COURSENOTES

# CS4104: Data and Algorithm Analysis

Clifford A. Shaffer
Department of Computer Science
Virginia Tech

# CS4014 Prereqs and Major Topics

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-$\Omega$, $\Theta$
- Most of what was covered in CS2606
  - Basic data structures
  - Algorithms for searching and sorting

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

## Introduction to Problem Solving

- 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
- 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.
- Getting your hands dirty
  - Example: Repairing a wobbly table
  - Get underneath and look
  - Example: Repairing a dryer
  - Open up back panel and look

## 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

## Heuristics for Problem Solving

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

- Puzzles
- Math and CS test or homework problems

A list of standard Heuristics:

- 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 $7 \pm 2$ pieces of information
  - Take advantage of your environment to get around this
  - Write things down
  - Manipulate problem (good representation)
- Get your hands dirty
  - "Play around" with the problem to get some initial insight.

## Heuristics (2)

- Look for special features
  - Example: Cryptogram addition problems.

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

- Go to the extremes
  - Study problem boundary conditions
- Simplify
  - This might give a partial solution that can be extended to the original problem.
- 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.
- Lateral thinking
  - Don’t be lead into a blind alley.
  - Using an inappropriate problem solving strategy might blind you to the solution.

## Heuristics (3)

- 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
- Symmetry
  - Symmetries in the problem might give clues to the solution

## Problem Solving “In the Large”

- Problem Definition
  - Reformulate problem statement to get at the ”real problem”.
- Generate Solutions
  - Getting around mental blocks.
  - Blockbusting.
  - Brainstorming.
- Decide the course of action.
  - Situation analysis.
  - Pareto analysis.
  - K.T. Problem analysis.
  - Decision analysis.
- Implement the solution.
  - Getting approval
  - Planning
  - Gannt charts
  - Critical path analysis
  - Experimental design
  - Report results.
- Evaluation
  - Make it an ongoing process at all stages

## 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

## 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.

Myths about reading: These are some popularized misconceptions

- Don’t subvocalize when you read
- Read only key words
- Don’t be a word-by-word reader
- Read in thought groups
- You can be a speed reader without loss of comprehension
- Don’t re-read

## 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?

## Algorithm Efficiency

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.

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.

## Cost Model

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 $2n + 1$.)

```
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 assigning strings by copying. Is this a good model?

## Big Issues

How do we create an efficient algorithm?

How do we recognize a "good" algorithm?

How "hard" is a problem?

General Plan:

- Define a PROBLEM.
- Build a MODEL to measure the cost of a solution to the 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.

## Problems

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.

An **algorithm** executes the mapping.

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

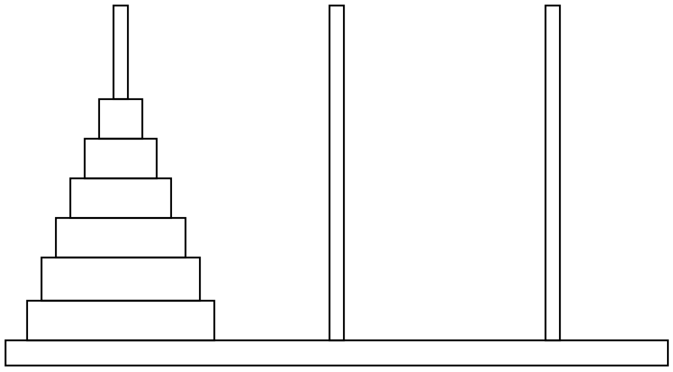
GOAL: Solve problems with as little computational effort per instance as possible.

## Categories of Hard Problems

- 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.
- 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.

# 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.

## 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?

## Recursive Solutions

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)?

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

## TOH Solution

```
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
}
```

## Algorithm Upper Bounds

**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 $I_n$.

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

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

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

## 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?

## 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.)

## Recurrence Relation

The following is a <u>**recurrence relation**</u>:

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

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?

## 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**.

## Proof by Induction

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

- We can get started (<u>**base case**</u>).
- 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 can conclude that the formula is correct for all $n \geq 1$.

Is this a good algorithm?

## Lower Bound of a Problem

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.

$\mathcal{A}_M$ is the set of algorithms within model $M$ that solve the problem.

Lower Bound on Problem P

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

## 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 $I$s in $I_n$ can have different costs is the reason why we must use the qualifier of “best” “worst” or “average” cases.

## Lower Bound (cont.)

Lower bounds are harder than upper bounds 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?

## 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).

## 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.

## 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.

# Factorial Growth

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 |

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?

## Using Logarithms

$n! \geq 2^{2n}$ 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.

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

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?

## 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.

## A Simple Sum

```
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:

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

Give a good estimate.

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

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?

## 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:

The number of pairs is:

Thus, the solution is:

## 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: $f(n) = c_1n^2 + c_2n + c_3$.
$f(0) = 0$ 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 $c_2 = 1/2$.

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?

## 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

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

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?

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

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

## Solving Summations (cont.)

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

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

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

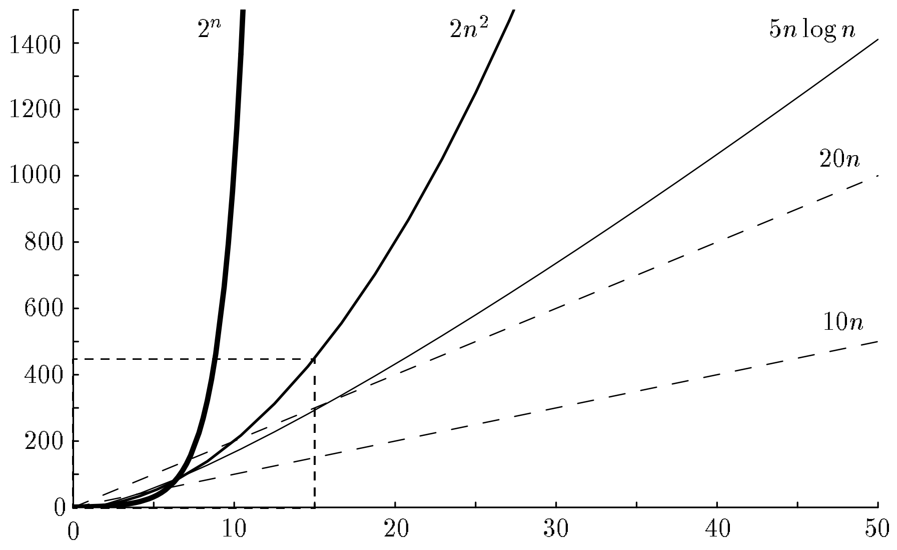
## Solving Summations (cont.)

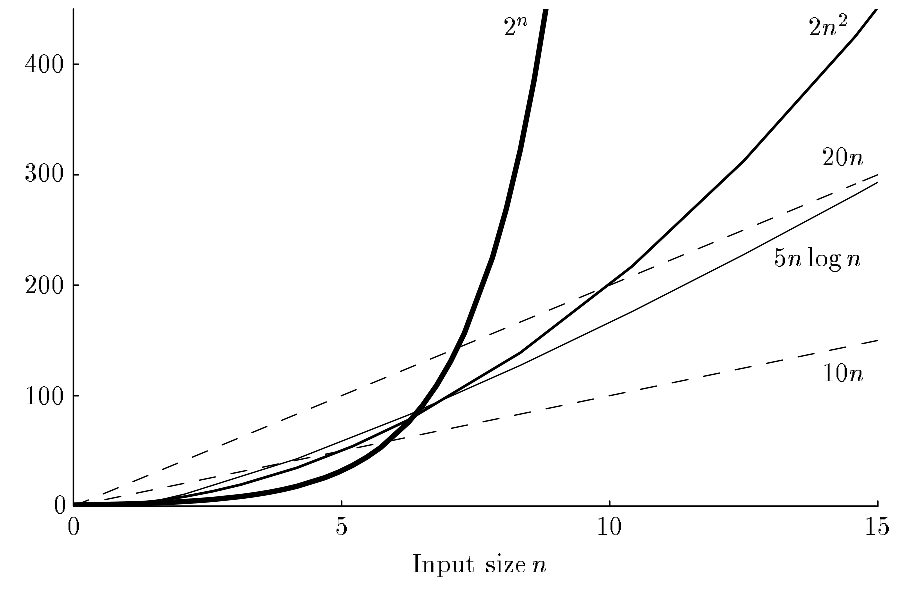
$$\begin{aligned}
\frac{f(n)}{f(n-1)} &= \frac{n+1}{n-1} \\
f(n)(n-1) &= (n+1)f(n-1) \\
f(n)(n-1) &= (n+1)(f(n)-n) \\
nf(n)-f(n) &= nf(n)+f(n)-n^2-n \\
2f(n) &= n^2+n=n(n+1) \\
f(n) &= \frac{n(n+1)}{2}
\end{aligned}$$

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

# 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.





## 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.

## 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.

## Some Growth Rates

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 $\log n$.

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 <u>no slower</u> than $n$.

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 $2^n$.

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

## Fibonacci Numbers

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

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

```
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:

What about Fibr?

# Analysis of Fibr

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

| $n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $f(n)$ | 1 | 2 | 3 | 5 | 8 | 13 | 21 |
| $f(n)/f(n-1)$ | 1 | 2 | 1.5 | 1.666 | 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.

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

For large $n$,

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

## Analysis of Fibr (cont.)

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

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 $f$?

## 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 $\mathcal{B} = \Theta(\log n)$.

## 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.

## 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.

## Searching

Assumptions for search problems:

- Target is well defined.
- Target is fixed.
- Probes are accurate (hit or miss).
- Search domain is finite.
- We (can) remember all information gathered during search.

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

## A Search Model

**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.
- Comparison determine $<$, $=$, $>$.
- Comparison is our only way to find ordering information.
- Every comparison costs the same.

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?

# 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?

## 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.

## 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.

## 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?

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

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

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

Now, prove using induction.

## 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?

# Fixing the Proof

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.

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.

## 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?

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

## 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 $p_0$)?

## 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 $p_0$, $\frac{n+1}{2} \leq f(n) \leq n$.

## 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.

## 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?

## Jump Search

What is the right amount to jump?

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.

## 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 $2\sqrt{n}$.

## 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

## 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
}
```

## Worst Case for Binary Search

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

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:

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

Collapse to

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

Now, prove it with induction.

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

## Lower Bound

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 $n$ 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.

## Average Cost of Binary Search

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?

## Average Cost (cont.)

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

$$= \frac{1}{n} \sum_{i=1}^{\log n} i2^{i-1}$$

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

Now what? Subtract from the original!

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

## Result

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

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

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

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

## Average Cost Lower Bound

Use decision trees again.

<u>**Total Path Length**</u>: 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.

## 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.

## 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.

## 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?

## QBS Probe Count

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:

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

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

## QBS Probe Count (cont.)

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.

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?

## Comparison

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 |

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?

## 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

## Hashing Insertion Analysis

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? $\alpha^2$

$$\sum_{i=0}^{i=N} i(\frac{N}{M})^i \frac{M-N}{M}$$

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

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

How does this grow?

## Searching Linked Lists

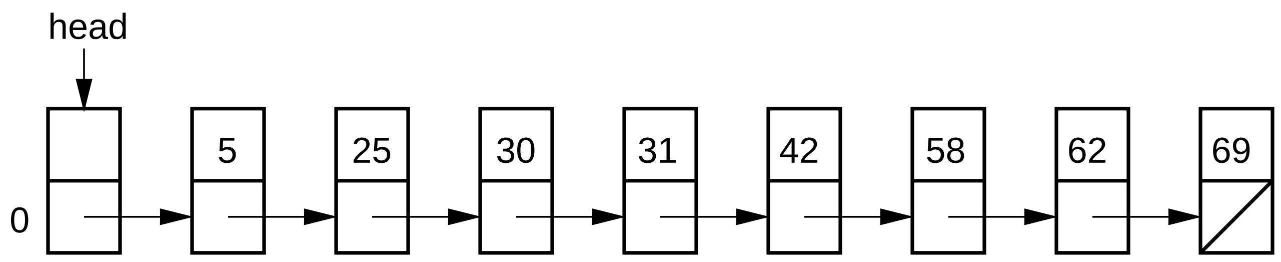
Assume the list is sorted, but is stored in a linked list.

Can we use binary search?

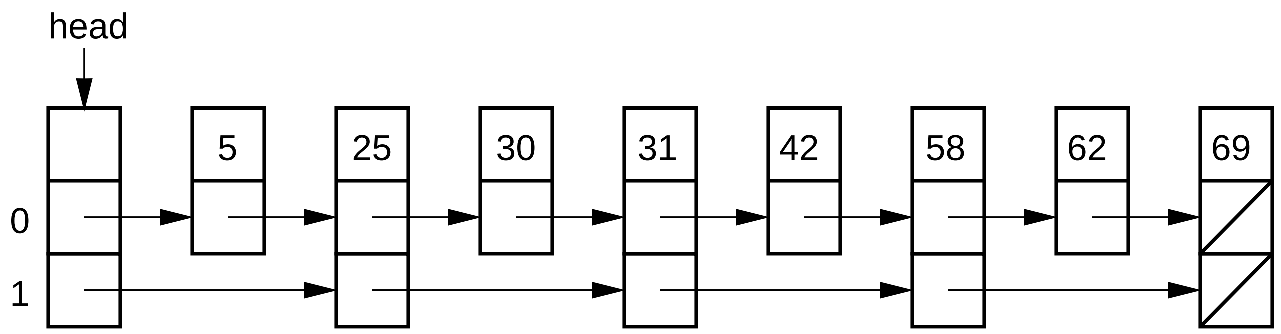
- Comparisons?
- “Work?”

What if we add additional pointers?

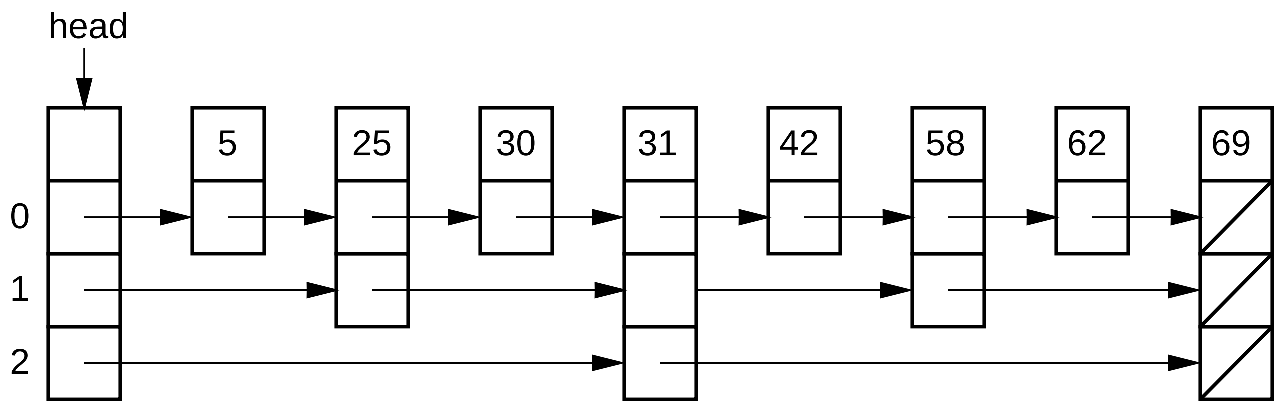
# “Perfect” Skip List



(a)



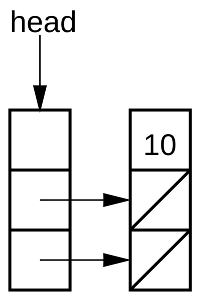
(b)



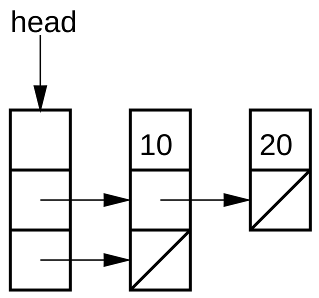
(c)

# Building a Skip List

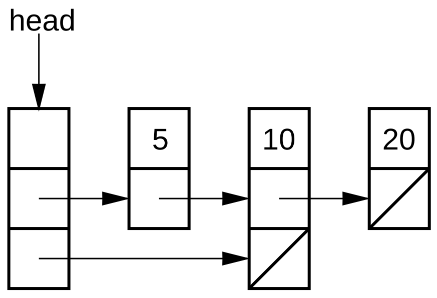
Pick the node size at random (from a suitable probability distribution).



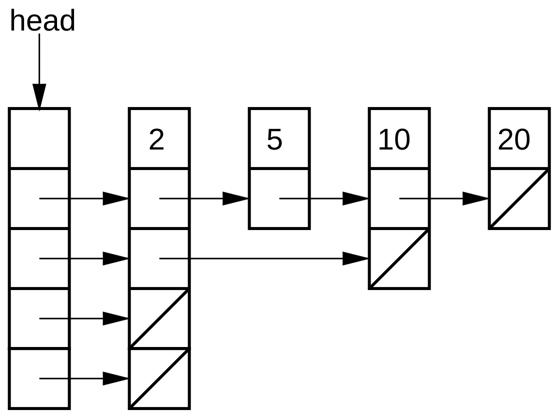
(a)



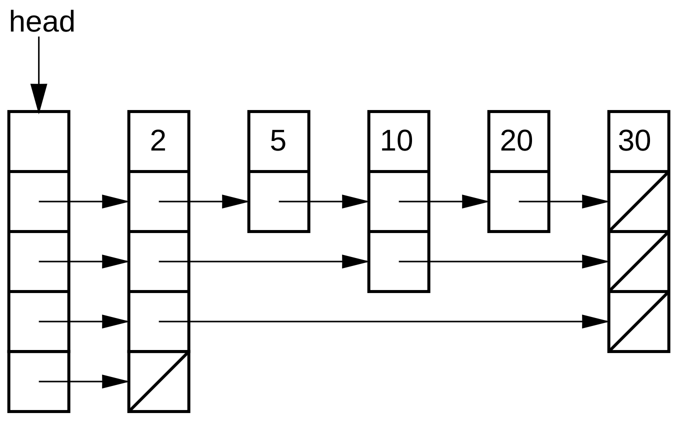
(b)



(c)



(d)



(e)

## Skip List Analysis

What distribution do we want for the node depths?

```
int randomLevel(void) { // Exponential distribution
  for (int level=0; Random(2) == 0; level++); // No-op
  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?

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?

## 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?

## Selection

How can we find the $i$th 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**.

## Properties of Relationships

**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.

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**.

## 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 $i$th largest.
- Find several ranks simultaneously.

## 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?

## Proof of Lower Bound

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.

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.

## 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 $i$th 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%.

## Average Number of Assignments

`Find_max` does an assignment on the $i$th iteration iff $L[i]$ is the biggest 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 $n$th <u>**harmonic number**</u>: $\mathcal{H}_n$.

## Technique

Since $i \le 2^{\lceil \log i \rceil}$, $1/i \ge 1/2^{\lceil \log i \rceil}$.

Thus, if $n = 2^k$

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

Using similar logic, $\mathcal{H}_{2^k} \le k + \frac{1}{2^k}$.

Thus, $\mathcal{H}_n = \Theta(\log n)$.

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

## Variance

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

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

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?

## 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?

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!

## Lower Bound for Second

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?

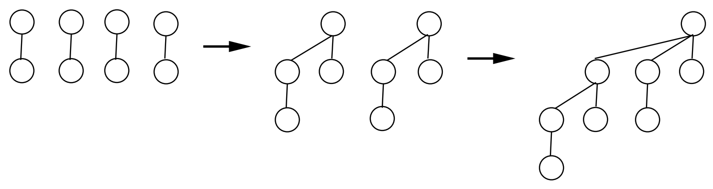
## Binomial Trees

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.



Algorithm:

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

Cost?

## Binomial Tree Representation

We could store the binomial tree as an explicit tree structure.

We can also store the binomial tree implicitly: In an array.

Assume two trees, each with $2^k$ nodes, are in the 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.

## Adversarial Lower Bounds Proof

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.

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.

## 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 $k$?

## 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?

## 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.

## Find Min and Max

Find them independantly: $2n-2$.

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

Should be able to do better(?)

Try divide and conquer.

```
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]); // Only 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:

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

## Solving the Recurrence

Assume $n = 2^k$.

Let's expand the recurrence a bit.

$$\begin{aligned}
f(n) &= 2f(n/2) + 2 \\
&= 2[2f(n/4) + 2] + 2 \\
&= 4f(n/4) + 4 + 2 \\
&= 4[2f(n/8) + 2] + 4 + 2 \\
&= 8f(n/8) + 8 + 4 + 2 \\
&= 2^i f(n/2^i) + \sum_{j=1}^{i} 2^j \\
&= 2^{k-1} f(n/2^{k-1}) + \sum_{j=1}^{k-1} 2^j \\
&= 2^{k-1} f(2) + \sum_{j=1}^{k-1} 2^j \\
&= 2^{k-1} + \sum_{j=1}^{k-1} 2^j \\
&= n/2 + 2^k - 2 \\
&= 3n/2 - 2
\end{aligned}$$

## Looking Closer

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?

## 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 $n$.

# Algorithms from Recurrences

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.

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

Cost:

What is the corresponding algorithm?

## The Lower Bound

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.

Who can get ignored?

What is the initial state?

What is the final state?

How is this relevant?

## Lower Bound (cont.)

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.

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

| | | | | |
|---|---|---|---|---|
| $N : N$ | $(i-2,$ | $j+1,$ | $k+1,$ | $l)$ |
| $W : W$ | $(i,$ | $j-1,$ | $k,$ | $l+1)$ |
| $L : L$ | $(i,$ | $j,$ | $k-1,$ | $l+1)$ |
| $L : N$ | $(i-1,$ | $j+1,$ | $k,$ | $l)$ |
| *or* | $(i-1,$ | $j,$ | $k,$ | $l+1)$ |
| $W : N$ | $(i-1,$ | $j,$ | $k+1,$ | $l)$ |
| *or* | $(i-1,$ | $j,$ | $k,$ | $l+1)$ |
| $W : L$ | $(i,$ | $j,$ | $k,$ | $l)$ |
| *or* | $(i,$ | $j-1,$ | $k-1,$ | $l+2)$ |

# 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:

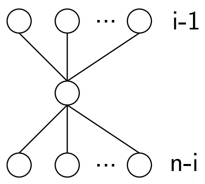
| | | | | |
|---|---|---|---|---|
| $N : N$ | $(i - 2,$ | $j + 1,$ | $k + 1,$ | $l)$ |
| $L : N$ | $(i - 1,$ | $j + 1,$ | $k,$ | $l)$ |
| $W : N$ | $(i - 1,$ | $j,$ | $k + 1,$ | $l)$ |
| $W : W$ | $(i,$ | $j - 1,$ | $k,$ | $l + 1)$ |
| $L : L$ | $(i,$ | $j,$ | $k - 1,$ | $l + 1)$ |

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.

## Finding the $i$th 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?

## 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 is $i$th best, we are done.

If not, solve the subproblem recursively.

## Cost

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

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

Let $f(n, i)$ be the average time to find the $i$th best of $n$ elements.

- Array bounds go from 1 to $n$
- Call $j$ the position of the pivot

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

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.

## Technique

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

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

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

## Technique (cont.)

Therefore,

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

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

## Solving the Recurrence

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

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

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?

## 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.

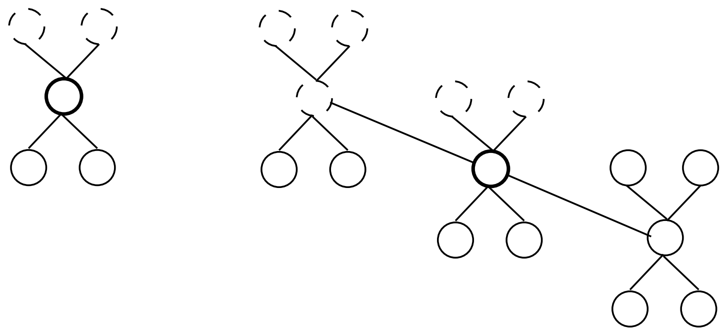
Choose a constant $c$, pick the median of a sample of size $n/c$ elements.

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

## 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 SPLIT 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 $(3n + 5)/10$

## Constructive Induction

Is the following recurrence linear?

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

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

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

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

Thus, we can use induction to prove that,

$$\forall n \ge 395, f(n) \le 23n.$$

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

## Changing the Model

What if we settle for the "approximate best?"

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

1. $X = Y$.
2. $X$'s rank is "close to" $Y$'s rank:

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

3. $X$ is "usually" $Y$.

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

4. $X$'s rank is "usually" "close" to $Y$'s rank.

We often give such algorithms names:

1. Exact or deterministic algorithm.
2. Approximation algorithm.
3. Probabilistic algorithm.
4. Heuristic.

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).

## 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 $\frac{mn}{m+1}$.

## 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**.

## 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 the running time.

## 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?

## Swap Sorts

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.

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.

# Heap Sort

**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.

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.

## 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:

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

## 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.

## Quicksort Average Cost

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

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

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

## Average Cost (cont.)

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

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

## Average Cost (cont.)

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

$$\begin{aligned}
f(n{+}1) &\leq 2 + \frac{n+2}{n+1} f(n) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n} f(n-1)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1} f(n-2)\right)\right) \\
&= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2} f(1))\right) \\
&= 2\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots \right. \\
&\qquad \left. + \frac{n+2}{n+1}\frac{n+1}{n} \cdots \frac{3}{2}\right) \\
&= 2\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right) \\
&= 2 + 2(n+2)\left(\mathcal{H}_{n+1} - 1\right) \\
&= \Theta(n \log n).
\end{aligned}$$

## Lower Bound for Sorting

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?

**Information theory** 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?

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 $\Theta(\log n!)$.

## 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.)

# Optimal Sorting

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

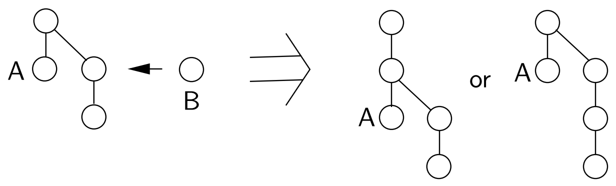
What is the absolute minimum number of comparisons needed to sort?

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 $n$?

Build the following poset:



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

Now, put in the off-element.

Total cost?

## 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:

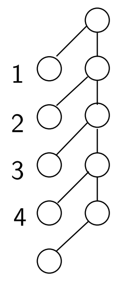
- Pair up all the nodes with $\lfloor \frac{n}{2} \rfloor$ comparisons.
- Recursively sort the winners.
- Fold in the losers.

## Finishing the Sort

We will use binary insert to place the losers.

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

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



Pick the order of inserts to optimize the binary searches.

- 3 (2 compares: size 3)
- 4 (2 compares: size 3)
- 1 (3 compares: size 7)
- 2 (3 compares: size 7)

We can form an algorithm: Binary Merge.

This sort is called **merge insert 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 that 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.

## 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?

## 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.

## 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.

If you consider the 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).

## Exponentiation

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 $m^n = m^{\lfloor n/2 \rfloor} m^{\lfloor n/2 \rfloor} m$.

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

## 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 $\beta$ is the number of 1's in the 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).

## Largest Common Factor

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

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

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

## Analysis of LCF

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

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

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

Total cost:

## 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: $\Omega(n^2)$.

## 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.

## Strassen's Algorithm

(1) Trade more additions/subtractions for fewer multiplications in $2 \times 2$ case.

(2) Divide and conquer.

In the straightforward implementation, $2 \times 2$ 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.

## Strassen's Algorithm (cont)

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.

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?

## Divide and Conquer Recurrences

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.

## Divide and Conquer Recurrences (cont)

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

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

$a^m = a^{\log_b n} = n^{\log_b a}$

The summation is a geometric series whose sum depends on the ratio

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

There are 3 cases.

## D & C Recurrences (cont)

(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) $r = 1$

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

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

(3) $r > 1$

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

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

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

## 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: $T(n) = \Theta(n^2)$.

## 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?

## 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?

## Probablistic Algorithms

Some probablistic algorithms:

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

Using number theory, we can create a cheap test that will determine that a number is composite (if it is) 50% of the time.

Algorithm:

```
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!

## 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.

## 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 range:

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

Must pick good values for $b$ and $t$.

- $t$ should be prime.

## Random Number examples

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

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

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

Suggested generator:

$$r(i) = 16807r(i-1) \text{ mod } 2^{31} - 1.$$

## Introduction to the Slide Rule

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?

The slide rule 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.

## Representing Polynomials

A vector $\mathbf{a}$ 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 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**.

## 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 (at the same points), we can simply multiply the corresponding pairs of values to get the corresponding values for polynomial $AB$.

Process:

- Evaluate polynomials $A$ and $B$ at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.

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. (Why?)

## An Example

Polynomial A: $x^2 + 1$.
Polynomial B: $2x^2 - x + 1$.

Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

Note that evaluating a polynomial at 0 is easy.

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?

$$\begin{aligned} AB(-1) &= (2)(4) = 8 \\ AB(0) &= (1)(1) = 1 \\ AB(1) &= (2)(2) = 4 \end{aligned}$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

## An Observation

In general, we can write $P_a(x) = E_a(x) + O_a(x)$ where $E_a$ is the even powers and $O_a$ is the odd powers. So,

$$P_a(x) = \sum_{i=0}^{n/2-1} a_{2i}x^{2i} + \sum_{i=0}^{n/2-1} a_{2i+1}x^{2i+1}$$

The significance is that when evaluating the pair of values $x$ and $-x$, we get

$$\begin{array}{rcl} E_a(x) + O_a(x) & = & E_a(x) - O_a(-x) \\ O_a(x) & = & -O_a(-x) \end{array}$$

Thus, we only need to compute the E's and O's once instead of twice to get both evaluations.

# 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 $z$ is a **primitive nth root of unity** if

1. $z^n = 1$ and
2. $z^k \neq 1$ for $0 < k < n$.

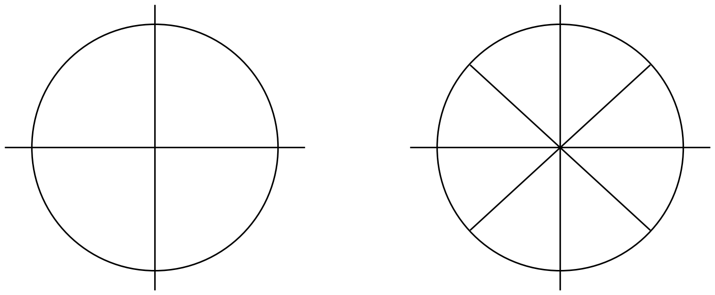
$z^0, z^1, ..., z^{n-1}$ are the **nth roots of unity**.

Example: For $n = 4$, $z = i$ or $z = -i$.

Identity: $e^{i\pi} = -1$.

In general, $z^j = e^{2\pi ij/n} = -1^{2j/n}$.

- Significance: We can find as many points on the circle as we need.



## Evaluation

Define an $n \times n$ matrix $A_z$ with row $i$ and column $j$ as

$$A_z = (z^{ij}).$$

Example: $n = 4$, $z = i$:

$$A_z = \begin{array}{rrrr} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{array}$$

Let $a = [a_0, a_1, ..., a_{n-1}]^T$ be a vector.

We can evaluate the polynomial at the $n$th roots of unity:

$$F_z = A_z a = b.$$

$$b_i = \sum_{k=0}^{n-1} a_k z^{ik}.$$

## Another Example

For $n = 8$, $z = \sqrt{i}$. So,

$$A_z = \begin{array}{rrrrrrrr} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i} \end{array}$$

We still have two problems:

1. We need to be able to do this fast. Its still $n^2$ multiplies to evaluate.
2. If we multiply the two sets of evaluations (cheap), we still need to be able to reverse the process (interpolate).

## Interpolation

The interpolation step is nearly identical to the evaluation step.

$$F_z^{-1} = A_z^{-1} b' = a'.$$

What is $A_z^{-1}$? This turns out to be simple to compute.

$$A_z^{-1} = \frac{1}{n} A_{1/z}$$

In other words, do the same computation as before but substitute $1/z$ for $z$ (and multiply by $1/n$ at the end).

So, if we can do one fast, we can do the other fast.

## Fast Polynomial Multiplication

An efficient divide and conquer algorithm exists to perform both the evaluation and the interpolation in $\Theta(n \log n)$ time.

- This is called the **Discrete Fourier Transform** (DFT).
- It is a recursive function that decomposes the matrix multiplications, taking advantage of the symmetries made available by doing evaluation at the $n$th roots of unity.

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 $AB$.

## Discrete Fourier Transform

```
Fourier_Transform(double *Polynomial, int n) {
  // Compute the Fourier transform of Polynomial
  // with degree n. Polynomial is a list of
  // coefficients indexed from 0 to n-1. n is
  // assumed to be a power of 2.
  double Even[n/2], Odd[n/2], List1[n/2], List2[n/2];

  if (n==1) return Polynomial[0];
  for (j=0; j<=n/2-1; j++) {
    Even[j] = Polynomial[2j];
    Odd[j] = Polynomial[2j+1];
  }
  List1 = Fourier_Transform(Even, n/2);
  List2 = Fourier_Transform(Odd, n/2);
  for (j=0; j<=n-1, j++) {
    Imaginary z = pow(E, 2*i*PI*j/n);
    k = j % (n/2);
    Polynomial[j] = List1[k] + z*List2[k];
  }
  return Polynomial;
}
```

This just does the transform on one of the two polynomials. The full process is:

1. Transform each polynomial.
2. Multiply resulting values ($O(n)$ multiplies).
3. Do the inverse transformation on the result.

Cost: $\Theta(n \log n)$

## Fibonacci Revisited

Consider again the recursive function for computing the $n$th 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?

If we could eliminate redundancy, cost would be 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.

## Dynamic Programming

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".

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.

## 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 $K$.

## 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!

## 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: $\Theta(nK)$.

## Knapsack Example

$K = 10$.

Five items: 9, 2, 7, 4, 1.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $k_1 = 9$ | $O$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $I$ | $-$ |
| $k_2 = 2$ | $O$ | $-$ | $I$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $O$ | $-$ |
| $k_3 = 7$ | $O$ | $-$ | $O$ | $-$ | $-$ | $-$ | $-$ | $I$ | $-$ | $I/O$ | $-$ |
| $k_4 = 4$ | $O$ | $-$ | $O$ | $-$ | $I$ | $-$ | $I$ | $O$ | $-$ | $O$ | $-$ |
| $k_5 = 1$ | $O$ | $I$ | $O$ | $I$ | $O$ | $I$ | $O$ | $I/O$ | $I$ | $O$ | $I$ |

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.

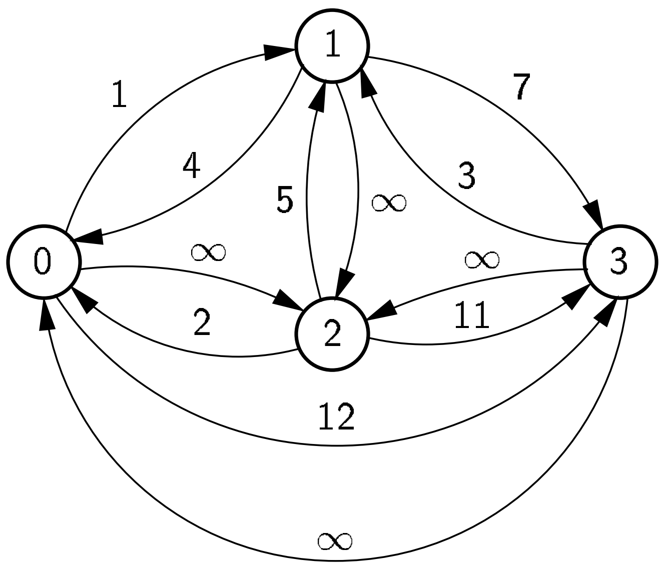
Example: $M(3, 9)$ contains O because $P(2, 9)$ has a solution. It contains I because $P(2, 2) = P(2, 9 - 7)$ has a solution.

How can we find a solution to $P(5, 10)$?
How can we find ALL solutions to $P(5, 10)$?

## All Pairs Shortest Paths

For every vertex $u, v \in$ 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 $k$.



```
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];
}
```

## Reductions

A **reduction** is a transformation of one problem to another.

Purposes: To compare the difficulty of two problems.

- To use one algorithm to solve another problem (upper bound).
- To compare the relative difficulty of two problems (lower bound).

Notation: A problem is a mapping of inputs to outputs.

A definition 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 $i < j$.

## 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.

## 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?

## PAIRING Lower Bound

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.

What is the cost of this algorithm?

What does this say about the existence of an $O(n)$ time 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 **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 $P1$ (**S**). Note that **S** MUST BE THE CORRECT SOLUTION for $\mathbf{I}$!

## 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 $P1$.

## 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.

# 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?

$$\begin{bmatrix} 0 & \mathsf{A} \\ \mathsf{A}^\mathsf{T} & 0 \end{bmatrix} \begin{bmatrix} 0 & \mathsf{B}^\mathsf{T} \\ \mathsf{B} & 0 \end{bmatrix} = \begin{bmatrix} \mathsf{AB} & 0 \\ 0 & \mathsf{A}^\mathsf{T}\mathsf{B}^\mathsf{T} \end{bmatrix}.$$

## 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?

## Complexity and Computability

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?

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

## Hard Problems

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.

## Hard Problems (Cont.)

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.

## 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!

# Nondeterministic Algorithm

An algorithm is <u>**nondeterministic**</u> 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.

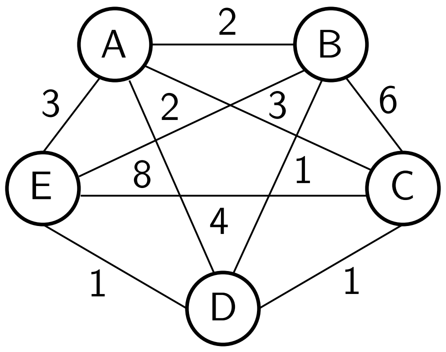
A problem that can be solved in polynomial time by a nondeterministic machine is said to be “in $\mathcal{NP}$.”

Is Towers of Hanoi in $\mathcal{NP}$?

## Traveling Salesman Problem

TRAVELING SALESMAN (1):

- Input: A complete, directed graph $G$ with distances assigned to each edge in the graph.
- Output: The shortest simple cycle that includes every vertex.



Problem: How to tell if a proposed solution is *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.

## $\mathcal{NP}$-complete Problems

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.

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!

## 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.

## 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.

## $\mathcal{NP}$-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 $\mathbf{I}$ of $A$ to an instance $\mathbf{I}'$ of $X$.
    - Describe a polynomial-time transformation from **S**$'$ to **S** such that **S** is the solution for $\mathbf{I}$.

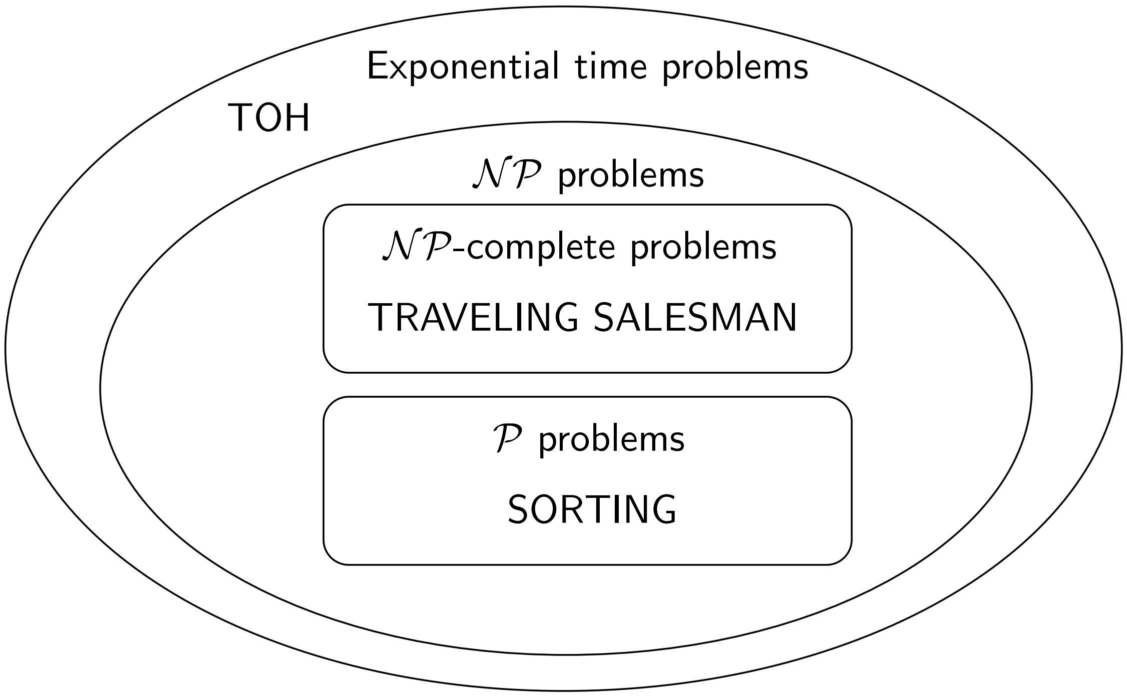
## Cook's Proof Outline

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 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.

## The World of Exponential-time(?) Problems



Question: Does $\mathcal{P} = \mathcal{NP}$?

# 3-SATISFIABILITY (3 SAT)

Input: Boolean expression E in CNF such that each clause contains exactly 3 literals.

Output: YES if the 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 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.

## Replacement

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.

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.

## Third Case

If $E$ is satisfiable, then $E'$ is satisfiable:

- Assume $x_m$ is assigned true.
- Then 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.

## Two Problems

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).

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 $G'$.

## K CLIQUE is $\mathcal{NP}$-Complete

Easy to show that K CLIQUE is in $\mathcal{NP}$.

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.

$$V = \{v[i,j] | 1 \le i \le m, 1 \le j \le k_i\}.$$

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 $k = m$.

## 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 a clique of size $\geq k$, then the clique must have size exactly $k$ and there is one vertex $v[i, j_i]$ in the clique for each $i$.
- There is a truth assignment making each $y[i, j_i]$ true. That truth assignment satisfies $B$.

We conclude that K CLIQUE is $\mathcal{NP}$-hard, therefore $\mathcal{NP}$-complete.

## 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.

## 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.

## 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."

## 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) <u>**matching**</u> 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.

## 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.

Heuristic: First fit

- Place a number in the first bin that fits.
- The number of bins used is no more than twice the sum of the numbers.
- First fit can be much worse than optimal.
- Consider 6 of $1/7 + \epsilon$, 6 of $1/3 + \epsilon$, 6 of $1/2 + \epsilon$.

Better Heuristic: Decreasing first fit

- Sort the items, then use first fit.
- This can be proven to yield no more than 11/9 the optimal number of bins.

## 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.

## 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, $|\text{Odd}| = |\text{Even}|$ because the function $f : |\text{Odd} \rightarrow \text{Even}|$ defined by $f(x) = x - 1$ is a bijection.

How about $|\text{Even}| = |\mathbb{N}|$?

## 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.

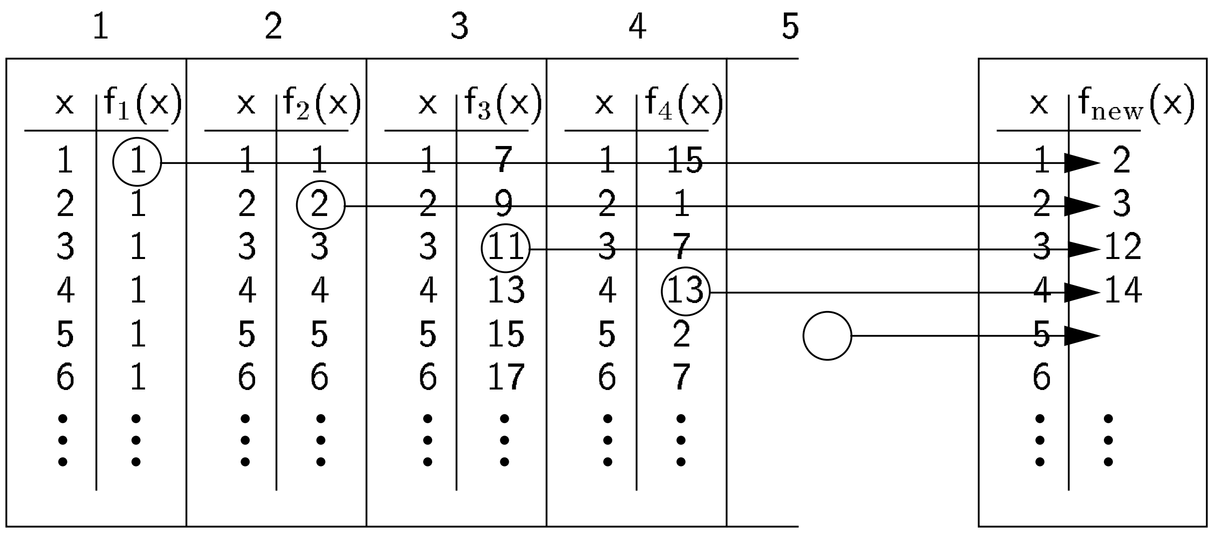
As a corollary, the set of all computer programs is countable.

# There are more Functions than Programs

Consider the set of functions $f(x) = y$ for $x, y$ natural numbers.

The set of such functions is uncountable.

Diagonalization argument:



| 1 | | 2 | | 3 | | 4 | | 5 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| x | $f_1$(x) | x | $f_2$(x) | x | $f_3$(x) | x | $f_4$(x) | | x | $f_{new}$(x) |
| 1 | 1 | 1 | 1 | 1 | 7 | 1 | 15 | | 1 | 2 |
| 2 | 1 | 2 | 2 | 2 | 9 | 2 | 1 | | 2 | 3 |
| 3 | 1 | 3 | 3 | 3 | 11 | 3 | 7 | | 3 | 12 |
| 4 | 1 | 4 | 4 | 4 | 13 | 4 | 13 | | 4 | 14 |
| 5 | 1 | 5 | 5 | 5 | 15 | 5 | 2 | | 5 | |
| 6 | 1 | 6 | 6 | 6 | 17 | 6 | 7 | | 6 | |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | | ⋮ | ⋮ |

Thus, not all functions on natural numbers are computable.

## Halting Problem for Programs

Does this 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 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);
}
```

## 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
}
```

## 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.

## 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.

## Another Reduction Proof

Does there exist ANY 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 ANY 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.

## Other Noncomputable Functions

Does a program halt on EVERY input?

Do two programs compute the SAME function?

Does a particular line in a program get executed?

Does a program compute a particular function?

Does a program contain a “computer virus”?

## 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

## Turing Machines

A tape, divided into squares.

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.

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.

## Formal definition of Turing Machine

A **Turing Machine** is a quadruple ($K$, $\Sigma$, $\delta$, $s$) where

- $K$ is a finite set of **states** (not including $h$).
- $\Sigma$ is an alphabet (containing $\#$, not $L$ or $R$).
- $s \in K$ is the **initial** state.
- $\delta$ 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.

## Turing Machine Example 1

$M = (K, \Sigma, \delta, s)$ where

- $K = \{q_0, q_1\}$,
- $\Sigma = \{a, \#\}$,
- $s = q_0$,
- $\delta =$

| $q$ | $\sigma$ | $\delta(q,\sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(q_1, \#)$ |
| $q_0$ | $\#$ | $(h, \#)$ |
| $q_1$ | $a$ | $(q_0, a)$ |
| $q_1$ | $\#$ | $(q_0, R)$ |

## Turing Machine Example 2

$M = (K, \Sigma, \delta, s)$ where

- $K = \{q_0\}$,
- $\Sigma = \{a, \#\}$,
- $s = q_0$,
- $\delta =$

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(q_0, L)$ |
| $q_0$ | $\#$ | $(h, \#)$ |

## 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** $n$.

$$\begin{aligned}
(q_0, \underline{a}aaa) &\vdash_M (q_1, \underline{\#}aaa) \\
&\vdash_M (q_0, \#\underline{a}aa) \\
&\vdash_M (q_1, \#\underline{\#}aa) \\
&\vdash_M (q_0, \#\#\underline{a}a) \\
&\vdash_M (q_1, \#\#\underline{\#}a) \\
&\vdash_M (q_0, \#\#\#\underline{a}) \\
&\vdash_M (q_1, \#\#\#\underline{\#}) \\
&\vdash_M (q_0, \#\#\#\#\underline{\#}) \\
&\vdash_M (h, \#\#\#\#\underline{\#})
\end{aligned}$$

## Computations

Convention:
$M$ 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 $\Sigma_0^*$ to $\Sigma_1^*$. 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: $f(w_1, ..., w_k) = u$,
$(s, \#w_1\#w_2\#...\#w_k\underline{\#}) \vdash_M^* (h, \#u\underline{\#})$.

# Functions on Natural Numbers

Represent numbers in <u>**unary**</u> 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 $n \in \mathbb{N}$.

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $I$ | $(h, R)$ |
| $q_0$ | $\#$ | $(q_0, I)$ |

$$(q_0, \#II\underline{\#}) \vdash_M (q_0, \#II\underline{I}) \vdash_M (h, \#III\underline{\#}).$$

In general, $(q_0, \#I^n\underline{\#}) \vdash_M^* (h, \#I^{n+1}\underline{\#})$.

What about $n = 0$?

# Turing-decidable Languages

A language $L \subset \Sigma_0^*$ is **<u>Turing-decidable</u>** iff function $\chi_L : \Sigma_0^* \to \{\boxed{\text{Y}}, \boxed{\text{N}}\}$ 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.

## 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: $\Sigma_0 = \{a, b\}$,
$L = \{w \in \Sigma_0^* : w$ contains at least one $a\}$.

| $q$ | $\sigma$ | $\delta(q, \sigma)$ |
|---|---|---|
| $q_0$ | $a$ | $(h, a)$ |
| $q_0$ | $b$ | $(q_0, L)$ |
| $q_0$ | # | $(q_0, L)$ |

Every Turing-decidable language is Turing-acceptable.

## 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

## 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.

## 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.

## 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