1. CSI 3105 Fall/2014
DESIGN AND ANALYSIS OF ALGORITHMS
Author: Sylvia Boyd

SECTION 1: INTRODUCTION

Why do we need to analyze algorithms?

Choosing an algorithm: For a particular problem there may exist many different algorithms. Methods for analyzing algorithms give us a way to compare them.

Example:

Designing an algorithm: Analyzing algorithms gives us insight into how to design "good" algorithms, or improve a given algorithm.

What does it mean to say an algorithm is good?

For the purposes of this course, when we say an algorithm is "good" we will mean

Comparing Algorithms in Terms of Time (and Space) Requirements

Suppose we have two different algorithms for scheduling exams--call them algorithms A and B. I implement A and run it for the Science Faculty exams, and it takes 4 hours and 30 minutes to run. I implement B and run it for the Faculty of Engineering exams, and it takes 2 hours. Which is the better algorithm in terms of the time required to solve a problem?

The time and space required by an algorithm depends on three things:

1. The size of the input.

2. How good or bad the implementation is, and the machine used for that implementation.

3. The particular input of fixed size used.

Example: Algorithm 1.3 Exchange Sort (Page 7 in textbook)

How do we deal with the three above problems in order to fairly analyze and compare algorithms?

1. Dependence on size of input: For a problem, we will choose a reasonable measure for the size of the input, and fix that size for the analysis.

   Example: For exchange sort, what would be a reasonable measure for the size of the input?

2. Dependence on implementation and machine: Given a problem and a fixed input size n, we will find expressions in terms of n which are likely to **predict** the time (and space) which will be required by an algorithm, without actually implementing the algorithm.

   Why would we want to avoid implementing two algorithms in order to compare them?

   To predict time usage, we will use a good approximation of the number of **elementary** or **basic** steps which the algorithm performs.

   Some examples of elementary or basic steps:

3. Dependency on type of input : Even if two inputs are the same size, the algorithm may perform differently on them. We use two types of analysis in dealing with this problem in order to compare algorithms.

   Worst-case analysis: Compute the maximum number of basic operations performed by the algorithm for any input of a fixed size.

   Average-case analysis: Compute the number of basic operations performed for each input of a fixed size, and then take the average.

   Example: Algorithm 1.2 Add Array Members (Page 7 in text)

SECTION 2: EFFICIENCY, ANALYSIS AND ORDER

Worst-Case Analysis of an Algorithm In Terms of Time

Most often we describe the behavior of an algorithm by stating its performance in the worst case. Given an algorithm A and a fixed input size n, let D_n represent the set of all inputs of size n for the problem under consideration. Let t(I) represent the number of basic operations performed by algorithm A on input I. We will denote the worst-case performance of algorithm A on inputs of size n by $W_A(n)$, where

$$W_A(n) = \max \{t(I) \text{ such that I is in } D_n\}.$$

$W_A(n)$ gives the maximum number of basic operations performed by algorithm A on any input of size n. Thus if $W_A(n) = k$, then for some input of size n, algorithm A requires k basic steps, and for all other inputs the algorithm requires <= k basic steps. Thus $W_A(n)$ provides an upper bound on the number of basic steps performed by the algorithm. Note that if it is obvious what algorithm we are talking about (which is most of the time!) we will simply use W(n) instead of $W_A(n)$. Also we usually choose to just count one type of basic operation rather than every basic operation, where this one basic operation accurately reflects the amount of work done by the algorithm.

Examples for worst-case analysis:

Problem: Look for an item in a list (Example 1.2 in text, Page 3).
Algorithm 1.1 Sequential Search (Page 4)
Algorithm 1.5 Binary Search (Page 10)
Algorithm 1.3 Exchange Sort (Page 7)
Table 1.1 Comparing Algorithms 1.1 and 1.5 (Page 11)
Algorithm 1.6 Nth Fibonacci Number (Recursive) (Page 12)
Algorithm 1.7 Nth Fibonacci Number (Iterative) (Page 15)
Table 1.2 Comparing 1.6 and 1.7 (Page 16)
Algorithm 2.3 Merging Two Sorted Arrays (Page 55)
Algorithm 1.4 Matrix Multiplication (Page 8)
Travelling Salesman Problem (TSP) Algorithm (not in text in this form)
Algorithm 2.1 Binary Search (Recursive)) (Page 49)

NOTE: Whenever you do a worst case analysis, you must state:

1) Input Size:
2) Basic Operations Counted:
3) Worst Case Input:

Problem Example 1.2 (from text) Determine whether the number x is in the list S of n numbers. The answer is yes if x is in S and no if it is not.

What algorithm would you use?

Algorithm 1.1 in text: Sequential Search

Worst Case Analysis of Sequential Search

Assumption:

Input size:

Basic operation counted:

Inputs that give worst case:

Analysis:

Algorithm 1.5 Binary Search (non-recursive)

NOTE: 1) Data in S must already be sorted for this algorithm to work..
2) Data must be in an array.

Algorithm in words:

Small Example: n = 7, x = 10

S = [5, 7, 10, 11, 19, 24, 30]

Question: How does the algorithm recognize the search range is empty?

Worst case analysis of Binary Search (non-recursive)

Input size:

Basic operation counted:

Input that gives worst case:

Analysis:

**Assumption:

Look at Table 1.1 in text, comparing Sequential Search and Binary Search:

Algorithm 1.3 Exchange Sort

Worst case Analysis of Exchange Sort

Input size:

Basic operation counted:

Input that gives worst case:

Analysis:

NOTE:

You should read Appendix A

Algorithm 1.6 Finding the nth Fibonacci number

NOTE: This is a recursive algorithm (uses the Divide and Conquer strategy)

The series f_n:

Problem: Given an integer n>=0, find f_n.

Recursive call tree for recursive algorithms:

- one node for every call to algorithm (including the initial call)
- each node is labelled with the name of the algorithm and the value being called
- leaf nodes are base cases
- root node is initial call

Recursive call tree for Algorithm 1.6, n=5:

Total number of calls in tree:

Let T(n) = total number of calls of algorithm required for value n.

Some values of T(n):

We won't find an exact formula, instead we will show

$$T(n) > 2^{n/2} \qquad \text{for } n \geq 2 \qquad \circledast$$

Proof of $\circledast$: By induction.

First Step: Prove $\circledast$ is true for base case(s).

Second Step: Induction Hypothesis (I.H.)

Third Step: Induction Step

Worst case analysis of Algorithm 1.6 Fibonacci numbers

Input size:

Basic operation counted:

Input that gives worst case:

Analysis:

Looking at Table 1.2 in text:

Conclusion:

Algorithm 1.7 Fibonacci numbers (iterative)

NOTE: This is an example of a dynamic programming algorithm

Algorithm in words:

Worst case analysis of Algorithm 1.7 Fibonacci numbers

Input size:

Basic operation counted:

Inputs that give worst case:

Analysis:

Algorithm 2.3 Merge

Problem**:**

Small example: U = [1, 5, 7] h = 3
V = [2, 3] m = 2

Find S =

Example to illustrate algorithm:

I give you two piles of exams, each in sorted order. Put them into one sorted pile. How would you do this?

Notes on algorithm in text:

Small example: h = 8, m = 5

U = [2, 7, 9, 12, 13, 19, 21, 27]

V = [1, 3, 4, 15, 16]

Worst case analysis of algorithm 2.3 Merge

Input size:

Basic operation counted:

Input that gives the worst case:

Analysis:

Best case?

Algorithm 1.4 Matrix multiplication

Problem:

Example:

Worst case analysis of Algorithm 1.4 Matrix multiplication

Input size:

Basic operation counted:

Input that gives worst case:

Analysis

Algorithm for the Travelling Salesman Problem (TSP)

Problem:
Given a weighted complete graph on n nodes, find a minimum weight tour (a tour is a Hamiltonian cycle, i.e. a cycle which visits each node exactly once).

Example: For n = 4

Algorithm: TSP

Input: An integer $n \geq 3$ and an nxn edge-weight matrix C of non-netative weights.
Output: A minimum weight tour.

```
Min:= +∞
For all distince cyclic permutations π of {1,2,3,…,n} do:
    Cost:= the weight of the tour associated with π
    If Cost < Min then
        Min:= Cost
        Besttour:= π
    End{if}
End{for}
```

Worst case analysis TSP algorithm

Input size:

Basic operation counted:

1.

Input that gives worst case:

Analysis:

Algorithm 2.1 Binary Search (recursive version)

Discussion:

Worst case analysis of Algorithm 2.1 Binary Search

Input size:

Basic operation counted:

Input that gives worst case:

Analysis:

First few values of W(n):

Guess at formula:

Now prove it:

Average Case Analysis of an Algorithm In Terms of Time

In average case analysis, we find the average amount of time required by an algorithm for an input of size n (this is denoted by A(n)).

What would be the advantages/disadvantages of this type of analysis over worst case analysis?

How to find A(n)

Let Dn be the set of all inputs of size n for the problem.
Let p(I) be the probability that input I occurs.
Let t(I) be the time required for input I.

Then A(n) =

Aside: What should the sum of all the p(I)'s be?

Does it make sense to consider every possible input?

Usually, instead of considering each input separately, we partition the set D_n into sets of inputs I_1, I_2, I_3,…, I_k in such a way that for each set I_i, t(I) is the same for all inputs $I \in I_i$. We then let $p(I_i)$ be the probability that any input of type I_i occurs, and we let $t(I_i)$ be the time required for any input in I_i . Then we have

$$A(n) =$$

Examples for average case analysis:

Example 1: Find the ball under the cup

Average case analysis for Example 1:

Input Size:
Basic Operation Counted:

Analysis:

Example 2: Algorithm 1.1 Sequential Search (Page 4)

Recall the algorithm:

Input Size:
Basic Operation Counted:

Average Case Analysis:

To start: We will assume x is one of the numbers in the list S.
We will also assume that all numbers in S are distinct.

Note: D_n is infinite in size, so we will need to partition it. How?

ORDER: A Classification of Algorithms into Complexity Classes (covered in Chapt. 1 in textbk)

1. Suppose that we have two algorithms A1 and A2 such that $W_{A1}(n) = 2n$ and $W_{A2}(n) = 4.5n$. Which is better (i.e. faster)?

Because our counting is imprecise, we really should consider two functions f(n) and g(n) which differ by a constant factor as being in the same complexity class, i.e. if g(n) = c*f(n) for some constant c>0 then g(n) and f(n) should be considered to be in the same class.

2. Suppose that we have two algorithms A1 and A2 such that $W_{A1}(n) = 2000n$ and $W_{A2}(n) = 2n^2$. Which is better?

For n small:

For n large: Eventually, once n gets large enough, A1 is always faster than A2, i.e. for some number n_o, we have that $W_{A1}(n) <= W_{A2}(n)$ whenever
$n \geq n_o$.

What is n_o in this case?

Examples 1 and 2 above suggest that it would be helpful to have a way to classify functions that ignores constant factors and small input sizes, i.e. we want to talk about the complexity (running time) of an algorithm as being "roughly proportional" to some function.

Usual Complexity Classes
(Note: $\Theta(g(n))$ is read "order g(n)". Also note that your text uses a slightly different notation.)

$\Theta(1)$ Constant time.

$\Theta(\lg n)$ Logarithmic time.

$\Theta(n)$ Linear time.

$\Theta(n \lg n)$

$\Theta(n^2)$ Quadratic time.

$\Theta(n^3)$ Cubic time.

$\Theta(2^n)$ Exponential time.

Some Examples: (Deciding the complexity class intuitively)

Figure 1.3 in text (Page 28): Growth rates of some common complexity functions

Table 1.4 in text (Page 29): Execution times for algorithms with the given time complexities.

A More Formal Explanation of Order

To understand order more formally, we study the asymptotic growth rate of a function. This is formalized by the "Big Oh" and "big Omega" notations.

"Big Oh"

Formal Definition: A function g(n) is said to be "Big Oh of f(n)", written $g(n) \in O(f(n))$ if and only if there exists some positive real constant c and some nonnegative integer n_o such that for all $n \geq n_o$, we have $g(n) <= c*f(n)$.

What this means: For large enough n, the growth rate of g(n) is less than or equal to the growth rate of f(n). In terms of the graphs of f(n) and g(n), g(n) is O(f(n)) if and only if there exists a constant c such that the graph of g(n) is at or beneath that of c*f(n) after a certain point n_o is reached on the horizontal axis. In this sense we can think of f(n) as being an asymptotic upper bound on a function.

Some examples from text:

Example 1.7 (Page 29)
Example 1.9 (Page 30)
Example 1.11 (Page 31)

Some Theorems Regarding "Big Oh"

1. Rule of Scaling:

   If g(n) is O(f(n)), then for any constant k > 0, g(n) is O(k*f(n)).

2. Rule of Sums:

   If $g_1(n)$ is $O(f_1(n))$ and $g_2(n)$ is $O(f_2(n))$, then $g_3(n) := g_1(n) + g_2(n)$ is $O(\max_{n \to \infty} (f_1(n), f_2(n)))$.

Complexity analysis with "Big Oh" is rarely done directly from the definitions. Instead the above theorems can often be used.

Example: Show that $g(n) = 2n^3 + 5n^2 + 3$ is $O(n^3)$.

"Big Omega"

Definition: A function g(n) is said to be "Big Omega of f(n)", written $g(n) \in \Omega(f(n))$ if and only if there exists some positive real constant c and some nonnegative integer n_o such that for all $n \geq n_o$, we have $g(n) \geq c*f(n)$.

What this means: For large enough n, the growth rate of g(n) is greater than or equal to the growth rate of f(n).

Definition of the Order of a Function

A function g(n) is "order f(n)", written $g(n) \in \Theta(f(n))$ or g(n) is $\Theta(f(n))$, if g(n) is O(f(n)) AND g(n) is $\Omega(f(n))$.

This means that the asymptotic growth rate of g(n) is THE SAME AS that of f(n).

Note that the scale rule and sum rule apply to Θ as well.

Using a limit to Determine Order

For complicated functions, it may be necessary to use a limit to help us derive the complexity classification.

Theorem: We have the following:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \begin{cases} c & \text{implies g(n) is } \Theta(f(n)) \text{ if } c > 0 \\ 0 & \text{implies g(n) is O(f(n))} \\ \infty & \text{implies g(n) is } \Omega(f(n)) \end{cases}$$

Theorem 1.4 in text (Page 40): L'Hôpital's Rule

Some examples of determining order using limits:

The Order Classification of an Algorithm

Suppose we have an algorithm A such that $W_A(n) = g(n)$, and $g(n)$ is $\Theta(f(n))$ for some one of our complexity categories $f(n)$. We say that algorithm A is $\Theta(f(n))$ if we know of an input for which the algorithm actually requires $g(n)$ basic steps. If we only know that $W_A(n) <= g(n)$ (i.e. we don't know if there is an input for which the algorithm actually performs that badly), then we say that the algorithm is $O(f(n))$.

Complexity classification of the algorithms we have looked at so far this term:

SECTION 3: (Chapter 2 in textbook)
THE DIVIDE AND CONQUER PROBLEM SOLVING STRATEGY

Strategy:

1. DIVIDE an instance of a problem into one or more smaller instances.
2. CONQUER (i.e. solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.
3. If necessary, COMBINE the solutions to the smaller instances to obtain the solution to the original instance.

Which algorithms have we already looked at that use this strategy?

Some other examples of the divide and conquer strategy:

- Mergesort for sorting a list
- Strassen's Algorithm for matrix multiplication

Example 1 for Divide and Conquer: Mergesort

Mergesort is an algorithm used to sort an array of n keys into non-decreasing order. It has the following steps:

1. DIVIDE the array into two subarrays which are close to equal in size (about n/2).
2. CONQUER (i.e. solve) each subarray by sorting it. Unless the array is sufficiently small use recursion to do this.
3. COMBINE the solutions to the subarrays by merging them into a single sorted array, using the Merge algorithm (Algorithm 2.3).

See Example 2.2 and Figure 2.2 in text (pages 53, 54).

Algorithm 2.2 in text (page 53): Mergesort.

Recall what we got for W(n) for the Merge Algorithm 2.2:

Input Size:

Basic Operation Counted:

W(n):

Worst-Case Complexity Analysis for Mergesort:

Input Size:

Basic Operation Counted:

Worst Case = ?

Analysis:

For n = 1, W(n) = _____

For n > 1,
W(n) = time to sort U + _____ + _____

Recurrence relation we must solve: $W(n) = 2W(n/2) + n - 1$, for $n>1$,
$W(1) = 0$.

Solution: $W(n) = n\lg n - (n-1)$

Therefore Mergesort is complexity _____

Definition: An *inplace* sort is a sorting algorithm which doesn't need any extra space beyond that needed for the input.

Is Mergesort an inplace sort?

How much extra space is used up?

Example 2 for Divide and Conquer: Strassen's Matrix Multiplication Algorithm (Page 67 in text)

Recall the worst-case analysis for the "usual" matrix multiplication algorithm, Algorithm 1.4 (page 8 in text):

Input size:
Basic operation counted:
Worst case input:

W(n) = _____
(Note that this makes this algorithm impractical for large values of n.)

So for n=2, W(n) = 8, i.e. Algorithm 1.4 takes 8 multiplications to solve C = A x B, i.e.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Is there some way to solve the above using only 7 multiplications?
Yes!! Using Strassen's Algorithm (developed in 1969)

Idea of Strassen's Algorithm: Calculate 7 values $m_1, m_2, \ldots, m_7$, each one of which requires one multiplication. Then express each entry c_{ij} as additions and subtractions of these m's.

Strassen determined that, if we let

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

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Check some values:

But—what about additions and subtractions??

If we check Algorithm 1.4, we see that the number of additions is n^3. With a small change, we can improve that to $n^3 - n^2$. How?

For n = 2: Algorithm 1.4 (improved) does 4 additions, while Strassen's does _____. Is Strassen's algorithm worthwhile for n = 2?

For greater values of n: Suppose that n is a power of 2. Then we can apply Strassen's algorithm recursively (i.e. using the divide and conquer strategy) by splitting the matrix into 4 equal pieces, and treating these pieces like the entries of a 2x2 matrix.

Figure 2.4, page 68:

Using Strassen's method, we compute

$M_1 = (A_{11} + A_{22})\ (B_{11} + B_{22})$ and $M_2,\ldots,M_7$ using the formulas from before.
Next we compute $C_{11} = M_1 + M_4 - M_5 + M_7$ and C_{12}, C_{21} and C_{22}.

See Example 2.5 (page 68).

Algorithm 2.8: Strassen's Algorithm (Page 69)

Analysis of Algorithm:

Short Recap of the Divide and Conquer Strategy

Sometimes this design technique can lead to good algorithms:

Other times, it does not:

If we develop a divide and conquer strategy for a problem, how can we tell when it will lead to an impractical (too slow!) algorithm?

SECTION 4:
THE DYNAMIC PROGRAMMING PROBLEM SOLVING STRATEGY (Chapter 3 in textbook)

The dynamic programming technique for solving problems is similar to the divide and conquer technique in that an instance of a problem is divided into smaller instances. However, instead of solving subproblems top-down as in the divide and conquer technique, we solve the subproblems bottom-up, i.e. we solve small instances first, store the results in some sort of a table, and later, whenever we need a result, look it up. Thus whenever we have a divide and conquer algorithm where the recursive call tree shows the same subproblem being solved more than once, the dynamic programming technique may provide a much more efficient algorithm. The dynamic programming technique will only solve a subproblem once, but please note that it must solve every subproblem once.

The steps in the development of a dynamic programming algorithm are as follows:

1) Establish a recursive property that gives the solution to an instance of the problem.

2) Solve and store solutions for ALL instances of the problem smaller than the one you are interested in, in a bottom-up fashion. Note that the order you solve these subproblems must be part of the strategy, as you must always have all the subproblems solutions you need for your recursive formula at every stage of the algorithm.

Example 1 For Dynamic Programming: The Binomial Coefficient
(Page 92 in text)

The binomial coefficient, "n choose k", is given by the formula

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

This number represents the number of possible combinations of n objects taken k at a time.

Example:

Why not just calculate this directly?

What to try next: We could look for a recursive property, and then decided if this should be used in a divide and conquer methodology or a dynamic programming methodology.

Development of this idea:

Algorithm 3.1: A divide and conquer algorithm for binomial coefficient
(Algorithm 3.1, page 93 in text)

Problem with this algorithm: Look at the recursive call tree—how big can it get?

Using the dynamic programming approach instead:

Step 1: Establish a recursive property for problem
Step 2: Build a table containing all the values of n choose k , i.e. all i choose j for $j \leq i, i \leq n$.

What order should we calculate the values for the table?

NOTE: We will never need to calculate past the value k for a row.

Example: For n=5, k=3.

Algorithm 3.2: Dynamic programming algorithm for binomial coefficient (Algorithm 3.2, Page 95)

Worst case analysis of algorithm:

Example 2 For Dynamic Programming: Chained Matrix Multiplication (Page 107 in text)

Example:

Suppose we have three matrices, A_1, A_2, and A_3, where A_1 has dimensions 50 x 5, A_2 has dimensions 5 x 100, and A_3 has dimensions 100 x 10. We wish to calculate

$$A_1 \times A_2 \times A_3$$

Recall that if A is m x k and B is k x n, then A * B requires _____ multiplications.

Number of multiplications for $A_1 * A_2 * A_3$:

Can we change the parenthesis?

Why would we want to?

How many different ways can we parenthesize 3 matrices?

Chained Matrix Multiplication Problem

Given a chain of n matrices A_1 , A_2, .., A_n where matrix A_i has dimension d_{i-1} x d_i, for i = 1, 2, …, n, fully parenthesize the product

$$A_1 * A_2 * A_3 * \ldots * A_n$$

in a way that minimizes the number of multiplications.

A recursive solution

Warm up: Look at all of the ways to fully parenthesize for n = 4.

Generalization:

Some notation: Let M[i][j], $i \leq j$ be the minimum number of multiplications for the subchain

$$A_i * A_{i+1} * \ldots * A_j$$

and let M[i][i] be 0.

Note that we want to find:

Let P[i][j] be the value of k (split point) which gives the minimum (assuming $i \leq j$).

Base Case:

General recursive algorithm:

How many recursive calls are required for M[1][n]?

How many possible M[i][j] subproblems are there?

So a dynamic programming version of this recursive algorithm will give an efficient algorithm for this (assuming we can calculate each M[i][j] efficiently).

Build up the table for all M[i][j] values iteratively (starting from the base case).

Example

Worst case analysis of our dynamic programming algorithm for chained matrix multiplication:

Input size:

Counting:

Worst case:

Analysis:

Example 3 For Dynamic Programming: The 0-1 Knapsack Problem
(Page 177 in text)

A small example

Formal definition of the problem

Suppose there are n items to consider. Let

S = the set of items, i.e. S = {item 1, item 2, …, item n},
w_i = weight of item i,
p_i = profit of item i,
W = maximum weight the knapsack can hold,

where w_i, p_i and W are positive integers. The 0-1 knapsack problem is to determine a subset A of S which will fit in the knapsack and has maximum total profit, i.e. find a subset A of S such that

Σ (p_i: item i is in A) is maximized subject to Σ (w_i: item i is in A) $\leq$ W.

Applications:

Dynamic programming algorithm:

Table:

Let P[i][k] = the maximum profit obtained when choosing from the first i items under the restriction that the total weight cannot exceed weight k.

What is it we want to find?

A recursive property for the problem:

Now consider finding P[i][k] recursively. We have two possible cases—either the optimal packing which gives P[i][k] uses item i or it doesn't.

If the optimal packing uses item i:

If the optimal packing doesn't use item i:

Putting it all together:

Base Case:

Recursive formula:

Try our example:

Item #	**1**	**2**	**3**	**4**	**5**
$\mathbf{p_i}$	10	5	5	10	3
$\mathbf{w_i}$	5	5	2	2	2

W = _____ n= _____

Table will be _____ rows, and _____ columns.

The entry [i][k] in the table will be P[i][k].

k / i	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											
5											

For each table entry we put
I--if we included item i to obtain the profit P[i][k]
N--if we did not include item i to obtain the profit P[i][k]

To get the list of items which give the profit P[n][W]:

We backtrack in the table, using the N's and the I's to tell us where to go next.

If N: Go to the entry directly above.

If I: Go to the entry in the row above, and w_i entries back.

For every row we have circled one entry, which has an I if that item should be included, and an N otherwise.

Final solution for our example:

Question: What is the smallest sized knapsack for which we can obtain this profit?

Worst case analysis of algorithm:

Input size:

Basic operation:

Worst case:

Analysis:

ASIDE:

Practical Practice Problems: For the following problems, design the most efficient algorithms you can. You can make use of any of the algorithms learned so far.

1. You are given a pile of thousands of telephone bills and thousands of cheques sent in to pay the bills. Assume that the telephone numbers are on the cheques. Find out who did not pay.

2. You are given an array A of numbers. Remove all the duplicates from A.

3. Find the nth smallest number in a list of 2n distinct numbers.

4. You are given all of the book checkout cards used in the campus library during the past year. Determine how many distinct people checked out at least one book.

5. You are given two arrays of numbers, A and B. The numbers in each array are distinct. Find the intersection of A and B.

A SIDE TOPIC: GRAPH TERMINOLOGY

A *graph* G = (V,E) is defined as a finite set of *nodes* (called vertices in your text) V which are interconnected by a finite set of *edges* E. A graph is a very useful tool which can be used to model many different problems.

Note: We will assume no loops, and no multiple edges.

Each edge e in E corresponds to two nodes in V, called the *ends* of e. An edge e in E with ends u and v can be denoted by uv. Two nodes u and v are said to be *adjacent* if uv is in E, and e is said to be *incident* with u and v.

The *neighbours* of a node v are all the nodes in V which are adjacent to v in G, and the *degree* of a node v is the number of neighbours of v.

Fact: $\sum (d(v): v \in V) = 2 * (\text{number of edges})$
Why?

A *weighted graph* is one in which every edge is assigned a number called its weight or cost.

A *subgraph* of a graph G = (V, E) is a graph G' = (V', E') such that V' is a subset of V, and E' is a subset of E. A subgraph is called *spanning* if V' = V.

A graph is *complete* if there is an edge between every pair of nodes.

A *path* from node v_0 to node v_k in a graph is a sequence of edges v_0v_1, v_1v_2, …, $v_{k-1}v_k$, where $v_0,v_1,\ldots,v_k$ are all distinct. Such a path can also be represented by v_0, v_1, v_2, …, v_k, and has length k. A single node is a path of length 0. A *cycle* in a graph is like a path except $v_0 = v_k$.

A graph is *connected* if there is a path between every pair of nodes. A *component* of a graph is a maximal connected subgraph.

A graph is called *acyclic* if it contains no cycles. A *forest* is an acyclic graph. A *tree* is a connected forest.

A *directed graph*, or *digraph*, is a graph in which all of the edges have been given a direction. In a digraph, an edge is represented by an ordered pair <u, v> where the edge (also called an *arc*) is directed from u to v. The *indegree* of a node v is the number of edges directed into v, and the *outdegree* of v is the number of edges directed out of v.

Different Ways to Represent Graphs and Digraphs

1. A list of edges
   Example:

2. Adjacency Matrix
   Suppose G has n nodes. The adjacency matrix for G is an n x n matrix A such that A[i,j]= 1 if edge v_iv_j (or $<v_i, v_j>$, in the case of a digraph) is in E, and is 0 otherwise. If G is a weighted graph, we can use the weight on the edge instead of 1 to indicate that the edge exists, and use a very large number (infinity) if the edge does not exist.

Examples:

3. Adjacency List

   This is a data structure which contains, for every node v, a linked list indicating which nodes are adjacent to v. Each node in the linked list has a node field and a link field. We can use a pointer array for the list heads.

   Examples:

Comparison of Efficiency Using Different Graph Storage Techniques

Note: We will use n to denote the number of nodes in G, and m to denote the number of edges in G. So in analysis of algorithms concerning graphs, there are two input sizes to consider (and we often include both in our complexity analysis). Sometimes in order to compare two algorithms, we need to use the fact that m is at most_____.

1. Given a pair of nodes v_i, v_j, check to see if v_iv_j is an edge in G.

(example continued)

2/ Given a node u, find any edge uv incident with u (i.e. find a neighbour of u, or determine there is none).

3/ Given a weighted graph, find the edge of maximum weight.

(Example continued)

SECTION 5: (Chapt. 4 in textbook)
THE GREEDY DESIGN TECHNIQUE FOR ALGORITHMS

Greedy algorithms repeatedly select data items, each time taking the one that is deemed "best" at that particular moment according to some criterion, without regard for the choices it has made before , or will make in the future (so it is making a "greedy" choice each time). Said another way, greedy algorithms make a sequence of locally optimal choices, in the hopes of obtaining a globally optimal solution.

NOTE: The greedy technique does not always result in a correct algorithm for a problem (you cannot assume that an algorithm designed this way works—you would need to prove it). To prove such an algorithm doesn't work, you .need to provide an example of the problem where the algorithm fails to find the optimal (i.e. best) solution.

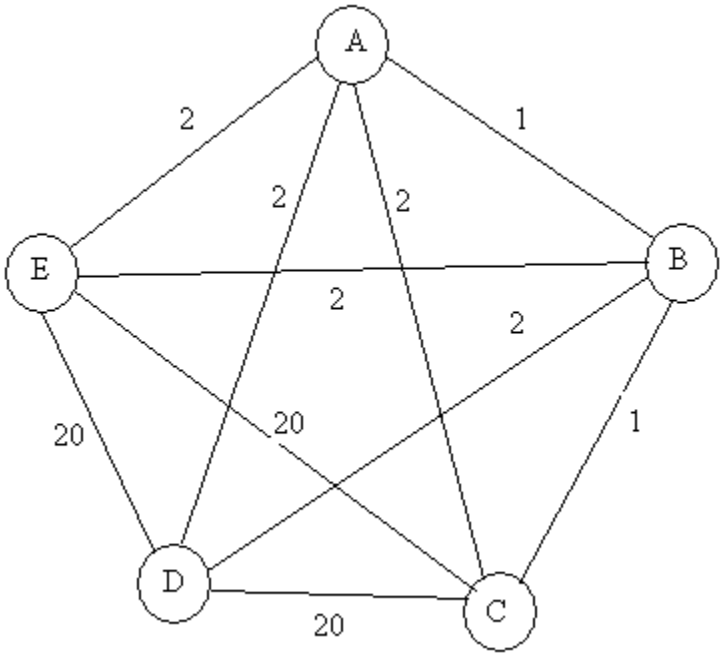
Example 1: A greedy algorithm for the Travelling Salesperson Problem (TSP)—Does it work?

Nearest Neighbour Algorithm:

1) Pick a starting city
2) **Repeat**
   Pick as the next city the closest city to where you are now
   which has not yet been visited
   Until all the cities have been visited
3) Join the last city to the first city to complete the tour

Example: (For this example, the nodes are point in the plane, and the edge distances are the actual Euclidean distances)

Another Example:

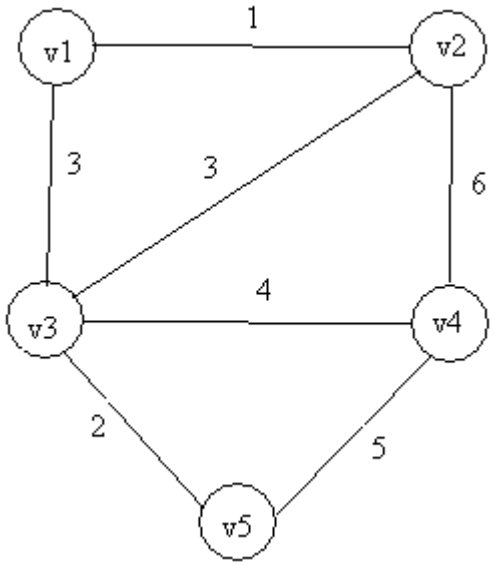


We can see from both examples that the Nearest Neighbour algorithm finds a tour for the TSP, but not necessarily an optimal (i.e. minimum cost/weight) tour. So this algorithm doesn't work, in general, to solve the TSP.

Example 2 For the Greedy Technique: Minimum Spanning Tree (pg 140)

Application:
Suppose that you are setting up communications centers. For every pair of communication centers, you are given the cost of laying the cable to link the two centers. You are asked to decide where to lay the cables (i.e. between which centers) such that you can ensure that every center can communicate with every other center. Moreover, you wish to do this as cheaply as possible.

Graphical Representation:



Solution—What will it look like on the graph?

If we model our communication problem graphically, then we can solve it by solving the following problem.

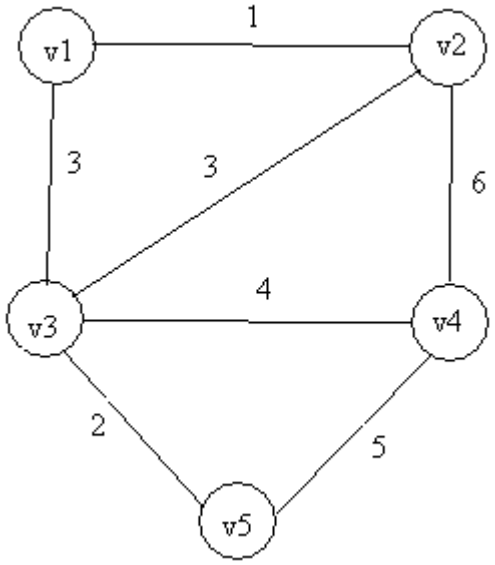
Minimum Spanning Tree Problem (MST)
Given a weighted connected graph G = (V,E) (note that G is undirected), find a minimum cost spanning tree of G.

A Solution Using the Greedy Technique: Kruscal's Algorithm (1957)

To start: Include all the nodes and none of the edges.

While the subgraph is not connected:
Pick the next cheapest edge that doesn't create a cycle.

Example:



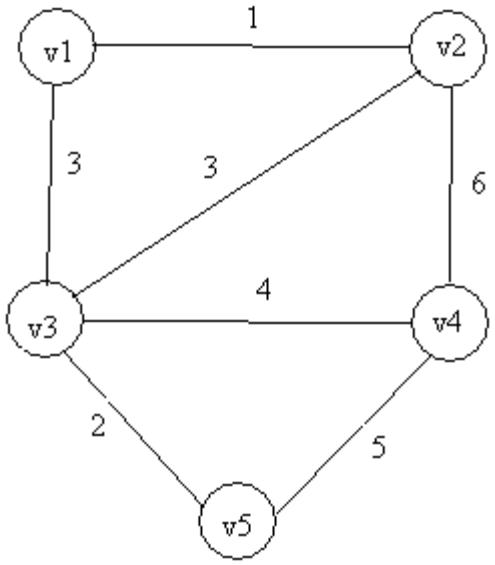
Another Solution Using the Greedy Technique: Prim's Algorithm (1957)

We will build a subgraph H=(Y,F) of G, which is a tree at every stage of the algorithm.

Initially, let F = empty set and let Y={v} for some node v.

While H is not spanning do:
    Pick the cheapest edge e you can add to F and still have H is a tree
        (i.e. pick the cheapest edge you can which has one end u in Y,
        one end w not in Y)
    Add e to F and w to Y
{endwhile}

Example:



Discussion about the Implementation of the Algorithm

Let the nodes for G be $v_1, v_2, \ldots, v_n$, and let v_1 be the initial node in Y.
We will store G as an adjacency matrix W:

W[i][j] =

For our example:

We will also maintain two arrays indexed by the nodes, called *nearest* and *distance*, where for each node v_i <u>not</u> in Y:

nearest[i] = index of the node in Y nearest to v_i

distance[i] = weight on the edge between v_i and the node indexed by nearest [i]

Note that distance[i] = -1 is used to indicate that node v_i is in Y (i.e. is part of the current tree). To be more efficient, after we add a node v_k to Y we will update the labels nearest[i] and distance[i] for all nodes not in Y by checking if the new node v_k is closer to v_i than what we had before. (If we did not use this update idea, our algorithm would end up being $O(n^3)$.)

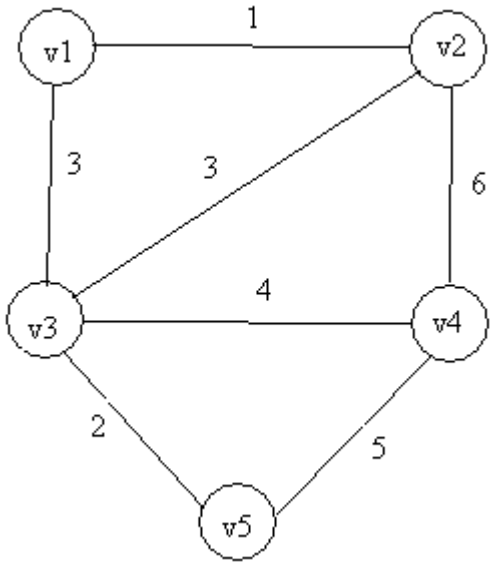
Note on updating the labels for distance, nearest arrays:

Simplest way to update:

Improvement:

Algorithm 4.1 (page 144): Prim's algorithm

Example:

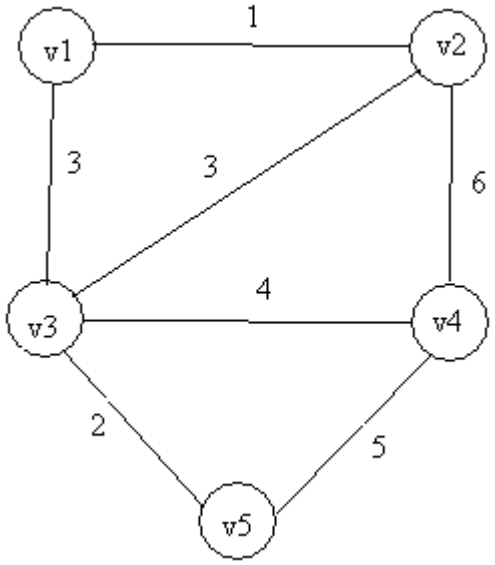


Worst case analysis of algorithm:

Proof that Prim's algorithm works: Page 149, 150 in text.

At every stage of the algorithm:

Definition:
A set of edges F that form a tree is called *promising* if there is some MST T that contains F.



Lemma 4.1 (textbook)

Let F be a promising set of edges. Let e be an edge that would get added to F using Prim's idea (i.e. the cheapest edge with one end in the nodeset for F, one end not in that nodeset). Then $F \cup \{e\}$ is also a promising set of edges.

Proof of Lemma:

Theorem 4.1: Prim's algorithm always produces a minimum spanning tree.

Example 2 For the Greedy Technique: The Stable Marriage Problem (not in textbook)

A problem that often arises at the co-op office is to "pair up" students with employers according to preference relationships that are likely to conflict. Each student lists several employers in order of preference, and each employer lists several students in order of preference. The problem is to assign students to employers in a fair way, respecting all the stated preferences as much as possible. A sophisticated algorithm is required because the best students are likely to be preferred by several employers, and the best employers are likely to be preferred by several students.

This problem can be solved as an instance of the Stable Marriage Problem, which can be modelled as follows: We assume we have N men and N women. Each person lists, in order of preference, all the people of the opposite sex. We wish to find a set of N marriages which respects everyone's preferences.

Example:

(2, 5, 1, 3, 4)	A	1	(E, A, D, B, C)
(1, 2, 3, 4, 5)	B	2	(D, E, B, A, C)
(2, 3, 5, 4, 1)	C	3	(A, D, B, C, E)
(1, 3, 2, 4, 5)	D	4	(C, B, D, A, E)
(5, 3, 2, 1, 4)	E	5	(D, B, C, E, A)

A set of marriages is called *unstable* if two people who are not married both prefer each other to their spouses.

Example: Consider the set of marriages A1, B3, C2, D4, E5 for the above example. This is an unstable set, because it is likely that _____ would leave _____ for _____, and _____ would leave _____ for _____.

We will look at an algorithm which finds a stable set of N marriages. Moreover, for one of the two sets (either the men or the women), this set of marriages will be "optimal" in the sense that no other stable configuration will give any person from that set a better choice from her (or his) list.

Algorithm

Each woman, in turn, becomes the "suitor" and seeks a husband. The suitor first proposes to the first man on her list. If he is already engaged to a woman whom he prefers to the suitor, he turns her down. The suitor then must try the next man on her list, continuing until she finds a man who is not engaged, or who prefers her to his current fiancé. This man accepts the suitor's proposal and they become engaged. If the man was not previously engaged, then the next woman becomes the next suitor, otherwise the "jilted" woman becomes the suitor once again, continuing along her preference list from where she left off.

Example:

(2, 5, 1, 3, 4)	A	1	(E, A, D, B, C)
(1, 2, 3, 4, 5)	B	2	(D, E, B, A, C)
(2, 3, 5, 4, 1)	C	3	(A, D, B, C, E)
(1, 3, 2, 4, 5)	D	4	(C, B, D, A, E)
(5, 3, 2, 1, 4)	E	5	(D, B, C, E, A)

(2, 5, 1, 3, 4)	A	1	(E, A, D, B, C)
(1, 2, 3, 4, 5)	B	2	(D, E, B, A, C)
(2, 3, 5, 4, 1)	C	3	(A, D, B, C, E)
(1, 3, 2, 4, 5)	D	4	(C, B, D, A, E)
(5, 3, 2, 1, 4)	E	5	(D, B, C, E, A)

Analysis of Algorithm**:**

Input size:

Basic operation counted:

Input that gives worst case:

Analysis:

Consider the number of times a woman proposes and asks “will you marry me?”.

Quick example for our worst case input:

Amount of work required for each proposal:

1) Deciding which woman asks next:

2) Deciding which man she asks:

3) The man being asked decides his answer (yes/no):

Total complexity:

W(N) =

Question: How do we know that the algorithm terminates?

Question: How do we know the set of marriages formed by the algorithm is a stable set?

Aside: Biases of the algorithm

The algorithm will be "optimal" for the women but not for the men. (If we started with the men as suitors, the marriages are 1E, 2D, 3Ad, 4C, 5B).

It might seem that the order in which the women become suitors matters--it turns out that this order doesn't matter at all--the same stable set of marriages result from any ordering.

Some variations:

Example 4 For the Greedy Technique: Scheduling With Deadlines
(page 159)

Suppose that you have a set of jobs to process on a single machine. Each job requires one unit of time on the machine, and each has an associated profit p and deadline d. The profit p is the amount you make if that job is scheduled to start on the machine on or before time period d. The time periods are numbered 1, 2, … (i.e. there is no time period 0). Schedule the jobs so as to maximize profit.

Example 1

Suppose we work in a lab where we test soil samples for different companies. We have a machine which analyses the sample for toxic substances, and it takes a day to process a sample. In order to be accurate, these samples must be analyzed within a certain time frame, so each has a deadline for processing. Currently we have 4 samples waiting for processing:

Job	Deadline (day)	Profit
1	2	30
2	1	35
3	2	25
4	1	40

All possible schedules:

Day	1	2	3	Profit
Schedule 1:	1	3		55
Schedule 2:	2	1		65
Schedule 3:	2	3		60
Schedule 4:	3	1		55
Schedule 5:	4	1		70
Schedule 6:	4	3		65

One solution strategy: Use brute force, i.e. try all the possible schedules, and find the one with the maximum profit.

What is wrong with this strategy?

Solution using the greedy technique:

Suppose I already have a partial schedule (underlines indicate time periods, in order):

Job K _____ Job J _____ _____ Job L

Which job do we schedule next? (Hint—do the greedy thing!)

Where do we schedule the next job?

Note: If that next job can't be scheduled before its deadline, go to the next biggest profit job.

Another example:

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

Another example:

Job	Deadline	Profit
1	4	40
2	4	15
3	3	60
4	1	20
5	3	10
6	1	45
7	1	55
8	3	15

Some details of the algorithm implementation:

Does the greedy algorithm always work for this problem?

Worst case analysis of algorithm:

Traversing graphs and digraphs

1. Depth First Search (DFS)

Input: G =(V,E), a graph or digraph represented by an adjacency list structure with V = {1,2,...,n}; v∈V, the vertex from which the search begins.

Comment: For a stack S, we assume that the function call Top(S) returns the value of the top item on S (without popping it).

```
procedure DepthFirstSearch (AdjacencyList: HeaderList; v: VertexType);
var
   S: Stack;
   w: VertexType;

begin
   initialize S to be empty;
   visit, mark, and stack v;
   while S is nonempty do
      while there is an unmarked vertex w adjacent to Top(S) do
         visit, mark, and stack w
      end {while there is an unmarked...}
      Pop(S)
   end {while S is nonempty}
end {DepthFirstSearch}
```

2. Breadth First Search (BFS)

Comment: For a queue Q, we assume that the function call RemoveFromQ(Q) returns the value of the front item on Q and removes that item from Q.

```
procedure BreadthFirstSearch(AdjacencyList: HeaderList; v:VertexType);
var
   Q:  Queue;
   w:  VertexType;
begin
   initialize Q to be empty;
   visit and mark v; insert v in Q;
   while Q is nonempty do
      x:= RemoveFromQ(Q);
      for each unmarked vertex w adjacent to x do
         visit and mark w;
         insert w in Q;
      end {for}
   end {while}
end {BreadthFirstSearch}
```

SECTION 6: Computational Complexity of a Problem: An Introduction to the Theory of NP and NP Completeness (Chapt 7 in text)

Up until now, we have been looking at the complexity of an algorithm for a problem. Now we will consider the complexity of a problem--is it an easy or hard problem? Are we likely to find a "good" (i.e. efficient) algorithm for it? If we have an algorithm, how do we know we have the most efficient algorithm for the problem (should we look for a better one, or can we somehow show that there isn't a more efficient algorithm for this problem?).

Lower Bounds on the Complexity of a Problem

Given a problem P, we say that an algorithm A is *optimal* (in the worst case) for P if we know that it is impossible to find another algorithm for P that performs fewer basic operations (in the worst case). Here we do not mean that A is optimal if it is the best *known* algorithm for P.

Why would we want to know if an algorithm is optimal for a problem?

How do we show an algorithm is optimal for a problem?

We establish a *lower bound* k on the number of operations needed to solve a problem P (in the worst case). We say that k is a lower bound for a problem P of input size n if

$$W_A(n) \geq k$$

for ANY algorithm A which solves problem P. This means that if we have an algorithm A* which solves P such that $W_{A^*}(n) = k$, then we know that A* is an optimal (i.e. the best possible) algorithm for P.

Examples

Example 1. Matrix Multiplication

The best know algorithm for matrix multiplication is the one by Coppersmith and Winograd, '87, which has $W(n) = \Theta(n^{2.376})$ (counting multiplications.

Someone has also established a lower bound for this problem (but we will not go over the proof). This lower bound is $\Theta(n^2)$, i.e. it has been proven that any algorithm for this problem must do at least $\Theta(n^2)$ multiplications in the worst case.

What does this mean?

It may be that there is a way to solve the problem in $\Theta(n^2)$ multiplications, but no one has found it yet. Or, there may be some more clever argument which shows that $\Theta(n^{2.376})$ is a lower bound for this problem, which would then show that Coppersmith and Winograd's algorithm is optimal. OR.....

Example 2: Sorting

We will look at the problem of sorting an array of n "keys" (i.e. numbers) using any algorithm which uses comparisons of keys.

Some examples of algorithms for this problem:

Can we find an algorithm that is more efficient than $\Theta(n \lg n)$? To prove that this is optimal for this problem, we would need to prove that any sorting algorithm must use at least this many comparisons. First we need some background material.

Decision Trees

With any algorithm that sorts by using comparisons, we can associate a *decision tree*. In a decision tree, the non-leaf nodes represent a comparison of two of the keys, i.e. $a < b$? The tree is a binary tree (so every node in the tree has at most 2 children) but it is not necessarily a complete binary tree. The left branch gives the next comparison if $a < b$ is true. The right branch gives the next comparison if $a < b$ is false. The leaf nodes give the final sorted permutations.

Example: Algorithm Sortthree (page 297) for sorting three numbers
S = (a, b, c), and the corresponding decision tree, page 299.

Given an algorithm and its decision tree, any particular input maps to a root-to-leaf path in the tree.

For example, try S = (5, 7, 2) in Algorithm Sortthree.

How many comparisons are required for the input S = (5, 7, 2) in Sortthree?

Given the decision tree for a sorting algorithm, how can we find the maximum number of comparisons required by the algorithm (over all inputs)? -- Note that this gives us W(n)

Definition: The *depth* of a tree is the number of edges in a longest root to leaf path.

Another example of a decision tree: Exchange Sort (Algorithm 1.3 in text, pg. 7). Draw the tree for n=3, i.e. S = (a, b, c)

Recall the order of the indices i, j for the algorithm:

Decision Tree (Page 300 in text)

Finding a Lower Bound For Sorting Algorithms Which Use Comparison of Keys

We would like to show that $W_A(n) \geq \Theta(n \lg n)$ for any algorithm A for sorting n keys which uses comparison of keys.

First two facts about general binary trees.

Fact 1: $2^d \geq k$, where d is the depth of the tree, and k is the number of leaves.

Proof:

Fact 2: $d \geq \lceil \lg k \rceil$

Proof:

Obtaining a Lower Bound:

Let A be any sorting algorithm and let T be the decision tree for A for input size n. We know that

$W_A(n) =$

Since T is a binary tree, we can apply Fact 2, and thus

$W_A(n) =$ (*)

where t is the number of leaves in T.

How many leaf nodes are there in T?

T must have at least 1 leaf node for every permutation of the keys $x_1, x_2, ..., x_n$.

Why?

Thus t is greater than or equal to the number of permutations of $x_1, x_2, ..., x_n$, which is _____.

Applying this to (*) gives

$$W_A(n) \geq \lceil \lg (n!) \rceil$$

and thus $\lceil \lg (n!) \rceil$ is a lower bound for the worst case performance of ANY sorting algorithm.

To finish the proof: An estimate for $\lceil \lg (n!) \rceil$:

The Theory of P, NP and NP Complete (Chapter 9 in textbook)

An algorithm is said to be *polynomial* (or is said to run in polynomial-time) if it has complexity $\leq \Theta(p(n))$, where $p(n)$ is some polynomial in the size of the input n.

Examples:

The set of all problems for which there exists a polynomial time algorithm is denoted by **P**.

Examples:

A problem for which we know a polynomial algorithm is called *tractable*.

There are many problems for which no polynomial-time algorithm is known. For some of these problems, polynomial algorithms do exist, but we just haven't discovered them yet (i.e. we haven't tried hard enough!!). However, for other such problems, we strongly suspect they cannot be solved in polynomial time.

NP-Complete Problems (NP stands for Nondeterministic Polynomial)

The class of NP-complete problems is a set of problems for which the following properties hold (and please note that this is NOT a definition, just properties that hold):

i) We don't know of a polynomial time algorithm for any problem in the class.

ii) There exists a polynomial-time algorithm for a problem in this class if and only if there exists a polynomial time algorithm for ALL NP-complete problems.

Why is the notion of NP-completeness important?

Some examples of NP-complete problems (and their definitions):

1) Graph Colouring
2) Knapsack Problem
3) CNF Satisfiability
4) TSP
5) Bin Packing
6) Clique

NOTE:
The theory of NP-completeness deals with the decision version of a problem, i.e. a version of the problem with a yes/no answer, rather than an optimization form. Every optimization form of a problem has a decision form you get by adding one parameter. So when we say a problem is NP-complete, we are talking about the decision form)

1. Graph Colouring

Given: A graph G=(V,E), a value C.

Definition: A k-colouring of G is a colouring of the nodes V of G with at most k colours such that adjacent nodes have different colours.

Problem (decision form):
Does there exist a k-colouring of G for some k<=C?

Aside: The optimization form of the problem: Given a graph G=(V,E), find the minimum k for which there exists a k colouring of G.

Example, and application for exam scheduling:

2. 0-1 Knapsack problem
(We have already seen the optimization form of this problem)

Given: A knapsack capacity W, a set S of n items, and for each item i a weight w_i and profit p_i. Also given is a target profit P.

Problem (decision form):
Is there a packing of the knapsack with profit >= P?

3. CNF Satisfiability Problem (SAT)

Given: A set of Boolean variables, x_i for i = 1, 2, ..., n (these have value true or false)

Definitions:
If x is a Boolean variable, then its negation $\bar{x}$ is true if and only if x is false.

A literal is a Boolean variable x_i or its negation $\bar{x}_i$

A clause is a a sequence of literals joined by OR's ($\vee$)

A logical expression is in CNF (Conjunctive Normal Form) if it is a sequence of clauses joined by AND's ($\wedge$).

Example:

Decision Problem: Is there an assignment of true (T) and false (F) values to our variables which makes the expression true?

For our example--answer?:

Another example:

If there are n variables, how many possibilities must we try?

Special cases:

K-SAT: All clauses have at most K literals.

Note:
2-SAT is in P.
3-SAT is NP-complete.

Applications:

4. TSP (Travelling Salesman Problem)
(we have already seen the optimization form)

Decision Form of the Problem: Given a value K, does there exist a tour of graph G of weight at most K?

5. Bin Packing

Given: A value K, a set of bins of capacity 1, and a set of n items with sizes $s_1, s_2, ..., s_n$ where $0<=s_i<=1$.

Problem (decision form): Can I pack all the items in at most K bins?

Example and applications:

6. Clique

Given: A graph G=(V,E) and a value K.

Definition: A clique is a subset W of the nodes V such that all pairs of nodes in W are joined by an edge in G.

Example:

Problem (Optimization form): Find the largest clique.

Problem (Decision form): Does there exist a clique of size >= K?

Example:

The Class NP (Nondeterministic Polynomial): Formal Definition

The theory of NP-completeness deals with problems in the decision form only (i.e. in a form where the answer is true or false).

Example: TSP in decision form

Given a complete weighted graph G = (V, E) and a number d, does G have a tour of weight $\leq$ d?

Instance of this problem:

Now, suppose someone CLAIMS the answer is true for the above question. It would be natural for us to want him to verify his claim. What could he provide for us in order to prove his claim?

What would we need to do to verify S?

Formal Definition of **NP**: A decision problem is in the set **NP** if, for any "yes" instance I of the problem, there exists a string S for I (called the *certificate*) and a verification algorithm A which can verify S is a yes answer to I in polynomial time.

So, for example TSP is in **NP**.

Two other examples: 1) Clique
2) MST (decision form)

Showing CLIQUE is in NP:

1) Form of the certificate S:

2) Find algorithm A to verify S (A must be polynomial time)

Show MST (minimum cost spanning tree) is in NP:

Decision form of problem:

Note: MST is in P.

Proving MST is in NP:

A slightly different version of the definition of NP:

Remember: If a problem is in the class NP, it doesn't say anything about how hard/easy it is to solve the problem, it just means it is easy to verify a yes answer.

Relationship between **P** and **NP**

Note that a problem being in NP DOES NOT mean that we can solve the problem itself in polynomial time--it simply means that we can verify a yes answer to the problem in polynomial time. However, we have the following theorem:

Theorem: $P \subseteq NP$

Proof:

Diagram of Relationship Between P and NP:

Does P = NP? $$$$$

The Class NP-Complete

The formal definition of NP-complete uses the idea of transforming or reducing one problem into another problem. This is a useful idea, even outside of the theory of NP-completeness, for reusing algorithms and code.

Suppose you have two different decision problems, A and B. You do not have an algorithm for A, but you do have an algorithm for B. If you could find a way to transform any instance x of A into an instance y of B such that the correct answer for x is yes if and only if the correct answer for y is yes, then this transformation algorithm (call it *tran*), along with the algorithm for B, provides an algorithm for problem A.

Definition: If the transformation algorithm *tran* is polynomial-time, then we say that problem A is polynomial-time reducible to problem B. We write this as $A \propto B$.

Theorem (*): If decision problem B is in **P** and $A \propto B$, then decision problem A is in **P**.

Proof:

Example:

Problem B: Given a directed weighted graph G = (V,E), a root node r, a target node v, and a number K, is there a directed path from r to v of weight <=k ?

Problem A: Given a weighted graph G' = (E', V'), a root node s, a target node t, and a number L, does there exist a path from s to t of length <= L?

We want to show A $\propto$ B.

Transformation:

Formal definition for NP-complete:

A decision problem B is ***NP-complete*** if

1. It is in **NP**, and
2. For every other problem A in **NP**, A $\propto$ B.

IMPORTANT: Note what this implies!!! If we could show any NP-complete problem is in **P**, then we could conclude that ALL problems in **NP** (and in particular, all other NP-complete problems) are in **P** by Theorem (*)).

BUT--How can we possibly show a decision problem B is NP-complete by this definition?

To show 2), it is enough to show that for some problem C which is already known to be NP-complete, we have C $\propto$ B. Since C is NP-complete, we know that for all problems K that are in **NP**, K $\propto$ C. Combining the polynomial-time transformation of K to C and C to B, it follows that K $\propto$ B for all problems K in **NP**, as required.

Summary: To show a problem B is NP-complete, do the following:

1) Show B is in NP

2) Show C $\propto$ B for some problem C which is known to be NP-complete.

Steps to show C $\propto$ B (using the transformation algorithm **tran** described):

We must show:

1) tran is a polynomial-time algorithm.

2) If the answer is yes for instance y = tran(x) for problem B, then the answer is yes for instance x of problem C.

3) If the answer is yes for instance x of problem C, then the answer is yes for instance y=tran(x) of problem B.

What about the very first time a problem was shown to be NP-complete?

Example
Recall the decision form of TSP:

Suppose I want to prove TSP is NP-complete. I would need to do the following:

1) Show that TSP $\in$ NP.
2) Pick some problem Q $\in$ NP-complete, and show Q $\propto$ TSP, which means showing:
   a) There exists a transformation algorithm TRAN from Q to TSP which is polynomial time.
   b) The answer is yes for an instance of problem Q if and only if the answer is yes for the TSP problem it is transformed into by TRAN.

Problem that we will use for Q:

Hamilton Cycle (HC): Given a graph G' = (V', E'), no weights, not necessarily complete, does there exist a cycle that goes through all the nodes in G'?

Example of HC:

So we will show HC $\propto$ TSP.

Transformation TRAN for HC $\propto$ TSP

1. To get G for TSP, take G' for HC and add in the missing edges (i.e. make it complete).

2. Edge weights: Give the edges that were in G' weight 1, and give new edges just added weight 2.

3. Let K for the TSP problem be n, the number of nodes in G'.

Examples of transformation TRAN:

Proving HC α TSP:

Consider the transformation TRAN from class. There are 3 things we must prove:

a) TRAN is a polynomial-time algorithm.

b) If the answer is yes for HAMILTON CYCLE, then the answer is yes for the TSP problem we formed

c) If the answer is yes for the TSP problem we formed, then the answer is yes for the HAMILTON CYCLE problem.

Proof of a): Given an adjacency matrix for G', it will be at most $\Theta(n^2)$ work to create the adjacency matrix for the TSP problem graph G we form. So TRAN is a polynomial-time algorithm.

Proof of b): Suppose we have a yes answer for the HAMILTON CYCLE problem, i.e. we have a cycle v1, v2, ..., vn in G' which visits all the nodes, and has n edges. This cycle is a tour in our TSP problem graph G, and it's cost is n. So we have a yes answer for the TSP problem.

Proof of c): Suppose we have a tour of cost <= n in the TSP problem we formed. Since this tour contains n edges, and each edge has weight >=1 in G, we must have that this tour has cost exactly n, and every edge in it has cost 1. This means that all the edges in the tour are edges in the HAMILTON CYCLE graph G', and thus the tour corresponds to a cycle in G' which visits all the nodes.

IMPORTANT TO REMEMBER: If we can show a problem B is NP-complete, then it doesn't mean that there exists no polynomial-time algorithm for it. It is simply a strong indication that the problem is very difficult, and that it is considered highly unlikely that we will ever find a polynomial time algorithm for B.

Another example: (example 9.9 in textbook, page 395)

We will show CLIQUE is NP-complete.

What must we show?

1) Show CLIQUE is in NP.
2) Show that for some problem B that is known to be NP-complete,
   B α CLIQUE.

We will pick SAT to be B (it is known that SAT is NP-complete).

Algorithm TRAN to transform an instance of SAT into an instance of CLIQUE: Let K = number of clauses in our instance of SAT. For each literal y in clause i, create a node (y,i) in $\underline{G}$. Join two nodes if they are in different clauses, and the literals are not x_i, x_i.

Example 1 to illustrate TRAN:

Example 2 to illustrate TRAN:

We must show three things about TRAN:

1) TRAN is polynomial time.

2) If the answer is yes for the CLIQUE problem TRAN creates, then the answer is yes for the SAT problem we started with.

3) If the answer is yes for the SAT problem we started with, then it is yes for the CLIQUE problem created by TRAN.

Techniques Used For Dealing With NP-Complete Problems

Many hundreds of important application problems are NP-complete. What can we do if we must solve one of these problems?

Approach 1:

While acknowledging the apparent inevitability of an exponential time algorithm, we seek to obtain as much improvement over straight-forward exhaustive search as possible.

Example: Some sort of branch and bound approach.

Approach 2:

We no longer focus on finding an optimal solution to our problem, but instead try to find a "good" solution within an acceptable amount of time. Algorithms that do this are called *heuristics*, and are frequently based on sensible "rules of thumb". Heuristics which guarantee to be within some fixed percentage of the optimal solution value are called *approximation algorithms*.

Example: The Nearest Neighbour heuristic for the Travelling Salesman Problem: Choose a starting city, then always choose as the next city the closest one which has not yet been visited.

Heuristic algorithms are usually evaluated through a combination of empirical studies. Some have theoretical performance guarantees, while others have no such performance guarantees and can be made to perform extremely badly on certain problems.

Using lower bounds on optimal solutions to gain information the the "goodness" of a heuristic solution:

Suppose we are working on a minimization problem, and we obtain a heuristic solution. We know that

The value of an optimal solution to our problem	$\leq$	The value of a heuristic solution to our problem

If the heuristic solution has no theoretical performance guarantee, these two values can be far apart; i.e. we have no way of knowing how good (or bad) our heuristic solution is.

Now suppose we have some way of obtaining a bound on the value of an optimal solution from below; i.e.

The value of our bound	≤	The value of an optimal solution	≤	The value of a heuristic solution

If GAP is small, this tells us:

If GAP is large, this tells us:

Example: Finding a lower bound on solutions for TSP which uses MST (assuming all costs are ≥ 0)

Let T be a minimum cost tour for our TSP problem, and let T′ be obtained from T by removing any edge.

We have that: Cost of T′ $\leq$ Cost of T (*)

Why?

Notice that T′ forms a spanning tree for our graph (but not necessarily a minimum cost spanning tree). So we have

Cost of a MST $\leq$ Cost of T′ (**)

By combining (*) and (**) we have

Cost of a MST $\leq$ Cost of an optimal TSP tour.

So the cost of a MST provides a lower bound for TSP.

Demo of software to illustrate some of these ideas for TSP: TRAVEL

TRAVEL was designed as a research/teaching tool for finding "provably good" TSP solutions.

It consists of 3 types of modules:

1) Tour construction methods (heuristics)
2) Tour improvement methods
3) Lower bound methods

It works on random problems, or you can give it a problem (plus a cost matrix).

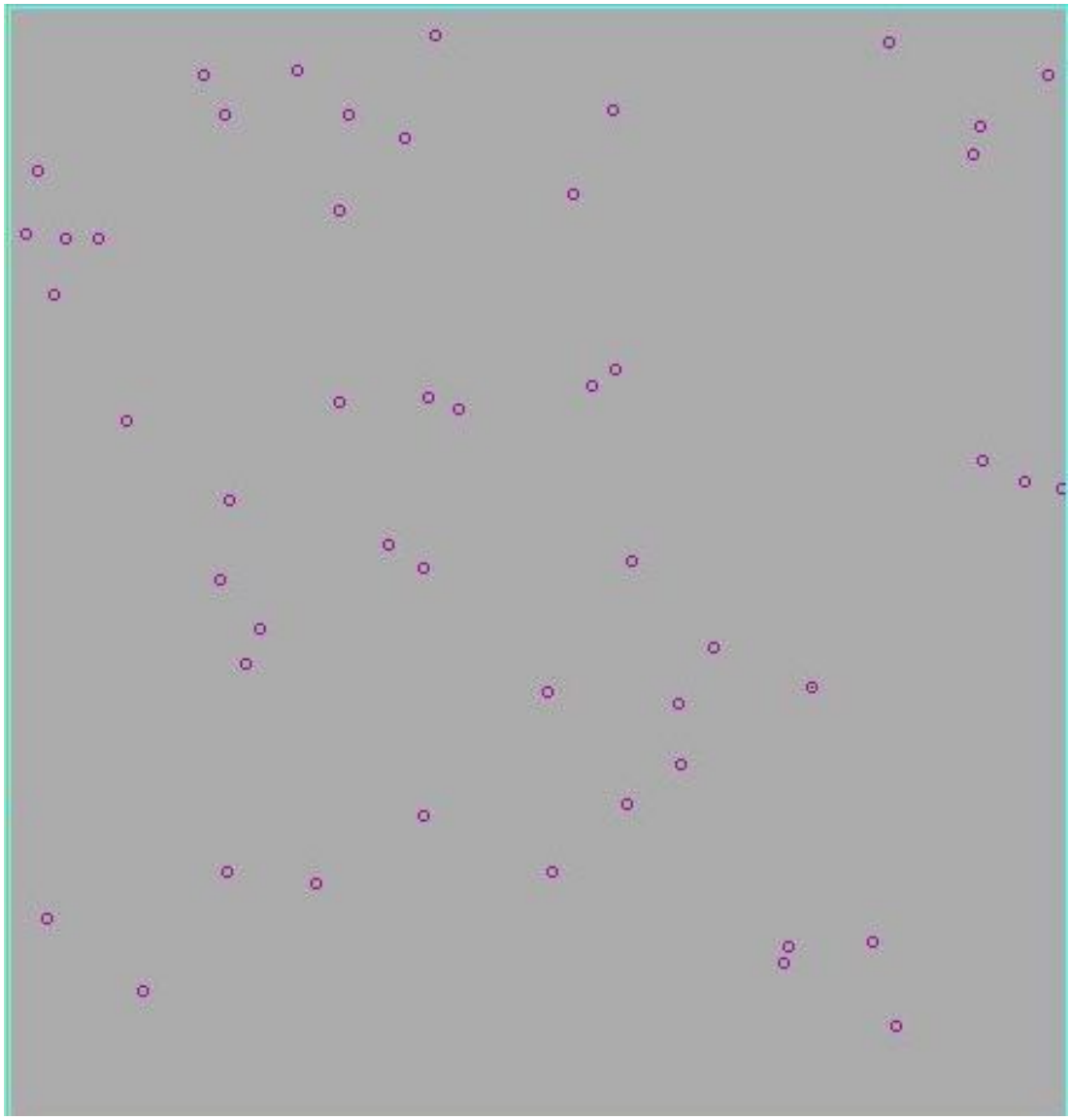
1) and 2) are used to find a "good" tour T of cost HCOST
3) is used to find a "good" lower bound of value LB

Using LB, we can calculate the "goodness" of the heuristic solution T as a percentage as follows:

$$\%GAP = \frac{(HCOST - LB)}{HCOST} * 100$$

An example of a tour improvement method: 2OPT

TSP example for you to try:



Approximation Heuristics: Performance Guarantees

Some heuristics have *performance guarantees*, which means that they are guaranteed to be within some percentage of optimal. For example, for a minimization problem we would have

$$\text{Heuristic solution value} \leq K * (\text{Optimal solution})$$

for some value given value of $K > 1$.

Such heuristics are often called *approximation algorithms*.

Example of an approximation algorithm for TSP

For the general TSP problem, no such heuristic is known, and it is considered highly unlikely we will find one (it would imply **P** = **NP**, which is considered highly unlikely).

A special case of TSP: The metric TSP

All costs satisfy the "triangle inequality property":

(Note that the metric TSP is also an NP-complete problem)

The double-the-MST-heuristic for the metric TSP:

1. Find an MST T
2. Create a "path" that visits every city by going trice around the tree T
3. Create a tour by taking the path in 2 and "shortcutting".

Example: page 408

Theorem: The tour found by the above algorithm has cost at most 2 times the cost of an optimal tour.

Proof:

Note: Best known approximation algorithm for Metric TSP:

Christofides method (1975) (gives a 3/2 approximation)