CS 4104: Data and Algorithm Analysis

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Fall 2010

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.)

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 (1)

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

We need a yardstick.

Algorithm Efficiency (1)

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

We need a yardstick.

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

Algorithm Efficiency (2)

We need:

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

Cost Model (1)

To get a measurement, we need a model.

Example:

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

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

Cost Model (1)

To get a measurement, we need a model.

Example:

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

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

One assignment was made, so the cost is 1.

Cost Model (1)

To get a measurement, we need a model.

Example:

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

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

One assignment was made, so the cost is 1.

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

Cost Model (1)

To get a measurement, we need a model.

Example:

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

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

One assignment was made, so the cost is 1.

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

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

Cost Model (2)

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

Cost Model (2)

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

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

Cost Model (2)

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

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

What makes a model “good”?

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

Big Issues

How do we create an efficient algorithm?

Big Issues

How do we create an efficient algorithm?

Q: How do we recognize a “good” algorithm?

Big Issues

How do we create an efficient algorithm?

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

Big Issues

How do we create an efficient algorithm?

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

How “hard” is a problem?

Big Issues (2)

General Plan:

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

Problems (1)

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

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

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

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

Problems (2)

An **algorithm** executes the mapping.

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

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

Categories of Hard Problems (1)

- A conceptually hard problem.
  - If we understood the problem, the algorithm might be easy. [Natural Language Processing]
  - Artificial Intelligence.

Categories of Hard Problems (1)

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

Categories of Hard Problems (2)

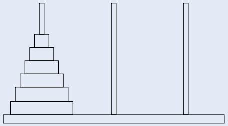
- A computationally hard problem.
  - The algorithm is expensive.
  - Class 1: No inexpensive algorithm is possible. [TOH]
  - Class 2: We don’t know if an inexpensive algorithm is possible. [Traveling Salesman]
  - Complexity Theory

Categories of Hard Problems (2)

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

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?

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 (1)

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

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

Recursive Solutions (1)

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

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

Problem-solving heuristics used:

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

Recursive Solutions (1)

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

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

Problem-solving heuristics used:

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

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

Recursive Solutions (2)

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

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

Algorithm Upper Bounds (1)

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

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

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

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

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

Algorithm Upper Bounds (2)

Examples:

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

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 **recurrence relation**:

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

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

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

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

How do we prove that this ALWAYS works?

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

Here again is the proof for TOH:

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

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

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

Is this a good algorithm?

Lower Bound of a Problem (1)

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

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

Consider a graph of cost for each possible algorithm.

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

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

Lower Bound of a Problem (2)

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

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

Growth Rate vs. I_n

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

Examples: Consider the graphs for each of these

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

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

Lower Bound (cont.)

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

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.