Analysis and Design of Algorithms
Algorithms & Assessment

Lecture Notes by Dr. Wang, Rui
Fall 2008
Department of Computer Science
Ocean University of China

October 22, 2009

What are Algorithms?

- “algorithm” is derived from Mohammed Al-Khowarizmi, 9th century Persian, mathematician credited with formalizing pencil-and-paper methods for addition, subtraction, multiplication, and division.
- Examples of “algorithms” in the nature:
  - Your DNA
  - Cook book
- Informally, an algorithm is a well-defined procedure that takes a set of objects as input and produces a set of objects as output. An algorithm is sequence of steps that transform the input into output.
- An algorithm A solves (is for) a problem Π:
  - For any instance I of Π, A is applicable on I and eventually stops and produces a solution for I.

Why is the study of algorithms worthwhile?

Graduates should have been well aware of the role algorithms play in Computer Science. If not, please listen to what Canadian high-school students complained on a forum.

- Could the concept of a repeated task, until a condition is met, be explained without using the term “for loop” or a paramount concern of how many semi-colons it requires? I’ve heard of Computer Science classes where the same material was taught in grades 10, 11, and 12 - just with a different programming languages each year. Great, kids will know how to write “for loop” in 3 different ways, and still not understand as to why. It seems that technical content takes preference over creativity and logic. I think that we should concentrate on the science and art parts of the subject.
- “Programming should be about concepts, and knowing what a for loop does, not how to write the same for loop in 3 different languages.”– exactly. Knowing the concepts of programming is more important than knowing any particular language.

Why...Cont.

I agree entirely with what you are saying. Just learning the syntax of a language doesn’t teach you anything about computer science. Computer Science is more about problem solving and algorithms, than pure coding. Nobody wants to end up being a code monkey, but that’s what I see computer science classes teaching you sometimes. I remember that in grade 10, I decided to skip the grade 10 computer science course and go to the grade 11 one. Boy, that was a smart decision. Besides being particularly slow, my computer science teacher was very good in that he focused on more than just writing code. I remember that the very first assignment he gave everyone taking the course had nothing to do with programming. It was purely problem solving. When I took the grade 12 class, we spent a lot of time learning algorithms. The teacher assumed that you know most of the syntax already and didn’t waste class time teaching us how to code what he’s talking about. I found that to be a great approach. Clearly, it was. By the end of Grade 12, all of us were writing simple AI’s for a game called Connect 4 or Hex. After taking grade 11 and grade 12, I feel that I’ve learned quite a lot about computer science, and that is shown by my good performance on programming contests. To compare, I sometimes go into the grade 10 class that is being taught by a teacher who has very little experience in problem solving and algorithms. All I ever see the grade 10 class doing is writing programs to display some sort of text on the screen. Here is the typical class assignment: “Ok class So, today you’re going to be making a program to read in the name of several items that can be sold in a shop. Each item will be assigned a price and a quantity. Your output will be the total price.” Perhaps, when you’re in the first month of Computer Science and you’re just learning basic syntax, that would be a good assignment. However, if you spend an entire semester doing assignments similar to that, you will never want to take Computer Science again. I feel that my school’s grade 10 computer science course discourages people to continue with the grade 11 and grade 12 courses. That is a great loss, because people who would potentially grow up to become great Computer Scientist go on to do something else.

How to Assessment an Algorithm?

- Two factors: time complexity and space complexity.
- (time) complexity of a Alg. is measure with number of operations as a function of the size of input.
  - size: number of bits. Or conveniently choose a parameter relevant to the problem:
    - Sorting: number of items.
    - Graph problems: number of vertices and edges.
  - number of operations: This depends on model.
    - RAM (Random Access Machine): instructions, like ADD, MULT, STORE, each takes 1 unit of time (unit-cost RAM).
- Space complexity of a Alg: # of memories as a function of the size of input.
- Time complexity is the dominator, Time$\geqslant$ Space.

Types of complexity

- For an algorithm A:
  - Let $T(I)$ be the time the Alg A takes on instance I.
    - Worst case complexity: $T(n) = \max\limits_{|I|=n} T(I)$.
    - Average case complexity: $T(n) = \sum\limits_{|I|=n} T(I)\,\mathrm{Pr}(I)$.
  - We stress "worst case".
- For a problem Π:
  - Complexity: $\min\limits_{A}\{T_A(n) \mid A \textit{ solves } \Pi\}$.
  - Upper bound $f(n)$: there exist an algorithm for Π with complexity $\leqslant f(n)$.
  - Lower bound $f(n)$: any algorithm for Π must have complexity $\geqslant f(n)$.

Comparison of Complexity Functions

Suppose an algorithm A with time complexity $T(n)$ runs on a computer that performs one million (1000000) operations/second.

$T(n)$	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 seconds	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Distinction between Polynomial Functions and Exponential Functions

- There is a significant distinction between polynomial time algorithms and exponential time algorithms.
- The two exponential complexity functions have much more explosive growth rates.
- Intuitively:
  - *polynomial time = efficient,*
  - *exponential time = inefficient*.
- Can we rely on the improvement of computer technology for exponential functions?

Effect of Improved Technology

- Suppose in a time interval, algorithm of complexity $T(n)$ can performs M operations and thus can solve instances of size $n = T^{-1}(M)$, i.e. $T(n) = M$.
- Then, on 1000 times faster computer and within the same time, the algorithm can execute $1000M = 1000T(n)$ operations and can solve instances of size $N = T^{-1}\big(1000T(n)\big)$.
- $T(n) = n^2$ gives us

$$N = \sqrt{1000n^2} = 31.6n,$$

  - the ability is 30 times increased.
- $T(n) = 2^n$ gives us $N = \log(1000 \times 2^n) = n + 9.97$,
  - the ability is only 10 added.
- Next table, listing the sizes of problem instance solvable in one hour for several polynomial and exponential time algorithms, reveals more.

$T(n)$	present computer	100 times faster	1000 times faster
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_2$	$3.89N_2$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

RWang @ CS of OUC Algorithms – 10 / 34

Conclusion

The two tables reveal fundamental distinction between polynomial time algorithms and exponential ones.

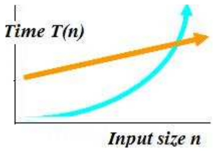


Figure 2.1: $M_{6\times7}$.

- There is wide agreement that
- Intuitively:
  - "polynomial time = good = efficient = fast",
  - a problem has not been well-solved until a polynomial time algorithm is known for it.

RWang @ CS of OUC Algorithms – 11 / 34

Asymptotic Notations

- $f(n) = O\big(g(n)\big) \Leftrightarrow \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = C < \infty.$
  - f grows the same or slower than g.
- $f(n) = \Omega\big(g(n)\big) \Leftrightarrow \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = C > 0.$
  - f grows the same or faster than g.
- $f(n) = \Theta\big(g(n)\big) \Leftrightarrow \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = C,\ 0 < C < \infty.$
  - f grows the same as g.
- $f(n) = o\big(g(n)\big) \Leftrightarrow \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = 0.$
  - f grows slower than g.

E.g., $n^3 + n^2 - n + 7 = O(n^3)$, $21 = O(1)$, $\sin n = \Theta(n) = O(n)$.

Polynomial and Exponential Time Algorithms

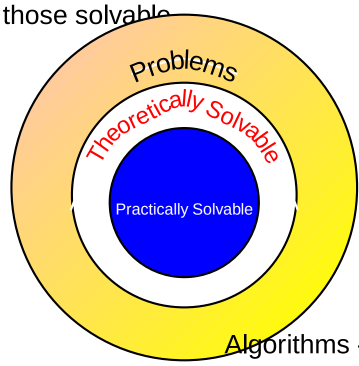
- An algorithm runs in POLYNOMIAL TIME if there exists a constant k such that its worst-case time complexity is $O(n^k)$.
- Put it another way:
  - A polynomial time algorithm is defined to be one whose (worst) time complexity function is $O\big(p(n)\big)$ for some polynomial function p, where n is used to denote the input length.
- Any algorithm whose time complexity function cannot be so bounded is called an exponential time algorithm.
  - Although it should be noted that this definition includes certain non-polynomial time complexity functions, like $n^{\log n}$, which are not normally regarded as exponential functions.

Algorithm Analysis (Omitted) 14 / 34

Algorithm Design 15 / 34

Introduction

- In this section, we are going to learn some useful techniques for designing algorithms.
- But, can we algorithmically solve any given problem?
- No. There are problems that have no algorithms.
  - E.g. the Halting problem:
    - given a program p and a input x, decide whether p, taking x as input, will eventually stop.
    - Halting problem has been proved undecideable.
  - Another problem: given an integer n, is there n consecutive 5's appearing in π? This problem is open (to me).
- In real world, there are uncountable number of problems.
- But, only a countable number of problems are solvable with algorithm?
- Thus, there are more problems unsolvable than those solvable.



Procedure of Algorithm Design

1. Design an algorithm.
   - Need deep insight into the problem, fully understand the structure of the problem.
2. Prove the algorithm is correct.
3. Analyze the complexity of the algorithm.

Paradigms for designing Algorithms

1. Reduction (transformation).
2. Greedy.
3. Divided-and-Conquer.
4. Dynamic Programming.
5. Data Structure Invention.

There are some other techniques worth study. Students should keep this in mind:

> *When the only tool you own is a hammer, every problem begins to resemble a nail.*
>
> Abraham Maslow

Reduction

The Idea: Reduce the problem to a known problem (i.e. by using algorithms for other problems).

Remark: Though this seems trival, it is indeed powerful and activates the foundation of the theory of NP-Completeness.

The idea will be demonstrated by examples.

Reduction Examples

Example 2.1 *Determine if an array of n numbers contains repeated elements.*

- **Solution 1:** Compare each element to every other element. This uses $\Theta(n^2)$ steps.
- **Solution 2:** Sort (by Heapsort) the n numbers. Then, determine if there is a repeat in $O(n)$ steps. Total: $\Theta(n \log n)$ steps!

Reduction Examples

Example 2.2 *Given a list of n points in the plane, determine if any 3 of them are collinear (lie on the same line).*

- **Solution 1:** Using a triple loop, compare all distinct triples of points, so this takes $O(n^3)$ time.
- **Solution 2:** $O(n^2 \log n)$.

1: **for** each point P in the list **do**
2:    **for** each point Q in the list **do**
3:       compute the slope of the line connecting P with Q and save it in a list
4:    **end for**
5:    determine (Example 1.1) if there are any duplicated slops in the list
6: **end for**

Reduction Examples

Example 2.3 *Find a minimum vertex cover for a given graph $G = (V, E)$.*

Note that:

- a **vertex cover** of $G = (V, E)$ is a subset $C \subseteq V$ such that for every edge $e = \{u, v\} \in E$ either u or v belongs to C.
- an **independent set** of $G = (V, E)$ is a subset $U \subseteq V$ such that for every edge $e = \{u, v\} \in E$ either u or v does not belong to U.
- C is a vetex cover iif $V - C$ is an independent set.

The last assertion gives us a reduction way that finds minimum vertex cover by computing maximum independent set.

Greedy

- A greedy algorithm always makes the choice that looks best at the moment.
- Every two year old knows this:
  - In order to get what you want, just start grabbing what looks best.
- Greedy algorithms are direct, simple, and fast.
- Greedy algorithms do not always yield optimal solution.
- But for many problems they do.

Greedy Example
–Job Scheduling

Example 2.4 *The Job Scheduling Problem*

Instance: n *Jobs with starting and finishing times* $(< s_1, f_1 >, < s_2, f_2 >, , < s_n, f_n >)$.

Solution: *A set of jobs that do not overlap.*

Cost of Solution: *The number of jobs scheduled.*

Goal: *Given a set of jobs, schedule as many as possible.*

Greedy Job Scheduling Algorithm

The Greedy Idea: First, sort the activities by finish time. Then, starting with the first, choose the next possible activity that is compatible with previous ones.

```
1: Sort the activities according to finish time f_i, ascendingly. Suppose the result is (J_1, J_2, , J_n, )
2: F = 0; i = 0
3: for k = 1 to n do
4:   if s_k > F then
5:     i = i + 1; T[i] = k; F = f_k
6:   end if
7: end for
```

J_i	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9	J_{10}	J_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Greedy Job Scheduling Algorithm

Theorem 2.1 *The greedy algorithm always produces a feasible schedule with the maximum number of jobs.*

Proof: The feasibility (no overlaps) is obviously guaranteed by the fourth line in the algorithm. Suppose the algorithm produces $T = (t_1, t_2, \ldots, t_i,)$ is not optimal. Then there exists a feasible schedule $B = (b_1, b_2, \ldots, b_j)$ with $j > i$, having more activities.

- But for all k, $1 \leqslant k \leqslant i$, we can justify (inductively) that $f_{t_k} \leqslant f_{b_k}$.
- Thus, $s_{b_{i+1}} > f_{b_i} \geqslant f_{t_i}$.
- Moreover, $f_{b_{i+1}} > s_{b_{i+1}} > f_{t_i} > f_{t_{i-1}} > f_{t_{i-2}} > \ldots > f_{t_1}$, so b_{i+1} is not in T.

The above implies that the algorithm should have chosen $J_{b_{i+1}}$ after J_{t_i}, a contradiction. □

Theorem 2.2 *The greedy algorithm completes its work within time* $O(n \log n)$.

Greedy Example
–Minimum Spanning Tree

Example 2.5 *Constructing a minimum spanning tree (M.S.T) for a given graph $G = (V, E)$ of which each edge $e \in E$ is assigned a weight $W(e)$.*

Kruskal's greedy algorithm:

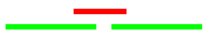
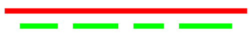
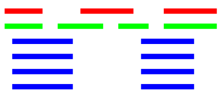
1: Order the edges non-decreasingly by weight, $(e_1, e_2, \ldots, e_m)$, such that $W(e_1) \leqslant W(e_2) \leqslant \ldots \leqslant W(e_m)$
2: Set T to be the empty tree
3: For $i = 1$ to m put edge e_i in T if it does not create a cycle.

Theorem 2.3 *The above algorithm builds an M.S.T. within time $O(nlogn + m) = O(NlogN)$, where $N = m + n$.*

Proof: Omitted. □

Greedy Criteria

Wrong criteria may not work. Take the Job Scheduling problem for example:

- **Shortest Job:** 
- **Earliest Starting Time:** 
- **Conflicting with the Fewest Other Jobs:** 
- **Earliest Finishing Time:** works!

Making Change Problem

Greedy Algorithms do not necessarily yield optimal solution. Locally greedy choice may have negative global consequences.

- **Making Change:** Problem: Find the minimum number of quarters, dimes, nickels, and pennies that total to a given amount.
- **Pu it another way:** You are given an integer x and you are expected to find four non-negative integers a, b, c, and d, such that $x = 25a + 10b + 5c + d$, and such that $a + b + c + d$ (the # of coins) is minimized.

Making Change Problem

- **The greedy algorithm for Making Change problem:** Choose as many quarters as possible (such that $25a \leqslant x$), then for $x - 25a$, choose as many dimes as possible, then nickels, then pennies, so that:

$$\begin{aligned} a &= \left\lfloor \frac{x}{25} \right\rfloor; \quad b = \left\lfloor \frac{x-25a}{10} \right\rfloor; \\ c &= \left\lfloor \frac{x-25a-10b}{5} \right\rfloor; \quad d = x - 25a - 10b - 5c. \end{aligned}$$

- Does this lead to an optimal # of coins?
  - For the currency system of denominations (25,10,5,1), it does.
  - But not for the system of denominations (11,5,1) with $x = 15$.
    - the greedy algorithm provides the solution $15 = 1 * 11 + 4 * 1$, using 5 coins.
    - a better solution is 15=3*5, using only 3 coins.

Divide & Conquer

- **The Idea:**
  - **DIVIDE** problem up into smaller subproblems.
  - **CONQUER** by solving each subproblem.
  - **COMBINE** results together to solve original problem.
- **Examples you know:** binary search, merge sort. . .

Merge Sort

Algorithm 1 MERGE-SORT(A, p, r)

Require: An array A and two index p and r.
Ensure: The elements in $A[p..r]$ is sorted.

1: **if** $p < r$ **then**
2: $\quad q = \left\lfloor \frac{p+r}{2} \right\rfloor$
3: $\quad$ MERGE-SORT(A, p, q), MERGE-SORT$(A, q+1, r)$
4: $\quad$ MERGE(A, p, q, r)
5: **end if**

- Let $T(n)$ denote the number of comparisons performed by MERGE-SORT on n numbers. Then $T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & n > 1 \\ 1 & n = 1 \end{cases}$, giving us that $T(n) = O(n \log n)$.

Multiplying

- The naive pencil-and-paper algorithm for multiplying two n bit numbers uses n^2 multiplications, n^2 additions (+ carrier).
- Karatsuba's 1962 algorithm does the same in $O(n^{1.59})$ steps.

```
       xxxxxxxxxxx
       xxxxxxxxxxx
--------------------
      xxxxxxxxxxxx
     xxxxxxxxxxxx
        ....
xxxxxxxxxxxx
--------------------
xxxxxxxxxxxxxxxxxxx
```

Figure 2.2: Naive Multiplying Alg.

Karatsuba's Algorithm

- Let X and Y each contains n bits. Write $X = ab$, $Y = cd$, where a, b, c, and d are $n/2$ bit numbers. Then

$$XY = \left(a2^{n/2} + b\right)\left(c2^{n/2} + d\right) \tag{2.1}$$
$$= ac2^n + (ad + bc)2^{n/2} + bd \tag{2.2}$$

- This breaks the problem up into 4 subproblems of size n/2, which doesn't do us any good. Instead, Karatsuba observed that

$$XY = (2^n + 2^{n/2})ac + 2^{n/2}(a - b)(d - c) + (2^{n/2} + 1)bd.$$

- Here the problem has been broken into THREE subproblems of size $n/2$ and some adds and shifts. Recursively solve these subproblems, forming an algorithm with time complexity

$$T(n) \leqslant 3T(n/2) + O(n) = O\left(n^{\log_2 3}\right) \approx O\left(n^{1.59}\right).$$