CS3233 - Competitive Programming (Sem II 2001/2002)

Course Information

Instructor:

A/Prof Andrew Lim

Email:	alim@comp.nus.edu.sg
Office:	S16 #08-05
Office Hours:	Monday 1330 - 1530hrs
Tel:	+65 874-6891

Lab Instructor:

Mr Andy Kurnia

Email:	andykurn@comp.nus.edu.sg
Office:	S16 #04-15A
Office Hours:	T.B.A.
Tel:	+65 874-6998

Course Web:

`www.comp.nus.edu.sg/~cs3233/2001`

Online Judge:

`aileen.comp.nus.edu.sg/~cs3233j`

Reference Books

- Introduction to Algorithms, T. Cormen, C. Leiserson, and R. Rivest. MIT Press/McGraw Hill.
- The Algorithm Design Manual, S. Skiena. Springer-Verlag, 1998.
- Algorithms, R. Sedgewick. Addison-Wesley. Many Versions.
- Computer Algorithms, E. Horowitz and S. Sahni, Computer Science Press.
- Programming Pearls, Jon Bentley, Addison-Wesley, 2000.
- The Art of Computer Programming, D. Knuth, Addison-Wesley.

Aims:

- To prepare students in competitive problem solving in computing – we will cover techniques for attacking and solving challenging computational problems.
- To remove students' inertia in writing programs.
- To prepare students for research work in experimental aspects of problem solving and algorithms.

Pre-requisities

- CS1102 or experience in NOI/IOI related activities.

How does it differ from CS3230 or CS4234/CS5234?

- Hands on, broader scope, lesser theoretical Stuff, much more programming, more sweat and fun!
- NO EXAM!
- NO Quota on A+/A grades
- A Chance to represent NUS in the ACM ICPC Contests.

GRADING (tentative)

Programming Tests	50%
Fortnightly. Usually (2 to 3 hours long)	
Graded Homeworks	30%
Written and Programming Assignments	
Tutorial Discussions	10%
Spanish Judge Contributions	10%
Bonus Marks	

Generous marks will be given for challenge questions

Cheating:
10% of your total score for EACH infringement. For example, if you are caught k times for cheating, we will deduct $10k\%$ from your total score.

Late Policy:
Late assignments will not be accepted unless stated otherwise.

SYLLABUS

Techniques	**Applications**
Data Structures	
Binary Search and Sorting	Number Theoretic Problems
Greedy Method	Combinatorial Problems
Divide and Conquer	Graph Problems
Dynamic Programming	Geometry Problems
Backtracking	Set and String Problems
Branch and Bound	
Graph Algorithms	
Heuristic Methods	
NP-Completeness	
- Recognition and Analysis	

Homework 1 (Completion Date: 19 Jan 2002)

```
Question 1 (Local Judge):

Note that the local judge can be found at
aileen.comp.nus.edu.sg/~cs3233j.

a) Explore the local judge.
b) Please attempt question 1 on the local judge.

Question 2 (Spanish Judge):

Note that the Spanish Judge can be found at
acm.uva.es/problemset.

a) Explore the Spanish Judge.
b) Please register yourself with the Spanish
   Judge. Note that your name must take the
   following format: CS3233-02 (your name).

For example, CS3233-02 Andrew Lim

Question 3 (Discussion)

a) Prove that the method permute
   (in lecture 1 notes pg 24) is correct.
b) Prove that the method choose
   (in lecture 1 notes pg 25) is correct.

Question 4 (Discussion)

Why is Master's Theorem Correct?
```


Let's get cracking

Simple Question : "How do I sort a disk file?"
Simple Answer : "Check the thick CLR book!"
Another Answer : "External merge sort"

The **first step** should be to understand the problem.
The **second step** is to get a precise problem statement.

Without a proper and precise definition of the problem, the battle is 90% lost!

Disk Sorting Problem

Input: A file containing at most n positive integers, each less than n, where $n = 10^7$. It is a fatal error if any integer occurs twice in the input. No other data is associated with the integer.

Output: A sorted list in increasing order of the input integers.

Constraints: At most two megabytes of storage is available in main memory; ample disk storage is available. The run time can be at most several minutes; a run time of ten seconds need not be decreased.

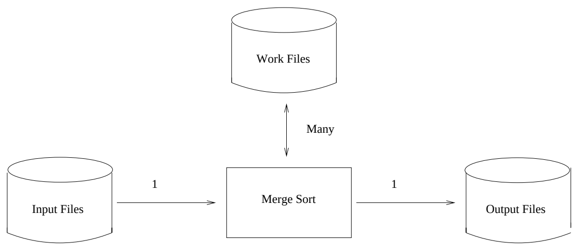
So how?

Some Calculations

8746891 as a string will take up 7 bytes (depends on the programming language). 8746891 as a 32-bit integer will take 4 bytes.

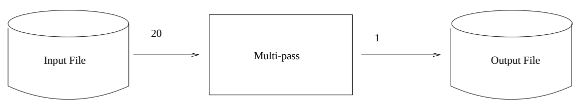
Using Strings, you can store 285k strings in the main memory. If you use, integers, you can store about 500k integers.

Method 1:



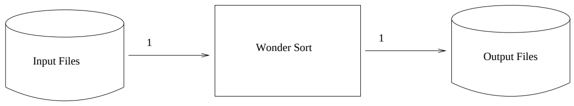
Disk Sort. Many work files. Merging process reads the disk many times. This can be slow. Can be very difficult to code.

Method 2:



Multipass method. Sorts 500,000 integers at a time. Read the input file 20 times. Easy to code, but can be slow.

Method 3:



Use the bitmap/bit vector method.

For example, we can store the set $\{1,3,5,7,10\}$ by using

0 1 0 1 0 1 0 1 0 0 1

Ten million bits is less than 2 megabytes.

This method is simple, efficient, and easy to code. Clearly the best method.

In this course, we must seek the most elegant solution that is easy enough to code.

Exercise : Another Three Problems

Divide yourselves into groups of three and discuss the following questions:

1. Given a sequential file that contains at most four billion 32-bit integers in random order, find a 32-bit integer that isn't in the file (and there must be at least one missing - why?). How would you solve this problem with ample quantities of main memory? How would you solve it if you can use several external "scratch files" files but only a few hundred bytes of main memory?

2. Rotate a one-dimensional vector of n elements left by i positions. For instance, with $n = 8$ and $i = 3$, the vector *abcdefgh* is rotated to *defghabc*. Simple code uses an $n - element$ intermediate vector to do the job in n steps. Can you rotate the vector in time proportional to n using only a few dozen extra bytes of storage?

3. Given a dictionary of English words, find all sets of anagrams. For instance, "pots", "stop", and "tops" are all the anagrams of one another because each can be formed by permuting the letters of the others.

What is An Algorithm?

The word Algorithm comes from the name of a Persian author `Abu Ja'far Mohammed ibn Musa al Khowarizmi` who wrote a textbook on mathematics.

The dictionary (from Dictionary.com) defines it as "a step-by-step problem solving procedure, especially an established, recursive computational procedure for solving a problem in a finite number of steps."

Basically, algorithms are just the ideas behind computer programs

Algorithms Forever

Algorithms are independent of

- the languages that describe them
- the machines that run them
- the locations in which they are being used

They are the only important, durable, and original part of computer science.

An Algorithmic Problem : Sorting

An algorithm should also solve a general specified problem.

An algorithmic problem is specified by providing the set of instances that the algorithm must handle and what the correct output must have.

For Example:
Input: A Sequence of N numbers $a_1, \ldots, a_n$
Output: A permutation of the input sequence such that $a_1 \leq a_2 \ldots \leq a_n$

We would like to have algorithms which are *correct* and *efficient*.

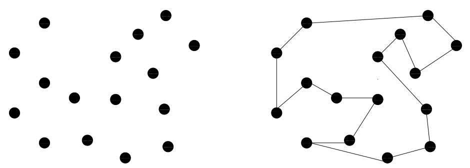
Correctness

For any algorithm, we must prove that it *always* returns the desired output for all valid and legal instances of the problem.

In the sorting example, the algorithm must work, even if the input has only one number, or if the input is already sorted, or if the input contains repeated elements.

Proving Correctness is Hard!

Let's look at the *Traveling Salesman Problem*. The problem has applications in many different domains.

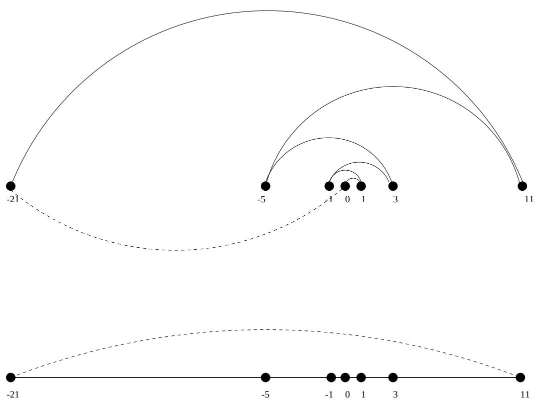


You are given the job to find the best tour for the above problem!

An Idea : Nearest Neighbor Tour

A very popular approach would be to start at any point, say p_0, and then go to its nearest neighbor p_1 that has not been visited before, then repeats the process from p_1 until all the points have been visited. This algorithm is efficient, easy to understand and implement. However, it is not correct!

An instance to show that NN-based algorithm is not correct



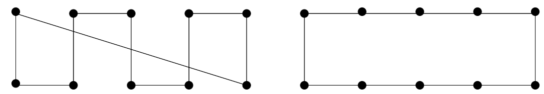
Always starting from the leftmost or rightmost point will not fix the problem.

Another Idea : Closest Pair Tour

Always selecting the closest point is too restrictive. It traps us into making moves that we don't want.

Another idea is to repeatedly connect the closest pair of points whose connection will not cause a cycle or a three-way branch to be formed. This is done until we have a single chain with all the points in it.

Although it works correctly on the previous example, the input below will cause it to have some problems:



Once again, this algorithm is not correct!

A Correct Algorithm

We can first try all possible ordering of the points. Following that, we then select the ordering which minimizes that total length. Since all possible orderings are considered, we are guaranteed to get the shortest possible tour.

$d = \infty$
for each of the n! permutations $\prod_i$
    if $(cost(\prod_i) < d$ then
        $d = cost(\prod_i)$
return d

Because the above algorithm tries all permutations, it is very very slow. It cannot solve problems with more than 20 points.

In fact, no known efficient algorithm exists for the TSP problem.

Permutations Generator

```
public class Permutate{
// Generate all possible permutation of elements in
// array a. We assume that all elements in array a
// are distinct
public static int a[], n;
public static void process(int a[]){
  for (int i=0; i<n; i++)
    System.out.print(a[i]+" ");
  System.out.println();
}
public static void permute(int i){
  int temp;
  if (i==n-1) process(a);
  else{
     permute(i+1);
     for (int j=i+1;j<n;j++){
        temp=a[i]; a[i]=a[j]; a[j]=temp;
        permute(i+1);
        temp=a[i]; a[i]=a[j]; a[j]=temp;
     }
  }
}
public static void main(String args[]){
  n=Integer.parseInt(args[0]);
  a = new int[n];
  for (int i=0; i<n; i++) a[i]=i;
  permute(0);
}
}
```

Combinations Generator

```
public class Choose{
//generate all r-elements subset from the array e
//which contains n elements. The r-elements subset
//is stored in array a.
public static int e[], n;
public static int a[], r;
public static void process(int a[]){
  for (int i=0; i<a.length; i++)
    System.out.print(a[i]+" ");
  System.out.println();
}
public static void choose(int b, int c){
//choose c+1 elements out of b..n-1
  if (c<0) process(a);
  else
     for (int i=b; i<n-c; i++){
         a[c]=e[i]; choose(i+1,c-1);
     }
}
public static void main(String args[]){
  n=Integer.parseInt(args[0]);
  r=Integer.parseInt(args[1]);
  e = new int[n];
  a = new int[r];
  for (int i=0; i<n; i++) e[i]=i;
  choose(0,r-1);
}
}
```


Exercise

1. Prove that the method Permutate is correct

2. Prove that the method Choose is correct

Efficiency and Speedup

"Why not just use the Supercomputer?"

In general, Supercomputers are for people who are too rich and too lazy to design efficient algorithms.

A faster algorithm running on a slower computer will always win for sufficiently large instances.

A study also indicates that most speedups can be obtained at the algorithms and data structure level, rather than at algorithms tuning, data structure reorganization, system-independent code tuning, system-dependent code tuning or a change in the hardware.

However, if improvements can be made at all avenues, then tremendous speedup can be obtained. Speedup of 1000 times is not uncommon.

Expressing Algorithms

We need some means of expressing the algorithms. In order of increasing precision, we have English, pseudo-code, and programming languages.

Unfortunately, the ease of expression is in the reverse direction.

Usually, I will use English or Pseudo-code unless I feel that that particular piece of code is useful to you.

The RAM Model

We need a common language to discuss the complexity of algorithms. We will use the RAM Model of computation.

In the RAM Model:

- Each "Simple" operation takes exactly 1 step.
- Each memory access takes exactly 1 step.
- Loops, Procedure Calls are *NOT Simple* operations.

We measure the run time of an algorithm by counting the number of steps.

Not a perfect model, but good enough in practice.

Best, Worst, and Average-Case

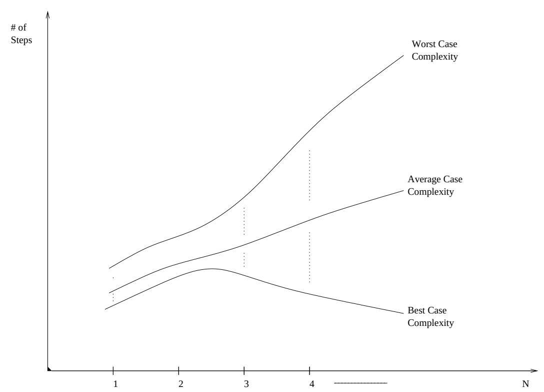
The *worst case complexity* of an algorithm is the function defined by the maximum number of steps taken on any instance of size n.

The *best case complexity* of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.

The *average-case complexity* of an algorithm is the function defined by the average number of steps taken on any instance of size n.

Each of these complexities defines a numerical function – time vs size.

Graphical Illustration



Insertion Sort

$a_1 \leq a_2 \leq \ldots \leq a_k | a_{k+1} \ldots a_n$

The insertion sort starts with an empty list, then successively inserts new elements in the proper position.

After each insertion, the length of the sorted list increases by 1. At the end of n insertions, the list contains exactly n sorted elements. Hence, the algorithm is correct.

But how *efficient* is it?

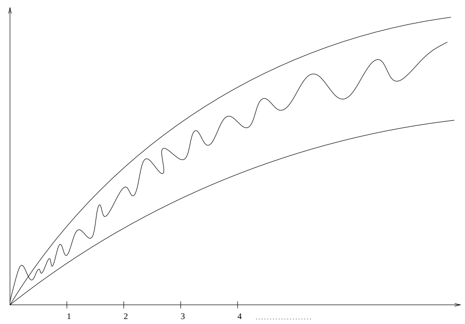
Note that the run time changes with the input instance (even for a fixed size problem).

Exercise

1. Which input instance is the worst for insertion sort? And what is the time complexity of the insertion sort on that instance?
2. Which input instance is the best? What is the time complexity of insertion sort?
3. How can you modify almost any algorithm to have a good best-case running time?
4. Discuss pros and cons of the best, average and worst case complexity measures.

Exact Analysis is hard

Exact analysis requires lots of details. It is to tedious and difficult.



It is usually cleaner and easier to talk about *upper* and *lower bounds.*

This is where the big O notation comes in!

Since running an algorithm on a machine which is twice as fast will, in effect, cut the running times by a constant factor of 2, we will have to ignore constant factors anyway.

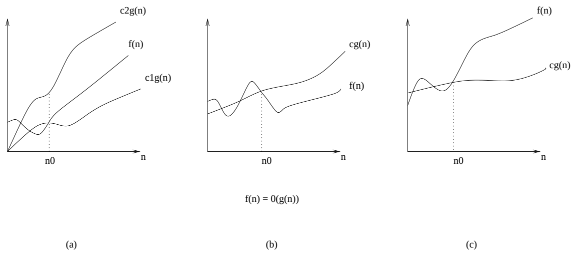
Bounding Functions

- $g(n) = O(f(n))$ means $C \times f(n)$ is an *upper bound* on $g(n)$.
- $g(n) = \Omega(f(n))$ means $C \times f(n)$ is a *lower bound* on $g(n)$.
- $g(n) = \Theta(f(n))$ means $C_1 \times f(n)$ is an upper bound on $g(n)$ and $C_2 \times f(n)$ is a lower bound on $g(n)$.

C, C_1, and C_2 are constants independent of n. All the above conditions need only to hold for all values of n beyond a constant n_0.

We usually do not care about small values of n when will talk about bounding functions.

O, Ω, and Θ



In the above diagrams, the n_0 is the minimum possible. Any bigger value will suffice too.

- $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1.g(n)$ and $c_2.g(n)$ inclusive.
- $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ is always lesser or equal to $c.g(n)$.
- $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ is always greater or equal to $c.g(n)$.

Some Examples

$2n^2 - 10n + 10 = O(n^2)$
$2n^2 - 10n + 10 = O(n^3)$
$2n^2 - 10n + 10 \neq O(n)$
$2n^2 - 10n + 10 = \Omega(n^2)$
$2n^2 - 10n + 10 \neq \Omega(n^3)$
$2n^2 - 10n + 10 = \Omega(n)$
$2n^2 - 10n + 10 = \Theta(n^2)$
$2n^2 - 10n + 10 \neq \Theta(n^3)$
$2n^2 - 10n + 10 \neq \Theta(n)$

Testing Dominance

$f(n)$ dominates $g(n)$ if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

which is the same as $g(n) = o(f(n))$. The little-oh means "grows strictly slower than".

If

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = C$$

where $C > 0$ is a constant, then $g(n) = O(f(n))$.

Randomized Algorithms - An Example

Identifying Repeated Elements

Consider an array $a[]$ of n numbers that has $\frac{n}{2}$ distinct elements and $\frac{n}{2}$ copies of another element. The problem is to identify the repeated element. You may assume that n is even and $n \geq 4$.

What is the minimum number of steps in the worst case for the best algorithm that you can think of?

Any deterministic algorithm will need at least $\frac{n}{2} + 2$ steps.

Why?

Can we take another approach?

The Randomized Approach

```
public static int RepeatedElement(int a[], int n){
  while (true){
    int i = (int)(Math.random()*n);
    int j = (int)(Math.random()*n);
    if ((i!=j) && (a[i]==a[j])) return a[i];
  }
}
```

Does it work?

How many tries

$P_s = \frac{n/2(n/2-1)}{n^2}$.
$P_s \geq \frac{1}{5}$ for all $n \geq 10$.
$P_f < \frac{4}{5}$.
If you perform the above 100 times, the chances of failing is $< (\frac{4}{5})^{100} < 2.04 * 10^{-10}$
It is almost certain that our routine will quit in 100 iterations or less.
So, if $n = 2000000$, any deterministic algorithm, in the worst case, may take 1000000 steps. But our algorithm will take no more than 100 steps!

Advantages and Disadvantages of Randomized Algorithms

Two of the most important advantages of using randomized algorithms are their simplicity and efficiency. In many applications, randomized algorithms not only yield superior asymptotic run time bounds, but they are also competitive in practice.

But how dependable are randomized algorithms? After all there is a nonzero probability that they may fail. But so may your hardware.

If the failure probability is much less than the probability that your hardware may crash, it should be ok for most applications.

If failure cannot be tolerated at all then randomized algorithms may not be suitable.

Recurrence Relations

Many algorithms, especially divide and conquer algorithms, have time complexities which are naturally modeled by recurrence relations.

For example,
$a_n = a_{n-1} + 1$, $a_1 = 1 \rightarrow a_n = n$
$a_n = 2a_{n-1}$, $a_1 = 1 \rightarrow a_n = 2^{n-1}$
$a_n = na_{n-1}$, $a_1 = 1 \rightarrow a_n = n!$

In general, solving complex recurrence relations is an art. Although, most of the cases can be solved using mathematical induction, differential equations, back-substitution, recursion trees, and Master's theorem.

Master's Theorem

Let $a \geq 1$ and $b > 1$ be constants. Let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$ where we interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$.

3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Examples of Usage

$T(n) = 4T(n/2) + n$
$a = 4, b = 2$, and $f(n) = n$.
Is $n = O(n^{2-\varepsilon})$?
Yes, So, $T(n) = \Theta(n^2)$, case 1.

$T(n) = 4T(n/2) + n^2$
$a = 4, b = 2$, and $f(n) = n^2$.
Is $n^2 = O(n^{2-\varepsilon})$?
No if $\varepsilon > 0$
Yes if $\varepsilon = 0$, case 2
Therefore, $T(n) = \Theta(n^2 \log n)$.

$T(n) = 4T(n/2) + n^3$
$a = 4, b = 2$, and $f(n) = n^3$.
Is $n^3 = \Omega(n^{2+\varepsilon})$?
Yes, for $0 < \varepsilon < 1$, so case 3 might apply.
Is $4(n/2)^3 \leq c.n^3$?
Yes, for $c \geq 1/2$, so there exists a $c < 1$ to satisfy the regularity condition, so case 3 applies and $T(n) = \Theta(n^3)$.

Exercise: Famous Algorithms and Their Recurrences

Matrix Multiplication : Strassen's divide and conquer algorithm takes
$T(n) = 7T(n/2) + O(n^2)$.

Polygon Triangulation : Chazelle's algorithm runs in $T(n) = 2T(n/2) + \sqrt{n}$.

Mergesort : Classical method that runs in
$T(n) = 2T(n/2) + O(n)$.

Why is Master's Theorem correct?