Slides for COMP 202 Fall 1996

Copyright ©1996 by David A. Plaisted

TOPICS FROM TEXTBOOK CHAPTER 1

1. Pseudocode conventions
2. Θ notation
3. Divide and Conquer
4. Analysis of D and C using recurrences

CHAPTER 2, ASYMPTOTIC NOTATION

1. Asymptotic notation Θ, O, Ω, o

CHAPTER 3, SUMMATIONS

1. Formulas for sums and products
2. Estimating sums by integrals

CHAPTER 4, RECURRENCES

1. Recursion trees

2. The iteration method

3. The master method

4. Linear recurrences (?)

THE MASTER 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 to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

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

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

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

Examples of the Master Theorem

$$T(n) = 2T(n/2) + 1$$

Here $a = b = 2$ so $n^{\log_b a} = n^{\log_2 2} = n$. Also $f(n) = 1$. Thus the first case applies, since $1 = O(n^{1-\epsilon})$ for say $\epsilon = 1/2$. Thus $T(n) = \Theta(n)$. In this case we can verify that $T(n) = n - 1$ is a solution, for example.

$$T(n) = 2T(n/2) + n$$

This for example is a recurrence for the merge sort which we know to be $\Theta(n \log(n))$. Here again $a = b = 2$ so $n^{\log_b a} = n^{\log_2 2} = n$. Also $f(n) = n$. Thus the second case applies, since $n = \Theta(n^1)$. Thus $T(n) = \Theta(n \lg n)$. This agrees with what we know about merge sort. One can also derive this result for powers of two by repeated expansion (iteration).

$$T(n) = 2T(n/2) + n^2$$

This for example is a recurrence for generating a complete graph on n vertices recursively, which is quadratic in complexity. Here again $a = b = 2$ so $n^{\log_b a} = n^{\log_2 2} = n$. Also $f(n) = n^2$. Thus the third case applies, since $n^2 = \Omega(n^{1+\epsilon})$ for say $\epsilon = 1/2$. Thus $T(n) = \Theta(n^2)$. This agrees with what we know about complete graphs. One can also derive this result by repeated expansion especially for powers of two.

CHAPTER 5, SETS, RELATIONS, ...

1. Sets

2. Relations

3. Functions

4. Graphs (vertices, edges etc.)

5. Trees

CHAPTER 6, COUNTING AND PROBABILITY

1. Permutations and combinations
2. Binomial coefficients
3. Probability distributions
4. Conditional probability
5. Bayes' theorem
6. Random variables
7. Expected values
8. Variance and standard deviation

CHAPTER 7, HEAPSORT

A heap is a binary tree stored in an array. One can insert, delete, and check membership in worst case logarithmic time. One can sort in $n \log(n)$ time. A heap can be used for a priority queue.

CHAPTER 8, QUICKSORT

Quicksort has average case $n \log(n)$ running time to sort a list of n elements but in the worst case can be $\Theta(n^2)$. It partitions an array into two sub-arrays, and sorts them recursively. The recurrence for the random case is a little tricky.

```
QUICKSORT(A,p,r)
    if p < r
       then q <- PARTITION(A,p,r)
            QUICKSORT(A,p,q)
            QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
    x <- A[p]
    i <- p - 1
    j <- r + 1
    while TRUE
        do repeat j <- j - 1
            until A[j] .le. x
           repeat i <- i + 1
            until A[i] .ge. x
           if i < j
               then exchange A[i] <-> A[j]
               else return j
```

```
  p               r
  4 3 6 8 2 7 9 1 5
i                   j
        x = 4
  p               r
  4 3 6 8 2 7 9 1 5
  i             j
  p               r
  1 3 6 8 2 7 9 4 5
  i             j
  p               r
  1 3 6 8 2 7 9 4 5
      i   j
  p               r
  1 3 2 8 6 7 9 4 5
      i   j
  p               r
  1 3 2 8 6 7 9 4 5
      j i
  return j
```

Worst case $\Theta(n^2)$
Best case $\Theta(n \log(n))$
Average case $T(n)$ recurrence

$$T(n) =$$

$$\frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right)$$

$$+\Theta(n)$$

Justification of the Quicksort recurrence

Suppose we start with an array in which the first element is the smallest. Then the partition proceeds as follows:

```
      p           r
      4 8 6 9 10 7 5
   i                j
          x = 4
      p           r
      4 8 6 9 10 7 5
      i
      j
      return j
```

Suppose we start with an array in which the first element is the next to the smallest. Then the partition proceeds as follows:

```
  p             r
  4 8 6 3 10 7 5
i                 j
      x = 4
  p             r
  4 8 6 3 10 7 5
  i     j
  p             r
  3 8 6 4 10 7 5
  i     j
  p             r
  3 8 6 4 10 7 5
  j i
  return j
```

In both cases the first block of the partition has only one element. Thus the case $T(1) + T(n-1)$ gets counted twice.

Sample solutions for Quicksort recurrence

Let's simplify it to

$$T(n) = \frac{1}{n}\left(\sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + n$$

Then $T(1) = 1$,

$T(2) = 1/2(T(1) + T(1)) + 2 = 3$,

$T(3) = 1/3(T(1) + T(2) + T(2) + T(1)) + 3 = 1/3(1 + 3 + 3 + 1) + 3 = 5\frac{2}{3}$.

$T(4) = 1/4(T(1) + T(3) + T(2) + T(2) + T(3) + T(1)) + 4 = 1/4(1 + 5\frac{2}{3} + 3 + 3 + 5\frac{2}{3} + 1) + 4 = 1/4(8 + 10\frac{4}{3}) + 4 = 8\frac{5}{6}$.

Is there a pattern?

Solution to the recurrence by integration

First, we simplify the recurrence to

$$T(n) = \frac{2}{n}\left(\sum_{q=1}^{n-1} T(q)\right) + \Theta(n)$$

using the fact that $T(n)$ is $O(n^2)$. We assume $T(n) \le an \lg n + b$ for some $a, b > 0$, and then try to prove this by induction on n.

Let c be $\lg e$, that is, $\log_2 e$. Then $\lg x = c \ln x$. We want to estimate $\sum_{q=1}^{n-1} T(q)$, and using the bound $an \lg n + b$ for $T(n)$, we want to estimate $\sum_{k=1}^{n-1}(ak \lg k + b)$.

We estimate the sum $\sum_{k=1}^{n-1} ak \lg k$ by an integral, using the fact that $\displaystyle\sum_{k=1}^{n-1} T(k) \le \int_{k=1}^{n} T(x)\,dx$ since T is monotonically increasing. Integrating $x \lg x$, we obtain $\frac{c}{2}x^2 \ln x - \frac{c}{4}x^2$. Evaluating this from 1 to n, we obtain

$$\sum_{k=1}^{n-1} ak \lg k \le \frac{n^2}{2} \lg n - \frac{c}{4}(n^2 - 1).$$

Putting this in the recurrence, our induction hypothesis yields that

$$T(n) \leq$$

$$\frac{2a}{n}\left(\frac{n^2}{2}\lg n - \frac{c}{4}(n^2-1)\right) + \frac{2b(n-1)}{n} + \Theta(n),$$

and so we need to show that the quantity on the right is bounded by $an \lg n + b$ to complete the induction. This inequality simplifies to (ignoring lower order terms)

$$T(n) \leq an \lg n - \frac{acn}{2} + 2b + \Theta(n)$$

If we choose a large enough (curiously), $-\frac{acn}{2} + b + \Theta(n)$ is negative, which means that $T(n) \leq an \lg n + b$, and the recurrence is proved by induction.

CHAPTER 9, SORTING IN LINEAR TIME

We first prove that any comparison based sorting algorithm requires $n\log(n)$ comparisons to sort n elements. Then three non-comparison based methods that run in linear time are presented. Counting sort works when the elements are integers in a small range. Radix sort sorts bit strings, least significant digit first. Bucket sort assumes the input list is random and divides it into small intervals.

Consider a decision tree for a comparison-based sorting method on lists of n elements. That's a binary tree with comparisons at the nodes and permutations at the leaves. Since there are $n!$ permutations of n elements, the tree has $n!$ leaves, so its depth must be $\log_2(n!)$, which is $\Omega(n \log n)$. This proves that comparison-based sorting requires $\Omega(n \log n)$ comparisons to sort n elements.

Consider counting sort on the sample list 2,3 6,1,4. We have an array A of 10 elements and put 2 in A(2), 3 in A(3), et cetera. Finally, we go through the elements of the array in order and extract the list in order, 1,2,3,4,6. This is linear time, but only works for small universes.

Suppose we radix sort the numbers 23, 56, 41, 52, 10. First we sort on the last digit, giving us 10,41,52,23,56. Then we sort on the first digit, giving us 10,23,41,52,56. This is linear time.

Let's bucket sort a set of 100 character strings. Consider each one as an integer in the range 0 - 1. Put them in 100 buckets, namely, $0 \leq x < .01$, $.01 \leq x < .02$, $\cdots$, $.99 \leq x \leq 1$. Then sort those in each bucket using say quicksort. Then put the buckets in order. On the average, each bucket will have one element, so this is linear time on the average.

CHAPTER 10, MEDIANS AND ORDER STATISTICS

To find the i^{th} largest element of a list, one can always sort the list and use indexing. However, there are faster methods. The minimum and maximum elements in a list can be found in linear time. Selecting the i^{th} largest element can be done in expected linear time similar to quicksort using a randomized partition. Selection can be done in worst-case linear time by a complicated procedure that divides the list into groups of 5 elements and then works recursively on their medians.

For example, suppose we want to find the minimum of the list 3,4,2,1,6. We start with 3, compare it with 4. We save 3, which is smallest. We compare 3 with 2, and now save 2 since it is smaller. We compare 2 with 1, and save 1 since is smaller. We compare 1 with 6 and save 1 since it is smaller. Thus we have the minimum (1) in four comparisons. In general we can find the minimum or maximum in $n-1$ comparisons for an n element list.

Suppose we want to find the 5^{th} smallest element of a list of 15 elements using quicksort-style partitioning. Suppose the first partition divides the list into two sublists of size 7 and 8, respectively. So we just look at the first sublist of size 7 and want its fifth smallest element. Suppose the partition this times divides the list of 7 into a list of 2 and a list of 5. We now want the third smallest element of the list of 5. We continue this way until we eventually have a list of size 1 which is the desired element.

In general, to find the i^{th} smallest (or largest) element, each partition step will divide the list in half on the average, so after $\log n$ partition steps we will have the desired element. The work will be about $O(n) + O(n/2) + O(n/4) + \ldots$ which is linear. This is, however, not a rigorous analysis.

Now we want to analyze this RANDOMIZED-SELECT algorithm rigorously, because it is another example of a random algorithm. Letting $T(n)$ be the expected time taken for a list of n elements, we obtain

$$T(n) = \frac{1}{n}$$

$$\left(max(T(1), T(n-1)) + \sum_{q=1}^{n-1} (max(T(q), T(n-q))) \right)$$

$$+\Theta(n)$$

since we only work on one side of the partition. Assuming $T(n)$ is monotonically increasing, this is bounded by

$$\frac{1}{n}\left(T(n-1) + 2 \sum_{q=n/2}^{n-1} T(q) \right) + \Theta(n)$$

As before, we can absorb the $T(n-1)$ term in $\Theta(n)$, since $T(n-1)$ is at worst quadratic.

Thus we obtain

$$T(n) \leq \frac{2}{n} \sum_{q=n/2}^{n-1} T(q) + \Theta(n).$$

Assuming by induction that $T(q) \leq cq$ for $q < n$, we can prove this also for $T(n)$, assuming that c is chosen large enough. This completes the proof. This gives expected linear time for selection. However, it can be done in worst-case linear time.

The algorithm is as follows: Divide the set of n elements into lists of 5 elements. Find the median of each list of 5. Then we have about $n/5$ medians. Recursively find their median x. Partition the elements around x, and look in whichever sublist is appropriate to find the desired element. Then we can show that about $(1/2) * (3/5)$ or 3/10 of the elements will be known to be smaller than this median of medians x and about 3/10 will be known to be larger. Thus the partition is guaranteed to be balanced, in a sense. We now show that this algorithm is linear time.

Suppose we use some known sorting method on lists of size 80 or less. We assume that $T(n) \leq cn$ for some constant c and all $n \leq 80$. For $n > 80$ we have

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

where the $O(n)$ is from sorting the lists of size 5. Assuming that $T(k) \leq ck$ for $k < n$ we can also prove it for n, assuming c is large enough. This gives the general formula $T(n) \leq cn$ by induction.

We give some of the details of this derivation. Assuming $T(k) \leq ck$ for $k < n$, we derive that

$$T(n) \leq 9cn/10 + 7c + O(n).$$

We want to show that $T(n) \leq cn$ to complete the induction. In order to show this, we need to show that

$$9cn/10 + 7c + O(n) \leq cn.$$

Now, $O(n)$ means a function bounded by Kn for some constant K. Thus it is enough to show that

$$9cn/10 + 7c + Kn \leq cn.$$

Subtracting $9cn/10$ from both sides, this becomes

$$7c + Kn \leq cn/10$$

or

$$Kn \leq cn/10 - 7c$$

or

$$Kn \leq c(n/10 - 7).$$

Dividing by n and c, this becomes

$$K/c \leq (1/10 - 7/n).$$

If $n \geq 80$, then $(1/10 - 7/n) \geq (1/10 - 7/80) = 1/80$. So we want $K/c \leq 1/80$, or, $c \geq 80K$. Thus if we choose $c \geq 80K$ then the induction goes through, and we have proved that this method of selection has worst case linear time.

CHAPTER 11, ELEMENTARY DATA STRUCTURES

This chapter is concerned with stacks and queues, linked lists, implementing pointers, and representing rooted trees.

CHAPTER 12, HASH TABLES

Direct addressing has a pointer from each element to the place it is stored, and works when the keys are small. For large universes of keys, we can use hash functions. Collisions can be resolved by chaining. The performance depends on the load factor. A number of techniques for obtaining good hash functions are given. Open addressing avoids pointers and saves space by computing a sequence of addresses for each element.

CHAPTER 13, BINARY SEARCH TREES

This data structure supports the operations of search, minimum, maximum, predecessor, successor, insert, and delete. Properties of random binary search trees are discussed.

CHAPTER 14, RED BLACK TREES

These are like binary trees but are guaranteed to be balanced and thus many operations have worst case logarithmic time. These are binary trees where the nodes are colored red or black in such a way that no path is more than twice as long as any other. The rotation operation can restore the red-black properties, in worst case logarithmic time. Operations of insertion and deletion are defined, taking worst case logarithmic time.

The properties that a red-black tree must satisfy are the following:

1. Every node is either red or black
2. Every leaf is black
3. If a node is red, both children are black
4. Every simple path from a node to a descendent leaf has the same number of black nodes

Here are some examples of red-black trees:

```
   B          R          B
  / \        / \        / \
 B   B      B   B      R   B
                      / \
                     B   B
```

From these conditions, one can easily see that at most half the nodes on a path from the root to a leaf can be red, and that all paths will differ in length by at most a factor of two. Thus the tree will never be highly unbalanced. It is possible to preserve these properties on insertion and deletion by complicated algorithms given in the book which run in logarithmic time.

CHAPTER 15, AUGMENTING DATA STRUCTURES

First, it is shown how to augment red-black trees to permit logarithmic retrieval of the i^{th} smallest element and determining the rank of an element. A general approach to augmenting data structures is sketched, consisting of adding some data and operations to a data structure. Finally, a data structure for intervals based on red-black trees is given; this permits one to find all intervals that overlap a specified one, in logarithmic time.

The intervals are stored in a tree as follows: We use their left endpoint as the key to insert them in the tree, so they are ordered by their left endpoint. Also, at each node in the tree, we store the maximum right endpoint of any interval in the subtree at that node. This information can be updated in logarithmic time on insertion and deletion. Then given an arbitrary interval i (not necessarily in the tree), in logarithmic time we can find an interval in the tree that overlaps i, if there is one. The idea is always go to the left (smaller) subtree unless it is empty or all intervals in that subtree have a right endpoint smaller than the left endpoint of i.

We give an example of some intervals stored in the tree and how the search would work:

```
            [10,11]
            /     \
        [3,7]    [12,15]
        /    \
    [1,2] [5,6]
```

If we are looking for an overlap to [8,9] we don't go to the left subtree, since it's highest point is 7, which is less than 8. We go to the right subtree, and find no overlap there. If we are looking for an overlap to [4,8], we go to the left subtree, since 7 is larger than or equal to 4. We then find an overlap right away.

CHAPTER 16, DYNAMIC PROGRAMMING

Dynamic programming is used for optimization problems, that is, one wants to find a solution that maximizes or minimizes the value of some function. This can be used when optimal solutions to subproblems can be combined to obtain an optimal solution to the whole problem, that is, any optimal solution to the whole problem has also optimal solutions to sub-problems. Also, it requires that the same subproblems need to be solved over and over. This technique can reduce exponential time problems to polynomial time. Several examples are given, namely, matrix-chain multiplication, longest common subsequences, and optimal polygon triangulation.

Suppose we are given matrices $A_1, A_2, \ldots, A_n$, and matrix A_i has dimensions $p_{i-1} \times p_i$. We want to find the fastest way to do the multiplication $A_1 A_2 \ldots A_n$. Let $m[i,j]$ be the cost of multiplying matrices $A_i A_{i+1} \ldots A_j$ in an optimal way. We can show that $m[i,j] = 0$ if $i = j$, and if $i < j$,

$$m[i,j] = min_{i \leq k < j}\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\}.$$

Let $s[i,j]$ be a value k yielding an optimal $m[i,j]$. By computing $m[i,j]$ in the correct order, one can solve this problem in $O(n^3)$ time for n matrices.

We illustrate this algorithm for matrices of dimensions $(2,5),(5,3),(3,6),(6,4)$.

Suppose we multiply the 5×3 and 3×6 matrices first. This takes 90 multiplications. We obtain a 5×6 matrix. Multiplying this by the 6×4 matrix takes 120 multiplications. We obtain a 5×4 matrix. Multiplying this by the 2×5 matrix takes 40 multiplications. The total number of multiplications is $90+120+40$, or, 250 multiplications. If we first multiply 2×5 by 5×3 it takes 30 multiplications. Then 3×6 by 6×4 takes 72 multiplications. Then 2×3 by 3×4 takes 24 multiplications. In all there are $30+72+24$ or 126 multiplications. So the order can make a large difference.

We compute $m[i,j]$ as follows:

```
m[1,2] = 30
m[2,3] = 90
m[3,4] = 72
m[1,3] = min(30+2*3*6, 90+2*5*6)
       = min(66, 150) = 66
m[2,4] = min(90+5*6*4, 72+5*3*4)
       = min(210, 132) = 132
m[1,4] = min(132+2*5*4,30+72+2*3*4,66+2*6*4)
       = min(172,126,114) = 114
```

The optimal is 114, so we obtain this by m[1,3] which is obtained from m[1,2]. So we multiply $((A_1 * A_2) * A_3) * A_4$, which is the optimal order.

We can solve the longest common subsequence algorithm by dynamic programming using the following recursive properties:

Let $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$ be sequences, and let $Z =< z_1, z_2, \ldots, z_k >$ be any longest common subsequence of X and Y. We then have the following:

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a longest common subsequence of X_{m-1} and Y_{n-1}.

- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is a longest common subsequence of X_{m-1} and Y.

- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is a longest common subsequence of X and Y_{n-1}.

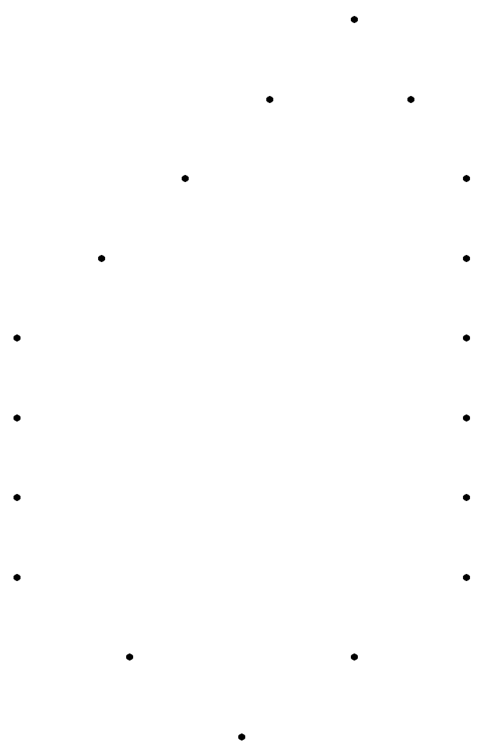
Based on this, we can compute an LCS of X and Y quickly if we know an LCS of X_{m-1} and Y_{n-1}, an LCS of X_{m-1} and Y, and an LCS of X and Y_{n-1}. We can solve these subproblems recursively and remember solutions to avoid repeated work, obtaining an efficient dynamic programming solution to the general longest common subsequence property. We can illustrate this on aab and bab:

```
LCS table:
     a   aa   aab

b    _   _    b

ba   a   a    a or b

bab  a   a    ab
```

The recursion for aab and bab says that since they have the same last character, we only have to find a LCS for aa and ba and then add b to it. To find a LCS for aa and ba, we do the same since they also have the same last character, so we have to find an LCS of a and b. The last characters differ, so we find either an LCS of (empty) and b, or we find an LCS of a and (empty). But the LCS of (empty) and anything is empty.

Let's try abab and baba. We have to find either an LCS of abab and bab or one of baba and aba, since the last characters differ. For abab and bab, we have the same letters on the end, so we recurse several times to the case a and (empty), which is empty. Unwinding the recursion, the answer is bab. For baba and aba, we have several cases of the same last letter, and recurse down to b and (empty), which has (empty) as the LCS. Unwinding the recursion, the answer is aba. Then for abab and baba, we pick the longest of these two, that is, of aba and bab. Since they are the same length, either will do as an LCS.

Optimal polygonal triangulation can be done by repeately using this kind of a recurrence:



```
            .
         .    .
      .        ,.
   .        ,,,,.
.        ,,,,,,,.
.     ,,,,,,,,,,.
.  ,,,,,,,,,,,,,.
.,,,,,,,,,,,,,,,.
    .       .
        .
```

CHAPTER 17, GREEDY ALGORITHMS

A greedy algorithm makes a sequence of locally optimal choices in an attempt to obtain a globally optimal solution. Even when this does not yield a global optimum, it may yield something not far away. An example (the activity selection problem) is given where this approach yields optimal solutions. Huffman codes are another example. Matroids give a general framework in which greedy algorithms work – I don't expect you to be familiar with them. Matroids are then applied to give a greedy algorithm for unit time task scheduling.

Examples where greedy algorithms aren't optimal

- Traveling salesman problem – always go to the closest city
- Making change with various coins – always give the largest coin possible
- Coloring a graph – always try to use as few colors as possible
- 0-1 knapsack problem – counterexample has \$60/(10 lbs), \$100/(20 lbs), \$120/(30 lbs) in size 50 knapsack

The text gives some applications where greedy algorithms always produce optimal solutions. The first such application is an activity selection problem. We have a set $S = \{1, 2, \ldots, n\}$ of *activities* with *start times* s_i and *finish times* f_i, where $s_i \leq f_i$. We say two activities i and j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$. The *activity selection problem* is to select a maximum size set of mutually compatible activities.

Our algorithm assumes the activities are sorted so that $f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_n$.

The algorithm is as follows:

```
GREEDY-ACTIVITY-SELECTOR(f)
    n <- length[s]
    A <- {1}
    j <- 1
    for i <- 2 to n
        do if s[i] .ge. f[j]
            then A <- A U {i}
                 j <- i
    return A
```

The idea is that at every choice, we pick the compatible activity with the earliest finishing time. This is greedy because the earliest finishing time is a locally optimal property. One can show that this is a (globally) optimal algorithm.

The text also formulates **Huffman codes** in terms of a greedy algorithm.

The text also discusses **Matroids**, which give a general set of axioms guaranteeing that a problem can be optimally solved by a greedy algorithm. This theory is then applied to devise a greedy algorithm to solve a task scheduling problem involving unit time tasks with specified deadlines and penalty functions.

CHAPTER 18, AMORTIZED ANALYSIS

This is a technique for analyzing the worst-case performance of a data structure. Instead of looking at the worst case time for each separate operation, it has various ways of subtracting times from some operations and adding them to other operations to make the analysis better for the total time taken. This is useful when some operations may be expensive, but they are only rarely performed. Three such techniques are given, the aggregate method, the accounting method, and the potential method. These are illustrated on two examples: stacks with a multipop operation and binary counting. These techniques are also used on dynamic tables. It's possible that many of you have not seen this material yet.

The *aggregate method* looks at sequences of n operations and estimates their total cost. The average time per operation is then computed.

Consider the following operations:

PUSH(S,x) - pushes x onto stack S
POP(S) pops the top element off of S
MULTIPOP(S,k) pops the top k elements off of S

We assume that the cost for PUSH and POP is 1 and the cost of MULTIPOP(S,k) is k. We want to analyze the worst case cost of a sequence of these operations, assuming the stack is empty at the start. Just looking at worst case per operation won't do (why?).

The aggregate method gives a better bound. We note that each call to MULTIPOP(S,k) corresponds to k PUSH operations done earlier. So if a sequence of operations has m PUSH operations, then the total cost of POP and MULTIPOP operations can be at most m, and the total cost is at most $2m$, which is still $O(m)$.

The *accounting method* tries to get a better bound by clever accounting (massaging the accounts). The cost of some operations is reduced and the cost of others is increased. We just have to be sure that this will not decrease the total cost.

We charge PUSH operations a cost of 2 and POP and MULTIPOP are free. Thus the *amortized cost* of PUSH is 2, which means there is a *credit* of 1 to assign to something else. POP has a debt of 1 and MULTIPOP(S,k) has a debt of k. Summing the amortized costs is clearly linear; then we have to show that the total credit has to be larger than the debt.

The *potential method* associates a *potential function* with each data structure, and this is a real number. The *amortized cost* of an operation with respect to the potential function is its actual cost plus the change in potential. Thus the sum of the amortized costs is bounded by the sum of the actual costs plus the change in potential. If the change in potential is positive, then the sum of the amortized costs is an upper bound on the true cost.

For our example, the potential can be the size of the stack, and if we start with an empty stack, the change in potential is positive.

This is applied to incrementing a binary counter, too. Sometimes one has to change many bits, but the average cost is small. This same formalism is applied to dynamic tables, which may have to be expanded when they become full and contracted when they become empty. The costs of these operations are large, but they are only rarely performed, if the data structure is properly designed.

Material for Tuesday, September 10, 1996 follows.

The potential method seems to be the best formalized of the three. So I'd like to spend a little more time on it.

Let D_0 be an initial data structure on which n operations are performed. Let c_i be the cost of the i^{th} operation and let D_i be the data structure that results after performing the i^{th} operation to data structure D_{i-1}. We define a **potential function** Φ that maps each D_i onto a real number $\Phi(D_i)$ called the *potential* associated with D_i. An example of a potential function might be the length of a list or the number of elements in a stack or the number of elements in a table. Then the *amortized cost* $\widehat{c_i}$ of the i^{th} operation with respect to Φ is defined by

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

So the amortized cost is the actual cost plus the change in potential. From the above equation, we obtain that

$$\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

from which we obtain

$$\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

This shows that the sum of the amortized costs equals the sum of the actual costs plus the change in potential.

If we define Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

and so we can use the sum of the amortized costs as an upper bound on the sum of the actual costs.

For the stack example, we can let $\Phi(D_i)$ to be the number of elements in the stack D_i. Then the amortized cost of a PUSH is two, the amortized cost of a POP is zero, and the amortized cost of a MULTIPOP is also zero. Hence the sum of the amortized costs is at most $2n$. If we start with an empty stack, this shows that $2n$ is an upper bound on the sum of the actual costs. Therefore the sum of the actual costs is bounded by $2n$, which is linear. Note that this result holds even though MULTIPOP operations can be expensive.

For the binary counter example, we can let the potential function be the number of one's in a bitstring. Then the amortized cost of any increment is one. So if we start with zero, this shows that the work for binary counting is proportional to the number of operations.

For the dynamic table example, we can let the potential be 2∗(number of elements in the table) - (size of the table). Then, inserting an element has a constant cost if there is no expansion. Suppose the table has size n and n elements. If we insert an element, the new table has $n+1$ elements and size $2n$. The work to expand the table is n. The difference in potentials is $(2*(n+1)-2*n)-(2n-n)$ or $2-n$. The work to insert the element after expansion is 1. So the amortized cost is ACTUAL COST + CHANGE IN POTENTIAL, or

$$n+1+2-n=3$$

This shows that the cost per operation on the average is no larger than 3.

B-TREES

These trees are adapted for situations where the tree is too big to fit in memory and disk accesses must be used. Therefore each node can have many data elements. Typically all of these will be stored on a page in disk so the whole node can be read in one disk access, reducing the number of disk accesses needed.

A node x of a B-tree with n elements a_1, a_2, $\ldots$, a_n must satisfy $a_1 \leq a_2 \leq \ldots \leq a_n$ and has $n+1$ children, in the ranges $[-\infty, a_1]$, $[a_1, a_2]$, $\ldots$, $[a_{n-1}, a_n]$, $[a_n, \infty]$.

Each leaf must have the same depth (the tree is balanced).

A node must have at least t children and can have at most $2t$ children.

One can show that such trees have logarithmic heights. Also, algorithms are given in the text to insert, search, and delete, all in worst-case logarithmic time. A basic operation for this is that of *splitting*, that is, splitting a node in two pieces and moving one of its elements to the parent.

The general idea is that one has to keep the number of elements of nodes between $t-1$ and $2t-1$. When inserting, the number can get too big, so one has to split nodes to make them smaller first. When deleting, the number can get to small, so one has to merge nodes to make them bigger first. (This is the reverse of splitting.) One also has to make sure that the tree remains balanced.

INSERTION

B-TREE-SPLIT-CHILD(x,i,y) splits full node y around its middle element, and y is the i-th child of x. We assume x is not full. The node y becomes two nodes, and the median element of y is moved up into x.

Here's an example of splitting a node:

```
Before:        0  1  20 30
                     |
                     |
             2   4   5   8   9
            |   |   |   |   |   |
            T1  T2  T3  T4  T5  T6

After:        0  1  5 20 30
                   /   \
                  /     \
            2   4         8   9
           |   |   |     |   |   |
           T1  T2  T3    T4  T5  T6
```

This can also create a new root.

B-TREE-INSERT(T,k) inserts a key k into a tree T. It splits the root if it is full, then calls B-TREE-INSERT-NONFULL to insert k into the resulting tree.

B-TREE-INSERT-NONFULL(x,k) inserts k into the subtree rooted at node x, which is assumed to be nonfull. This procedure splits full nodes encountered in the path to a leaf. When it reaches a leaf it inserts k there.

DELETION

This is sufficiently complicated that even the book does not give a complete algorithm. The ideas are these:

We go down the path to the node being deleted. Along the way we insure that each node has an extra key, thus, t keys in all. If we find the key k to delete in a leaf, we just delete it, which is OK since the leaf has an extra key.

If we find k in the internal node x being examined, we delete the predecessor or successor of k from a subtree and replace k by this predecessor or successor. However, this can't work if the two nearby children of k both have t-1 nodes. In this case, we merge them with k, remove k from x (which is OK since it had at least t elements), and recursively delete k from the merged node.

Here's an example of the merging operation, assuming 5 is being deleted and has been found in an internal node:

```
Before:       0  1  5  20 30
                  /   \
                 /     \
           2   4         8   9
         |   |   |     |   |   |
         T1  T2  T3    T4  T5  T6

After:         0  1  20 30
                    |
                    |
            2   4   5   8   9
          |   |   |   |   |   |
          T1  T2  T3  T4  T5  T6
```

If we don't find k at the internal node x being examined, we find which child y of x is on the path to k. We then do a preprocessing step to insure that this child y has at least t keys, then recurse to this child.

This preprocessing step involves a kind of rotation, in case one of the neighbor siblings of y has t keys. If y and both neighbor siblings of y have t-1 keys, we do a merge of one of them with y and move k down to this new node.

Here's how this rotation works:

```
Before:       0  1  5  20 30
                  /   \
                 /     \
           2   4         8   9  10
         |   |   |     |   |   |   |
         T1  T2  T3    T4  T5  T6  T7


After :       0  1  8  20 30
                  /   \
                 /     \
        2   4   5         9  10
      |   |   |   |     |   |   |
      T1  T2  T3  T4    T5  T6  T7
```

Note that the left child is now bigger.

Transparencies for Sept. 12 1996 lecture

BINOMIAL HEAPS

Binomial heaps are like ordinary (binary) heaps, but they permit some operations to be done faster; in particular, two heaps can be merged in logarithmic time. The price for this is that extracting the minimum is a little slower (logarithmic as opposed to constant time). This data structure makes efficient implementations of some algorithms possible.

Binomial heaps are collections of **Binomial trees**. The Binomial tree B_k has a root node and the children (left to right) $B_{k-1}, B_{k-2}, \ldots, B_0$. One can show that B_k has the following properties:

- It has 2^k nodes.
- It has height k.
- There are $\binom{k}{i}$ nodes at depth i.

All these properties can be proved by induction. We can also show that the maximum degree of any node in B_n is n.

Here are some examples of Binomial trees:

```
B_0     *     B_1     *
                      |
                      *


B_2     *     B_3      *
       /|             '/|
      / |           ' / |
     *  *         '  *  *
     |         *     |
     |        /|     |
     *       / |     *
            *  *
            |
            |
            *
```

A **Binomial heap** H is a set of binomial trees such that

- The key of a node is greater than or equal to the key of its parent
- There is at most one binomial tree in H of any given degree.

For example, if H has 5 nodes, then it consists of B_2 and B_0, and the binary representation of 5 is 101.

This is represented as a data structure by pointers from nodes to siblings and parent, and pointers from the parent to one child.

The operations performed on Binomial heaps are these:

- **Finding the minimum key**, by looking at the roots
- **Merging two Binomial trees of the same degree**, easy to do
- **Merging two Binomial heaps**, by merging some of their trees of the same degree; like adding binary numbers
- **Inserting a node**, by merging a heap and a node B_0
- **Remove minimum node**, by making subtrees into a new heap and merging with the original one

- **Decreasing a key**, bubbling it up in the tree

- **Deleting a key**, by decreasing it to $-\infty$, then removing the minimum node

Transparencies for Tuesday, Sept. 17 lecture

Examples

```
Binomial Heap              Binomial Heap

D           A              F           B
           /|                          |
          / |                          |
         E  G                          P
         |
         |
         H
```

Find smallest element
Make an element smaller
Merge the two heaps
Removing a minimum element
Removing a non-minimal element
Inserting an element
Q: Identical keys?

```
D       B          A
|       |         /|
|       |        / |
F       P       E  G
                |
                |
                H
```

```
      B          A
     /|         /|
    / |        / |
   D  P       E  G
   |          |
   |          |
   F          H
```

```
             A
          , /|
      ,    / |
   B      E  G
  /|      |
 / |      |
D  P      H
|
|
F
```

This is like binary addition.

Tradeoffs in number of binomial trees allowed in a heap:

- Many – minimum is slow

- One – union is slow

FIBONACCI HEAPS (CHAPTER 21)

This material is not required – but I wanted to give you just an idea of what they are and why they are interested. A Fibonacci heap is like a Binomial heap but with less structure. It permits some operations to be performed faster. In particular, insertion, minimum, union, and decrease-key are all constant amortized time operations, but all are worst case logarithmic for Binomial heaps. Thus, Fibonacci heaps permit some algorithms to run faster. However, they have a higher overhead, so they may not be practical for implementation purposes.

Disjoint Set Operations

We have a collection of disjoint sets of elements. Each set is identified by a **representative** element. We want to perform union operations, and tell which set something is in. This is useful in a minimum spanning tree algorithm and many other applications. Formally, we have the following operations:

1. MAKE-SET(x): Create new set $\{x\}$ with representative x.
2. UNION(x,y): x and y are elements of two sets. Remove these sets and add their union. Choose a representative for it.
3. FIND-SET(x): return the representative of the set containing x.

Example

MAKE-SET(1)	
MAKE-SET(2)	
MAKE-SET(3)	
MAKE-SET(4)	
FIND(3)	(returns 3)
FIND(2)	(returns 2)
UNION(1,2)	(representative 1, say)
FIND(2)	(returns 1)
FIND(1)	(returns 1)
UNION(3,4)	(representative 4, say)
FIND(4)	(returns 4)
FIND(3)	(returns 4)
UNION(1,3)	(representative 4, say)
FIND(2)	(returns 4)
FIND(1)	(returns 4)
FIND(4)	(returns 4)
FIND(3)	(returns 4)

Implementation

Linked List Implementation with Concatenation

Each cell has an element, a pointer to the next member, and a pointer to the first element in the list, which is the set representative.

For union, we append one list to another, changing the pointers to the set representative on the list on the end.

MAKE-SET	$O(1)$ time
FIND-SET	$O(1)$ time
UNION	$O(n)$ time

Thus we get quadratic worst case performance.

Refined Linked List Implementation
(Concatenate longer list onto shorter one)

MAKE-SET	$O(1)$ time
FIND-SET	$O(1)$ time
UNION	$O(\log n)$ time (rough bound)

Proof of the bound for UNION: A set has to double in size each time it is concatenated onto the end. Thus we get $O(n \log n)$ worst case performance for n operations.

Forest representation

Here we represent each set as a tree, and the representative is the root. For example, the following forest represents the sets {1,2,3}, {4,5}, {6}:

```
    2     5   6
   / \    |
  1   3   4
```

Implementation

MAKE-SET(x)	Create a tree
FIND-SET(x)	Go to the root
UNION(x,y)	Add a pointer

Thus we would get the following from UNION(1,4):

```
   2           6
  /|\
 1 3 5
     |
     4
```

This representation does not improve the running time in the worst case over the linked list representation.

Path compression and ranks

These are refinements of the forest representation which make it significantly faster.

FIND-SET	Do path compression
UNION	Use ranks

Path compression means that when we do FIND-SET(x), we make all nodes encountered point directly to the representative element for x. Initially, all elements have rank 0. The ranks of representative elements are updated so that if two sets with representatives of the same rank are unioned, then the new representative rank is incremented by one.

Example

We show a sequence of operations and what the forest would look like.

```
MAKE-SET(1) ... MAKE-SET(6)

    1  2  3  4  5  6   RANKS: 0

UNION(1,2) UNION(4,5)

       2   3   5  6    RANK(2)=1
       |       |       RANK(5)=1
       1       4

UNION(1,3)

       2       5  6    RANK(2)=1
      / \      |       RANK(5)=1
     1   3     4
```

UNION(5,6)

```
    2       5        RANK(2)=1
   / \     / \       RANK(5)=1
  1   3   4   6
```

UNION(4,3)

```
    2                RANK(2)=2
   /|\               RANK(5)=1
  1 3 5
      |\
      4 6
```

FIND(4)

```
    2
   /|\ .
  1 3 5 4
       \
        6
```

FIND(3) no change

FIND(6) (path compression)

```
     2
    /|\ . .
   1 3 5 4 6
```

Costs

MAKE-SET	$O(1)$ time
FIND-SET	Path length
UNION	Path length + constant

One can show that this is nearly linear time, much less than $n \log n$.

Note the tremendous difference between the worst method (quadratic) and the best one (nearly linear), which could make a tremendous difference in running time for 100 or more element lists.

Breadth-first and Depth-first Search on Graphs

We illustrate two search techniques and apply the second of them to topological sort and strongly connected components.

Only one lecture is allocated to Chapter 23 which has a lot of material, so we can't cover it all in class.

- Breadth-first search

- Depth-first search

Breadth-first search

- Initially, vertices are colored white.
- Discovered vertices are colored gray.
- When done, discovered vertices are colored black.
- $d[u]$ stores the distance from s to u.
- $\pi[u]$ is a predecessor to u on its shortest path.
- Q is a first-in first-out queue.

```
BFS(G,s)
    for each vertex u in V[G] - {s}
        do color[u] <- white
           d[u] <- infinity
           pi[u] <- NIL
    color[s] <- gray
    d[s] <- 0
    pi[s] <- NIL
    Q <- {s}
    while Q .ne. {}
        do u <- head[Q]
           for each v in Adj[u]
               do if color[v] = white
                     then color[v] <- gray
                          d[v] <- d[u]+1
                          pi[v] <- u
                          Enqueue(Q,v)
           Dequeue(Q)
           color[u] <- black
```

Example graph

```
 1 - 2 - 3 - 4
 |   |   |   |
 5 - 6 - 7 - 8
 |   |   |   |
 9 -10 -11 -12
 |   |   |   |
13 -14 -15 -16
```

The running time of BFS is $O(V + E)$.

After execution of BFS, all nodes reachable from the source s are colored black.

After execution of BFS, $d[v]$ is the distance of a shortest path from the source s to v for vertices v reachable from s.

After execution of BFS, if v is reachable from s, then one of the shortest paths to v passes through the edge $(\pi[v], v)$ at the end.

After execution of BFS, the edges $(\pi[v], v)$ for v reachable from s form a **breadth-first tree**.

Example.

Depth-first search

- Nodes are initially white
- Nodes become gray when first discovered
- Nodes become black when they are done
- $d[v]$ records when v is first discovered
- $f[v]$ records when v is done
- $d[u] < f[u]$

```
DFS(G)
    for each vertex u in V[G]
        do color[u] <- white
        pi[u] <- NIL
    time <- 0
    for each vertex u in V[G]
        do if color[u] = white
              then DFS-Visit(u)

DFS-Visit(u)
    color[u] <- gray
    d[u] <- time <- time + 1
    for each v in Adj[u]
        do if color[v] = white
              then pi[v] <- u
                   DFS-Visit(v)
    color[u] <- black
    f[u] <- time <- time + 1
```

Example graph

```
 1 - 2 - 3 - 4 -17-18
 |   |   |   |
 5 - 6 - 7 - 8
 |   |   |   |
 9 -10 -11 -12
 |   |   |   |
13 -14 -15 -16 -19-20
```

The running time of DFS is $O(V + E)$.

After execution of DFS, all nodes are colored black.

After execution of DFS, the edges $(\pi[v], v)$ form a collection of **depth-first trees**, called a **depth-first forest**.

Edge classification:

1. Tree edges (u, v): v was discovered first using (u, v)
2. Back edges (u, v): v is an ancestor of u in the DFS tree
3. Forward edges (u, v): v is a descendent of u, not a tree edge
4. Cross edges (u, v): Other edges

Example.

In a depth-first search of an undirected graph, every edge is either a tree edge or a back edge.

TOPOLOGICAL SORT

A *topological sort* of a dag $G = (V, E)$ is a linear ordering of all the vertices of G such that if G contains an edge (u, v), then u appears before v in the ordering.

This can be done using DFS – just insert each vertex on the front of an (initially empty) list as the vertex is finished.

Example.

Note that this algorithm is linear (that is, $O(V + E)$) time.

ACYCLICITY

Theorem. A graph may be topologically sorted iff it is cyle-free.

Proof. If every vertex has an outgoing edge, then the graph has a cycle. Otherwise, one can repeatedly remove vertices with no outgoing edges.

Thus the topological sort routine is a linear time test for the presence of a cycle in a graph. More simply, we do a depth-first search, and then see if the DFS tree has a back edge.

STRONGLY CONNECTED COMPONENTS

A *strongly connected component* of a graph G is a maximal set of vertices U such that for any vertices u and v in U, there is a path from u to v in G and a path from v to u.

Example.

Let G^T be G with the directions of the edges reversed.

Algorithm for strongly connected components using DFS:

1. Call $DFS(G)$ to compute finishing times $f[u]$ of vertices u.
2. Compute G^T.
3. Call DFS(G^T), considering vertices in reverse order of $f[u]$ from line 1.
4. Output the vertices of each DFS tree obtained from step 3 as a separate strongly connected component.

Note that this algorithm is linear $(O(V+E))$ time.

Intuition about the algorithm.

The text gives a long proof of correctness – here is the idea:

Strongly connected components form a tree – those nearest the leaves get smaller f values. Thus in step 3 we start near the roots, and reverse the edges, getting one component at a time.

Example.

MINIMUM SPANNING TREES

A **spanning tree** is a subset of the edges of a graph that is a tree and includes every vertex.

The **cost** of a spanning tree of an edge-weighted graph is the sum of the costs of its edges.

A **minimum spanning tree** is a spanning tree of minimum cost.

The **minimum spanning tree problem** is to find a minimum spanning tree in a graph. It is not always unique.

One can think of applications such as power line construction.

We say an edge e in a graph G is **safe for** A if $A \cup \{e\}$ is a subset of *some* minimum spanning tree for G.

We have the following generic algorithm for minimum spanning trees:

```
GENERIC_MST(G,w)
    A <- {}
    while A does not form a spanning tree
        do find an edge (u,v) that is
           safe for A
            A <- A U {(u,v)}
    return A
```

We now need to know which edges are safe. For this we have:

A **cut** of a graph G is a partition $(S, V - S)$ of the vertices of G into two sets. An edge (u, v) **crosses** the cut if its two endpoints are in different sets of the cut. An edge is a **light edge** crossing a cut if its weight is minimum among all edges crossing the cut.

Any light edge is in some MST.

If A has no edges that cross a cut, and A is a subset of some MST, and e is a light edge for the cut, then e is safe for A.

KRUSKAL'S ALGORITHM

```
MST-Kruskal(G,w)
    A <- {}
    for each vertex v in V[G]
        do MAKE-SET(v)
    sort the edges of E by nondecreasing
         weight w
    for each edge (u,v) in E, in order of
                        nondecreasing weight,
        do if FIND-SET(u) .ne. FIND-SET(v)
            then A <- A U {(u,v)}
                Union(u,v)
    return A
```

Time is dominated by sorting and is therefore $O(E \log E)$.

PRIM'S ALGORITHM

```
MST-Prim(G,w,r)
    Q <- V[G]
    for each u in Q
        do key[u] <- infinity
    key[r] <- 0
    pi[r] <- NIL
    while Q .ne 0
        do u <- EXTRACT-MIN(Q)
            for each v in Adj[u]
                do if v in Q
                      and w(u,v) < key[v]
                      then pi[v] <- u
                           key[v] <- w(u,v)
```

Time is dominated by the heap operations and is therefore $O(E \log V)$. By use of Fibonacci heaps this can be improved to $O(E + V \log V)$.

There are faster algorithms for sparse graphs, one of which runs in time $O(E \lg^*(V))$.

SINGLE-SOURCE SHORTEST PATHS

The weight of a path is the sum of the weights of its edges.

A **shortest path** from u to v is a path of minimum weight from u to v.

The **single-source shortest-paths problem**: Find a shortest path from a given source vertex s to every vertex v in a graph.

There are many applications – costs can be time or expense, for example.

We consider several topics:

- Dijkstra's algorithm – non-negative edge weights, using breadth-first search

- Bellman-Ford algorithm – permits negative edge weights, detects negative cycles, otherwise finds shortest paths

- Shortest paths in directed acyclic graphs; PERT charts

- Difference constraints and shortest paths

The algorithms all work by **relaxation**.

Define $\delta(u,v)$ to be the weight of a shortest path from u to v.

$d[v]$ is an upper bound on the length of a shortest path from the source to v.

First d is initialized:

```
INITIALIZE-SINGLE-SOURCE(G,s)
    for each vertex v in V[G]
        do d[v] <- infinity
           pi[v] <- NIL
    d[s] <- 0
```

Here $\pi[v]$ is a predecessor of v on a shortest path, eventually.

Then we relax on edges in some order:

```
RELAX(u,v,w)
    if d[v] > d[u] + w(u,v)
       then d[v] <- d[u] + w(u,v)
       pi[v] <- u
```

This updates the estimate of distance to u using the edge (u, v) and preserves the fact that $d[v]$ is still an upper bound on $\delta(s, v)$. It also updates $\pi[v]$, so that by consecutive π we can obtain a path of length $d[v]$. Thus when we are done, we can obtain the shortest path using π.

Dijkstra's algorithm

```
DIJKSTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    S <- {}
    Q <- V[G]
    while Q .ne. {}
        do u <- EXTRACT-MIN(Q)
           S <- S U {u}
           for each vertex v in Adj[u]
               do RELAX(u,v,w)
```

At any time, shortest paths to elements of S are known.

Running time

For dense graphs, the running time is $O(V^2)$ if we maintain the priority queue as a linear array. Each edge is processed once, and each vertex may be seen V times.

For sparse graphs, we can maintain Q as a binary heap, which yields $O(E \log V)$ time. It takes $O(\log V)$ time to decrease $d[v]$, and this must be done E times.

With Fibonacci heaps, the time can be reduced to $O(V \log V + E)$, since the decrease-key operations take only $O(1)$ amortized time.

Correctness

At any time, $d[v] = \delta(s, v)$ for vertices v in S.

This is true at the beginning, when S is empty.

We prove it for each iteration by induction.

Consider the vertex v^* in $V - S$ whose value $\delta(s, v^*)$ is minimal. It must have a shortest path all of whose vertices other than v^*, are in S. Now, EXTRACT-MIN(Q) will find some such v^* and add it to S. Then relaxation will update the distance estimates of vertices in $V - S$.

Example.

Bellman-Ford algorithm

This relaxes over all the edges of the graph V times.

```
BELLMAN-FORD(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    for i <- 1 to |V[G]| - 1
        do for each edge (u,v) in E[G]
               do RELAX(u,v,w)
    for each edge (u,v) in E[G]
        do if d[v] > d[u] + w(u,v)
              then return FALSE
    return TRUE
```

The Boolean value returned tells if there is no negative weight cycle reachable from the root. If TRUE is returned, then the shortest paths are returned.

Running time

The running time is $O(VE)$ because each edge is processed V times.

Correctness

We assume that for each vertex v, there is a shortest path from s to v, i.e., there are no negative cycles.

We show that after i iterations of the loop, for all vertices v having a shortest path from s with i or fewer edges, $d[v]$ is correctly computed.

This is true at the beginning when $i = 0$, obviously.

Assume it is true before a given iteration. We prove it is true afterwards as well.

Before the iteration, we have correctly computed shortest paths of i edges. Suppose v has a shortest path with last edge (u, v). Then u has a shortest path with $i-1$ edges, which will have been correctly computed in the previous iteration. Thus one step of relaxation will compute the shortest path for v also.

If there is a negative cycle reachable from s, then the values $d[v]$ will continue to decrease, and so the Boolean flag will be set to FALSE.

Example.

Single-source shortest paths in directed acyclic graphs

In this case only one iteration over the vertices is needed, and the algorithm is $\Theta(V+E)$ time. Negative edges are no problem, since there are no cycles.

```
DAG-SHORTEST-PATHS(G,w,s)
    topologically sort the vertices of G
    INITIALIZE-SINGLE-SOURCE(G,s)
    for each vertex u taken in
             topologically sorted order
        do for each vertex v in Adj[u]
               do Relax(u,v,w)
```

Application to PERT charts – find **longest paths** (critical paths) in dags – reverse signs of weights.

Correctness

We show that for each vertex u, its shortest path will have been correctly computed when u is considered in the loop.

We assume by induction that this is true before u is processed.

The shortest path for u will consist of a shortest path to some other vertex u' followed by the edge (u', u). Then u' occurs before u in the topological sort, so it will be processed earlier, and by induction we can assume that its shortest path will be computed correctly. Therefore, a relaxation step when u' is considered will also correctly find a shortest path for u. So when u is considered its shortest path will have already been correctly computed.

Example of shortest path for dags

Difference constraints

These are inequalities of the form

$$x_j - x_i \leq b_k$$

where $1 \leq i, j \leq n$ and $1 \leq k \leq m$. Here the x_i are variables and b_k are constants. Such systems have obvious scheduling applications. These systems can be solved by shortest path algorithms, too.

For this we use the **constraint graph** $G = (V, E)$ where

$$V = \{v_0, \ldots, v_n\}$$

and

$$E = \{(v_i, v_j) : x_j - x_i \leq b_k \textit{ is a constraint}\}$$

$$\cup\{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\}.$$

Also, $w(v_i, v_j) = b_k$ if $x_j - x_i \leq b_k$ is a difference constraint, and $w(v_0, v_i) = 0$ for all i.

We have that setting $x_i = \delta(v_0, v_i)$ gives a feasible solution for the difference constraints, unless G has a negative weight cycle, in which case there is no feasible solution.

Thus we can use the Bellman-Ford algorithm to solve difference constraints.

Correctness

We need to show that $x_j - x_i \leq b_k$, that is, $\delta(v_O, v_j) - \delta(v_O, v_i) \leq b_k$. However, there is an edge (v_i, v_j) of weight b_k, which guarantees this.

If there are negative cycles, then there can be no feasible solution, because by adding inequalities we obtain that $x_j - x_j < 0$ for some j.

Example

ALL-PAIRS SHORTEST PATHS

The problem is to find the lengths of the shortest paths between u and v, for all vertices u and v of a graph.

For non-negative edge weights, repeated running of Dijkstra's algorithm is $O(V^3)$, and none of the algorithms presented here improve on this running time.

For negative edge weights, Dijkstra's algorithm cannot directly be applied. We could run the Bellman-Ford algorithm repeatedly, but this takes $O(V^2E)$ time. We can do better than this.

We present several methods for the negative edge weight case:

- A matrix multiplication based method, that is a little slow

- The Floyd-Warshall algorithm, which is $O(V^3)$

- Johnson's algorithm, which is fast for sparse graphs and eliminates negative weights, then applies Dijkstra's algorithm repeatedly

A DYNAMIC PROGRAMMIC APPROACH

Let $d_{ij}^{(m)}$ be the minimum weight of any path from vertex i to j that contains at most m edges. Then we have

$$d_{ij}^{(0)} = \begin{cases} 0 & if \ i = j \\ \infty & if \ i \neq j \end{cases}$$

$$d_{ij}^{(1)} = \begin{cases} 0 & if \ i = j \\ w_{ij} & if \ i \neq j \end{cases}$$

where w_{ij} is the weight of the edge between i and j, ∞ if there is no such edge, and

$$d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(p)} + d_{kj}^{(q)} \right\}$$

where $p + q = m$ and $m > 0$.

The reason for this: The shortest path between i and j with at most m edges must consist of an initial portion of at most p edges and a final portion of at most q edges. Let k be the vertex at the end of the initial portion. Then $d_{ij}^{(m)} = d_{ik}^{(p)} + d_{kj}^{(q)}$. Note how we use dynamic programming here.

We can use powers of two, so,

$$d_{ij}^{(2^{r+1})} = \min_{1 \le k \le n} \left\{ d_{ik}^{(2^r)} + d_{kj}^{(2^r)} \right\}$$

Also, if $m > n$, then $d_{ij}^{(m)} = d_{ij}^{(n)}$. So we just need to compute all $d_{ij}^{(1)}$, then all $d_{ij}^{(2)}$, then all $d_{ij}^{(4)}$, until we compute $d_{ij}^{(2^r)}$ where $2^r \ge n$. It suffices for $r \ge \log_2 n$, so we need $\log_2 n$ iterations. Each iteration takes work $O(V^3)$, for a total work of $O(V^3 \log_2 V)$.

The Floyd-Warshall Algorithm

WARNING: We now change notation, so that $d_{ij}^{(k)}$ is the weight of a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. Then we have

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & if\ \ k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & if\ \ k \geq 1 \end{cases}$$

The reason for this: If the shortest path between i and j using the vertices $\{1, \ldots, k\}$ passes through vertex k, it need do so only once, and thus it consists of an initial portion passing through the vertices $\{1, \ldots, k-1\}$, followed by k, followed by a final portion again passing through the vertices $\{1, \ldots, k-1\}$. Thus $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ in this case. If the shortest path between i and j using the vertices $\{1, \ldots, k\}$ does not pass through vertex k, then $d_{ij}^{(k)} = d_{ij}^{(k-1)}$. Whichever of these is smallest is the correct value to use.

The time required to compute all of the $d_{ij}^{(k)}$ values for a fixed k is $O(n^2)$. This must be done for $k = 1, 2, 3, \ldots, n$, for a total work of $O(n^3)$.

This algorithm can be applied to the **transitive closure** problem; this is, given a graph $G = (V, E)$, to compute E^*, which is defined as

$$\{(i, j) : \text{there is a path from } i \text{ to } j \text{ in } G\}$$

We can assign a weight of 1 to each edge and run the Floyd-Warshall algorithm. If there is a path from i to j, then we find out that $d_{ij} < n$, else $d_{ij} = \infty$. Thus we can compute E^* in $O(V^3)$ time. This can be done faster using bit operations and avoiding explicitly computing the distances, but still takes $O(V^3)$ time.

Johnson's Algorithm

This uses a very clever technique of *reweighting*, which changes the edge weights to eliminate negative cycles without changing the shortest paths.

First we modify G to obtain G' by adding a new vertex s with zero weight edges to all other edges in the graph. Then the shortest distances from s to other vertices are used to obtain the reweighting function.

```
JOHNSON(G)
  compute G', where V[G'] = V[G] U {s} and
      E[G'] = E[G] U {(s,v): v in V[G]}
      and w(s,v) = 0 for v in V[G]
```

```
if BELLMAN-FORD(G',w,s) = FALSE
   then print ``the graph has a negative
                               weight cycle''
   else for each vertex v in V[G']
             do set h(v) to delta(s,v)
                 computed by the Bellman-Ford
                 algorithm on G'
        for each edge (u,v) in E[G']
             do w'(u,v) <- w(u,v)
                 + h(u) - h(v)
        for each vertex u in V[G]
             do run DIJKSTRA(G,w',u) to
                 compute delta'(u,v)
                 for all v in V[G]
                for each vertex v in V[G]
                    do d_{uv} <- delta'(u,v)
                         + h(v) - h(u)
return(D)
```

Correctness of the weighting technique

We show that $w'(u,v) \geq 0$ for all vertices u,v in G. First, $h(u) = \delta(s,u)$ and $h(v) = \delta(s,v)$. Thus $w'(u,v) = w(u,v) + h(u) - h(v)$ $= w(u,v) + \delta(s,u) - \delta(s,v)$. Now, $\delta(s,v) \leq \delta(s,u) + w(u,v)$, by properties of shortest paths. Subtracting $\delta(s,v)$ from both sides, we obtain that $0 \leq \delta(s,u) + w(u,v) - \delta(s,v)$, that is, $0 \leq w'(u,v)$ as desired.

We now show that $\delta(u,v) = \delta'(u,v) + h(v) - h(u)$, where δ' is the shortest distance computed using the weights w' and δ is the shortest distance computed using the weights w.

Consider any path $p = v_0 v_1 v_2 \ldots v_k$. Adding the edge weights, the length $w(p)$ of this path p is

$$w(p) = w(0,1) + w(1,2) + \ldots + w(k-1,k).$$

Adding the modified edge weights, we obtain

$$w'(p) = w'(0,1) + w'(1,2) + \ldots + w'(k-1,k),$$

that is,

$$w(0,1)+h(0)-h(1)+w(1,2)+h(1)-h(2)+\ldots$$

$$+w(k-1,k) + h(k-1) - h(k).$$

These terms cancel out, for the most part, giving finally

$$w'(p) = w(p) + h(0) - h(k).$$

Therefore, the shortest path is unchanged; also, $w(p) = w'(p) - h(0) + h(k)$, implying that $\delta(u,v) = \delta'(u,v) - h(u) + h(v)$.

Example

The running time is $O(V^2 \log V + VE)$ if a Fibonacci heap is used, $O(VE \log V)$ if a binary heap is used, which is asymptotically faster than the Floyd-Warshall algorithm for sparse graphs.

MAXIMUM FLOW

- Motivation
- Ford-Fulkerson algorithm
- Breadth-first search variant
- Application to maximum matching
- Preflow push method (not covered)
- Lift-to-front algorithm (not covered)

A **flow network** $G = (V, E)$ is a directed graph in which each edge (u, v) has a nonnegative **capacity** $c(u, v) \geq 0$. There is also a **source** s and a **sink** t. A **flow** is a function from pairs of vertices to the reals such that

- $f(u, v) \leq c(u, v)$ all u, v (The capacity constraint)

- $f(u, v) = -f(v, u)$ (Skew symmetry)

- For all u other than s and t,

  $$\sum_{v \in V} f(u, v) = 0 \text{ (Flow conservation)}$$

$f(u, v)$ is called the **net flow** from u to v. The **value** $|f|$ of a flow f is

$$\sum_{v \in V} f(s, v)$$

The **maximum flow problem** is, given G, s, and t, to find a flow f of maximum value.

Example and motivation.

Multiple sources and sinks.

Implicit summation notation: For sets X and Y of vertices,

$$f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$$

$$f(s,Y) = \sum_{y \in Y} f(s,y)$$

$$c(X,Y) = \sum_{x \in X} \sum_{y \in Y} c(x,y)$$

Lemma 27.1 Let $G = (V, E)$ be a flow network, and let f be a flow in G. Then, for $X \subseteq V$,

$$f(X, X) = 0.$$

For $X, Y \subseteq V$,

$$f(X, Y) = -f(Y, X).$$

For $X, Y, Z \subseteq V$ with $X \cap Y = \phi$,

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

and

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y).$$

Sample proof with summation notation:

Theorem. $|f| = f(V, t)$.

Proof.

$$|f| = f(s, V) = f(V, V) - f(V - s, V) = f(V, V - s)$$

$$= f(V, t) + f(V, V - s - t) = f(V, t)$$

Another useful fact often used in the text: If S is a set not including s or t, then $f(S, V) = 0$ (by flow conservation).

Ford-Fulkerson Method

- Start with initial flow of zero
- Find an augmenting path
- Augment the flow along the path

The **residual capacity** $c_f(u,v)$ of pair (u,v) of vertices is defined by

$$c_f(u,v) = c(u,v) - f(u,v).$$

The **residual network** is $G_f = (V, E_f)$ where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}.$$

This may have edges not present in the original graph (going in the reverse direction). The residual network is also a flow network.

Example.

Additivity of flows.

An **augmenting path** p is a simple path from s to t in the residual network G_f. The **residual capacity** of p is

$$c_f(p) = \text{min}\{c_f(u,v) : \text{edge } (u,v) \text{ is on } p\}.$$

There is a flow in G_f of value $c_f(p)$ if p is an augmenting path in G. This flow can be added to f, obtaining a larger flow.

Example.

MAX-FLOW MIN-CUT THEOREM

A **cut** (S, T) of a flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the **net flow** across the cut (S, T) is defined to be $f(S, T)$. The **capacity** of the cut is $c(S, T)$. (Note the use of the summation notation here.)

Example.

Lemma 27.5 Let f be a flow in a flow network G with source s and sink t. Let (S,T) be a cut of G. Then the net flow $f(S,T)$ across (S,T) is $|f|$.

Proof.

$f(S,T) = f(S,V) - f(S,S)$ (summation notation)

$= f(S,V)$ (by skew symmetry)

$= f(s,V) + f(S-s,V)$ (summation notation)

$= f(s,V)$ (flow conservation)

$= |f|$ (definition of $|f|$).

Thus the value of f is bounded by the capacity of any cut.

Example.

Theorem 27.7 (slightly modified) The value of a maximum flow f in G is equal to the minimum capacity of any cut (S, T) of G.

Proof. By lemma 27.5, the value of f cannot be larger than $c(S, T)$. Also, if f is a maximum flow, then there can be no augmenting paths. In this case, we obtain a cut (S, T) with capacity equal to $|f|$, as follows: Let S be the vertices v such that there is a path from s to v in G_f. This cannot include t. Let T be the other vertices. Then (S, T) is a cut. Also, if $u \in S$ and $v \in T$, then $f(u, v) = c(u, v)$ (or else (u, v) would be in the residual graph E_f). Thus $f(S, T) = c(S, T)$. But we just proved in lemma 27.5 that $f(S, T) = |f|$. Thus $|f| = c(S, T)$.

If no augmenting path for f exists, then f is a maximum flow.

```
FORD-FULKERSON(G,s,t)
    for each edge (u,v) in E[G]
        do f[u,v] <- 0
           f[v,u] <- 0
    while there exists a path p from s to t
          in the residual network G_f
        do c_f(p) <- min{c_f(u,v): (u,v) is in p}
           for each edge (u,v) in p
               do f[u,v] <- f[u,v] + c_f(p)
                  f[v,u] <- -f[u,v]
```

Example.

Running time

Can be very long, may not even terminate. May not even converge to the right value.

For integer flows, performance is better. Running time $O(E|f^*|)$ where f^* is the flow found by the algorithm.

Bad example.

With breadth-first search we obtain the Edmonds-Karp algorithm, which runs in $O(VE^2)$ time. This can be further improved by the preflow push method and lift-to-front algorithm, which we will not cover.

Example for Edmonds-Karp algorithm.

Application to maximum matching

For bipartite graphs, add a source and sink vertex, and let all edge capacities be one; then a maximum flow gives a maximum matching.

Example.

Since any matching has cardinality at most $O(V)$, and the capacities are integers, the running time is $O(VE)$. Faster methods are known, however.

The exam may cover any topic emphasized in the lectures or homework but will not cover NP completeness.

Do you want to delay the exam till October 21 to permit the homework solutions to be discussed?

Vote about open or closed book.

Review topics for maximum flow

- General idea of Edmonds-Karp algorithm proof

- Use of breadth-first search for Edmonds-Karp and its relation to shortest paths and the running time

- Explanation of $E|f^*|$ bound

- Explanation of bound on running time for matching algorithm

- A couple of new examples for the matching algorithm

NP Completeness

- Polynomial time and its significance
- Largely independent of machine model
- Largely independent of the representation
- Not all problems are solvable in polynomial time
- NP-complete problems and their relationship to one another
- P versus NP

- Importance of the P versus NP problem

- Value of showing a problem NP-complete

- My early research in this area

Abstract problems

Abstract problem Q – relation on a binary set I of problem instances (mathematical objects) and problem solutions (mathematical objects).

Minimum spanning tree example

Decision problems – have a yes/no solution

Example – is there a minimum spanning tree with cost not larger than 100?

Optimization problems

Minimum spanning tree example

Relation of decision problems to optimization problems

Implications of NP completeness for optimization problems

Applications of algorithms covered

- Use of maximum matching for figure identification
- Use of maximum matching to match processes to processors
- Use of maximum flow to maximize throughput on a multiprocessor system

Of course, geometric algorithms (Chapter 35 and beyond) have many, many applications to graphics.

Encodings

An encoding is a mapping from abstract objects to character strings over some finite alphabet.

Examples

A **concrete problem** is a problem whose instances are sets of binary strings.

Example

An algorithm **solves** a concrete problem in time $O(T(n))$ if, given an input of length n, produces the (encoding of the) solution in at most $O(T(n))$ time.

Polynomial time solvability: $O(n^k)$ some k.

Complexity class P: decision problems solvable in polynomial time.

This definition is independent of encoding as long as one only considers encodings that are **polynomially related**.

Unary-binary as non-polynomially related encodings.

Formal language framework

- Alphabets (finite)

- Languages over an alphabet Σ

- A decision problem as the set of strings with a yes answer

- An algorithm accepts (rejects) a string (or loops)

- A language is *decided* by an algorithm

- Acceptance in polynomial time

- Decidability in polynomial time

- Definition of complexity class

Polynomial time verification

Checking (verifying) versus solving

> Example: Existence of a path between vertices s and t

Verification algorithm $A(x, y)$

Certificates y

Example – a path between two vertices verifies that they are connected. Thus x could be the connectedness question and y could be a path that verifies that two vertices are connected.

Language verified by A

The class NP

NP is the set of languages L such that there is a polynomial time verification algorithm A such that

$$x \in L \text{ iff there exists “short” } y : A(x, y) = 1$$

We say A verifies L in polynomial time.

Example – a coloring y verifies that a graph x can be colored with 4 colors. But it may be hard to find such a coloring y.

The P = NP question.

P $\subseteq$ NP: A can ignore y

Closure under complementation is also unknown – would imply that there is always a verification of non-solutions i.e. verification of non-4 colorability for problems in NP

NP-Completeness

NP-complete problems: If any one of them has a polynomial time solution then all do, and P = NP. But no one has found such a solution, nor has anyone proven that it does not exist. Many problems of practical importance are NP-complete.

These are in a sense the hardest problems in NP.

Examples

- Graph isomorphism is in NP (but maybe not NP complete); a verification is an isomorphism

- The hamiltonian cycle problem is in NP (and NP complete); a verification is a cycle

- The tautology problem is in co-NP.

A language L_1 is **polynomial-time reducible** to L_2 ($L_1 \leq_P L_2$) if there is a polynomial time computable function f from languages to languages such that for all x, $x \in L_1$ iff $f(x) \in L_2$.

If L_1 is polynomial time reducible to L_2, then an algorithm to solve L_2 can be used to obtain an algorithm to solve L_1. Also, the time required will be the same to within a polynomial. That is, L_1 cannot be much harder than L_2.

Lemma 36.3 If L_1 and L_2 are languages such that $L_1 \leq_P L_2$, then $L_2 \in P$ implies $L_1 \in P$.

Proof.

NP-completeness

A language L is **NP-compete** if

1. $L \in NP$, and

2. $L' \leq_P L$ for every $L' \in NP$

Thus every other language in NP is polynomial reducible to L. We say L is **NP-hard** if it satisfies property 2.

Theorem 36.4 If any NP complete problem is in P, then P = NP. If any NP complete problem is not in P, then no NP complete problem is in P, and P $\neq$ NP.

Thus the whole P = NP question boils down to the complexity of a single NP-complete problem (any one of them).

Cook [1971] found the first NP-complete problem and became famous for it. He showed that the satisfiability problem was NP complete. The book shows a slightly different problem to be NP-complete, that is, **circuit satisfiability**.

A circuit has **AND**, **OR**, **NOT**, and possibly other gates, with outputs of some gates connected to inputs of others in a directed acyclic graph (no cycles). The circuit satisfiability problem is to determine whether there is an input to the circuit leading to an output of 1 (true). The book shows this to be NP-complete using a slightly informal argument.

Review the argument.

Once we have one NP-complete problem we can obtain more using the following lemma:

Lemma 36.8 If L' is NP-complete and $L' \leq_P L$, then L is NP-hard. If L is also in NP, then L is NP-complete.

Proof.

Five-step recipe for showing NP-completeness of L:

1. Prove $L \in NP$

2. Select NP-complete L'

3. Find a reduction f from L' to L

4. Prove $x \in L'$ iff $f(x) \in L$

5. Prove that f is polynomial time computable

Other problems shown to be NP-complete

- Formula satisfiability
- 3-CNF satisfiability
- The clique problem
- The vertex cover problem (minimize size of cover)
- The subset sum problem

- The hamiltonian cycle problem

- The traveling salesman problem

- Graph coloring

and many, many more ...

In fact, some journals no longer accept NP completeness proofs, since there are so many of them and they are now becoming routine and uninteresting – of course, it still helps a lot to know how hard a problem is!

Many of the reductions are uninteresting, too.

Proof that formula satisfiability is NP complete (follow the 5 step recipe). Reduce from circuit satisfiability.

Idea for avoiding duplication of subformulas

Suppose we have the circuit

```
                 |
                NOT
                 |
                 |
                 OR   .
             .     .        .
          .         .             .
       .             .                 .
    AND              AND              AND
   /   \            /   \            /   \
  /     \          /     \          /     \
X1       X2       X3      X4       X5       X6
```

The reduction to NOT(OR(AND(X1, X2), AND(X3, X4), AND(X5, X6))) won't always work because we can have exponential size increase due to duplicated subformulas, making the reduction non polynomial time, as in this example:

```
   OR
  /  \
AND  AND
 | \/ |
 | /\ |
 OR  OR
 | \/ |
 | /\ |
AND  AND
 | \/ |
 | /\ |
 OR  OR
 | \/ |
 | /\ |
AND  AND
 | \/ |
 | /\ |

   ...
```

So we have to add new variables for the wires, as follows:

```
                    Z5|
                     NOT
                    Z4|
                      |
                      OR    .
                  .      .         .
         Z1   .       Z2.              .Z3
           .             .                  .
        AND              AND                AND
       /   \            /   \              /   \
      /     \          /     \            /     \
    X1        X2      X3       X4       X5        X6
```

Then we create formulas relating the values of the variables, as follows:

$$Z1 \equiv AND(X1, X2)$$

$$Z2 \equiv AND(X3, X4)$$

$$Z3 \equiv AND(X5, X6)$$

$$Z4 \equiv OR(Z1, Z2, Z3)$$

$$Z5 \equiv NOT(Z4)$$

We then concatenate them all together, with Z5, as follows:

$(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4)) \wedge (Z3 \equiv AND(X5, X6)) \wedge (Z4 \equiv OR(Z1, Z2, Z3)) \wedge (Z5 \equiv NOT(Z4)) \wedge Z5$

Note that Z5 is added on, too. In this way, a circuit $\mathcal{C}$ is mapped on to a formula $\mathcal{F}$. This is the reduction. We need to show it is polynomial time (easy), and preserves solvability. (Also not difficult.)

We now show that 3 CNF is NP complete by reducing from formula satisfiability. An example of a formula in 3 CNF is the following:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_2 \vee x_1)$$

The general steps in the reduction are as follows:

- All Boolean connectives as binary
- New variables for subformulas
- Reduce to conjunctive normal form
- Add variables to clauses that are too small

We illustrate these steps on the formula obtained above: $(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4)) \wedge (Z3 \equiv AND(X5, X6)) \wedge (Z4 \equiv OR(Z1, Z2, Z3)) \wedge (Z5 \equiv NOT(Z4)) \wedge Z5$

First we make all connectives binary:

$(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4))$
$\wedge\ (Z3 \equiv AND(X5, X6))\ \wedge$
$(Z4 \equiv OR(Z1, OR(Z2, Z3))) \wedge (Z5 \equiv NOT(Z4))$
$\wedge\ Z5$

Then we add new variables for subformulas so that the number of variables in each conjunct is 3 or less. For our formula that is already true for each conjunct except one, so we obtain this formula:

$$(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4)) \wedge (Z3 \equiv AND(X5, X6)) \wedge (Z4 \equiv OR(Z1, Y1)) \wedge (Y1 \equiv OR(Z2, Z3)) \wedge (Z5 \equiv NOT(Z4)) \wedge Z5$$

Next we put each subformula in conjunctive normal form, obtaining:

$$(Z1 \supset AND(X1, X2)) \wedge (Z1 \subset AND(X1, X2)) \wedge (Z2 \supset AND(X3, X4)) \wedge (Z2 \subset AND(X3, X4)) \wedge (Z3 \supset AND(X5, X6)) \wedge (Z3 \subset AND(X5, X6)) \wedge (Z4 \supset OR(Z1, Y1)) \wedge (Z4 \subset OR(Z1, Y1)) \wedge (Y1 \supset OR(Z2, Z3)) \wedge (Y1 \subset OR(Z2, Z3)) \wedge (Z5 \supset NOT(Z4)) \wedge (Z5 \subset NOT(Z4)) \wedge Z5$$

Continuing, we obtain:

$(\neg Z1 \vee X1) \wedge (\neg Z1 \vee X2) \wedge (\neg X1 \vee \neg X2 \vee Z1) \wedge$
$(\neg Z2 \vee X3) \wedge (\neg Z2 \vee X4) \wedge (\neg X3 \vee \neg X4 \vee Z2) \wedge$
$\ldots \wedge Z5$

The problem now is that some of the disjunctions have too few variables. So we add extra variables to them; for example, we replace $(\neg Z1 \vee X1)$ by $(\neg Z1 \vee X1 \vee W) \wedge (\neg Z1 \vee X1 \vee \neg W)$ for a new variable W. This has to be done twice if we start with just one variable.

After all these steps, we obtain a 3 CNF formula that is satisfiable iff the original formula was satisfiable. Also, the transformation is polynomial time.

Proof that the clique problem is NP complete:

Reduce from 3 CNF; suppose there are k clauses $C_1, \ldots, C_k$. Create a graph G for the clique problem, as follows:

- Three vertices for each clause, one for each literal

- Edges between vertices in different clauses if the literals are not complementary

- A k clique exists iff S is satisfiable

Proof that the vertex cover problem is NP complete.

Reduce from the clique problem, using graph complement.

Suppose G has a clique of size k. Let $\overline{G}$ be the complementary graph, with an edge between (u, v) iff G does not have this edge. Then G has a clique of size k iff $\overline{G}$ has a vertex cover of size $V - k$.

Example of vertex cover reduction:

```
G:    a---b
      |\ /|\
      |   | \
      |/ \|  e
      c---d
```

G has a clique of size 4

G complement:

```
      a    b
        ' .
           ' e
         . '.'
      c    d
```

G complement has a vertex cover of size 1

Proof that the subset sum problem is NP complete:

Reduce from the vertex cover problem.

Create base 4 numbers, with a digit position for each edge in the graph. Each vertex has ones for digits of its incident edges. Each edge has a one in its own position. Then we want to sum so that there is a 2 in every position. Thus every position has to appear twice. This cannot come only from the edge digits, so there must be an incident vertex. But if there is only one incident vertex, then the edge has to be included, too; otherwise it is not. Also, the total number of vertices is controlled by a significant digit.

Example of the subset sum construction:

```
    e1
  v1--v2
  |    |
e2|    |e3
  v3--v4
    e4
```

This has a vertex cover of {v1,v4}. The subset sum instance:

```
    4^4    4^2    4^0
      4^3    4^1


       e4  e3  e2  e1  Value
v1:  1  0   0   1   1    4^4 + 4^1 + 4^0
v2:  1  0   1   0   1    4^4 + 4^2 + 4^0
v3:  1  1   0   1   0    4^4 + 4^3 + 4^1
v4:  1  1   1   0   0    4^4 + 4^3 + 4^2
e1:  0  0   0   0   1    4^0
e2:  0  0   0   1   0    4^1
e3:  0  0   1   0   0    4^2
e4:  0  1   0   0   0    4^3
---------------------
    (2) 2   2   2   2    2*4^4 + 2*4^3
                         + 2*4^2 + 2*4^1
                         + 2*4^0
```

So we obtain the following subset sum problem: Is there a subset of the integers $\{4^4 + 4^1 + 4^0, 4^4 + 4^2 + 4^0, 4^4 + 4^3 + 4^1, 4^4 + 4^3 + 4^2, 4^0, 4^1, 4^2, 4^3\}$ that adds to $2 * 4^4 + 2 * 4^3 + 2 * 4^2 + 2 * 4^1 + 2 * 4^0$? Or, is there a subset of the integers $\{261, 273, 324, 336, 1, 4, 16, 64\}$ that sums to 682? And the answer is yes, the subset corresponding to $\{v1, v4, e4, e3, e2, e1\}$, that is, $\{261, 336, 1, 4, 16, 64\}$, and one can verify that $261 + 336 + 1 + 4 + 16 + 64 = 682$. We add enough edges (one or two in each column) to insure that each column has two ones altogether.

NP completeness of the Hamiltonian circuit problem:

Very involved.

The idea is that one constructs A-widgets, that permit exactly one of two edges to be traversed, and B widgets, that permit any proper subset of three edges (but not all of them) to be traversed in a Hamiltonian cycle. Using these, we can convert a 3 satisfiability problem into a graph that has a Hamiltonian cycle iff the set of clauses is satisfiable.

NP completeness of the traveling salesman problem:

Reduce from the Hamiltonian circuit problem. Convert a graph G to a graph G' with edge costs $c(i,j)$. G' is a complete graph. Let the cost be zero if edge (i,j) is in G, one otherwise. Then it is easily seen that G' has a traveling salesman tour of cost 0 iff G has a Hamiltonian circuit.

The Turing Machine Approach

The class NP is often defined in terms of nondeterministic Turing machines instead of polynomial time verifiers, so you should be familiar with this approach, too. A Turing machine has an infinite tape on which the input is written. It has a finite set of states, and can read, write, and move to the left or right, while changing state. It can also be nondeterministic.

The class NP is defined as the class of problems that can be accepted by nondeterministic Turing machines in polynomial time. That is, there is *some* sequence of transitions that leads to an accepting state. One can show that this is equivalent to the verifier definition given in the text.

This permits an exponential number of states and all states are processed in parallel. For example, if there are 10 electrons and each are in 2 states simultaneously, then the whole system has 2^{10} states. It is known that integers can be factored in polynomial time on quantum computers (which have never been built), but this is not known for conventional computers. It could even be that problems in NP are solvable in polynomial time on quantum computers, though this hasn't been shown. If this were true, it might make the P = NP question of secondary importance.

Approximation Algorithms
Chapter 37

Even if a problem is NP complete, we can find solutions that are close to optimal in polynomial time in many cases. Sometimes one can prove that even this is impossible if P $\neq$ NP.

One can bound the **ratio** of the cost of a solution to the cost of an optimal solution, or the **relative error**, as a function of n, the size of the input. One looks for an approximation algorithm that runs in polynomial time and has a specified ratio, for example.

A **polynomial-time approximation scheme** is an algorithm that accepts as an input a problem instance and a relative error ϵ and runs in polynomial time and produces a solution within ϵ of optimal. (May not exist.)

A **fully polynomial-time approximation scheme** has a running time that is polynomial both in n and in $1/\epsilon$. This guarantees that one can improve the accuracy of the approximation without too much extra work. (May not exist, too.)

The Vertex Cover Problem

For this we can obtain a ratio of two to an optimal solution in polynomial time by a very simple algorithm.

```
APPROX-VERTEX-COVER(G)
    C <- 0
    E' <- E[G]
    while E' .ne. 0
         do let (u,v) be an arbitrary edge of E'
            C' <- C U {u,v}
            remove from E' every edge incident
                   on either u or v
    return C
```

Example.

Proof of approximation.

The Traveling Salesman Problem

We obtain a ratio of two to an optimal solution in polynomial time assuming the **triangle inequality**:

$$c(u, w) \leq c(u, v) + c(v, w).$$

The approximation algorithm makes use of minimum spanning trees.

```
APPROX-TSP-TOUR(G,c)
    select a vertex r in V[G] as a root
    grow a MST T for G from r using
        Prim's algorithm
    Let L be the list of vertices of T
        visited in preorder
    return the Hamiltonian cycle that
        visits the vertices in the order L
```

Example and proof of approximation.

Without the triangle inequality, no polynomial time approximation algorithm is possible unless P = NP:

The idea is to take a graph G and produce another graph G' in which the costs of the edges in G are 1 but the costs of edges not in G are very large. Then an approximation algorithm for the TSP G' can be used to test if G has a Hamiltonian circuit.

Example.

The Set Covering Problem

Here we obtain an approximation algorithm with a logarithmic bound.

An instance $(X, \mathcal{F})$ of the set covering problem consists of a finite set X and a family $\mathcal{F}$ of subsets of X such that every element of X belongs to at least one subset in $\mathcal{F}$. We want to find a minimum size subset $\{S_1, S_2, \ldots, S_k\}$ of elements of $\mathcal{F}$ such that $S_1 \cup S_2 \cup \ldots S_k = X$. That is, k should be as small as possible.

The approximation algorithm is greedy:

```
GREEDY-SET-COVER(X,F)
    U <- X
    C <- 0
    while U .ne. 0
        do select an S in F that
           maximizes |S /\ U|
           U <- U - S
           C <- C U {S}
    return(C)
```

The ratio obtained is $H(max\{|S| : S \in \mathcal{F}\})$ where $H(d) = 1 + 1/2 + 1/3 + \ldots + 1/d$.

Example.

Proof of ratio.

The Subset Sum Problem

This is interesting because a fully polynomial approximation scheme is known. For this we want to find a subset of a set S of integers which sums to t (or as close as possible to t), where S and t are given as input. However, it is not permitted for the sum to exceed t.

First we give an exact (exponential) algorithm:

```
EXACT-SUBSET-SUM(S,t)
    n <- |S|
    L_0 <- <0>
    for i <- 1 to n
        do L_i <-
               MERGE-LISTS(L_(i-1),L_(i-1)+x_i)
           remove from L_i all elements
                                   larger than t
    return the largest element in L_n
```

This returns a sum as large as possible but not larger than t.

The idea of the approximation algorithm is to remove from the lists elements that are close to one another in value:

```
APPROX-SUBSET-SUM(S,t,epsilon)
    n <- |S|
    L_0 <- <0>
    for i <- 1 to n
        do L_i <-
               MERGE-LISTS(L_(i-1),L_(i-1)+x_i)
           L_i <- Trim(L_i,epsilon/n)
           remove from L_i all elements
                           larger than t
    return the largest element in L_n
```

Here we use TRIM defined as follows:

```
TRIM(L, delta)
    m <- |L|
    L' <- <y_1>
    last <- y_1
    for i <- 2 to m
         do if last < (1 - delta)y_i
                then append y_i onto
                        the end of L'
                     last <- y_i
    return L'
```

Example.

Proof of approximation.

Matrix Operations, Chapter 31

We will present ways of multiplying matrices asymptotically faster than the obvious method, and algorithms for some other matrix problems, too.

- Matrix

- Transpose of a matrix

- Vector (vertical, column vector)

- Square matrix

- Diagonal matrix

- Identity matrix

- Upper triangular matrix
- Lower triangular matrix
- Permutation matrix (one 1 in each row and column)
- Symmetric matrix

Matrix operations

- Matrix addition

- Scalar multiple of a matrix

- Matrix subtraction

- Matrix multiplication. Requires time $\Theta(n^3)$ on $n \times n$ matrices. Associative and distributes over addition, but not commutative.

- Ax for matrix A and vector x

- Inner product $x^T y$ for vectors x, y

- Outer product xy^T for vectors x, y

- Euclidean norm $||x||$

+ +

Matrix inverses, ranks, and determinants

- Inverse of an $n \times n$ matrix

- Singular or noninvertible matrices have no inverses

- Linear dependence: x_i are linearly dependent iff there exist real c_i, not all zero, such that $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n = 0$

- Linear independence

- Column rank, row rank, rank

- Determinant $det(A)$ of a matrix A

An $n{\times}n$ matrix A is **positive definite** if $x^T Ax > 0$ for all n-vectors x different from 0.

Strassen's Algorithm for Matrix Multiplication

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Conventional method:

$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned}$$

This gives $\Theta(n^3)$ work.

Faster method:

$$
\begin{aligned}
&P_1 = a \cdot (g - h)\\
&P_2 = (a + b) \cdot h\\
&s = P_1 + P_2\\
&P_3 = (c + d) \cdot e\\
&P_4 = d \cdot (f - e)\\
&t = P_3 + P_4\\
&P_5 = (a + d) \cdot (e + h)\\
&P_6 = (b - d) \cdot (f + h)\\
&r = P_5 + P_4 - P_2 + P_6\\
&P_7 = (a - c) \cdot (e + g)\\
&u = P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

Running time (used recursively) is $\Theta(n^{\lg 7})$ or $\Theta(n^{2.81})$, faster than $\Theta(n^3)$. The constant is high, though. The best known bound is $\Theta(n^{2.376})$.

It's probably better to use the conventional matrix multiplication unless the matrices are large, and there may also be error problems with Strassen's method and similar methods for real number arithmetic.

Boolean matrix multiplication

(This is from a starred section) Boolean matrices can also be multiplied in $O(M(n))$ arithmetic operations. This is done by treating 0 and 1 as integers, multiplying the integer matrices quickly, then replacing all non zero elements by one. One can do all computations modulo $n+1$ to prevent intermediate elements from growing large.

(The succeeding material is, I assume, covered in other courses; from now, on we will be somewhat brief.)

Solving systems of linear equations

Consider a set of n equations in n unknowns:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\cdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

This can be rewritten as

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{pmatrix}$$

Thus, we have reformulated the problem as follows:

> Given matrix A and vector b, find vector x such that $Ax = b$.

We assume that A is nonsingular. One way to solve for x is to compute A^{-1} and let x be $A^{-1}b$. This is numerically unstable and slower than necessary.

A better way is essentially Gaussian elimination. This is to compute the LUP decomposition of A and use it to find x.

The LUP decomposition of A is defined as follows: L, U, and P are three $n \times n$ matrices such that P is a permutation matrix, L is unit lower triangular, and U is upper triangular, and such that

$$PA = LU.$$

Every nonsingular $n \times n$ matrix has an LUP decomposition.

This decomposition can be used to solve for x by solving two triangular systems. If $Ax = b$, then $PAx = Pb$, so $LUx = Pb$. We can compute Pb easily. We solve $Ly = Pb$ for y by forward substitution, since L is lower triangular. Then we solve $Ux = y$ for x by back substitution, since U is upper triangular. It follows that x is the desired solution.

We illustrate solving $Ly = Pb$. We have a system of this form:

$$\begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \cdots \\ b'_n \end{pmatrix}$$

This translates to equations of the form

$$(*)x_1 + 0x_2 + \cdots + 0x_n = b'_1$$

$$(*)x_1 + (*)x_2 + \cdots + 0x_n = b'_2$$

$$\cdots$$

$$(*)x_1 + (*)x_2 + \cdots + (*)x_n = b'_n$$

which can be easily solved – first for x_1, then for x_2 et cetera.

Solving $Ux = y$ is similar except we have to start at the bottom and work up. Both processes take $\Theta(n^2)$ time.

The problem now is to obtain the LUP decomposition. One can obtain an LUP decomposition by **Gaussian elimination**. This takes time $\Theta(n^3)$ for an $n \times n$ matrix.

The idea of this is as follows. Consider the matrix A again:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

We can subtract multiples of the top row from the other rows to make all a_{i1} zero except a_{11}:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & (*) & \cdots & (*) \\ \cdots & \cdots & \cdots & \cdots \\ 0 & (*) & \cdots & (*) \end{pmatrix}$$

This can be done with the second column, too, and continued to make all below diagonal elements zero. This gives the upper triangular matrix U. The lower triangular matrix L comes from remembering the transformation that must be applied to b to do this, since you also have to subtract some multiple of b_1 from b_i for $i > 1$. Also, one sometimes has to "pivot" to guarantee nonzero elements on the diagonal (to avoid division by zero), so one also gets a permutation matrix P. In fact, this all can be done "in place," overwriting A in the process, to save storage.

Given an LUP decomposition of A, the inverse of A can be computed in an additional $\Theta(n^3)$ time.

To do this, we solve the matrix equation

$$AX = I_n$$

for X, which then must be the inverse of A, since I_n is the identity matrix. This reduces to n equations of the form

$$AX_i = e_i$$

where X_i is the i^{th} column of X and e_i is the i^{th} column of I_n, that is, e_i has a 1 in the i^{th} row and zeroes elsewhere.

Each such equation can be solved using the LUP decomposition of A, which only needs to be computed once. Each equation solving takes $\Theta(n^2)$ time, and there are n of them to do, for an accumulated time of $\Theta(n^3)$. The LUP decomposition also takes $\Theta(n^3)$ work, for a total bound of $\Theta(n^3)$ to compute the inverse of A.

It is then shown that the asymptotic time required to invert a matrix is the same as the time required to multiply two matrices (assuming reasonable regularity conditions on the time bounds). Thus we can also invert a matrix in time $\Theta(n^{2.81})$ (or even in time $\Theta(n^{2.376})$. Thus we could in principle solve sets of linear equations this fast.

To show that matrix multiplication and matrix inversion have the same complexity, let D be the $3n \times 3n$ matrix

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Then it is claimed that the inverse of D is

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

so if we invert D we are computing the product AB, too. (Check this inverse claim.) This requires the same asymptotic time bound as matrix multiplication, too. Thus if we could find a fast way to invert a matrix, we could also use it to multiply matrices.

We now show that matrix inversion can be done using matrix multiplication. Suppose A is an $n \times n$ matrix and n is a power of two. (This does not affect the argument.) Suppose A is symmetric and positive definite. Write A as follows:

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}.$$

Let S be $D - CB^{-1}C^T$, which is called the *Schur complement* of A with respect to B. Then it is claimed that

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^T S^{-1} C B^{-1} & -B^{-1}C^T S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}.$$

(Check this!)

Thus, to compute the inverse of A, we have to invert B and S recursively, and then perform these multiplications:

$$C \cdot B^{-1}$$

$$(CB^{-1}) \cdot C^T$$

$$S^{-1} \cdot (CB^{-1})$$

$$(CB^{-1})^T \cdot (S^{-1}CB^{-1}).$$

This is four multiplications, so

$$I(n) \le 2I(n/2) + 4M(n) + O(n^2)$$

$$= 2I(n/2) + O(M(n))$$

$$= O(M(n)).$$

(by the Master Theorem).

If A is not symmetric and positive definite, we use the fact that $A^{-1} = (A^T A)^{-1} A^T$. (Why?) Also, $A^T A$ is symmetric and positive definite, so the preceding method can be applied to it. The total time is still $O(M(n))$.

Finally, it is shown how to do least-squares approximation using symmetric positive definite matrices. The following results are proved:

Any symmetric positive-definite matrix is nonsingular.

Leading submatrices of symmetric positive definite matrices are symmetric and positive definite.

Schur complements of symmetric positive-definite matrices are symmetric and positive definite.

LU decomposition of a symmetric positive-definite matrix never causes a division by zero.

For least squares approximation, we are given a non-square matrix A and a vector y, and want to choose vector c so that $Ac - y$ is as small as possible (in the Euclidean norm). We assume y has more elements than c here.

It turns out that the best c satisfies the equation

$$A^T Ac = A^T y.$$

The solution c is then

$$((A^T A)^{-1} A^T)y,$$

which is $A^+ y$ where A^+ is defined to be

$$(A^T A)^{-1} A^T.$$

This is called the *pseudoinverse* of A. To find c, we compute $A^T y$ and then solve for c by taking an LU decomposition of $A^T A$.

Least-squares approximations are common in statistics, where one wants to eliminate the "noise." This also has obvious applications to graphics, where one may want to reconstruct a shape from a noisy image. Or, one might want to eliminate the noise from a transmitted scene or sound track.

Polynomials and the Fast Fourier Transform

Many periodic real functions $f(x)$ can be expressed as an infinite sum

$$\sum_{j=0}^{\infty}(a_j \sin(2\pi jx) + b_j \cos(2\pi jx))$$

where a_j and b_j are real numbers. (This would be for a function f having a period of 1). This can be viewed as expressing the function f as a sum of sines and cosines. In essence, we are expressing a complicated function in terms of very simple ones. For example, a musical sound can be expressed in terms of harmonics, or pure tones. This is called a *Fourier transform*. This can be very useful, since some operations on signals are much easier in this *frequency domain* than in the *time domain*.

So there is some interest in computing a_j and b_j, given f. This can be done using integration, since the functions

$$sin(2\pi jx)$$

and

$$cos(2\pi jx)$$

are *orthogonal*. However, this can be done much faster using the **Fast Fourier transform**, which we will now study.

Instead of dealing with sines and cosines, we will deal with complex numbers, using the remarkable formula

$$e^{ix} = \cos(x) + i\sin(x).$$

To justify this, we recall the Taylor series for a function f:

$$f(x) = f(0) + xf'(0) + \frac{x^2}{2}f''(0) + \frac{x^3}{6}f'''(0) + \cdots$$

Since e^x has all derivatives equal to itself, and $e^0 = 1$, we have

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots$$

Also, $\frac{d}{dx}\sin(x) = \cos(x)$, $\frac{d}{dx}\cos(x) = -\sin(x)$, $\frac{d}{dx} - \sin(x) = -\cos(x)$, $\frac{d}{dx} - \cos(x) = \sin(x)$, and $\sin(0) = 0$ and $\cos(0) = 1$. Thus, by the Taylor series,

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

and

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Recall that $i = \sqrt{-1}$, so that $i^2 = -1$ and $i^3 = -i$ and $i^4 = 1$.

Thus we obtain that

$$
\begin{aligned}
e^{ix} &= 1 + ix + \frac{(ix)^2}{2} + \frac{(ix)^3}{6} + \frac{(ix)^4}{24} + \cdots \\
&= 1 + ix + \frac{i^2x^2}{2} + \frac{i^3x^3}{6} + \frac{i^4x^4}{24} + \cdots \\
&= 1 + ix - \frac{x^2}{2} - i\frac{x^3}{6} + \frac{x^4}{24} + \cdots \\
&= (1 - \frac{x^2}{2} + \frac{x^4}{24} + \cdots) + i(x - \frac{x^3}{6} + \cdots) \\
&= \cos(x) + i\sin(x)
\end{aligned}
$$

This formula is useful for deriving some trigonometric identities. Suppose you want to find another formula for $\sin(x) \cdot \cos(x)$. We note that

$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2}$$

and multiplying exponentials is easy, so we obtain a formula for $\sin(x) \cdot \cos(x)$ in terms of $\cos(2x)$ and $\sin(2x)$.

We also note that if $\omega = e^{2\pi i/n}$, then $\omega^n = e^{2\pi i} = \cos(2\pi) + i\sin(2\pi) = 1 + 0 = 1$. Thus $\omega^n = 1$, and such an ω is denoted by ω_n and called an n^{th} **root of unity**. In fact, any integer α such that $\alpha^n = 1$ is called an n^{th} root of unity, and these are all of the form ω_n^k for integers k between 0 and $n-1$.

So, with these preliminaries, instead of talking about sines and cosines, we discuss exponential functions of complex numbers, and instead of talking about arbitrary periodic functions $f(x)$, we discuss polynomials over the reals. Within this context, we can use a fast Fourier transform to multiply two polynomials quickly, and to interpolate. The whole discussion, however, could be phrased in terms of expressing a periodic function as a sum of pure frequencies.

A *polynomial* in the variable x over an algebraic field F (such as the complex numbers) is a function $A(x)$ as follows:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

The number n is called the *degree bound* and $a_0, a_1, a_2, \cdots$ are called the *coefficients*. The coefficients are from F, which is typically the complex numbers. The *degree* of A is the index of its highest nonzero coefficient.

We define *addition* and *multiplication* on polynomials as follows: Let

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

$$C(x) = \sum_{j=0}^{n-1} c_j x^j$$

We say $C(x) = A(x) + B(x)$ if $c_j = a_j + b_j$ for all j and $C(x) = A(x) \cdot B(x)$ if $c_j = \sum_{k=0}^{j} a_k b_{j-k}$ for all j.

The *coefficient representation* of a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

is the sequence $(a_0, a_1, a_2, \cdots)$ of coefficients, and the *point-value representation* is a set of n (distinct) *point-value pairs*

$$\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$$

such that $y_k = A(x_k)$.

We can add two polynomials in linear time using the coefficient representation. We can also evaluate a polynomial at a point x (that is, compute $A(x)$) in linear time using Horner's rule. Multiplying polynomials in coefficient form takes $\Theta(n^2)$ time, using the straightforward algorithm. Polynomials in point-value form can be multiplied in linear time. So there is a lot of interest in converting from coefficient form to point-value form and back, to find a faster polynomial multiplication algorithm.

The book proves that for any set $\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs, there is a unique polynomial $A(x)$ of degree bound n such that $y_k = A(x_k)$ for $k = 0, 1, 2, \cdots, n-1$. The problem of determining a polynomial from a point-value representation is called **interpolation**. (Motivate this.)

We will show how to convert between these two forms quickly and thus to multiply polynomials A and B of degree bound n quickly as follows:

- Add zeroes to make A and B of degree bound $2n$
- Compute point-value representations of A and B at the points ω_{2n}^k
- Compute a point-value representation of C by pointwise multiplication.
- Convert the point-value representation of C back to coefficient form

The points ω_{2n}^k are called the $2n^{th}$ *roots of unity*. The second step can be done in $O(n \log(n))$ time using the fast Fourier transform. The last step can be done by another application of fast Fourier transform, in $O(n \log(n))$ time. It's curious that the FFT is almost its own inverse.

Useful facts about complex roots of unity:

For $n \geq 0$, $k \geq 0$, $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k$$

For any even integer $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1$$

If $n > 0$ is even, then the squares of the n^{th} roots of unity are the $(n/2)^{th}$ roots of unity.

For any integer $n \geq 1$ and nonnegative integer k not divisible by n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

(Illustrate on a graph.)

The situation is something like this:

```
                 (0+1i)
                e^(i/2 pi)
                   |
                   |
                   |         e^(i/4 pi)
                   |
                   |
                   |
(-1+0i)---------(0+0i)---------(1+0i)
e^(i pi)           |             e^0
                   |             e^(2i pi)
                   |
                   |
                   |
                   |
               e^(3i/2 pi)
                 (0-1i)
```

We give some examples of ω_n for various n:

$$\begin{aligned}
\omega_1 &= e^{2i\pi/1} = \cos(2\pi) + i\sin(2\pi) = 1 + 0 = 1 \\
\omega_2 &= e^{2i\pi/2} = e^{i\pi} = \cos(\pi) + i\sin(\pi) \\
&= -1 + 0 = -1 \\
\omega_3 &= e^{2i\pi/3} = \cos(2\pi/3) + i\sin(2\pi/3) \\
&= -1/2 + i\sqrt{3}/2 \\
\omega_4 &= e^{2i\pi/4} = e^{i\pi/2} = \cos(\pi/2) + i\sin(\pi/2) \\
&= 0 + i = i \\
\omega_5 &= e^{2i\pi/5} = \cos(2\pi/5) + i\sin(2\pi/5) \\
\omega_6 &= e^{2i\pi/6} = e^{i\pi/3} = \cos(\pi/3) + i\sin(\pi/3) \\
&= 1/2 + i\sqrt{3}/2 \\
\omega_7 &= e^{2i\pi/7} = \cos(2\pi/7) + i\sin(2\pi/7) \\
\omega_8 &= e^{2i\pi/8} = e^{i\pi/4} = \cos(\pi/4) + i\sin(\pi/4) \\
&= 1/\sqrt{2} + i/\sqrt{2}
\end{aligned}$$

Recall that 2π is 360 degrees, so π is 180 degrees and $\pi/2$ is 90 degrees.

We give examples of computing powers of ω_n:

$\omega_6^2 = \omega_3$

$\omega_6^3 = \omega_2$

$\omega_8^2 = \omega_4$

$\omega_8^4 = \omega_2$

$\omega_8^8 = \omega_1 = 1$

These can be computed using complex arithmetic. For example,

$$\omega_8^2 = (1/\sqrt{2} + i/\sqrt{2})^2 =$$

$$(1/\sqrt{2} + i/\sqrt{2}) * (1/\sqrt{2} + i/\sqrt{2}) =$$

$$(1/2 + 2i/2 + i^2/2) = (1/2 + i - 1/2) = i = \omega_4.$$

Also, since $i^4 = 1$, we know that $\omega_8^8 = 1$.

As another example,

$$\omega_6^2 = (1/2 + i\sqrt{3}/2)^2$$

$$= 1/4 + 2i\sqrt{3}/4 + 3i^2/4 = 1/4 + \sqrt{3}i/2 - 3/4$$

$$= -1/2 + \sqrt{3}i/2 = \omega_3.$$

Those powers of ω_n that cannot be computed exactly like this can be computed using real number (floating point) arithmetic on complex numbers.

We wish to evaluate a polynomial $A(x)$ of degree bound n (a power of two, say) at the n complex n^{th} roots of unity. Thus we want to compute

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

for $k = 0, 1, 2, \cdots, n-1$. We call the vector $y = (y_0, y_1, \cdots, y_{n-1})$ the **discrete Fourier transform (DFT)** of $(a_0, a_1, \cdots, a_{n-1})$, the coefficient vector. We also write $y = DFT_n(a)$. The straightforward method would involve $\Theta(n^2)$ work. We can do this using the **fast Fourier transform (FFT)** as follows:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1},$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

Then we have

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2). \qquad (1)$$

Thus, to evaluate A at $\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1}$, we can evaluate $A^{[0]}(x)$ and $A^{[1]}(x)$ at the points

$$(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2 \qquad (2)$$

and then use equation (1). However, there are only $n/2$ distinct values in equation (2), so the subproblems are of half the size, saving work. Thus we have

$$T(n) = 2T(n/2) + \Theta(n)$$

which, by the Master Theorem, yields $T(n) = \Theta(n \lg n)$.

The book gives a detailed algorithm (RECURSIVE-FFT) on page 788 that implements this idea; it is fairly straightforward. A more efficient iterative version is given in section 3. A parallel circuit of depth $\Theta(\lg n)$ is also given in section 3, which would give a tremendous speedup. We will discuss these later.

Now, to multiply polynomials efficiently, we also need to interpolate quickly. In the book, the following formula is derived:

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj}$$

for $j = 0, 1, 2, \cdots, n-1$. This is the same as the FFT except:

- We switch the roles of a and y
- We use ω_n^{-1} instead of ω_n
- The final result is divided by n

Thus interpolation can be done by almost the same algorithm in $\Theta(n \lg n)$ work, too. Thus we can multiply two polynomials in $\Theta(n \lg n)$ work.

This can be expressed by the following **convolution theorem**:

> For any two vectors a and b of length n, a power of two,
>
> $$a \otimes b = DFT_{2n}^{-1}(DFT_{2n}(a) \cdot DFT_{2n}(b))$$

Here $\cdot$ denotes componentwise product, $\otimes$ denotes convolution, and a and b are padded with zeroes to length $2n$.

We want to illustrate how interpolation works. We will take a polynomial x^k with degree bound n and compute its Fourier transform. Then we will interpolate back to show that we end up with x^k again. By linearity, this will imply the result for any polynomial. Note that x^k corresponds to the coefficient vector

$$(0, 0, 0, \cdots, 1, \cdots, 0)$$

with a 1 in the k^{th} place.

We evaluate x^k at powers of ω_n to compute its Fourier transform, that is, the Fourier tranform is

$$(1, \omega_n^k, \omega_n^{2k}, \cdots, \omega_n^{(n-1)k}).$$

We then consider this as a polynomial in x, that is,

$$1 + \omega_n^k x + \omega_n^{2k} x^2 + \cdots + \omega_n^{(n-1)k} x^{n-1}.$$

We evaluate this polynomial at powers of $(\omega_n)^{-1}$ to interpolate. Now, $(\omega_n^{-1})^0 = 1$, so we evaluate first at 1 to obtain

$$1 + \omega_n^k + \omega_n^{2k} + \cdots + \omega_n^{(n-1)k},$$

which equals zero unless $k = 0$, and then it is n. Evaluation of this polynomial at $(\omega_n^{-1})^1$ gives

$$1 + \omega_n^k \omega_n^{-1} + \omega_n^{2k} \omega_n^{-2} + \cdots + \omega_n^{(n-1)k} \omega_n^{-(n-1)},$$

or,

$$1 + \omega_n^{k-1} + \omega_n^{2k-2} + \cdots + \omega_n^{(n-1)(k-1)},$$

which is zero unless $k = 1$, and then it is n.

This happens everywhere, but when we evaluate this polynomial at $(\omega_n)^{-k}$, we obtain

$$1 + 1 + 1 + \cdots + 1$$

which is n. Thus the inverse Fourier transform yields the sequence

$$(0, 0, 0, \cdots, n, \cdots, 0)$$

with the n in the k^{th} place. So when we divide by n, we obtain

$$(0, 0, 0, \cdots, 1, \cdots, 0)$$

back again, which is what we started with. So we can interpolate using the inverse Fourier transform.

Now we will do the Fourier transform and interpolation on the polynomial $1 + x - x^2$ from last class.

We now discuss the recursive FFT procedure of page 788:

```
RECURSIVE-FFT(a)
  n ← length(a)
  if n = 1
     then return a
  ω_n ← e^(2πi/n)
  ω ← 1
  a^[0] ← (a_0, a_2, ···, a_{n−2})
  a^[1] ← (a_1, a_3, ···, a_{n−1})
  y^[0] ← RECURSIVE-FFT(a^[0])
  y^[1] ← RECURSIVE-FFT(a^[1])
  for k ← 0 to n/2 − 1
       do y_k ← y_k^[0] + ω y_k^[1]

          y_{k+(n/2)} ← y_k^[0] − ω y_k^[1]
          ω ← ω ω_n
  return y
```

RECURSIVE-FFT(a)

$n \leftarrow length(a)$

if $n = 1$

then return a

$\omega_n \leftarrow e^{2\pi i/n}$

$\omega \leftarrow 1$

$a^{[0]} \leftarrow (a_0, a_2, \cdots, a_{n-2})$

$a^{[1]} \leftarrow (a_1, a_3, \cdots, a_{n-1})$

$y^{[0]} \leftarrow$ RECURSIVE-FFT$(a^{[0]})$

$y^{[1]} \leftarrow$ RECURSIVE-FFT$(a^{[1]})$

for $k \leftarrow 0$ to $n/2 - 1$

do $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$

$y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$

$\omega \leftarrow \omega\omega_n$

return y

To understand this, recall that

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2).$$

Therefore, letting x be ω_n^k,

$$A(\omega_n^k) = A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}).$$

Now, $y_k = A(\omega_n^k)$ and $y_k^{[0]} = A^{[0]}(\omega_{n/2}^k)$ and $y_k^{[1]} = A^{[1]}(\omega_{n/2}^k)$. Also, $\omega_{n/2} = \omega_n^2$. Thus $y_k^{[0]} = A^{[0]}(\omega_n^{2k})$ and $y_k^{[1]} = A^{[1]}(\omega_n^{2k})$. So, substituting in the above,

$$y_k = y_k^{[0]} + \omega_n^k y_k^{[1]}.$$

Also, we note that $\omega_n^{k+(n/2)} = -\omega_n^k$.

Let's define the procedure COMBINE-FFT($y^{[0]}, y^{[1]}$) as follows:

$$
\begin{array}{l}
n \leftarrow \text{length}(y^{[0]}) \\
\omega_n \leftarrow e^{2\pi i/n} \\
\omega \leftarrow 1 \\
\text{for } k \leftarrow 0 \text{ to } n/2 - 1 \\
\quad\quad \text{do } y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]} \\
\\
\quad\quad\quad\quad y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega y_k^{[1]} \\
\quad\quad\quad\quad \omega \leftarrow \omega\omega_n \\
\text{return } y
\end{array}
$$

We'll make use of this later.

RECURSIVE-FFT is made more efficient using the following ideas:

- We can avoid the double computation of $\omega y_k^{[1]}$ using a temporary variable;

- We can perform the recursive FFT iteratively in place in an array A

- We can make this more efficient using bit reverse code

To do the FFT in place is something like this (for $n = 8$):

$$
\begin{aligned}
&a_i \leftarrow \mathsf{FFT}(a_i) = a_i, 1 \leq i \leq 8\\
&(a_0, a_4) \leftarrow \mathsf{FFT}(orig(a_0, a_4))\\
&(a_2, a_6) \leftarrow \mathsf{FFT}(orig(a_2, a_6))\\
&(a_1, a_5) \leftarrow \mathsf{FFT}(orig(a_1, a_5))\\
&(a_3, a_7) \leftarrow \mathsf{FFT}(orig(a_3, a_7))\\
&(a_0, a_2, a_4, a_6) \leftarrow \mathsf{FFT}(orig(a_0, a_2, a_4, a_6))\\
&(a_1, a_3, a_5, a_7) \leftarrow \mathsf{FFT}(orig(a_1, a_3, a_5, a_7))\\
&(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) \leftarrow\\
&\quad \mathsf{FFT}(orig(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8))
\end{aligned}
$$

To make this more explicit, we use the procedure COMBINE-FFT:

$$
\begin{aligned}
& a_i \leftarrow \mathsf{FFT}(a_i) = a_i, 1 \le i \le 8 \\
& (a_0, a_4) \leftarrow \mathsf{COMBINE\text{-}FFT}((a_0), (a_4)) \\
& (a_2, a_6) \leftarrow \mathsf{COMBINE\text{-}FFT}((a_2), (a_6)) \\
& (a_1, a_5) \leftarrow \mathsf{COMBINE\text{-}FFT}((a_1), (a_5)) \\
& (a_3, a_7) \leftarrow \mathsf{COMBINE\text{-}FFT}((a_3), (a_7)) \\
& (a_0, a_2, a_4, a_6) \leftarrow \\
& \quad \mathsf{COMBINE\text{-}FFT}((a_0, a_4), (a_2, a_6)) \\
& (a_1, a_3, a_5, a_7) \leftarrow \\
& \quad \mathsf{COMBINE\text{-}FFT}((a_1, a_5), (a_3, a_7)) \\
& (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \leftarrow \\
& \quad \mathsf{COMBINE\text{-}FFT}((a_0, a_2, a_4, a_6), (a_1, a_3, a_5, a_7))
\end{aligned}
$$

This can be made more efficient if we store the entries in reverse binary order, which for $n = 8$ is $0, 4, 2, 6, 1, 5, 3, 7$. Reverse binary order is obtained by taking a binary sequence and reversing the bits, so we obtain 000, 100, 010, 110, ... by reversing 000, 001, 010, 011, ...

This gives the following iterative algorithm for FFT:

```
ITERATIVE-FFT(a)
  BIT-REVERSE-COPY(a,A)
  n ← length[a]
  for s ← 1 to lg n
      do m ← 2^s
         ω_m ← e^{2πi/m}
         ω ← 1
         for j ← 0 to m/2 − 1
           do for k ← j to n − 1 by m
                 do t ← ωA[k + m/2]
                    u ← A[k]
                    A[k] ← u + t
                    A[k + m/2] ← u − t
             ω ← ωω_m
  return A
```

The structure of the inner loops is a little tricky. Say m is 4. Then j goes from 0 to 1. Also, k increases from j to n-1 by 4. Thus k is first 0,4,8, ... and then 1,5,9, ... Then we compute A[k] and A[k+m/2] in terms of each other, that is, A[k] and A[k+2]. Thus we compute A[0] and A[2] together, then A[4] and A[6] together, then A[8] and A[10] together, et cetera. Then we compute A[1] and A[3] together, then A[5] and A[7] together, then A[9] and A[11] together, et cetera. Thus A[0] and A[1] are combined with A[2] and A[3], and A[4] and A[5] are combined with A[6] and A[7], et cetera. This represents combining two evaluations at the second roots of unity (-1 and 1) to obtain an evaluation of a polynomial at the fourth roots of unity (i,-1,-i,1).

It's fairly straightforward to convert this to a parallel circuit of depth $\Theta(\lg n)$, and the book does an excellent job of explaining and illustrating it. Also, the parallel circuit helps us to understand ITERATIVE-FFT.

Let's apply the fast Fourier transform (parallel version) to our problem of evaluating the polynomial $1 + x - x^2$ at the fourth roots of unity $i, -1, -i, 1$ and then inverting to obtain the original polynomial back again.

Now let's apply the parallel circuit to multiply two polynomials, say, $1 + x$ and $1 - x$. Their degree is one, and their degree bound is two. So we have to add two zero coefficients and express them as polynomials of degree bound four. Then we need to use the fourth roots of unity, that is, powers of ω_4, which is i. The powers of i are are just $i, -1, -i, 1$. We want to evaluate both polynomials at all four points, then multiply the values, then convert back.

Number Theoretic Algorithms

- An interesting area of mathematics
- Now of practical importance due to encryption algorithms
- By the way, leading computer science theoreticians recently found a way to break commercial encryption schemes using radiation and observing the error messages. This also broke the RSA scheme.
- If we get quantum computers, the game could change again, too

Size of input – number of digits in it

Measure work in terms of bit operations

Best known method for multiplying two β-digit integers has time $\Theta(\beta \lg \beta \lg \lg \beta)$ (measured in bit operations), which is polynomial. The straightforward method is $\Theta(\beta^2)$, which is still polynomial.

Fast Integer Multiplication

(Not from the book.)

Suppose we want to multiply the two binary numbers $b = b_0 + 2b_1 + \cdots + 2^n b_n$ and $c = c_0 + 2c_1 + \cdots + 2^n c_n$, where b_i and c_i are 0 or 1. Consider the polynomials $p(x) = b_0 + b_1 x + \cdots + b_n x^n$ and $q(x) = c_0 + c_1 x + \cdots + c_n x^n$. Then $b = p(2)$ and $c = q(2)$.

We can multiply to get $r(x) = p(x)q(x)$ using the Fast Fourier Transform, in $O(n \log n)$ operations. Each operation is on integers that may be $\log n$ in length, so the time per operation may be $\Theta(\log^2 n)$. This therefore takes $O(n \log^3 n)$ time. Then $r(2)$ is $b * c$.

This can be reduced by doing all operations in $\mathbf{Z}_k$ for suitable small k and precomputing in advance all sums and products. This reduces the total time to $O(n \log^2 n)$ time.

We only need to convert $r(2)$ to binary form. Suppose $r(x) = d_0 + d_1 x + \cdots + d_n x^n$. Then $r(2) = d_0 + 2d_1 + \cdots + 2^n d_n$ Each coefficient d_i of r will be between 0 and $n+1$, thus at most about $\log_2 n$ bits long. We can just do the addition $((d_0 + 2d_1) + 4d_2) + \cdots + 2^n d_n$. Since each d_i has a small number of one bits, this series of additions will give the final answer, in $O(n \log n)$ bit operations. The total time to multiply b and c is then $O(n \log^2 n)$.

By using parallel FFT circuits and divide and conquer on the final series of additions, this method of integer multiplication can be done in expected $O(\log n)$ parallel time, using the fact that carries will not usually propogate very far. This may be useful for really fast integer multiplication hardware, I don't know. By using a trick, we can reduce the expected $O(\log n)$ parallel time to worst-case parallel $O(\log n)$ time.

This can be improved a little. The idea is to break the integers into blocks of bits and apply the fast multiplication recursively. Also, one uses modular arithmetic. Still, $O(n \log^2 n)$ is not bad, and the $O(n \log^2 n)$ method may have a smaller constant factor and be faster for small values of n.

- Divisibility
- Primality
- Composites
- Quotients and remainders
- $a \equiv b \pmod{n}$
- $\mathbf{Z}_n$
- Greatest common divisors
- Relatively prime
- Unique factorization

- Euclid's algorithm for gcd
- $O(\beta^3)$ bit operations
- Extended Euclid algorithm
- Modular arithmetic
- Additive group modulo n
- Multiplicative group modulo n
- Euler's ϕ function

- Order of an element

- Solving modular linear equations

- The Chinese Remainder Theorem

- Powers of an element modulo n

- Euler's theorem

- Fermat's little theorem

- Raising to powers with repeated squaring

- RSA public-key cryptosystem

Philosophy of learning and repetition

Philosophy of thanksgiving

If a and b are any integers, not both zero, then gcd(a,b) is the smallest positive element of the set $\{ax + by : x, y \in \mathbf{Z}\}$ of linear combinations of a and b.

If $gcd(a, b) = 1$, we say that a and b are **relatively prime**. Thus 4 and 9 are relatively prime, but 4 and 10 are not.

Euclid's algorithm for gcd uses the fact that

$$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b).$$

The algorithm follows:

```
EUCLID(a,b)
  if b = 0
      then return a
      else return EUCLID(b,a mod b)
```

Example.

$$\begin{aligned}
\text{EUCLID}(0,5) &= \\
\text{EUCLID}(5, 0 \bmod 5) &= \\
\text{EUCLID}(5,0) &= \\
& 5.
\end{aligned}$$

$$\begin{aligned}
\text{EUCLID}(4,6) &= \\
\text{EUCLID}(6, 4 \bmod 6) &= \\
\text{EUCLID}(6,4) &= \\
\text{EUCLID}(4, 6 \bmod 4) &= \\
\text{EUCLID}(4,2) &= \\
\text{EUCLID}(2, 4 \bmod 2) &= \\
\text{EUCLID}(2,0) &= \\
& 2.
\end{aligned}$$

F_k is the k^{th} Fibonacci number (1,1,2,3,5,8, $\cdots$).

For any integer $k \geq 1$, if $a > b \geq 0$ and $b < F_{k+1}$, then EUCLID(a,b) makes fewer than k recursive calls.

Example where the bound is achieved:
EUCLID(F_{k+1},F_k).

Note: F_k is approximately $\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^k$.

In fact, F_k is exactly

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^k - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^k.$$

$$
\begin{aligned}
\text{EUCLID}(21,13) &= \\
\text{EUCLID}(13,8) &= \\
\text{EUCLID}(8,5) &= \\
\text{EUCLID}(5,3) &= \\
\text{EUCLID}(3,2) &= \\
\text{EUCLID}(2,1) &= \\
\text{EUCLID}(1,0) &= \\
1 &
\end{aligned}
$$

The extended form of Euclid's algorithm computes x and y such that $d = \text{gcd}(a, b) = ax + by$. This just requires some extra bookkeeping.

```
EXTENDED-EUCLID(a,b)
  if b = 0
      then return (a, 1, 0)
  (d', x', y') ← EXTENDED-EUCLID(b,a mod b)
  (d, x, y) ← (d', y', x' − ⌊a/b⌋y')
  return(d, x, y)
```

```
EXTENDED-EUCLID(6,4) :
  EXTENDED-EUCLID(4,2) :
    EXTENDED-EUCLID(2,0)
    return(2,1,0)
  return(2,0,1)
return(2,1,-1)
```

Note that $1 * 6 + (-1) * 4 = 2$.

We use $x \equiv y \pmod{n}$ to mean $x - y$ is a multiple of n, that is, x and y have the same remainder when divided by n. Thus $4 \equiv 19 \pmod{5}$. This notation will often be used in the sequel.

We use $x \bmod n$ (or sometimes $x \pmod{n}$) to denote the remainder when x is divided by n. Thus $19 \bmod 5 = 4$.

The advantage of modular notation is that one can often compute remainders without knowing the numbers themselves. For example, suppose that $x \equiv 1 \pmod 5$ and $y \equiv 2 \pmod 5$. Then we know that $x + y \equiv 3 \pmod 5$ regardless of what x and y are. Also, $xy \equiv 2 \pmod 5$.

As a justification, $x = 5a + 1$ for some a and $y = 5b + 2$ for some b. So $xy = (5a+1)*(5b+2) = 25ab + 5b + 10a + 2 = 5(5ab + b + 2a) + 2$ so $xy \equiv 2 \pmod 5$.

As another example, if $y \equiv 2 \pmod 5$, then $y^2 \equiv 4 \pmod 5$, and $y^4 \equiv 1 \pmod 5$.

Let's do another example of EXTENDED-EUCLID:

EXTENDED-EUCLID(21,13) :
  EXTENDED-EUCLID(13,8) :
    EXTENDED-EUCLID(8,5) :
      EXTENDED-EUCLID(5,3) :
        EXTENDED-EUCLID(3,2) :
          EXTENDED-EUCLID(2,1) :
            EXTENDED-EUCLID(1,0)
            return(1,1,0)
          return(1,0,1)
        return(1,1,-1)
      return(1,-1,2)
    return(1,2,-3)
  return(1,-3,5)
return(1,5,-8)

Note that $5 * 21 - 8 * 13 = 1$. In general, it appears that $F_k * F_{k+3} - F_{k+1} * F_{k+2} = \pm 1$.

Also, $5 * 21 \equiv 1 \pmod{13}$. We call 5 and 21 *multiplicative inverses* modulo 13, and EXTENDED-EUCLID can compute the multiplicative inverse of a modulo n if a and n are relatively prime (like 13 and 21).

The additive group modulo n is $\mathbf{Z}_n$, and has n elements. The group operation is addition followed by remainder modulo n.

Thus in $\mathbf{Z}_8$, $6 + 5 = 3$.

The multiplicative group modulo n is $\mathbf{Z}_n^*$, defined by

$$\mathbf{Z}_n^* = \{[a]_n \in \mathbf{Z}_n : \mathrm{gcd}(a, n) = 1\}.$$

The group operation is multiplication followed by remainder modulo n.

So $\mathbf{Z}_6^* = \{[1]_6, [5]_6\}$. We abbreviate this as $\{1, 5\}$.

Thus in $\mathbf{Z}_{15}^*$, $7 * 11 = 2$.

The size of $\mathbf{Z}_\mathbf{n}^*$ is denoted by $\phi(n)$, the number of integers between 1 and n and relatively prime to n.

$$\begin{array}{ll}
\phi(5) = 4 & 4 = 5(1 - 1/5) \\
\phi(6) = 2 & 2 = 6(1 - 1/2)(1 - 1/3) \\
\phi(7) = 6 & 6 = 7(1 - 1/7) \\
\phi(35) = 24 & 24 = 35(1 - 1/5)(1 - 1/7)
\end{array}$$

In general, if n has the distinct prime factors $p_1, p_2, \cdots, p_m$, then $\phi(n) = n(1-1/p_1)(1-1/p_2)\cdots(1-1/p_m)$.

The subgroup generated by an element: The set of powers of an element. For example, in $\mathbf{Z}_n$, it is the multiples $\pmod n$.

Thus the subgroup generated by 2 in $\mathbf{Z}_6$ is $\{2, 4, 0\}$.

We now consider the problem of solving $ax \equiv b \pmod{n}$ for x. This has an application to the Chinese Remainder Theorem.

Let $\langle a \rangle$ be the subgroup of $\mathbf{Z}_n$ generated by a. For example, the subgroup of $\mathbf{Z}_{12}$ generated by 8 is $\{0, 4, 8\}$, and $\gcd(8, 12) = 4$.

In general, if $d = \gcd(a, n)$, then $|\langle a \rangle| = n/d$.

```
MODULAR-LINEAR-E-S(a,b,n)
  (d, x', y') ← EXTENDED-EUCLID(a, n)
  if d|b
      then x0 ← x'(b/d) mod n
           for i ← 0 to d − 1
             do print(x0 + i(n/d)) mod n
      else print "no solutions"
```

Running time is $O(\lg n + \gcd(a, n))$.

Example: Suppose we want to solve the equation

$$9x \equiv 6 \pmod{12}.$$

Then $a = 9$, $b = 6$, $n = 12$, $d = \gcd(a, n) = \gcd(9, 12) = 3$, $x' = -1$, and $y' = 1$. Since $d|b$, $x_0 = x'(b/d) \bmod n = -1(6/3) \bmod 12 = -2 \bmod 12 = 10$.

Thus the solutions are $(x_0+i(n/d)) \bmod n$ for $0 \le i < d$, that is, $10+i(12/3) \pmod{12}$ for $0 \le i < 3$, or, $10, 10+4 \pmod{12}, 10+8 \pmod{12}$. This gives $10, 2, 6$.

We can test this: $9*10 = 90 = 84+6$, $9*2 = 18 = 12+6$, and $9*6 = 54 = 48+6$. There are no more solutions.

Corollary 33.25.
For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax \equiv b \pmod{n}$ has a unique solution modulo n.

If $b = 1$, then we call such an x a **multiplicative inverse** of a, modulo n. In this case, we can use the notation $(a^{-1} \bmod n)$ for such an x. Such an x can be computed by the procedure EXTENDED-EUCLID, too.

Corollary 33.28

If $n_1, n_2, \cdots, n_k$ are pairwise relatively prime and $n = n_1 n_2 \cdots n_k$, then for any integers $a_1, a_2, \cdots, a_k$, the set of simultaneous equations

$$x \equiv a_i \pmod{n_i}$$

for $i = 1, 2, \cdots, k$ has a unique solution modulo n for the unknown x.

The solution may be obtained in this way: Let $m_i = n/n_i$ for $i = 1, 2, \cdots, k$. Let

$$c_i = m_i(m_i^{-1} \bmod n_i)$$

for $i = 1, 2, \cdots, k$. Then

$$x \equiv (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \pmod{n}.$$

Here is an example:

$x \equiv 1 \pmod 5$
$x \equiv 3 \pmod 7$
$a_1 = 1 \qquad n_1 = 5 \quad n = 35$
$a_2 = 3 \qquad n_2 = 7$
$m_1 = 35/5 = 7$
$m_2 = 35/7 = 5$

$$\begin{aligned} c_1 &= m_1(m_1^{-1} \pmod{n_1}) \\ &= 7(7^{-1} \pmod 5) \\ &= 7 * 3 \\ &= 21 \end{aligned}$$

$$\begin{aligned} c_2 &= m_2(m_2^{-1} \pmod{n_2}) \\ &= 5(5^{-1} \pmod 7) \\ &= 5 * 3 \\ &= 15 \end{aligned}$$

$x = 1 * 21 + 3 * 15 = 66$
$(66 \pmod{35}) = 31$
$(31 \pmod 5) = 1$
$(31 \pmod 7) = 3.$

Euler's Theorem

For any integer $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ for all } a \in \mathbf{Z}_n^*.$$

For example, $a^{24} \equiv 1 \pmod{35}$ for all a relatively prime to 35, since 24 is $\phi(35)$. We can try 2^{24} to verify this. 2^{24} is 16777216, and $479349 * 35 = 16777215$.

Fermat's Theorem

If p is prime, then

$$a^{p-1} \equiv 1 \pmod{p} \text{ for all } a \in \mathbf{Z}_p^*.$$

For example, $a^6 \equiv 1 \pmod{7}$ for all a not divisible by 7. Thus $2^6 \equiv 1 \pmod{7}$, that is, $64 \equiv 1 \pmod{7}$. This is correct, since $9 * 7 = 63$.

Raising to powers by repeated squaring

We want to compute $a^b \bmod n$ efficiently where a and b are nonnegative and n is a positive integer. This is used in the RSA public-key cryptosystem. We can do this by repeated squaring, as follows:

Let $\langle b_k, b_{k-1}, \cdots, b_1, b_0 \rangle$ be the binary representation of b. We compute $a^b \bmod n$ this way:

```
MODULAR-EXPONENTIATION(a, b, n)
  c ← 0
  d ← 1
  let ⟨b_k, b_{k-1}, ···, b_1, b_0⟩ be
    the binary representation of b
  for i ← k downto 0
      do c ← 2c
        d ← (d · d) mod n
        if b_i = 1
          then c ← c + 1
               d ← (d · a) mod n
  return d
```

This takes $O(\beta)$ arithmetic operations and $O(\beta^3)$ bit operations.

As an example, to compute $a^5 \bmod n$ without repeated squaring, we would compute $a * a * a * a * a \bmod n$, which involves four multiplications and one remainder. We can use repeated squaring to reduce the number of multiplications, computing $a^4 * a \bmod n$, that is, $(a^2)^2 * a \bmod n$. This requires only three multiplications and one remainder. For larger exponents, the savings are much greater. We can do the divisions after each multiplication to keep the intermediate results small, which yields

$$(((a^2 \bmod n)^2 \bmod n) * a) \bmod n.$$

Public Key Cryptosystems

- S_A is secret.
- P_A is public.
- For all messages M, $S_A(P_A(M)) = M$.
- For all messages M, $P_A(S_A(M)) = M$.

To create S and P:

Let p and q be randomly selected large primes. Let n be pq. Let e be a small odd integer that is relatively prime to $\phi(n)$, that is, to $(p-1)(q-1)$. Let d be the multiplicative inverse of e modulo $\phi(n)$.

Publish $P = (e, n)$ as the **RSA public key**.

Use $S = (d, n)$ as the **RSA secret key**.

Then $P(M) = M^e \pmod{n}$, and $S(C) = C^d \pmod{n}$.

These functions can be computed efficiently using the procedure MODULAR-EXPONENTIATION.

We can prove that P and S are inverses of one another using Euler's theorem.

Note that if we could factor efficiently, we could recover p and q from n, and then compute d from e easily. This would enable us to figure out the secret key from the public key.

As an example, suppose we choose $p = 5$ and $q = 11$. Then $n = pq = 55$. Also, $\phi(n) = (p-1)(q-1) = 40$. We choose e to be a small odd integer relatively prime to 40. Let e be 3. Let d be the multiplicative inverse of e modulo 40. Then $d = 27$, since $3 * 27 = 81$, and the remainder of 81 divided by 40 is 1.

We then publish $P = (3, 55)$ as the RSA public key.

We use $S = (27, 55)$ as the RSA secret key.

Note that it is hard to figure out S from P if we cannot factor 55.

Then $P(M) = M^3 \pmod{55}$ and $S(C) = C^{27} \pmod{55}$. We want to show that for all M, $P(S(M)) = M$ and $S(P(M)) = M$ if $M < 55$.

Now, $S(P(M)) = (M^3 \pmod{55})^{27} \pmod{55}$ $= M^{3*27} \pmod{55} = M^{81} \pmod{55} = M*$ $(M^{40})^2 \pmod{55}$.

But $M^{40} \equiv 1 \pmod{55}$ by Euler's theorem since $40 = \phi(55)$. Thus $M^{80} \equiv 1 \pmod{55}$ and so $M^{81} \equiv M \pmod{55}$. Thus if $M < 55$, $S(P(M)) = M$. We note also that $P(S(M)) = S(P(M))$, so $P(S(M)) = M$ also for $M < 55$.

Let's do some examples. $P(2) = 2^3 \pmod{55} = 8 \pmod{55}$ which is 8. $P(8) = 8^{27} \pmod{55} = 2$.

```
| ?- X is 8 ^ 5 mod 55 .
X = 43 .
| ?- X is 43 ^ 5 mod 55 .
X = 43 .
| ?- X is 8 ^ 2 mod 55 .
X = 9 .
| ?- X is (43 * 9) mod 55 .
X = 2 .
```

Also, $S(2) = 2^{27} \pmod{55} = 134217728 \pmod{55}$ which is 18. $P(18) = 18^3 \pmod{55} = 5832 \pmod{55} = 2$.

Thus, public P can encode his secret message (2, say), by computing $P(2)$, which is $2^3 \pmod{55}$, that is, $8 \pmod{55}$, or 8. Everyone can read this. But only secret S can decode 8 by computing $8^{27} \pmod{55}$, back to the value 2.

Also, secret S can sign a message M' by appending $S(M')$ to it. When public P receives the message, he can compute $P(S(M'))$ and obtain M' and verify that secret S sent the message.

String Matching Algorithms (Chapter 34)

The problem is to find all occurrences of a *pattern* P of length m characters in a text T of length n characters.

The practical importance of this in text editors, text searching, and DNA matching should be apparent.

The brute-force method has worst case running time $O((n-m+1)m)$, but on the average should be $O(n-m+1)$:

$\textsc{Naive-String-Matcher}(T, P)$
  $n \leftarrow length[T]$
  $m \leftarrow length[P]$
  for $s \leftarrow 0$ to $n - m$
    do if $P[1 \cdots m] = T[s+1 \cdots s+m]$
      then print
      "Pattern occurs with shift" s

Rabin-Karp algorithm

The idea is to compute an integer function $f(T[s+1\cdots s+m])$ for all s and compare it to $f(P[1\cdots m])$. If there is a match, these will be equal, and we can check whether $P[1\cdots m] = T[s+1\cdots s+m]$. If these are not equal, we know there is no match. So the idea of the algorithm is just the following:

ABSTRACT-RABIN-KARP-MATCHER(T,P)
  $n \leftarrow length[T]$
  $m \leftarrow length[P]$
  for $s \leftarrow 0$ to $n-m$
      do if $f(P[1\cdots m]) = f(T[s+1\cdots s+m])$
        and $P[1\cdots m] = T[s+1\cdots s+m]$
          then print
          "Pattern occurs with shift" s

For this to work well, f must have two properties:

- $f(T[s+2 \cdots s+m+1])$ must be easy to compute from $f(T[s+1 \cdots s+m])$.
- It must be very unlikely for $f(P[1 \cdots m]) = f(T[s+1 \cdots s+m])$ when $P[1 \cdots m] \neq T[s+1 \cdots s+m]$. (This is called a *spurious hit*.)

The function chosen is

$$f(P[1 \cdots m]) =$$

$$P[m]+d*P[m-1]+d^2*P[m-2]+\cdots+d^{m-1}*P[1]$$

$$\bmod q,$$

where d is the size of the alphabet, each letter $P[i]$ is regarded as an integer, and q is a large integer. (Modular arithmetic strikes again!)

The expected running time is $O((n-m+1)m)$ in the worst case, since it can be that every shift is valid and all must be checked. If the number of valid shifts is small, and q is chosen larger than the pattern length, then Rabin-Karp "probably" runs in time $O(n+m)$.

Here is the detailed algorithm:

```
RABIN-KARP-MATCHER(T,P,d,q)
  n ← length[T]
  m ← length[P]
  h ← d^(m−1) mod q
  p ← 0
  t_0 ← 0
  for i ← 1 to m
      do p ← (d * p + P[i]) mod q
         t_0 ← (d * t_0 + T[i]) mod q
  for s ← 0 to n − m
      do if p = t_s
           then if P[1 ··· m] = T[s + 1 ··· s + m]
                  then
                  "Pattern occurs with shift" s
         if s < n − m
           then t_{s+1} ← (d(t_s − T[s + 1]h)
                      +T[s + m + 1]) mod q
```

Finite Automaton Approach

The idea of this is to compute a table. At each stage, scanning the i^{th} character of T, we want to know the longest prefix of P that matches the pattern so far, that is, $T[1 \cdots i]$. Formally, we want the longest prefix of P which is a suffix of $T[1 \cdots i]$.

If we know the longest prefix of P which is a suffix of $T[1 \cdots i]$, and we know $T[i+1]$, we can compute the longest prefix of P that is a suffix of $T[1 \cdots i+1]$. Thus this information can be stored in a table. We define $\delta(q, a)$ to be the longest prefix of P that is a suffix of $T[1 \cdots i+1]$, given that a is $T[i+1]$ and q is the length of the longest prefix of P that is a suffix of $T[1 \cdots i]$.

The result is the following simple algorithm:

FINITE-AUTOMATON-MATCHER(T, δ, m)
  $n \leftarrow length[T]$
  $q \leftarrow 0$
  for $i \leftarrow 1$ to n
    do $q \leftarrow \delta(q, T[i])$
      if $q = m$
        then $s \leftarrow i - m$
          print
          "Pattern occurs with shift" s

The running time is $O(n)$ on a text string of length n. But this does not include the time to construct the table. A straightforward method of constructing the table takes time $O(m^3|\Sigma|)$ where Σ is the alphabet. But this may be reduced to $O(m|\Sigma|)$. Thus the total running time for this procedure becomes $O(n + m|\Sigma|)$.

Here is a simple way to construct δ:

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
  m ← length[P]
  for q ← 0 to m
      do for each character a ∈ Σ
             do k ← min(m + 1, q + 2)
                repeat k ← k − 1
                  until P_k is a suffix of P_q a
                δ(q, a) ← k
  return δ
```

Here P_k is the first k characters of P.

Example for computing transition function and pattern matching.

Let the pattern be $ababaca$ as in the book:

```
a  b  a  b  a  c  a
```

Illustrate what some of the transitions would be. The final table in the book is like this:

```
         input

state    a b c  Letter

  0      1 0 0    a
  1      1 2 0    b
  2      3 0 0    a
  3      1 4 0    b

          ...
```

The Knuth-Morris-Pratt algorithm

This achieves a $\Theta(n+m)$ worst case running time by doing less work to compute the table δ, but slightly more work during pattern matching. For this, we need to compute ways in which the pattern P can match against itself. Formally, we compute

$\pi[q] = \max\{k : k < q$ and P_k is a suffix of $P_q\}$.

Recall that P_k is the first k characters of P.

We show the example of π from the book.

```
* * * * *      Matching characters
a b a b a c a  Pattern


    a b a b a c a    Next possible match
```

Thus $\pi[5] = 3$.

We have the following algorithm:

KMP-Matcher(T, P)
$n \leftarrow length[T]$
$m \leftarrow length[P]$
$\pi \leftarrow$ Compute-Prefix-Function(P)
$q \leftarrow 0$
for $i \leftarrow 1$ to n
    do while $q > 0$ and $P[q+1] \neq T[i]$
        do $q \leftarrow \pi[q]$
      if $P[q+1] = T[i]$
        then $q \leftarrow q+1$
      if $q = m$
        then print
          "Pattern occurs with shift" $i - m$
            $q \leftarrow \pi[q]$

Example.

We also need to compute π, which may be done as follows:

```
COMPUTE-PREFIX-FUNCTION(P)
  m ← length[P]
  π[1] ← 0
  k ← 0
  for q ← 2 to m
      do while k > 0 and P[k + 1] ≠ P[q]
             do k ← π[k]
         if P[k + 1] = P[q]
           then k ← k + 1
         π[q] ← k
  return π
```

This may be understood as applying the KMP matcher with the pattern P and text T both equal to P. Then q is the location being scanned in the text and P_k is the portion of the pattern P matching T_{q-1}, that is, P_k is a suffix of T_{q-1}.

We can show linear time using the potential method, with k as the potential function. Each time through the loop increases k by at most one and may execute $k \leftarrow \pi[k]$ some number of times, each of which decreases the potential and thus has zero amortized cost. So the cost of each time through the loop is $O(1)$. Since we traverse the loop m times, the total work is $O(m)$ for this algorithm.

Boyer-Moore algorithm

Just give some idea of how it works.

This algorithm matches the pattern from right to left. It often permits large sections of the input to be skipped altogether. Thus the running time is often much less than the length of the input. However, the worst case running time is $O((n - m + 1)m + |\Sigma|)$.

First we give the algorithm and then an intuition about how it works.

```
Boyer-Moore-Matcher(T, P)
  n ← length[T]
  m ← length[P]
  λ ← Compute-Last-Occurrence-Fn(P, m, Σ)
  γ ← Compute-Good-Suffix-Fn(P, m)
  s ← 0
  while s ≤ n − m
       do j ← m
          while j > 0 and P[j] = T[s + j]
             do j ← j − 1
          if j = 0
            then print "Pattern occurs at shift" s
                 s ← s + γ[0]
            else s ← s + max(γ[j], j − λ[T[s + j]])
```

We just give an example from the book to illustrate λ and γ and how this algorithm works.

```
 w r i t t e n   n o t i c e   t h a t
     r e m i n i s c e n c e
```

```
Bad character heuristic -- move four forward:
```

```
 w r i t t e n   n o t i c e   t h a t
             r e m i n i s c e n c e
```

```
Good suffix heuristic -- move three forward:
```

```
 w r i t t e n   n o t i c e   t h a t
           r e m i n i s c e n c e
```

Thus we choose the larger move of the two, and move four forward.

Profound words to end the course

[Substitute here something philosophical about what we have covered]

Bon Voyage!

Auf Wiedersehen!

Hasta Luego!

Arrivederci!

Aloha!

...

And, for the sake of old times, some animations ...