there any relationship between them?

A problem X is said to be $\mathcal{NP}$-hard if ANY problem in $\mathcal{NP}$ can be reduced to X in polynomial time.

- X is AS HARD AS any problem in $\mathcal{NP}$.

A problem X is said to be $\mathcal{NP}$-complete if

1. It is in $\mathcal{NP}$.
2. It is $\mathcal{NP}$-hard.

But also cannot prove that there is *no* polynomial-time algorithm.

Note that X can be outside (harder than) $\mathcal{NP}$. But that's not useful.

$\mathcal{NP}$-complete Problems (2)

To start the process we need to prove just one problem H is $\mathcal{NP}$-complete.

- To show that X is $\mathcal{NP}$-hard, just reduce H to X.
- DON'T GET IT BACKWARDS!

no notes

Why Care about $\mathcal{NP}$-Completeness?

Your boss asks you to write a fast program for TRAVELING SALESMAN.

- Its obviously an easy problem to understand.
- She can easily see some algorithm to solve the problem.
- It must be easy to speed up!

If you can't do the job, what do you tell her?

- I can't do it.
- I can't find evidence that anyone can do it.
- Nobody has been able to do it, despite the fact that many people have tried. Furthermore, if anyone solved any of this long list of problems, then they would be able to do this problem too.

no notes

Satisfiability

Let E be a Boolean expression over variables $x_1, x_2, ..., x_n$ in Conjunctive Normal form:

$$E = (x_5 + x_7 + \overline{x_8} + x_{10}) \cdot (\overline{x_2} + x_3) \cdot (x_1 + \overline{x_3} + x_6).$$

SATISFIABILITY (SAT):

- INPUT: A Boolean expression E over variables $x_1, x_2, ...$ in Conjunctive Normal Form.
- OUTPUT: YES if there is an assignment to the variables that makes E true, NO otherwise.

This is the "grand-daddy" $\mathcal{NP}$-complete problem.

Cook's Theorem: SAT is $\mathcal{NP}$-complete.

no notes

$\mathcal{NP}$-completeness Proof Model

Implication: If a polynomial time algorithm can be found for ANY problem that is $\mathcal{NP}$-complete, then by a chain of polynomial time reductions, ALL $\mathcal{NP}$-complete problems can be solved in polynomial time.

To show that a decision problem X is $\mathcal{NP}$-complete:

1. Show that X is in $\mathcal{NP}$.
   - Give a polynomial-time, nondeterministic algorithm.
2. Show that X is $\mathcal{NP}$-hard.
   - Choose a known $\mathcal{NP}$-complete problem, A.
   - Describe a polynomial-time transformation that takes an ARBITRARY instance **I** of A to an instance **I**$'$ of X.
   - Describe a polynomial-time transformation from **S**$'$ to **S** such that **S** is the solution for **I**.

└ $\mathcal{NP}$-completeness Proof Model

no notes

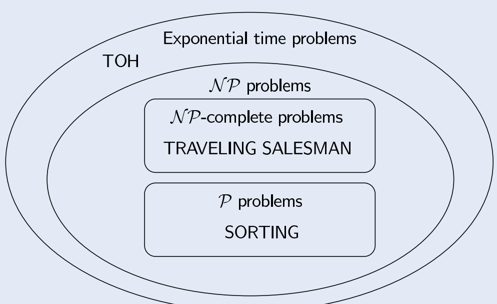
Cook's Proof Outline

1. Any decision problem can be recast as a language acceptance problem: $F(I) = \text{YES} \Leftrightarrow L(I') = \text{ACCEPT}$.
2. Turing machines are a simple model of computation for writing programs that are language acceptors.
3. There is a "universal" Turing machine that can take as input a description for a Turing machine, and an input string, and return the result of the execution of that machine on that string.
4. This in turn can be cast as a boolean expression such that the expression is satisfiable if and only if the Turing machine yields ACCEPT for that string.
5. Thus, *any* decision problem that is performable by the Turing machine is transformable to SAT: This is $\mathcal{NP}$-hard.

└ Cook's Proof Outline

no notes

The World of Exponential-time(?) Problems



Question: Does $\mathcal{P} = \mathcal{NP}$?

└ The World of Exponential-time(?) Problems

no notes

3-SATISFIABILITY (3 SAT)

Input: Boolean expression E in CNF such that each clause contains exactly 3 literals.
Output: YES if expression can be satisfied, NO otherwise.

A special case of SAT.

- Is 3 SAT easier than SAT?

Theorem: 3 SAT is $\mathcal{NP}$-complete.
Proof:

- 3 SAT is in $\mathcal{NP}$.
  - Guess (nondeterministically) values for the variables.
  - The correctness of the guess can be verified in polynomial time.
- 3 SAT is $\mathcal{NP}$-hard, by a reduction from SAT to 3 SAT.

└ 3-SATISFIABILITY (3 SAT)

2-SAT is polynomial.

3 SAT is $\mathcal{NP}$-hard

Find a polynomial time reduction from SAT to 3 SAT.

Let $E = C_1 \cdot C_2 \cdot ... \cdot C_k$ by any instance of SAT.

Strategy: Replace any clause C_i that does not have exactly 3 literals with two or more clauses having exactly 3 literals.

Let $C_i = x_1 + x_2 + ... + x_j$ where $x_1, ..., x_j$ are literals.

no notes

Replacement (1)

1. $j = 1$, so $C_i = x_1$. Replace C_i with
$$(x_1 + v + w) \cdot (x_1 + \overline{v} + w) \cdot (x_1 + v + \overline{w}) \cdot (x_1 + \overline{v} + \overline{w})$$
where v and w are new variables.
2. $j = 2$, so $C_i = (x_1 + x_2)$. Replace C_i with
$$(x_1 + x_2 + z) \cdot (x_1 + x_2 + \overline{z})$$
where z is a new variable.
3. $j > 3$. Replace C_i with
$$(x_1 + x_2 + z_1) \cdot (x_3 + \overline{z_1} + z_2) \cdot (x_4 + \overline{z_2} + z_3) \cdot ...$$
$$\cdot (x_{j-2} + \overline{z_{j-4}} + z_{j-3}) \cdot (x_{j-1} + x_j + \overline{z_{j-3}})$$
where $z_1, ..., z_{j-3}$ are new variables.

no notes

Replacement (2)

After appropriate replacements have been made for each C_i, a Boolean expression results that is an instance of 3 SAT.

Each replacement is satisfiable if and only if the original clause is satisfiable.

The reduction is clearly polynomial time.

no notes

Third Case

If E is satisfiable, then E' is satisfiable:

- Assume x_m is assigned true.
- Assign $z_t, t \leq m - 2$ as true and $z_k, t \geq m - 1$ as false.
- Then all clauses in Case (3) are satisfied.

If E' is satisfiable, then E is satisfiable:

- Proof by contradiction.
- If $x_1, x_2, ..., x_j$ are all false, then $z_1, z_2, ..., z_{j-3}$ are all true.
- But then $(x_{j-1} + x_{j-2} + \overline{z_{j-3}})$ is false, a contradiction.

(Not necessary for proof, but may help insight.)

Conversely, if E is not satisfiable, then E' is not satisfiable.

- E not satisfiable means all x_i are false.
- This leaves E' as
$$(z_1) \cdot (\overline{z_1} + z_2) \cdot ... \cdot (\overline{z_{j-4}} + z_{j-3}) \cdot (\overline{z_{j-3}})$$
which is NOT satisfiable.

no notes

Two Problems (1)

VERTEX COVER:
Input: An undirected graph G and an integer k.
Output: YES if there is a subset of vertices in G of size k or less such that every edge in the graph has at least one of its ends in the subset; NO otherwise.

K-CLIQUE:
Input: An undirected graph G and an integer k.
Output: YES if there is a subset of the vertices of size k or greater that is a complete graph (a clique).

no notes

Two Problems (2)

We can reduce either problem to the other by switching G to its inverse G'.

- If edge (i, j) is in G, it is NOT in G'.
- If edge (i, j) is NOT in G, it IS in G'.

Given a VC in G of size k, there is an $(n - k)$-sized clique in G' using the vertices *not* in the original vertex cover (and vice versa).
[The vertices *not* in the match cannot be connected, otherwise their connector edge would not be covered. So, the inverse graph must be a clique on those vertices.]

K CLIQUE is $\mathcal{NP}$-Complete (1)

```
procedure nd-CLIQUE(Graph G, int K) {
  VertexSet S = EMPTY;  int size = 0;
  for (v in G.V)
    if (nd-choice(YES, NO) == YES) then {
      S = union(S, v);
      size = size + 1;
    }
  if (size < K) then
    REJECT;      // S is too small
  for (u in S)
    for (v in S)
      if ((u <> v) && ((u, v) not in E))
        REJECT;  // S is missing an edge
  ACCEPT;
}
```


Guess a group of vertices and check that they form a complete graph.

K CLIQUE is $\mathcal{NP}$-Complete (2)

Now show that K CLIQUE is $\mathcal{NP}$-hard.

Reduce SAT to K CLIQUE.

An instance of SAT is a Boolean expression

$$B = C_1 \cdot C_2 \cdot ... \cdot C_m$$

where

$$C_i = y[i, 1] + y[i, 2] + ... + y[i, k_i].$$

Transform this to an instance of K CLIQUE as follows.

$$V = \{v[i, j] | 1 \leq i \leq m, 1 \leq j \leq k_i\}.$$

A vertex for every literal in every clause.

K CLIQUE is $\mathcal{NP}$-Complete (3)

All vertices $v[i_1, j_1]$ and $v[i_2, j_2]$ have an edge between them UNLESS they are two literals within the same clause ($i_1 = i_2$) OR they are opposite values for the same variable.

Set $k = m$.

no notes

Example

$$B = (y_1 + y_2) \cdot (\overline{y_1} + y_2 + y_3).$$

B is satisfiable if and only if G has a clique of size $\geq k$.

- B satisfiable implies there is a truth assignment such that $y[i, j_i]$ is true for each i.
- But then, $v[i, j_i]$ must be in a clique of size $k = m$.
- If G has clique of size $\geq k$, then clique must have size exactly k with one vertex $v[i, j_i]$ in clique for each i.
- There is a truth assignment making each $y[i, j_i]$ true. That truth assignment satisfies B.

Conclude that K CLIQUE is $\mathcal{NP}$-hard, therefore $\mathcal{NP}$-complete.

Need graph here

Co-$\mathcal{NP}$

- Note the asymmetry in the definition of $\mathcal{NP}$.
  - The non-determinism can identify a clique, and you can verify it.
  - But what if the correct answer is "NO"? How do you verify that?
- Co-$\mathcal{NP}$: The complements of problems in $\mathcal{NP}$.
  - Is a boolean expression **always** false?
  - Is there no clique of size k?
- It seems unlikely that $\mathcal{NP}$= co-$\mathcal{NP}$.

Co-$\mathcal{NP}$might be a bigger ("harder") class that includes $\mathcal{NP}$.

Is Everything in $\mathcal{NP}$ Either $\mathcal{P}$ or $\mathcal{NP}$-complete?

The following problems are not known to be in $\mathcal{P}$ or $\mathcal{NP}$-complete, but seem to be of a type that makes them unlikely to be in $\mathcal{NP}$-complete.

- GRAPH ISOMORPHISM: Are two graphs isomorphic?
- COMPOSITE NUMBERS: For positive integer K, are there integers $m, n > 1$ such that $K = mn$?
- LINEAR PROGRAMMING

These problems seem easier than typical $\mathcal{NP}$-complete problems, but are still probably harder than $\mathcal{P}$. They are obviously in $\mathcal{NP}$, but don't appear to be "hard" enough to solve any $\mathcal{NP}$-complete problem.

Subgraph Isomorphism (is a graph A isomorphic to some subgraph in graph B) is NP-complete. But it is understandable how this might be a harder problem (there are so many subgraphs to choose from).

Coping with $\mathcal{NP}$-Completeness

1. Organize to reduce costs.
   - Dynamic programming.
   - Backtracking.
   - Branch and Bounds.
2. Find subproblems of the original problem that have polynomial-time solutions.
   - Significant special cases that are useful to answer.
3. Approximation algorithms.
4. Randomized algorithms.
5. Use heuristics.
   - Greedy algorithms.
   - Simulated Annealing.
   - Genetic Algorithms.

See next slide.

Discussed later.

Knapsack Analysis Revisited

Fact: Knapsack is $\mathcal{NP}$-complete.

- But we have a $\Theta(nK)$ algorithm!!

Question: How big is K?

- Input size is typically $O(n \log K)$ since the item sizes are smaller than K.
- Thus, $\Theta(nK)$ is exponential on input size.

This algorithm is tractable if the numbers are "reasonable."

- nK can be thousands.
- This is different from TRAVELING SALESMAN which cannot handle $n = 100$.

Such an algorithm is called a **pseudo-polynomial** time algorithm.

$> 2^n$ is quite possible.

Subproblems and Special Cases

Some restricted cases of $\mathcal{NP}$-complete problems are useful, and not $\mathcal{NP}$-complete.

- VERTEX COVER and K CLIQUE have polynomial time algorithms for bipartite graphs.
- 2-SATISFIABILITY has a polynomial time solution.
- Several geometric problems are polynomial-time in two dimensions, but not in three or more.
- KNAPSACK is polynomial if the numbers are "small."

Example: Vertex cover on a bipartite graph. Best to pick the side with the greater number of vertices.

Approximation Algorithms

Seek algorithms for optimization problems with a guaranteed bound on quality of the solution.

For VERTEX COVER:

- Let M be a maximal (not necessarily maximum) **matching** in G.
  - A matching pairs vertices (with connecting edges) so that no vertex is paired with more than one match.
  - Maximal means pick as many pairs as possible.
- If OPT is the size of a minimum vertex cover, then

$$|M| \leq 2 \cdot \text{OPT}$$

because at least one endpoint of every matched edge must be in ANY vertex cover.

And, M is a vertex cover since no edge is free.

BIN PACKING

INPUT: Numbers $x_1, x_2, ..., x_n$ between 0 and 1, and an unlimited supply of bins of size 1.

OUTPUT: An assignment of numbers to bins that requires the fewest possible number of bins (no bin can hold numbers whose sum exceeds 1).

This problem is $\mathcal{NP}$-complete.

Example: Numbers 3/4, 1/3, 1/2, 1/8, 2/3, 1/2, 1/4.

Optimal solution: [3/4, 1/8], [1/2, 1/3], [1/2, 1/4], [2/3].

BIN PACKING

Optimal in that the sum is 3 1/8, and we packed into 4 bins. There is another optimal solution with the first 3 bins packed, but this is more than we need to solve the problem.

First Fit Algorithm

Place x_1 into the first bin.

For each $i, 2 \leq i \leq n$, place x_i in the first bin that will contain it.

No more than 1 bin can be left less than half full.
The number of bins used is no more than twice the sum of the numbers.

The sum of the numbers is a lower bound on the number of bins in the optimal solution.

Therefore, first fit is no more than twice the optimal number of bins.

First Fit Algorithm

Otherwise, the items in the second half-full bin would be put into the first!

First Fit Does Poorly

Let ϵ be very small, e.g., $\epsilon = .00001$.
Numbers (in this order):

- 6 of $(1/7 + \epsilon)$.
- 6 of $(1/3 + \epsilon)$.
- 6 of $(1/2 + \epsilon)$.

First fit returns:

- 1 bin of [6 of $1/7 + \epsilon$]
- 3 bins of [2 of $1/3 + \epsilon$]
- 6 bins of [$1/2 + \epsilon$]

Optimal solution is 6 bins of [$1/7 + \epsilon, 1/3 + \epsilon, 1/2 + \epsilon$].

First fit is 5/3 larger than optimal.

First Fit Does Poorly

no notes

Decreasing First Fit

It can be proved that the worst-case performance of first-fit is 17/10 times optimal.

Use the following heuristic:

- Sort the numbers in decreasing order.
- Apply first fit.
- This is called **decreasing first fit**.

The worst case performance of decreasing first fit is close to 11/9 times optimal.

Decreasing First Fit

no notes

Summary

The theory of $\mathcal{NP}$-completeness gives a technique for separating tractable from (probably) untractable problems.

When faced with a new problem, we might alternate between:

- Check if it is tractable (find a fast solution).
- Check if it is intractable (prove the problem is $\mathcal{NP}$-complete).

If the problem is in $\mathcal{NP}$-complete, then use one of the "coping" strategies.

no notes

Countable vs. Uncountably Infinite Sets

Two sets have the **same cardinality** if there is a bijection between them.

Notation: $|A| = |B|$.

This concept can also be applied to infinite sets.

Example: Let Odd and Even be the sets of odd and even natural numbers, respectively.
Then, $|\mathrm{Odd}| = |\mathrm{Even}|$ because the function $f : |\mathrm{Odd} \to \mathrm{Even}|$ defined by $f(x) = x - 1$ is a bijection.

How about $|\mathrm{Even}| = |\mathbb{N}|$?

no notes

Counting Infinite Sets

A set C is **countable** if it is finite or if $|C| = |\mathbb{N}|$.

If a set is not countable, then it is **uncountable**.

If A is a finite alphabet, then A^* is countably infinite.

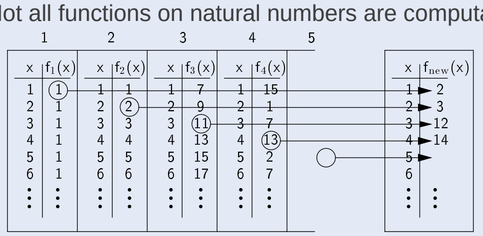
Proof: Arrange the strings in order by length, and within a given length by alphabetical order. This provides a bijection.

As a corollary, the set of all computer programs is countable.

Basically, any set that you can "put into an order" is countable.

More Functions than Programs

- Consider set of functions $f(x) = y$ for x, y natural numbers.
- The set of such functions is uncountable.
- Diagonalization argument
- Not all functions on natural numbers are computable.



We are taking the *i*th value from function *i* and changing it to create our new function. Which means that our new function is not the same as function *i*. And since we do this to every function, our new function is not any of the other functions.

Halting Problem for Programs

Does the following terminate?

```
while (n > 1)
  if (ODD(n))
    n = 3 * n + 1;
  else
    n = n / 2;
```

Can a **C**++ program be written to solve the following problem?

Halting Problem:

- Input: A program P and input X.
- Output: "Halts" if P halts when run with X as input. "Does not Halt" otherwise.

Halting Problem for Programs

It is interesting "in theory" that not all functions can have programs. But does this limit anything of interest in practice? After all, we are only interested in functions that we can somehow "describe", not functions with effectively no meaningful relationship between input and output.

Halting Problem Proof

Theorem: There is no program to solve the Halting Problem.

Proof: (by contradiction).

Assumption: There is a **C**++ program that solves the Halting Problem.

```
bool halt(char* prog, char* input)
{
  Code to solve halting problem
  if (prog does halt on input) then
    return(TRUE);
  else
    return(FALSE);
}
```


Halting Problem Proof

no notes

Two More Procedures

```
bool selfhalt(char *prog) {
  // Return TRUE if program halts
  //  when given itself as input.
  if (halt(prog, prog))
    return(TRUE);
  else
    return(FALSE);
}

void contrary(char *prog) {
  if (selfhalt(prog))
    while(TRUE); // Go into an infinite loop
}
```


Two More Procedures

Clearly these are real functions (because here they are!).

The Punchline

- What happens when function `contrary` is run on itself?
- Case 1: `selfhalt` returns TRUE.
  - `contrary` will go into an infinite loop.
  - This contradicts the result from `selfhalt`.
- `selfhalt` returns FALSE.
  - `contrary` will halt.
  - This contradicts the result from `selfhalt`.
- Either result is impossible.
- The only flaw in this argument is the assumption that `halt` exists.
- Therefore, `halt` cannot exist.

The Punchline

no notes

Computability Reduction Proof

Given arbitrary program M, does it halt on the EMPTY input?

This is uncomputable. Proof:

- Suppose that program M_0 determines if M halts on the EMPTY input.
- Given arbitrary program M and string w, we can create a new program M_w that operates as follows on empty input:
  - Write w into a static variable.
  - Simulate the execution of M.
- So, we can take arbitrary program M and string w, create M_w, and invoke M_0 on M_w (with empty input) to solve the original halting problem.
- Thus, M_0 must not exist.

no notes

Another Reduction Proof

Does there exist SOME input for which an arbitrary program halts?
Proof that this is uncomputable:

- Suppose that program M_0 could decide if arbitrary program M halts on SOME input.
- We can take an arbitrary program M and string w, and modify it so that it ignores its input before proceeding.
- Thus, arbitrary program M is modified to be M' that effectively is M operating on the empty input.
- Thus, we can take arbitrary program M and string w, modify it to become M' and feed that to M_0 to solve the problem of deciding if M halts on the empty input.
- We already know that is undecidable.
- Thus, M_0 cannot exist.

no notes

Other Noncomputable Functions

1. Does a program halt on EVERY input?
2. Do two programs compute the SAME function?
3. Does a particular line in a program get executed?
4. Does a program compute a particular function?
   pause
5. Does a program contain a “computer virus”?

1. EVERY: If I knew it always halted, then I would be able to answer if it halted on a specific input (the orginal halting problem)
2. SAME: Fix one program to perform the function “infinite loop”
3. Lines: Fix one program to loop on the selected line.
4. Functions: Fix the function to be “halts”.
5. Virus: This is essentially a complex behavior, an even vaguer problem than determining if a particular function is performed.

Parallel Algorithms

- **Running time**: $T(n,p)$ where n is the problem size, p is number of processors.
- **Speedup**: $S(p) = T(n,1)/T(n,p)$.
  - A comparison of the time for a (good) sequential algorithm vs. the parallel algorithm in question.
- Problem: Best sequential algorithm may not be the same as the best algorithm for p processors, which may not be the best for ∞ processors.
- Efficiency: $E(n,p) = S(p)/p = T(n,1)/(pT(n,p))$.
- Ratio of the time taken for 1 processor vs. the total time required for p processors.
  - Measure of how much the p processors are used (not wasted).
  - Optimal efficiency $= 1 =$ speedup by factor of p.

As opposed to $T(n)$ for sequential algorithms.

Question: What algorithms should be compared?

$pT(n,p)$ is total amount of “processor power” put into the problem.

If $E(n,p) > 1$ then the sequential form of the parallel algorithm would be faster than the sequential algorithm being compared against – very suspicious!

So there are differing goals possible: Absolute fastest speedup vs. efficiency.

Parallel Algorithm Design

Approach (1): Pick p and write best algorithm.

- Would need a new algorithm for every p!

Approach (2): Pick best algorithm for $p = \infty$, then convert to run on p processors.

Hopefully, if $T(n, p) = X$, then $T(n, p/k) \approx kX$ for $k > 1$.

Using one processor to **emulate** k processors is called the **parallelism folding principle**.

no notes

Parallel Algorithm Design (2)

Some algorithms are only good for a large number of processors.

$$\begin{aligned} T(n,1) &= n \\ T(n,n) &= \log n \\ S(n) &= n/\log n \\ E(n,n) &= 1/\log n \end{aligned}$$

For $p = 256$, $n = 1024$.
$T(1024, 256) = 4 \log 1024 = 40$.
For $p = 16$, running time $= (1024/16) * \log 1024 = 640$.
Speedup < 2, efficiency $= 1024/(16 * 640) = 1/10$.

Good in terms of speedup.

$1024/256$, assuming one processor emulates 4 in 4 times the time.
$E(1024, 256) = 1024/(256 * 40) = 1/10$.

But note that efficiency goes down as the problem size grows.

Amdahl's Law

Think of an algorithm as having a **parallelizable** section and a **serial** section.

Example: 100 operations.

- 80 can be done in parallel, 20 must be done in sequence.

Then, the best speedup possible leaves the 20 in sequence, or a speedup of $100/20 = 5$.

Amdahl's law:

$$\begin{aligned} \text{Speedup} &= (\mathcal{S} + \mathcal{P})/(\mathcal{S} + \mathcal{P}/N) \\ &= 1/(\mathcal{S} + \mathcal{P}/N) \leq 1/\mathcal{S}, \end{aligned}$$

for $\mathcal{S}$ = serial fraction, $\mathcal{P}$ = parallel fraction, $\mathcal{S} + \mathcal{P} = 1$.

See John L. Gustafson "Reevaluating Amdahl's Law," CACM 5/88 and follow-up technical correspondance in CACM 8/89.

Speedup is Serial / Parallel.
Draw graph, speed up is Y axis, Sequential is X axis. You will see a nonlinear curve going down.

Amdahl's Law Revisited

However, this version of Amdahl's law applies to a fixed problem size.

What happens as the problem size grows?
Hopefully, $\mathcal{S} = f(n)$ with $\mathcal{S}$ shrinking as n grows.

Instead of fixing problem size, fix execution time for increasing number N processors (and thus, increasing problem size).

$$\begin{aligned} \text{Scaled Speedup} &= (\mathcal{S} + \mathcal{P} \times N)/(\mathcal{S} + \mathcal{P}) \\ &= \mathcal{S} + \mathcal{P} \times N \\ &= \mathcal{S} + (1 - \mathcal{S}) \times N \\ &= N + (1 - N) \times \mathcal{S}. \end{aligned}$$

How long sequential process would take / How long for N processors.

Since $S + P = 1$ and $P = 1 - S$.

The point is that this equation drops off much less slowly in N: Graphing (sequential fraction for fixed N) vs. speedup, you get a line with slope $1 - N$.

All of this seems to assume the same algorithm for sequential and parallel. But that's OK – we want to see how much parallelism is possible for the parallel algorithm.

Models of Parallel Computation

Single Instruction Multiple Data (SIMD)

- All processors operate the same instruction in step.
- Example: Vector processor.

Pipelined Processing:

- Stream of data items, each pushed through the same sequence of several steps.

Multiple Instruction Multiple Data (MIMD)

- Processors are independent.

Vector: IBM 3090, Cray

Pipelined: Graphics coprocessor boards

MIMD: Modern clusters.

MIMD Communications (1)

Interconnection network:

- Each processor is connected to a limited number of neighbors.
- Can be modeled as (undirected) graph.
- Examples: Array, mesh, N-cube.
- It is possible for the cost of communications to dominate the algorithm (and in fact to limit parallelism).
- **Diameter**: Maximum over all pairwise distances between processors.
- Tradeoff between diameter and number of connections.

no notes

MIMD Communications (2)

Shared memory:

- Random access to global memory such that any processor can access any variable with unit cost.
- In practice, this limits number of processors.
- Exclusive Read/Exclusive Write (EREW).
- Concurrent Read/Exclusive Write (CREW).
- Concurrent Read/Concurrent Write (CRCW).

no notes

Addition

Problem: Find the sum of two n-bit binary numbers.

Sequential Algorithm:

- Start at the low end, add two bits.
- If necessary, carry bit is brought forward.
- Can't do ith step until $i-1$ is complete due to uncertainty of carry bit (?).

Induction: (Going from $n-1$ to n implies a sequential algorithm)

no notes

Parallel Addition

Divide and conquer to the rescue:

- Do the sum for top and bottom halves.
- What about the carry bit?

Strengthen induction hypothesis:

- Find the sum of the two numbers **with** or **without** the carry bit.

After solving for $n/2$, we have L, L_c, R, and R_c.

Can combine pieces in constant time.

Two possibilities: carry or not carry.

Also, for each a boolean indicating if it returns a carry.

If right has carry then
  Sum $= L_c|R$
Else
  Sum $= L|R$
If Sum has carry then
  Carry $=$ TRUE
For Sum$_c$

Do the same using R_c since it is computing value having received carry.

Parallel Addition (2)

The $n/2$-size problems are independent.
Given enough processors,

$$T(n, n) = T(n/2, n/2) + O(1) = O(\log n).$$

We need only the EREW memory model.

Not $2T(n/2, n/2)$ because done in parallel!

Maximum-finding Algorithm: EREW

“Tournament” algorithm:

- Compare pairs of numbers, the “winner” advances to the next level.
- Initially, have $n/2$ pairs, so need $n/2$ processors.
- Running time is $O(\log n)$.

That is faster than the sequential algorithm, but what about efficiency?

$$E(n, n/2) \approx 1/\log n.$$

Why is the efficiency so low?

Since $\frac{T(n,1)}{nT(n,n)} = \frac{n}{n \log n}$

Lots of idle processors after the first round.

More Efficient EREW Algorithm

Divide the input into $n/\log n$ groups each with $\log n$ items.

Assign a group to each of $n/\log n$ processors.

Each processor finds the maximum (sequentially) in $\log n$ steps.

Now we have $n/\log n$ “winners”.

Finish tournament algorithm.
$T(n, n/\log n) = O(\log n)$.
$E(n, n/\log n) = O(1)$.

In $\log n$ time.

More Efficient EREW Algorithm (2)

But what could we do with more processors?
A parallel algorithm is **static** if the assignment of processors to actions is predefined.

- We know in advance, for each step i of the algorithm and for each processor p_j, the operation and operands p_j uses at step i.

This maximum-finding algorithm is static.

- All comparisons are pre-arranged.

Cannot improve time past $O(\log n)$.

Doesn't depend on a specific input value.

As an analogy to help understand the concept of static: Bubblesort and Mergesort are static in this way. We always know the positions to be copmared next.
In contrast, Insertion Sort is not static.

Brent's Lemma

Lemma 12.1: If there exists an EREW static algorithm with $T(n,p) \in O(t)$, such that the total number of steps (over all processors) is s, then there exists an EREW static algorithm with $T(n,s/t) \in O(t)$.

Proof:

- Let $a_i, 1 \leq i \leq t$, be the total number of steps performed by all processors in step i of the algorithm.
- $\sum_{i=1}^{t} a_i = s$.
- If $a_i \leq s/t$, then there are enough processors to perform this step without change.
- Otherwise, replace step i with $\lceil a_i/(s/t) \rceil$ steps, where the s/t processors emulate the steps taken by the original p processors.

Note that we are using t as the actual number of steps, as well as the variable in the big-Oh analysis, which is a bit informal.

Brent's Lemma (2)

- The total number of steps is now

$$\begin{aligned} \sum_{i=1}^{t} \lceil a_i/(s/t) \rceil &\leq \sum_{i=1}^{t} (a_i t/s + 1) \\ &= t + (t/s) \sum_{i=1}^{t} a_i = 2t. \end{aligned}$$

Thus, the running time is still $O(t)$.

Intuition: You have to split the s work steps across the t time steps somehow; things can't **always** be bad!

If s is sequential complexity, then the modified algorithm has $O(1)$ efficiency.

Maximum-finding: CRCW

- Allow concurrent writes to a variable only when each processor writes the same thing.
- Associate each element x_i with a variable v_i, initially "1".
- For each of $n(n-1)/2$ processors, processor p_{ij} compares elements i and j.
- First step: Each processor writes "0" to the v variable of the smaller element.
  - Now, only one v is "1".
- Second step: Look at all $v_i, 1 \leq i \leq n$.
  - The processor assigned to the max element writes that value to MAX.

Efficiency of this algorithm is **very** poor!

- "Divide and crush."

Need $O(n^2)$*processors*
Need only constant time.
Efficiency is $1/n$.

Maximum-finding: CRCW (2)

More efficient (but slower) algorithm:

- Given: n processors.
- Find maximum for each of $n/2$ pairs in constant time.
- Find max for $n/8$ groups of 4 elements (using 8 proc/group) each in constant time.
- Square the group size each time.
- Total time: $O(\log \log n)$.

n/2 processors
n processors, using previous "divide and crush" algorithm.

This leaves n/8 elements which can be broken into n/128 groups of 16 elements with 128 processors assigned to each group. And so on.

Efficiency is $1/\log \log n$.

Parallel Prefix

- Let $\cdot$ be any associative binary operation.
  - Ex: Addition, multiplication, minimum.
- Problem: Compute $x_1 \cdot x_2 \cdot \ldots \cdot x_k$ for all $k, 1 \leq k \leq n$.
- Define $\mathrm{PR}(i, j) = x_i \cdot x_{i+1} \cdot \ldots \cdot x_j$.
  We want to compute $\mathrm{PR}(1, k)$ for $1 \leq k \leq n$.
- Sequential alg: Compute each prefix in order
  - $O(n)$ time required (using previous prefix)
- Approach: Divide and Conquer
  - IH: We know how to solve for $n/2$ elements.

1. $\mathrm{PR}(1, k)$ and $\mathrm{PR}(n/2 + 1, n/2 + k)$ for $1 \leq k \leq n/2$.
2. $\mathrm{PR}(1, m)$ for $n/2 < m \leq n$ comes from $\mathrm{PR}(1, n/2) \cdot \mathrm{PR}(n/2 + 1, m)$ – from IH.

We don't just want the sum or min of all – we want all the partials as well.

We have the lower half done, and the upper half values are each missing the contribution from the lower half.

Parallel Prefix (2)

- **Complexity**: (2) requires $n/2$ processors and CREW for parallelism (all read middle position).
- $T(n, n) = O(\log n)$; $\quad E(n, n) = O(1/\log n)$.
  Brent's lemma no help: $O(n \log n)$ total steps.

That is – no processors are "excessively" idle. This is because we needed to copy PR(1, $n/2$) into $n/2$ positions on the last step.

$$E = \frac{n}{n \cdot \log n} = \frac{1}{logn}$$

Better Parallel Prefix

- E is the set of all x_is with i even.
- If we know $\mathrm{PR}(1, 2i)$ for $1 \leq i \leq n/2$ then $\mathrm{PR}(1, 2i + 1) = \mathrm{PR}(1, 2i) \cdot x_{2i+1}$.
- Algorithm:
  - Compute in parallel $x_{2i} = x_{2i-1} \cdot x_{2i}$ for $1 \leq i \leq n/2$.
  - Solve for E (by induction).
  - Compute in parallel $x_{2i+1} = x_{2i} \cdot x_{2i+1}$.
- Complexity:
  $T(n, n) = O(\log n)$.
  $S(n) = S(n/2) + n - 1$, so $S(n) = O(n)$ for $S(n)$ the total number of steps required to process n elements.
- So, by Brent's Lemma, we can use $O(n/\log n)$ processors for $O(1)$ efficiency.

Since the E's already include their left neighbors, all info is available to get the odds.

There is only one recursive call, instead of two in the previous algorithm.

Need EREW model for Brent's Lemma.

Routing on a Hypercube

Goal: Each processor P_i simultaneously sends a message to processor $P_{\sigma(i)}$ such that no processor is the destination for more than one message.

Problem:

- In an n-cube, each processor is connected to n other processors.
- At the same time, each processor can send (or receive) only one message per time step on a given connection.
- So, two messages cannot use the same edge at the same time – one must wait.

Need a figure

Randomizing Switching Algorithm

It can be shown that any deterministic algorithm is $\Omega(2^{n^a})$ for some $a > 0$, where 2^n is the number of messages.

A node i (and its corresponding message) has binary representation $i_1 i_2 \cdots i_n$.

Randomization approach:

(a) Route each message from i to j to a random processor r (by a randomly selected route).

(b) Continue the message from r to j by the shortest route.

n-dimensional hypercube has 2^n nodes.

Remember that we want parallel algorithms with cost $\log n$, not cost n^a!
The distance from any processor i to another processor j is only $\log n$ steps.

Randomized Switching (2)

Phase (a):

```
for (each message at i)
cobegin
  for (k = 1 to n)
    T[i, k] = RANDOM(0, 1);
  for (k = 1 to n)
    if (T[i, k] = 1)
      Transmit i along dimension k;
coend;
```


no notes

Randomized Switching (3)

Phase (b):

```
for (each message i)
cobegin
  for (k = 1 to n)
    T[i, k] =
      Current[i, k] EXCLUSIVE_OR Dest[i, k];
  for (k = 1 to n)
    if (T[i, k] = 1)
      Transmit i along dimension k;
coend;
```


no notes

Randomized Switching (4)

With high probability, each phase completes in $O(\log n)$ time.

- It is possible to get a really bad random routing, but this is unlikely (by chance).
- In contrast, it is very possible for any correlated group of messages to generate a bottleneck.

no notes

Sorting on an array

Given: n processors labeled $P_1, P_2, \cdots, P_n$ with processor P_i initially holding input x_i.

P_i is connected to P_{i-1} and P_{i+1} (except for P_1 and P_n).

- Comparisons/exchanges possible only for adjacent elements.

```
Algorithm ArraySort(X, n) {
  do in parallel ceil(n/2) times {
    Exchange-compare(P[2i-1], P[2i]); // Odd
    Exchange-compare(P[2i], P[2i+1]); // Even
  }
}
```

A simple algorithm, but will it work?

Any algorithm that correctly sorts 1's and 0's by comparisons will also correctly sort arbitrary numbers.

Parallel Array Sort

7 3 6 5 **8** 1 4 2
3 7 5 6 1 **8** 2 4
3 5 7 1 6 2 **8** 4
3 5 1 7 2 6 4 **8**
3 1 5 2 7 4 6 **8**
1 3 2 5 4 7 6 **8**
1 2 3 4 5 6 7 **8**
1 2 3 4 5 6 7 **8**
1 2 3 4 5 6 7 **8**

Manber Figure 12.8.

Correctness of Odd-Even Transpose

Theorem 12.2: When Algorithm ArraySort terminates, the numbers are sorted.

Proof: By induction on n.

Base Case: 1 or 2 elements are sorted with one comparison/exchange.

Induction Step:

- Consider the maximum element, say x_m.
- Assume m odd (if even, it just won't exchange on first step).
- This element will move one step to the right each step until it reaches the rightmost position.

no notes

Correctness (2)

- The position of x_m follows a diagonal in the array of element positions at each step.
- Remove this diagonal, moving comparisons in the upper triangle one step closer.
- The first row is the nth step; the right column holds the greatest value; the rest is an $n-1$ element sort (by induction).

Map the execution of n to an execution of $n-1$ elements.

See Manber Figure 12.9.

Sorting Networks

When designing parallel algorithms, need to make the steps independent.

Ex: Mergesort split step can be done in parallel, but the join step is nearly serial.

- To parallelize mergesort, we must parallelize the merge.

no notes

Batcher's Algorithm

For n a power of 2, assume $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_n$ are sorted sequences.

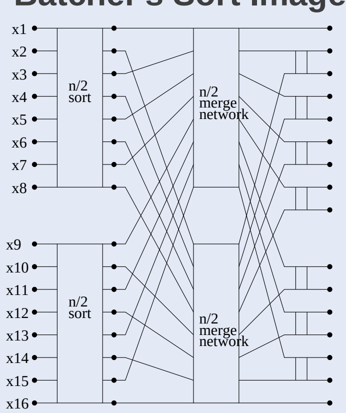
Let $x_1, x_2, \cdots, x_{2n}$ be the final merged order.

Need to merge disjoint parts of these sequences in parallel.

- Split a, b into odd- and even- index elements.
- Merge a_{odd} with b_{odd}, a_{even} with b_{even}, yielding $o_1, o_2, \cdots, o_n$ and $e_1, e_2, \cdots, e_n$ respectively.

No notes

Batcher's Sort Image



No notes

Batcher's Algorithm Correctness

Theorem 12.3: For all i such that $1 \leq i \leq n-1$, we have $x_{2i} = \min(o_{i+1}, e_i)$ and $x_{2i+1} = \max(o_{i+1}, e_i)$.

Proof:

- Since e_i is the ith element in the sorted even sequence, it is $\geq$ at least i even elements.
- For each even element, e_i is also $\geq$ an odd element.
- So, $e_i \geq 2i$ elements, or $e_i \geq x_{2i}$.
- In the same way, $o_{i+1} \geq i+1$ odd elements, $\geq$ at least $2i$ elements all together.
- So, $o_{i+1} \geq x_{2i}$.
- By the pigeonhole principle, e_i and o_{i+1} must be x_{2i} and x_{2i+1} (in either order).

See Manber Figure 12.11.

Batcher Sort Complexity

- Total number of comparisons for merge:
$$T_M(2n) = 2T_M(n) + n - 1; \quad T_M(1) = 1.$$
Total number of comparisons is $O(n \log n)$, but the depth of recursion (parallel steps) is $O(\log n)$.
- Total number of comparisons for the sort is:
$$T_S(2n) = 2T_S(n) + O(n \log n), \quad T_S(2) = 1.$$
So, $T_S(n) = O(n \log^2 n)$.
- The circuit requires n processors in each column, with depth $O(\log^2 n)$, for a total of $O(n \log^2 n)$ processors and $O(\log^2 n)$ time.
- The processors only need to do comparisons with two inputs and two outputs.

$O(\log n)$ sort steps, with each associated merge step counting $O(\log n)$.

Matrix-Vector Multiplication

Problem: Find the product $x = A\mathbf{b}$ of an m by n matrix A with a column vector $\mathbf{b}$ of size n.

Systolic solution:

- Use n processor elements arranged in an array, with processor P_i initially containing element b_i.
- Each processor takes a partial computation from its left neighbor and a new element of A from above, generating a partial computation for its right neighbor.

Cost: $O(n + m)$

See Manber Figure 12.17.

A General Model

Want a general model of computation that is as simple as possible.

- Wish to be able to reason about the model.
- "State machines" are simple.

Necessary features:

- Read
- Write
- Compute

Our key concern now is *ability* not *efficiency*.

Turing Machines (1)

A tape, divided into squares.

"States"

A single I/O head:
- Read current symbol
- Change current symbol

Control Unit Actions:
- Put the control unit into a new state.
- Either:
  1. Write a symbol in current tape square.
  2. Move I/O head one square left or right.

Cook used Turing machines to prove that Satisfiability is $\mathcal{NP}$-complete.

A Turing machine is sufficiently complex that a Turing machine can be built that can take as input a coding for an arbitrary Turing machine, along with an input, and simulate its execution on that input.

Turing Machines (2)

Tape has a fixed left end, infinite right end.
- Machine ceases to operate if head moves off left end.
- By convention, input is placed on left end of tape.

A **halt** state (h) signals end of computation.

"#" indicates a blank tape square.

no notes

Formal definition of Turing Machine

A **Turing Machine** is a quadruple (K, Σ, δ, s) where
- K is a finite set of **states** (not including h).
- Σ is an alphabet (containing #, not L or R).
- $s \in K$ is the **initial** state.
- δ is a function from $K \times \Sigma$ to $(K \cup \{h\}) \times (\Sigma \cup \{L, R\})$.

If $q \in K$, $a \in \Sigma$ and $\delta(q, a) = (p, b)$, then when in state q and scanning a, enter state p and
1. If $b \in \Sigma$ then replace a with b.
2. Else (b is L or R): move head.

is "space." Note including # in the language is for convenience only! We want to be able to read our specifications without being confused.

Turing Machine Example 1

$M = (K, \Sigma, \delta, s)$ where
- $K = \{q_0, q_1\}$,
- $\Sigma = \{a, \#\}$,
- $s = q_0$,
- $\delta =$

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, \#)$
q_0	$\#$	$(h, \#)$
q_1	a	(q_0, a)
q_1	$\#$	(q_0, R)

State (q_1, a) cannot happen if the start state is q_0. This is included only for completness (to make δ a total function).

Scan right, changing a's to #'s. When we hit first #, halt.

Turing Machine Example 2

$M = (K, \Sigma, \delta, s)$ where

- $K = \{q_0\}$,
- $\Sigma = \{a, \#\}$,
- $s = q_0$,
- $\delta =$

q	σ	$\delta(q, \sigma)$
q_0	a	(q_0, L)
q_0	$\#$	$(h, \#)$

Scan left to #. Then halt.

Notation

Configuration: $(q, aaba\#\underline{\#}a)$

Halted configuration: q is h.

Hanging configuration: Move left from leftmost square.

A **computation** is a sequence of configurations for some $n \geq 0$. Such a computation is of **length** n.

First symbol after the comma is the leftmost square of the tape. The underscore shows placement of the head. After the last symbol is an infinte series of spaces.

Execution

Execution on first machine example.

$$
\begin{array}{rll}
(q_0, \underline{a}aaa) & \vdash_M & (q_1, \underline{\#}aaa) \\
& \vdash_M & (q_0, \#\underline{a}aa) \\
& \vdash_M & (q_1, \#\underline{\#}aa) \\
& \vdash_M & (q_0, \#\#\underline{a}a) \\
& \vdash_M & (q_1, \#\#\underline{\#}a) \\
& \vdash_M & (q_0, \#\#\#\underline{a}) \\
& \vdash_M & (q_1, \#\#\#\underline{\#}) \\
& \vdash_M & (q_0, \#\#\#\#\underline{\#}) \\
& \vdash_M & (h, \#\#\#\#\underline{\#})
\end{array}
$$

No notes

Computations

- M is said to **halt on input** w iff $(s, \#w\underline{\#})$ yields some halted configuration.
- M is said to **hang on input** w if $(s, \#w\underline{\#})$ yields some hanging configuration.
- Turing machines compute functions from strings to strings.
- Formally: Let f be a function from Σ_0^* to Σ_1^*. Turing machine M is said to **compute** f if for any $w \in \Sigma_0^*$, if $f(w) = u$ then
$$(s, \#w\underline{\#}) \vdash_M^* (h, \#u\underline{\#}).$$
- f is said to be a **Turing-computable function**.
- Multiple parameters: $f(w_1, ..., w_k) = u$, $(s, \#w_1\#w_2\#...\#w_k\underline{\#}) \vdash_M^* (h, \#u\underline{\#})$.

These are the conventions.

Specify input conditions. Behavior is undefined for other initial conditions.

Either move left from left end *or* infinite loop.

Functions on Natural Numbers

- Represent numbers in **unary** notation on symbol I (zero is represented by the empty string).
- $f : \mathbb{N} \rightarrow \mathbb{N}$ is computed by M if M computes $f' : \{I\}^* \rightarrow \{I\}^*$ where $f'(I^n) = I^{f(n)}$ for each $n \in \mathbb{N}$.
- Example: $f(n) = n + 1$ for each $n \in \mathbb{N}$.

q	σ	$\delta(q, \sigma)$
q_0	I	(h, R)
q_0	$\#$	(q_0, I)

$$(q_0, \#II\underline{\#}) \vdash_M (q_0, \#II\underline{I}) \vdash_M (h, \#III\underline{\#}).$$

- In general, $(q_0, \#I^n\underline{\#}) \vdash_M^* (h, \#I^{n+1}\underline{\#})$.
- What about $n = 0$?

Works OK.

Turing-decidable Languages

A language $L \subset \Sigma_0^*$ is **Turing-decidable** iff function $\chi_L : \Sigma_0^* \rightarrow \{\boxed{\text{Y}}, \boxed{\text{N}}\}$ is Turing-computable, where for each $w \in \Sigma_0^*$,

$$\chi_L(w) = \begin{cases} \boxed{\text{Y}} & \text{if } w \in L \\ \boxed{\text{N}} & \text{otherwise} \end{cases}$$

Ex: Let $\Sigma_0 = \{a\}$, and let $L = \{w \in \Sigma_0^* : |w| \text{ is even}\}$.

M erases the marks from right to left, with current parity encode by state. Once blank at left is reached, mark $\boxed{\text{Y}}$ or $\boxed{\text{N}}$ as appropriate.

There are many views of computation. One is functions mapping input to output ($N \rightarrow N$, or strings to strings, for examples). Another is deciding if a string is in a language.

Turing-acceptable Languages

M **accepts** a string w if M halts on input w.

- M accepts a language iff M halts on w iff $w \in L$.
- A language is **Turing-acceptable** if there is some Turing machine that accepts it.

Ex: $\Sigma_0 = \{a, b\}$, $L = \{w \in \Sigma_0^* : w \text{ contains at least one } a\}$.

q	σ	$\delta(q, \sigma)$
q_0	a	(h, a)
q_0	b	(q_0, L)
q_0	$\#$	(q_0, L)

Every Turing-decidable language is Turing-acceptable.

Is this language Turing decidable? Of course. Instead of just running left, invoke another state that means "seen an a," and print $\boxed{\text{Y}}$ if we reach # in that state, $\boxed{\text{N}}$ otherwise.

If we would have printed $\boxed{\text{Y}}$, then halt.
If we would have printed $\boxed{\text{N}}$, then hang left.

Is every Turing-acceptible language Turing decidable? This is the Halting Problem.

Of course, if the TA language would halt, we write $\boxed{\text{Y}}$. But if the TA lang would hang, can we *always* replace it with logic to write $\boxed{\text{N}}$ instead? Ex: Collatz function.

Combining Turing Machines

Lemma: If

$$(q_1, w_1\underline{a_1}u_1) \vdash_M^* (q_2, ww_2\underline{a_2}u_2)$$

for string w and

$$(q_2, w_2\underline{a_2}u_2) \vdash_M^* (q_3, w_3\underline{a_3}u_3),$$

then

$$(q_1, w_1\underline{a_1}u_1) \vdash_M^* (q_3, ww_3\underline{a_3}u_3).$$

Insight: Since $(q_2, w_2\underline{a_2}u_2) \vdash_M^* (q_3, w_3\underline{a_3}u_3)$, this computation must take place without moving the head left of w_2

- The machine cannot "sense" the left end of the tape

And if it had moved left, it would have hung.

Combining Turing Machines (Cont)

Thus, the head won't move left of w_2 even if it is not at the left end of the tape.

This means that Turing machine computations can be combined into larger machines:

- M_2 prepares string as input to M_1.
- M_2 passes control to M_1 with I/O head at end of input.
- M_2 retrieves control when M_1 has completed.

no notes

Some Simple Machines

Basic machines:

- $|\Sigma|$ symbol-writing machines (one for each symbol).
- Head-moving machines R and L move the head appropriately.

More machines:

- First do M_1, then do M_2 or M_3 depending on current symbol.
- (For $\Sigma = \{a, b, c\}$) Move head to the right until a blank is found.
- Find first blank square to left: $L_\#$
- Copy Machine: Transform $\#w\underline{\#}$ into $\#w\#w\underline{\#}$.
- Shift a string left or right.

Show shift left machine and copy machine.

We know how to increment. How do we decrement? Add? Multiply?

Extensions

The following extensions do not increase the power of Turing Machines.

- 2-way infinite tape
- Multiple tapes
- Multiple heads on one tape
- Two-dimensional "tape"
- Non-determinism

Show figures for these.

Just bend infinite tape in the middle to get back to one-way tape, but with two layers. Now, expand the language. The new language is ordered pairs of the old language, to encode two levels of tape.

Again, expanded alphabet collapses multipe symbols to 1.

Encode the heads onto the tape, and simulate moving them around.

Convert to 1D, by diagonals.

Simulate nondeterministic behavior in sequence, doing all length -1 computations, then length -2, etc., until we reach a halt state for one of the non-deteriministic choices. Non-determinism gives us speed, not ability.