Algorithms - Exercise 1

Due Wednesday 1/11 24:00

1. Rank the following functions by their order of growth. Prove your answers.

$$\log^2(n),\ n\log n,\ n^{\log\log n},\ n^{1/3},\ n!,\ \log(n!),\ 4^n,\ \binom{2n}{n}$$

2. Define the iterative logarithm function as $\log^*(n) = \min\{i : \log^{(i)}(n) \leq 1\}$, where

$$\log^{(i)}(n) \quad = \quad \begin{cases} n & i = 0 \\ \log(\log^{(i-1)}(n)) & \text{if } i > 0 \text{ and } \log^{(i-1)}(n) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

   Which function grows asymptotically faster, $\log(\log^*(n))$ or $\log^*(\log n)$?

3. Consider the algorithm $\frac{1}{3}$-merge-sort that is exactly the same as merge-sort with the only difference that the recursive step sorts first the first 1/3 of the array then the last 2/3 and then merges the two.

   Write a pseudo-code for $\frac{1}{3}$-merge-sort and analyze its running time (as a function of the length of the array).

4. $X = [x_1, x_2, \ldots x_n]$ is an array of different integers. Give an algorithm that finds the second bigest number, using

   (a) at most $2n$ comparisons.

   (b) at most $n + O(\log n)$ comparisons. (Hint: put the integers on leaves of a binary tree.)

5. $X = [x_1, x_2, \ldots x_n]$ is a read-only array of integers, where $1 \leq x_i \leq n-1$. Give an algorithm that returns a value appearing at least twice in the array, subject to the following limitations:

(a) Time $O(n)$

(b) Space $O(1)$

(c) Time $O(n \log n)$ and Space $O(1)$

(d) (reshut) Time $O(n)$ and Space $O(1)$

Solution to Ex 1 in Algorithms

1.
   - $\log^2 n = o(n^{1/3})$
   - $n^{1/3} = o(n \log n)$
   - $n \log n = \Theta(\log(n!))$. from Stirling's formula.
   - $n \log n = o(n^{\log \log n})$
   - $n^{\log \log n} = o(\left(\begin{array}{c} 2n \\ n \end{array} \right))$ because $2^n \leq \left(\begin{array}{c} 2n \\ n \end{array} \right)$
   - $\left(\begin{array}{c} 2n \\ n \end{array} \right) = o(4^n)$ from Stirling's formula
   - $4^n = o(n!)$

2. Define the iterative logarithm function as $\log^*(n) = \min\{i : \log^{(i)}(n) \leq 1\}$, where

$$\log^{(i)}(n) \quad = \quad \begin{cases} n & i = 0 \\ \log(\log^{(i-1)}(n)) & \text{if } i > 0 \text{ and } \log^{(i-1)}(n) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

   Which function grows asymptotically faster, $\log(\log^*(n))$ or $\log^*(\log n)$?

   From the definition, $\log^*(\log n) = \log^*(n) - 1$, thus $\log(\log^*(n)) = o(\log^*(\log n))$

3. Analysing the running time of $\frac{1}{3}$-merge-sort: assume that merging of two arrays of combined length n takes less than cn time. Denote by $T(n)$ the running time of the $\frac{1}{3}$-merge-sort, we obtain the following equation:

$$T(n) \leq T(\frac{1}{3}n) + T(\frac{2}{3}n) + cn$$

   Take any $K > 3c$. We shall prove by induction that $T(n) \leq Kn \log n$. Assume correctness for numbers smaller than n. we have

$$T(n) \leq T(\frac{1}{3}n) + T(\frac{2}{3}n) + cn \leq \frac{1}{3}Kn \log(\frac{1}{3}n) + \frac{2}{3}Kn \log(\frac{2}{3}n) + cn =$$

$$= \frac{1}{3}Kn(\log n - \log 3) + \frac{2}{3}Kn(\log n + \log 2 - \log 3) + cn \leq Kn \log n - \frac{1}{3}Kn \log 3 + cn \leq Kn \log n$$

4. $X = [x_1, x_2, \ldots x_n]$ is an array of different integers. Give an algorithm that finds the second biggest number, using

(a) at most $2n$ comparisons.

(b) at most $n + O(\log n)$ comparisons. (Hint: put the integers on leaves of a binary tree.)

Solution for (b). As hinted, put the integers on the leaves of a binary tree with depth $\log n$. We now go on all the inner verices of the tree bottom up, placing at a vertix v the maximum between the numbers on the sons of v. Obviously, the biggest number M will end up on top. But what about the second biggest one m? It had to 'loose' to the biggest one. But the biggest one, M was only compared while computing the values on the vertices on the path from the leaf containg M to the root of the tree. The length of this path is $\log n$, thus we only need to find the maximum between $\log n$ numbers. Summing up, computing the values of all the vertices takes n comparisons, and another $\log n$ to find the maximum between all the numbers M was compared to.

5. $X = [x_1, x_2, \ldots x_n]$ is a read-only array of integers, where $1 \leq x_i \leq n-1$. Give an algorithm that returns a value appearing at least twice in the array, subject to the following limitations:

   (a) Time $O(n)$

   (b) Space $O(1)$

   (c) Time $O(n \log n)$ and Space $O(1)$

   (d) (reshut) Time $O(n)$ and Space $O(1)$

   We shall present the solution to (d), thus solving (a-d). For better intuition, think of the directed graph $G = (V, E)$ with $V = [1..n]$ and $(i, j) \in E$ iff $x_i = j$. Our goal then is to find a vertex which had two edges entering it. Start from the vertex n (which doesn't have edges entering it) and walk on the graph. The path has to intersect itself and continue in a loop. We are bound to be inside the loop after n steps. Call that vertex v. We can measure the length of the loop by starting at v and counting the number of steps k it takes to get to v again. Now start at vertex n and walk k steps to a vertex w. We then walk *at the same times* both from w and from n, till the paths meet. As the distance is between them is the length of the loop, they are bound to meet at the vertex which has two edges entering it.

   Input= $[x_1, \ldots, x_n]$
   $t \leftarrow n$ {Getting into the loop}
   for $i = 1$ to n **do**
   $t \leftarrow a_t$
   end for
   $p \leftarrow t$ {Computing loop's length k}
   $k \leftarrow 1$
   while $a_p \neq t$ **do**
   $p \leftarrow a_p$
   $k \leftarrow k + 1$
   end while
   $t \leftarrow n$ {Going k steps from n}
   for $i = 1$ to k **do**
   $t \leftarrow a_t$
   end for

$p \leftarrow n$ {Stepping together}
while $p \neq t$ **do**
  $t \leftarrow a_t$
  $p \leftarrow a_p$
end while
Output p

Algorithms – Exercise 2

Due Wednesday 8/11/06 24:00

Definition: A matroid is an ordered pair $M=(S,I)$ satisfying the following conditions:

a. S is a finite nonempty set.
b. I is a nonempty family of subsets of S, such that if $B \in I$ and $A \subseteq B$ then $A \in I$. (I is hereditary)
c. If $A \in I$, $B \in I$, and $|A| < |B|$, then there is some element $x \in B-A$ such that $A \cup \{x\} \in I$. (M satisfies the exchange property)

1. Prove that the following are matroids:
   a. (S,I) such that S is some finite set and I is the set of all subsets of S of size at most k, where $k \leq |S|$.
   b. (S,I) such that S is a finite set and let $S_1, S_2, \ldots, S_k$ a partition[1] of S into nonempty disjoint subsets, and $I = \{A: |A \cap S_i| \leq 1 \text{ for all } i=1,\ldots,k\}$.
   c. (Reshut) (S,I) such that S is a finite set and $I = \{X_1 \cup X_2 : X_1 \in I_1 \ \& \ X_2 \in I_2\}$, where $M_1=(S,I_1)$, $M_2=(S,I_2)$ are matroids.

2. Consider the following problem:
   Given $c_1,\ldots,c_n$ and $r_1,\ldots,r_n$, sequences of positive integers, output a matrix A whose entries are either 0 or 1, such that for every $1 \leq i \leq n$, r_i is the sum of the i-th row and c_i is the sum of the i-th column (if such A exists).
   Prove that the following greedy algorithm works:
   Go through the rows from 1 to n. For each row i, assign 1 to the r_i columns such that c_j-a_j is maximal, where a_j equals the number of 1s in the j-th column after the assignment of the first i-1 rows. Assign 0 to the other entries.

3.
   a. Build the Huffman code for the following input. Write down all the intermediate stages.
      The messages: $X = (a, b, c, d, e, f)$
      The frequencies (correspondingly): $F = (20, 75, 15, 32, 40, 7)$
   b. Build the Huffman code for the following input: (Prove your answer)
      The messages: $X = (a_1, \ldots, a_n)$
      The frequencies (correspondingly): $F = (1, 2, \ldots, 2^{n-1})$

[1] $S_1,..,S_n$ is a partition of S, if $\bigcup_{i=1}^{k} S_i = S$ such and $S_1,..,S_n$ are disjoint.

4. Given a binary tree T, and given a frequency function f which assigns positive weights to the leaves of T, we expand f recursively to give values to all the nodes of T by setting $f(w) = \sum_{x \in Sons(w)} f(x)$ for all w which are not leaves.
   Give an algorithm for the following problem:
   Given a list of messages X and a frequency function f, return a tree representing a prefix code for X (the members of X are assigned messages with their corresponding frequencies) such that the sum of the frequencies over all[2] the nodes of T is minimal.

5.
   a. Let C be a prefix code with codewords of lengths $l_1,..,l_n$. Show that the following inequality holds:
   $$\sum_{i=1}^{n} 2^{-l_i} \leq 1$$
   b. (Reshut) Show that if the inequality above holds for some positive integers $l_1,..,l_n$, then there exists a prefix code C whose codewords' lengths are $l_1,..,l_n$.

6. Give a dynamic programming algorithm that solves the following problem:
   Given strings u,v , find a string w of minimal length which contains u and v as subsequences (i.e. the characters of u,v in w need not be consecutive).

[2] i.e. the leaves and the non-leaves

Algorithms – Exercise 2 (Solution)

1.

a. It follows directly from the definition-
   i. I is hereditary since if $|A| \le k$, then if $B \subseteq A$ then $|B| \le k$.
   ii. If $A, B \in I$ such that $|A| < |B|$ then $|B| \le k$ and therefore, if we take an element $x \in A$ and add it to B then we get A' such that $|A'| = |A| + 1 \le |B|$. And therefore $A' \in I$.

b. By the definition-
   i. I is hereditary since if $A \in I = \{A: |A \cap S_i| \le 1 \text{ for all } i=1,\ldots,k\}$, and $B \subseteq A$ then for all i=1,..,k $|B \cap S_i| \le |A \cap S_i| \le 1$ and therefore $B \in I$.
   ii. I satisfies the exchange property, since if $A, B \in I$ are composed of one element of some of the S_i let's say that A is composed of elements of $S_{j1},..,S_{jsize(A)}$ $S_1,..,S_{size(A)}$ and B of elements of $S_{i1},..,S_{isize(B)}$ then if $|A| < |B|$ then there must be a set among $S_{i1},..,S_{isize(B)}$ which is not among $S_{j1},..,S_{jsize(A)}$, and therefore we can extend A by taking an element from that set.

c. We start with a lemma:
   Lemma: Given any matroid $M = (S', \boldsymbol{I}')$, and any function (not necessarily injective) $f: S' \to S$, then $M = (S, f(\boldsymbol{I}'))$ is a matroid, where $f(\boldsymbol{I}') = \{f(A): A \in \boldsymbol{I}'\}$.
   Proof: Since f is a function, it is clear that if $I \in f(\boldsymbol{I})$, then any subset of I is also in $f(\boldsymbol{I}')$. Now suppose $I, J \in f(\boldsymbol{I}')$, with $|I| < |J|$. We need to show that for some $f \in J \setminus I, I \cup \{j\} \in \boldsymbol{I}'$. By assumption (and definition) I and J must be images of two independent sets I', J' of M'. Since f is not injective, there may be many ways to choose such sets. We take I', J' such that $I = f(I')$ and $|I| = |I'|$, $J = f(J')$ and $|J| = |J'|$ and finally, such that $|I' \cap J'|$ is maximal.
   Since $|I'| < |J'|$ and M' is, by assumption, a matroid, then there exists an element $t \in J' \setminus I'$ such that $I' \cup \{t\} \in \boldsymbol{I}'$. If $f(t) \in f(I') \cap f(J')$ then there exists some $u \in I'$ such that $f(t) = f(u)$. Since $|J| = |J'|$, f maps J' injectively onto J, and thus $u \in I' \setminus J'$. But then the set $I'' = I' \cup \{t\} \setminus \{u\}$ is in $\boldsymbol{I}$ (because $I'' = I' \cup \{t\} \in \boldsymbol{I}$), and then $f(I'') = I$, $|I''| = |I|$ and $|I'' \cap J'| > |I' \cap J'|$ contradicting maximality.
   Therefore $f(t) \in f(J') \setminus f(I')$, and $f(I' \cup \{t\}) = f(I') \cup \{f(t)\} \in \boldsymbol{I}$ as required.

   Now we prove that $(S, \boldsymbol{I})$ where $\boldsymbol{I} = \{X_1 \cup X_2 : X_1 \in I_1 \ \& \ X_2 \in I_2\}$ is a matroid.
   First let us assume that S_1 and S_2 are two disjoint copies of S. Let I_1 and I_2 be the corresponding independent sets. Then, it can be easily verified

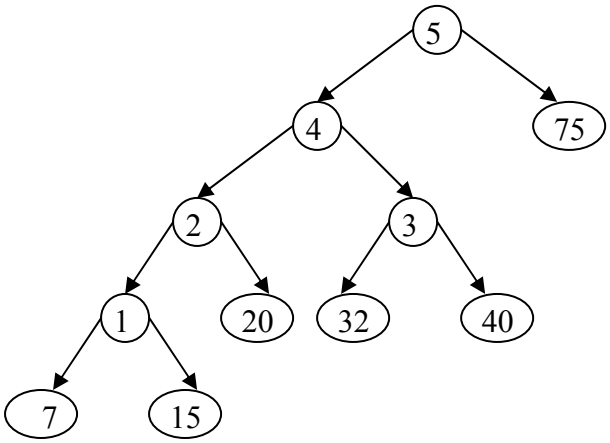
that $(S_1 \bigcup S_2, \boldsymbol{I})$ where $\boldsymbol{I}$ is defined as above (only the base sets are now disjoint) is a matroid. We now build a function $f : S_1 \bigcup S_2 \to S$ by mapping each element to its original copy. *(S,f(**I**))* is exactly the structure which we want to prove a matroid. By the above lemma, we conclude it is a matroid. As required.

2. Lemma: for every 0 <= k <= n, there exists a matrix A which obeys the constraints, who shares its first k rows with the matrix the algorithm outputs.

   Proof: by induction on k. It is trivial for k=0.
   Induction step: Let A' be a matrix which obeys the constraints and whose k-1 first rows are shared with the outputted matrix (such exists by the induction hypothesis). There are m=r_i 1's to distribute in the i-th row. Let's denote the algorithm's output matrix as B.
   Now the *i*-th row of A' is not necessarily identical to that of B. Let us assume that the entries in the *i*-th row of A' which has to be switched from 1 to 0 in order to receive B's *i*-th row are $r_{i(1)},...,r_{i(l)}$, and that $r_{j(1)},...,r_{j(l)}$ has to be switched from 0 to 1. Now we would like to switch the values of $r_{i(1)},...,r_{i(l)}$ and $r_{j(1)},...,r_{j(l)}$ one by one. In order to remain with a matrix that obeys the constraints, we must find another row $j > i$ in which the entries in the corresponding columns would be the other way around. For instance, in order to swap $r_{i(1)}$ and $r_{j(1)}$ we must find a row j > i in which $r_{i(1)}$ is 0 and $r_{j(1)}$ is 1. Such a row *j* indeed exists. Since the sum of the remaining entries in $r_{i(1)}$ is $c_{ri(1)} - a_{ri(1)}$, and the same with $r_{j(1)}$. Due to the algorithm's choice of where to put the 1's, we know that $c_{ri(1)} - a_{ri(1)} \le c_{rj(1)} - a_{rj(1)}$. Therefore, there is a row j > i in which $r_{i(1)}$ is 0 and $r_{j(1)}$ is 1, otherwise the previous inequality would not have held.
   We now swap the entries of $r_{i(1)},...,r_{i(l)}$ and $r_{i(1)},...,r_{i(l)}$ one by one, each time finding a row $j > i$ in which we swap entries of 1 and 0 in the other direction. We receive a matrix A whose *i* first rows are identical to that of B. as required.

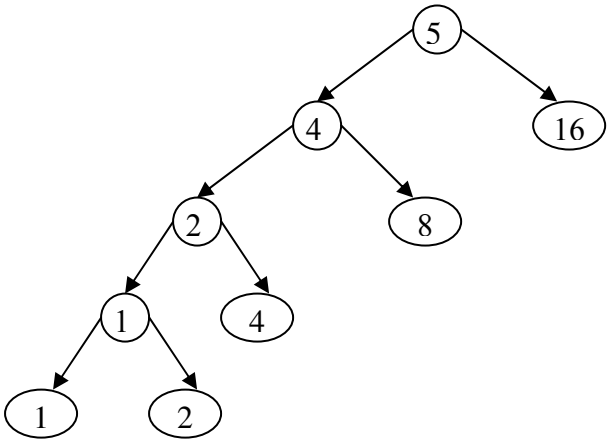
   This lemma, for k=n claims that the algorithm returns a matrix which obeys the constraints.

3. a.



The frequencies: *F = (20, 75, 15, 32, 40, 7)*
The ellipses are leaves corresponding to the frequencies written in them. The other leaves were added during the running of the algorithm, and are enumerated according to the order in which they were created.

b. The frequencies (correspondingly): *F = (1, 2, …, 2^{n-1})*. For n=5 the tree looks like that:



In general, the Huffman code will generate a tree that branches to one side. This can be seen since at any given stage, the remaining frequencies in the queue are: ($\sum_{i=0}^{k-1} 2^i = 2^k - 1$, 2^k,..,2^{n-1}), and therefore, the two lowest frequencies are 2^k-1,2^k. Now by induction, it follows that the tree is always right (or left, depending on the way we build the Huffman code) branching.

4. The expression we would like to minimize is the following: $\sum_{x \in V} f(x)$. From the recursive formula, we may conclude that each leaf x appears in this sum once for himself, then once for his father, and so on until the root. So, each leaf appears exactly its depth times in the sum. We rewrite

the sum as: $\sum_{x \in Leaves} d(x)f(x)$. But this is exactly the term which the Huffman code minimizes, and therefore the Huffman code algorithm solves the given problem.

5.

a. Given the prefix code, we will build a tree T which corresponds to that prefix code, the leaves correspond to the code words, and their length is the leaves' depth. Let us extend T to a tree T' by adding the all the possible descendents of the leaves in T up to the maximal level of T (which we shall denote by l_{max}). This tree has at most $2^{l\max}$ leaves. (since all leaves are of depth l_{max}) Each leaf in T which was in the level l_i has $2^{l\max - l_i}$ descendents which are leaves in T', and the descendents of different leaves of T are disjoint. We conclude the equality:

$$\sum_{i=1}^{n} 2^{l\max - l_i} \le 2^{l\max}$$

Dividing by $2^{l\max}$ yields the required inequality.

b. We'll prove it by induction on n. For n=1, it is trivial. The induction step: let's assume that the $l_1,..,l_n$ are sorted by size, so l_n is the largest. Now $l_1,..,l_{n-1}$ satisfy the induction hypothesis, so we'll get a tree T for the first n-1 lengths. We'll extend this tree in a similar manner to that in the last question, so all leaves in T will have their descendents in the l_n level. Now since $\sum_{i=1}^{n} 2^{l_n - l_i} \le 2^{l_n}$, we get $\sum_{i=1}^{n-1} 2^{l_n - l_i} < 2^{l_n}$. Therefore, T is not a full binary tree (otherwise we will get equality in the above inequality). We conclude that T has a node with only one son, we may attach the last leaf to it, which will be of depth $\le l_n$ (we'll denote it T'). A tree such that the depth of this leaf is exactly l_n can be easily constructed from T'.

6.

This algorithm is rather similar to the "Longest Common Subsequence" algorithm discussed in class.
For a string u we'll denote u_n the prefix of first n letters in u. The symbol ^ will denote the concat operator.
The following lemma justifies the dynamic programming approach:
Lemma: Let SCS(u,v) denote a shortest common superstring of u,v. Let us assume that u is of length n and v of length m. Then:

1. if u and v share a last letter c, then SCS(u,v) = SCS(u_{n-1},v_{m-1})^c.
2. if u and v don't share a last letter, but u terminates with c_1 while v terminates with c_2. Then SCS(u,v) = SCS(u_{n-1},v_m)^c_1 or SCS(u,v) = SCS(u_n,v_{m-1})^c_2.

Proof: First let us assume that u and v share a last letter c. Then their SCS has to end with c otherwise we can remove the final letter, and still be left with a superstring of u,v. Now, we have to prove that after removing c from the SCS, we are left with a SCS of u_{n-1} and v_{m-1}. Let us assume the contrary, then there

exists a string w which is shorter then what we are left with. Then we can construct a string $w\hat{}c$ which is a superstring of u,v and is shorter then SCS(u,v), which contradicts the definition of SCS. As required.
Now, we'll assume that u,v do not terminate with the same letter. First, let us remark that SCS(u,v) has to end with either c_1 or c_2, otherwise we may remove the last letter of SCS(u,v) and be left with a shorter superstring of u,v.
Now, if SCS(u,v) ends with c_1, then after removing it we are still left with a string that contains v as a subsequence (since v did not end with c_1). It is also a superstring of u_{n-1} , since if SCS(u,v) had u as a subsequence, then the after removing one letter, the first $n-1$ elements of that subsequence are still a part of SCS(u,v) without the final letter. Next, we will notice that SCS(u,v) without the final $c1$ is also the shortest superstring of u_{n-1},v_m since otherwise (as in the first article of this lemma), we can replace it with a shorter superstring, which after concatenating it with a terminal c_1 will yield a superstring of u,v which is shorter then SCS(u,v).

Now let us denote by OPT(i,j) – the length of SCS(u_i,v_j).
The lemma gives us this recursion formula:

$$OPT(i,j)=\begin{cases} 1+OPT(i-1,j-1) & \text{if } u\ i\text{'th letter equals } v\ j\text{'th} \\ 1+\min(OPT(i-1,j),OPT(i,j-1)) & \text{else} \end{cases}$$

This can also give us a dynamic programming algorithm to find SCS(u,v):
SCS-Length(u,v)
1. m = length(v)
2. n = length(u)
3. for $i=1..m$
    a. $c[i,0]=0$
4. for $j=1..n$
    a. $c[0,j]=0$
5. for $i=1..m$, $j=1..n$
    a. if $u(i)=v(j)$
        i. $c[i,j]=c[i-1,j-1]+1$
        ii. $b[i,j]=$ ↖
    b. *Else if* $c[i-1,j]\le c[i,j-1]$
        $c[i,j]=1+c[i-1,j]$
        $b[i,j]=$ ↑
    else
        $c[i,j]=1+c[i,j-1]$
        $b[i,j]=$ ←
6. return c,b

This code creates two charts c,b. c holds OPT(i,j) for every i,j. b holds the way to construct SCS(u_i,v_j) from shorter strings. In order to find SCS(u,v) one has to run Print-SCS(b,u,v,n,m):
Print-SCS(b,u,v,i,j)
1. if (i=0 or j=0) return;
2. if $b[i,j]=$ ↖
    a. Print-SCS($b,u,v,i-1,j-1$)
    b. Print $u(i)$
3. if $b[i,j]=$ ↑

a. Print-SCS(*b,u,v,i-1,j*)
b. Print *u(i)*

4. if *b[i,j]*= ←
   a. Print-SCS(*b,u,v,i,j-1*)
   b. Print *v(j)*

The lemma above implies the correctness of the algorithm.
The running time of complexity of this algorithm is O(*mn*) for SCS-Length and O(*m*+*n*) for Print-SCS.

Algorithms - Exercise 3

Due Wednesday 15/11 24:00

1. Definition: Given a graph $G = (V, E)$ a set S of vertices is called *independent* if every $v, w \in S$ are not connected in E.

   Give an algorithm for the following problem: Given a *tree* T with weight function $w : V \to R^+$, find an independent set in T with maximal weight.

2. 0-1 Knapsack. We have a sack of size N and a set of k objects, each with size x_i and value w_i. For every subset of objects its size is the total size of its elements and its value is the total value of its element. Give an algorithm finding a subset of maximal value that can be put in the sack.

3. (reshut) $G = (V, E)$ is a complete graph with $|V| = n$. A path $(v_{i_1}, \ldots, v_{i_n+1})$ in G is called a *bitonic cycle* if:

   - $i_1 = i_{n+1} = 1$
   - The path goes through all the vertices
   - There exists $1 < j < n + 1$ such that $i_k < i_m$ for $k < m \leq j$ and $i_k > i_m$ for $j \leq k < m$.

   Let $w : E \to R^+$ be a length function on the edges. Give an algorithm that finds the shortest bitonic path.

4. Give an algorithm finding the maximal flow in a flow network with multiple sources and sinks.

5. You are running the Mossad. You have n agents in Asia, each sitting in a different city. You want them to be in America, again, each one in a different city. There are only flights from Asia to Europe and from Europe to America. You have the full flight chart. To keep the secrecy level high, you don't want different agents to pass through the same city. Give an algorithm which gives a solution or says that none exists.

Note: in questions 4,5 assume that there exist an efficient algorithm finding the maximal flow in a flow network with one source and one sink.

Solution to Exercise 3 in Algorithms

Remark: in questions (1-3) we give an algorithm that finds the *value* of the optimal solution and not the solution itself. As in all dynamic programming questions, it can be handled by remembering the choices done while filling the table, and building the optimal solution based on them.

1. Definition: Given a graph $G = (V, E)$ a set S of vertices is called *independent* if for every $v, w \in S$ we have $(v, w) \notin E$.

   Give an algorithm for the following problem: Given a *tree* T with weight function $w : V \to R^+$, find an independent set in T with maximal weight.

   Pick any vertex v_0 in T and think of the tree as a rooted tree with root v_0. Now every vertex v has depth $d(v)$; it is connected to its father which has depth $d(v) - 1$ and sons with depth $d(v) + 1$. For any vertex $v \in T$ denote by T_v the tree containing v and all its descendants. We now define a set of subproblems, which we will use for the dynamic programming.

   $$A(v) = \max\{w(S) | S \subset T_v, v \in S\} \qquad B(v) = \max\{w(S) | S \subset T_v, v \notin S\} \quad C(v) = \max(A(v), B(v))$$

   That is, $A(v)$ is the maximal weight of an independent subset of T_v that contains v, and $B(v)$ is the maximal among those that don't contain v. Obviously, the solution to the big problem is $C(v_0)$. To get the recursion formulas, note the following: if S is an independent set in T_v and $v \in S$ then $u \notin S$ for every son u of v. On the other hand, any solution to the subtree T_u when u is a grandchild of v can be in S. Thus

   $$A(v) = w(v) + \sum_{u \text{ grandson of } v} C(u)$$

   If $v \notin S$ then S can contain any solution for the subtrees T_u with u a child of v. We obtain:

   $$B(v) = \sum_{u \text{ child of } v} C(u)$$

   We compute the values $A(v), B(v), C(v)$ bottom-up, starting from the leaves of the tree. Running-time: every node participates only in two sums, as it has only one father and only one grandfather. Thus the running time is $O(|T|)$.

2. 0-1 Knapsack. We have a sack of size N and a set of k objects, each with size x_i and value w_i. For every subset of objects its size is the total size of its elements and its value is the total value of its element. Give an algorithm finding a subset of maximal value that can be put in the sack.

   Define the subproblems:

   $$A(j, m) = \max\{\sum_{i \in S} w_i \mid S \subset [1..j], \sum_{i \in S} x_i \leq m\}$$

The solution to the big problem is clearly $A(k, N)$. To get the recursion formula, note the following: if subset S gives $A(j, m)$, it either contains the j-th object or not. If it does not, then $A(j, m) = A(j-1, m)$. If it does, the $A(j, m) = w_j + A(j-1, m-x_j)$. We obtain

$$A(j, m) = \max\{A(j-1, m), w_j + A(j-1, m-x_j)\}$$

We compute the values of $A(j, m)$ bottom up. Running time: every A(j,m) takes $O(1)$ time, thus the total is $O(kN)$.

3. $G = (V, E)$ is a complete graph with $|V| = n$. A path $(v_{i_1}, \ldots, v_{i_n+1})$ in G is called a *bitonic cycle* if:

   - $i_1 = i_{n+1} = 1$
   - The path goes through all the vertices
   - There exists $1 < j < n+1$ such that $i_k < i_m$ for $k < m \leq j$ and $i_k > i_m$ for $j \leq k < m$.

   Let $w : E \to R^+$ be a length function on the edges. Give an algorithm that finds the shortest bitonic cycle.

   A bit of intuitive notation: order the vertices on the plane such that if $i < j$ then v_i is to the left of v_j. What we look for is a path that starts at the left end, goes to the right (possibly skipping nodes on the way) up to v_n and then goes left till it gets to v_1. Note that every bitonic path contains the edge (v_{n-1}, v_n). We can thus rephrase the problem as follows: find a path $v_{i_1}, \ldots, v_{i_n}$ such that $i_1 = n$, $i_n = n-1$, the path goes through all the vertices and there exists $1 < j \leq n$ such that $i_k > i_m$ for $k < m \leq j$ and $i_k < i_m$ for $j \leq k < m$ (in other words, it goes left from v_n till v_1 and then right to v_{n-1}.

   For $i > j$ we shall call a path p from v_i to v_j *normal* if it starts at v_i, ends at v_j, goes through all vertices v_k for $k \leq i$ and first goes left and then right, as above. Our subproblems are:

   $$A(i, j) = \min\{w(p) \mid p \text{ is a normal path from } v_i \text{ to } v_j\}$$

   with $A(n, n-1)$ being the solution we seek. Now to the recursion formulas: note that if $i > j+1$ then a normal path p has to go through v_{i-1} on the way left, because that is its only chance to do so. In this case, $A(i, j) = w(i, i-1) + A(i-1, j)$. We now consider the case $i = j+1$. If we look at the path p backwards, it starts at v_j, goes left, then right and finally gets to v_{j+1}. Call v_k the last vertex before v_{j+1}. Now this path from v_j to v_k is a normal path. We obtain

   $$A(j+1, j) = \max_{1 \leq k < j} A(j, k) + w(k, j+1)$$

   Armed with the recursion formulas, we can compute the values of $A(i, j)$ bottom up. Running time: there are $O(n^2)$ values to compute. All but $O(n)$ of them (those with i=j+1) require $O(1)$ time each and those with $i = j+1$ require $O(n)$ each. In total, the running time is $O(n^2)$.

4. Give an algorithm finding the maximal flow in a flow network with multiple sources and sinks.

We are given a flow network on a graph $G = (V, E)$ and capacities c. Let S be the set of sources and T the set of sinks. We shall define a new flow network on graph $G' = (V', E')$, defined as follows:

$$V' = V \cup \{s_0\} \cup \{t_0\} \qquad E' = E \cup (\{s_0\} \times S) \cup (T \times \{t_0\})$$

the capacities on the old edges remain the same, and all the new edges have infinite capacities.

Let f be a flow on G. We shall define a flow f' on f. For every edge $e \in E$, $f'(e) = f(e)$. If $e = (s_0, s)$ for s a source in G, we set $f'(e) = \sum_{v \in V} f(s, v)$. If $e = (t, t_0)$ for t a sink in G, we set $f(e) = \sum_{v \in V} f(v, t)$. Clearly, f' is a flow in G', as now for every vertex except s_0 and t_0 the incoming flow is equal to the outgoing flow. We also have $|f'| = |f|$.

Let f' be a flow on G'. The restriction of f' to G gives a flow f on G which observes all the capacities. We again have $|f| = |f'|$. We obtain an $1 : 1$ and onto correspondence between flows on G and flows on G' which preserves the value of the flow. Thus if we find the maximal flow on G', we will get a maximal flow on G.

5. You are running the Mossad. You have n agents in Asia, each sitting in a different city. You want them to be in America, again, each one in a different city. There are only flights from Asia to Europe and from Europe to America. You have the full flight chart. To keep the secrecy level high, you don't want different agents to pass through the same city. Give an algorithm which gives a solution or says that none exists.

   Imagine that every city in Europe has two airports, one for flights to/from Asia and another one to/from America. Make a graph $G = (V, E)$ with a vertex for each airport, plus a source node s and a sink node t. s is connected to all n cities in Asia and t is connected to all n cities in America. Two airports are connected in the graph if there is a flight connecting them or they are in the same city. We give every edge in the graph capacity 1.

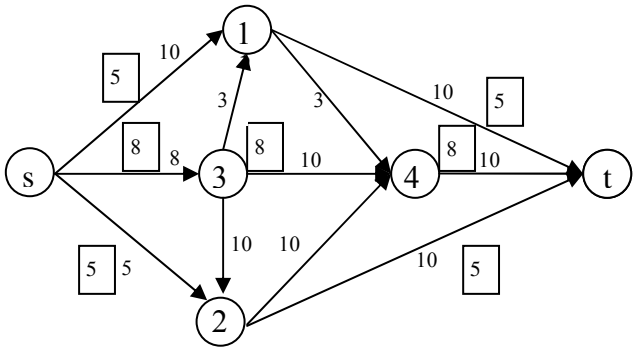
   Claim: a solution to the original problem exists iff the value of maximal flow in the network is n. Proof: Let f be an integer flow of value n in the network (we can assume it is an integer flow, as all the capacities are integers). Look at the set of all edges between two airports with flow$= 1$. This set consists of n paths of length 3 which are disjoint in their vertices: two paths in it cannot have a joint vertex v is Asia, as the capacity of (s, v) is 1, same goes for America. Two paths cannot have a joint airport v in Europe, as there is only one edge connecting v to the other side of town and its capacity is 1.

   On the other hand, if there exists a solution to the problem, it gives a flow of value n: put $f(e) = 1$ for every edge used, and put $f(e) = 1$ for every edge coming from the source or going to the sink.

Algorithms – Exercise 4

Due Wednesday 22/11/06 24:00

1. Consider the following flow network:



The capacity of each edge appears as a label next to the edge, and the numbers in the boxes give the amount of flow sent on each edge. (Edges without boxed numbers – specifically, the four edges of capacity 3 – have no flow being sent on them.)
Is the given flow a maximum flow? If not, run the Ford-Fulkerson algorithm and give the residual networks in all intermediate steps, until a maximum flow is achieved.

2.

Given a bipartite graph *G=(V,E)*.

a) Is *(E,I)* where *I* is the set of matchings in *G* a matroid ?
b) Prove that the following pair *(S,I)* is a matroid:
   S = L;
   I is the set of all subsets *A* of *L* such that there exists a matching in which all members of *A* participate. (A vertex *v* is said to participate in a matching *M* if there exists an edge $e \in M$ which is adjacent to *v*)
c) Assume there is a weight function on L, $v : L \to R^+$. Give an efficient algorithm that finds the matching in G which maximizes the sum of *v* over all vertices in L which participate in the matching.
   (Hint: To prove the exchange property notice that if you take the union of two matchings, the result is a graph made up of connected components that are either a cycle or a path. In each connected component the edges alternate between the two matchings.)

3. Let *G=(V,E)* be a network with source *s*, sink *t*, and integer capacities. Suppose we are given a maximal flow in *G*.
   a. Suppose the capacity of a single edge $(u,v) \in E$ is increased by 1. Give an *O(E)*-time algorithm to update the maximum flow.

b. Suppose the capacity of a single edge $(u,v) \in E$ is decreased by 1. Give an *O(E)*-time algorithm to update the maximum flow.

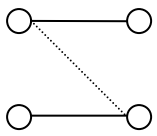
4. A perfect matching in a graph is a matching in which every vertex is matched. Let *G=($L \cup R$,E)* be an undirected bipartite graph with $|L| = |R|$. For every $A \subseteq L \cup R$ we define *N(A)* to be the set of neighbors of vertices from *A*, i.e. $N(A) = \{u \in L \cup R : \exists v \in A.(u,v) \in E\}$.
   Prove Hall's matching theorem using flow networks:
   If for every $A \subseteq L, |A| \le |N(A)|$ then there exists a perfect matching in *G*.

5. (reshut)
   Suppose we allow a flow network to have negative (as well as positive) edge capacities. In such a network, a feasible flow need not exist.
   Let *G=(V,E)* be a flow network with negative edge capacities, and let *s* and *t* be the source and sink of *G*. Construct the ordinary flow network *G′=(V′,E′)* with capacity function *c′*, source *s′*, and sink *t′*, where
   $V' = V \cup \{s', t'\}$
   $E' = E \cup \{(u,v) : (u,v) \in E\} \cup \{(s',v) : v \in V\} \cup \{(u,t') : u \in V\} \cup \{(s,t),(t,s)\}$
   We assign capacities to edges as follows:
   For each edge $(u,v) \in E$, we set
   $c'(u,v) = c'(v,u) = (c(u,v) + c(v,u))/2$
   For each vertex $u \in V$, we set
   $c'(s',u) = \max(0,(c(V,u) - c(u,V))/2)$
   $c'(u,t') = \max(0,(c(u,V) - c(V,u))/2)$
   We also set $c'(s,t) = c'(t,s) = \infty$.
   a. Prove that if feasible flow exists in *G*, then all capacities in *G′* are nonnegative and a maximum flow exists in *G′* such that all edges into the sink *t′* are saturated.
   b. Prove the converse of part (b). Your proof should be constructive, that is, given a flow in *G′* that saturates all the edges into *t′*, your proof should show how to obtain a feasible flow in *G*.
   c. Describe an algorithm that finds a maximum feasible flow in *G*.

Algorithms – Exercise 4 (Solution)

1. A sketch of the solution:
   We find augmenting paths, and augment the flow along them. We can see it is not a maximum flow since the path (s,1,t) is a path in the residual network, with capacity 5. Augmenting along this path will give us a residual network which does not contain any edges leaving s, and therefore no paths between s and t. This means that after augmenting along (s,1,t) the flow achieved is a maximal flow.

2.
   a. No, consider the following bipartite graph:

   Then the dotted edge is a matching (of size 1), the filled edges are also a matching (of size 2). But there is no filled edge you may add to the dotted edge, so they will together form a matching. Therefore the exchange property is not satisfied.
   b. We prove that $M=(S,I)$ is a matroid:

   I. S is final and isn't empty as $S = L$ and L is finite and non empty by the definition of the graph.

   II. **Hereditary**: Let $B \in I$, i.e. there's a perfect matching between B and a subset B' of R, and let A be a subset of B: $A \subseteq B$. To prove that $A \in I$ we need to show that there is a subset $A' \subseteq R$ and a perfect matching between A and A': We choose the vertices in B' that are matched to the vertices in A in the perfect matching between B and B'. Clearly there is a perfect matching between A and A'.

   III. **Exchange property:**
   Let $A \in I$, $B \in I$ and $|A| < |B|$. We're looking for a vertex $u \in B - A$ such that $A \cup \{u\} \in I$:
   Let M be a perfect matching between B and a subset of R, and let M' be a perfect matching between A and a subset of R. Let us look at the graph $M \cup M'$, and consider its connected components:

      1. If a vertex u is part of a **circle**, its degree is 2 (an edge exiting and an edge entering), which means it belongs to both A & B (in a single perfect matching the degree of every vertex is at most 1).
      2. In a **simple path of even length** (starting and ending on the same side), the number of vertices in A is equal to the number of vertices in B: all the vertices that are not

endpoints in the path must belong to both A and B (their degree is 2). As for the endpoints, one must belong to B and the other to A, otherwise it would contradict the perfect matching of A or B.

3. Finally we get to **simple paths of odd length**. They are the only candidates for connected components that contains more vertices from B than from A. So there must be such a component that has a vertex u at one of the path's ends such that $u \in B - A$.
   Also, the vertex on the other end cannot belong to A', i.e. the set in R that is matched to A (otherwise the edge entering it would be the second one exiting from one vertex in A, contradicting the perfect matching).
   We add vertex u to A and change the matching of the vertices in this component to use only the edges in this component that exited from B. We know such edges exist in the graph and we also know they give perfect matching for the vertices in this component. In other components we keep the matching of A as it was. So we get a perfect matching for all vertices in $A \cup \{u\}$.

c. After we proved that *(S,I)* defined in b is a matroid we may use the generic matroid algorithm:
   i. $A \leftarrow \varnothing$
   ii. Sort the vertices in L according to their weights v (descending order).
   iii. For each $x \in S$ taken in monotonically decreasing order by the weight $v(x)$
      1. if $A \cup \{x\}$ has a perfect matching in R then $A \leftarrow A \cup \{x\}$

   This algorithm runs in polynomial time: first we sort in $O(L*log(L))$, then the loop has L iterations. Each iteration tests if there is a perfect matching for a subset of L (done in $O(EV)$ with Edmonds-Karp, see Cormen p. 668). The unions can be done in $O(V)$. And therefore we get $O(EV^2)$ running time.

3.

1. Suppose we have a maximum flow f. So the residual network G_f does not contain any paths between s and t. We will increase the capacity of a single edge (u,v) by 1, and receive a new residual network G_f. We now search for an augmenting path, if we didn't find any, it is clear that f is still a maximum flow. If we did, we will augment f along this path. We claim that the new flow f' is a maximum flow:
   First, since the capacities are integers, the maximum flow must be an integer and therefore $|f'| \geq |f| + 1$. According to Max flow-Min cut theorem, the minimal cut in G before the capacity increment had capacity $|f|$. Since the capacity of only one edge in G was increased by 1 with the capacity increment, we find that the minimal cut in G now has

capacity $\leq |f|+1$. Therefore by introducing a flow with that capacity, we prove it is the maximal flow (again using the Min cut-Max flow theorem. The running time of the algorithm equals the sum of:

a. the time to find a path between *s* and *t* in G_f, this can be done in *O(E)*.
b. the time to update *f* along the augmenting path. Done in *O(V)*.

The running time is then *O(V+E)=O(E)*.

2. Now suppose we decrease the capacity of a single edge *(u,v)* by 1. Let *f* be a maximum flow before the decrement. If *f* did not saturate the edge *(u,v)*, then *f* is also a maximal flow after the decrement, and therefore we may output *f*. Else, let's consider a flow *f'* which is produced from *f* by reducing 1 from the flow of *f* along a path from *s* to *t* through which *f* sends positive flow. Formally: we define *G'* to be a graph with vertices identical to *G* and edges along all the edges of *G* in which *f* sends a positive flow. We now find a path *P* between *s* and *t* in *G'* which includes the edge *(u,v)* (how can this be done ?). We define a new flow in *G*, *f'* which is identical to *f* for every edge not in *P*, and equals one less then *f* in every edge of *P* (and one more in the other direction of course). It can be verified that *f* is a flow in *G* and that $|f'|=|f|-1$. Now since the maximal flow in *G* was $|f|$ before the decrement, then it is either $|f|$ or $|f|$-1 after the decrement (since min-cut =max-flow and the capacity of every cut was reduced by at most 1 following the decrement). Therefore, it we now try to find an augmenting path in $G_{f'}$, then either there are none and then *f'* maximal, or there is an augmenting path, we will augment *f'* along this path and achieve a flow of at least $|f'|+1=|f|$, which is maximal (since it was even maximal in *G* before the decrement).
   The running time is then:
   1. *O(E)* finding a path between *s* and *t* in the graph *G'*.
   2. *O(V)* calculating *f'* from *f*, and calculating $G_{f'}$.
   3. *O(E)* finding an augmenting path in $G_{f'}$.

   All together *O(E+V)=O(E)* running time.
   Note: notice that there must be a path between *s* and *t* that passes through *(u,v)* in *G'*. *f* saturates *(u,v)*, by conservation of flow, there is a vertex *w* such that *(w,u)* is an edge in *G'*. Again by conservation of flow, there is a vertex w_2 such that $f(w_2,w)>0$. Again by conservation of flow we find w_3 such that $f(w_3,\{w,w_2\})>0$, and so we continue iteratively to find a series $w,w_2,w_3,\ldots,w_n$ such that $f(w_n,\{w,w_2,..,w_n\})>0$. The series can only end when we have reached w_n=s. And so there is a path from *s* to *u* similarly we can find a path from *v* to *t*, and concatenate them.

4. We convert the given graph into flow network, and the existence of the perfect matching is equivalent to existence of the flow of size n (number of vertices in *L*). According to MAX-FLOW MIN-CUT theorem it is enough to prove that if for each $A \subseteq L$, $|A| \leq |N(A)|$ then there exists cut of size n. Look at the following cut: the source *s*, $n-k$ vertices in *L*, and $n-m$ vertices in *R*. In this cut there are *k* edges from *s* to *L* and $n-m$ edges from R to t. Let's see how many edges are from *L* to *R*. It's given that the $n-k$ vertices in L have at least $n-k$ neighbors in *R*. Assume that all $n-m$ vertices in *R* which are inside the cut are among them, but then there are at least $(n-k)-(n-m)$ vertices

outside the cut so this is the amount of edges from L to R crossing the cut. The total number of edges in the cut is at least $k + (n - m) + (n - k) - (n - m) = n$.

5.

a. Let f be a feasible flow in G. First we prove that all capacities in G' are non-negative. It is enough to prove that for every u,v, $c'(u,v) \geq 0$, namely that $c(u,v)+c(v,u) \geq 0$. This can be shown through:
$c(u,v)+c(v,u) \geq f(u,v)+f(v,u) = 0$.
Now, we will construct from f, a flow f' which saturates all the edges entering t'. We define for every $(u,v) \in E$, $f'(u,v)=f(u,v)+\frac{c(v,u)-c(u,v)}{2}$, for every $u \in V$, $f'(s',u) = max(0,\frac{c(V,u)-c(u,V)}{2})$, $f'(u,t') = max(0,\frac{c(u,V)-c(V,u)}{2})$ and finally $f'(s,t) = -f'(t,s) = -|f|$. This defines a flow in G', the skew symmetry and capacity constraints are immediate. The conservation of flow is shown as follows:
if $u \in V$, then

$$\sum_{v \in V'} f'(s,v) = f(s,V) + \frac{c(s,V)-c(V,s)}{2} + f(s,s') + f(s,t') + f(s,t) =$$

$$|f| + \frac{c(s,V)-c(V,s)}{2} - max(0,\frac{c(s,V)-c(V,s)}{2}) + max(0,\frac{c(s,V)-c(V,s)}{2}) - |f|$$

$$= 0$$

if $u=s$, then

$$\sum_{v \in V'} f'(u,v) = f(u,V) + \frac{c(u,V)-c(V,u)}{2} + f(u,s') + f(u,t') =$$

$$0 + \frac{c(u,V)-c(V,u)}{2} - max(0,\frac{c(u,V)-c(V,u)}{2}) + max(0,\frac{c(u,V)-c(V,u)}{2}) = 0$$

Similarly, if $u=t$.
b. Now we assume that we have a flow f' in G which saturates the edges entering t'. First we observe, that f' is a maximal flow and that (V\{t'},{t'}) is a minimal cut in G' (according to Min cut-Max flow theorem). We will show that ({s'},V\{s'}) is also a minimal cut. Indeed, we define
$A = \{u \in V' : c(V,u) - c(u,V) > 0\}$, then

$$c'(s',V \setminus s') = \sum_{u \in V'} c'(s',u) = \sum_{u \in A} \frac{c(V,u)-c(u,V)}{2} =$$

$$c(V,A) - c(A,V) = c(V \setminus A,V) - c(V,V \setminus A) =$$

$$\sum_{u \in V \setminus A} \frac{c(u,V)-c(V,u)}{2} = \sum_{u \in V'} c'(u,t')$$

and therefore, we conclude that f' saturates the cut ({s'},V\{s'}), as well. Now we simply define f to be $f(u,v)=f'(u,v)- \frac{c(v,u)-c(u,v)}{2}$. A similar calculation to that done in the last question proves that f is indeed a feasible flow in G. We conclude that maximal flows in G' match feasible flows in G, and the value of the flow is linked through the equation $|f| = -f'(s,t) = f'(t,s)$.

c. From the previous two articles we find that we can restate the problem of finding a maximal feasible flow in G, by finding the maximal flow in G' which maximizes the value $f'(t,s)$. We can now approach the problem as follows: first run the Edmonds-Karp algorithm on G' to obtain a maximal flow f', if f' does not saturate all the edges entering t', declare "No feasible flows in G''. Now produce from f' a flow f through the transformation given in article b. Now turn back to the flow network G, and start running the Edmonds-Karp algorithm on G with the only alteration that at initialization the flow is not identically 0, but rather f. It can be proven that the capacities of the residual network are positive, that the flows produced in this method are feasible flows in G, and that flow produced is a maximal flow in G. The proof is identical to the proof in the case where there are only non-negative capacities and can simply be replicated.
Running Time: The algorithm consists of two runs of the Edmonds-Karp algorithm on G and G', as well as building the graph G' in and constructing f from f'. This can all be done in $O(V^2E)$.

Algorithms - Exercise 5

Due Wednesday 29/11 24:00

1. The *edge connectivity* of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. Give an algorithm that determines the edge connectivity. (Hint: run the maximal-flow algorithm on $O(|V|)$ flow networks)

2. A *path cover* of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths can be of any length, including 0. A *minimal path cover* of G is a path cover containing the fewest possible paths.

   (a) Give an efficient algorithm to find a minimal path cover of a directed acyclic graph $G = (V, E)$. (Hint: look at the graph $G' = (V', E')$, assuming $|V| = n$

   $$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_o, y_1, \ldots, y_n\}$$

   $$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}$$

   (b) Does your algorithm work for directed graphs that contain cycles?

3. (a) Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications.

   (b) Give two algorithms for multiplying two polynomials with degree $\leq n$ that run in time $\Theta(n^{\log 3})$. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.

   (c) Show that two n-bit integers can be multiplied in $O(n^{\log 3})$ steps, where each step operates on at most a constant number of 1-bit values.

4. Explain what is wrong with the 'obvious' approach to polynomial division using a point-value representation, i.e. dividing the corresponding y values.

5. (a) Find the polynomial $p(z)$ with degree< 4 such that

   $$p(1) = 4 - i \quad p(i) = -1 + 2i \quad p(-1) = -6 - i \quad p(-i) = -1$$

   (b) Use the FFT algorithm to compute the values of $p(z) = iz^3 - z^2 + 2z - i$ in the 8-th roots of unity.

Solution to Exercise 5 in Algorithms

1. Notation: the edge connectivity of G is $C(G)$. Algorithm: Give capacity 1 to every edge in the graph G. Pick any vertex in V and call it s. Now for every $t \in V \setminus \{v\}$ find the maximal flow $h(t)$ of the network with t as a sink. Claim: $C(G) = \min_t h(t)$.

   Proof: Let A be a set of edges such that $(V, E \setminus A)$ is disconnected and $|A| = C(G)$. Pick t to be any vertex which does not belong to the connected component of s in $(V, E \setminus A)$. The maximal flow $h(t)$ is the size of a minimal cut between s and t, thus $h(t) \leq C(G)$. On the other hand, for any $t \neq v$, we have $h(t) \geq C(G)$, as $C(G)$ is the minimal number of edges to be taken away in order to disconnect the graph. We obtain $C(G) = \min_{t \neq v} h(t)$.

2. (a) We build the graph G' as hinted and use it to find the maximal matching M between V and itself. M is a collection of edges (x_i, y_j) for some i-s and j-s. Let A be the set of edges $A = \{(y_i, x_i) \mid 1 \leq i \leq n\}$. Let H be the graph with vertices $x_1 \dots x_n, y_1 \dots y_n$ and edges $M \cup A$. Every vertex in H has at most 2 edges: one incoming and one outgoing. There are no cycles in G, so there are no cycles in H. Thus the edges in H form a set of disjoint paths, each of the form: $y_{i_1}, x_{i_1}, y_{i_2}, x_{i_2}, \dots, y_{i_k}, x_{i_k}$. For each of these paths, we take the path $(i_1, i_2, \dots, i_k)$ as a path in our path cover of V.

      Proof of correctness: First, the set of paths we obtain is a path cover, as the paths are disjoint in vertices and every vertex sits on some path. To show optimality, we show that every path cover of G comes from a matching. Let $\{p_\alpha\}$ be a path cover of G of size k. Define $M = \{(x_i, y_j) \mid (i, j) \text{ is an edge in some path } p_\alpha\}$. M is a matching, because the paths in the path cover are disjoint. Now the size of the matching, $|M|$, is the total number of edges in the path cover and we have $k = n - |M|$ (the number of connected components in a forest with n vertices and M edges is $n - k$). It follows that finding the maximal matching will give the minimal path cover.

   (b) No

3. (a) Compute $a' = ac$, $b' = bd'$ and $e = (a + b)(c + d)$. Then $(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd = a'x^2 + (e - a' - b')x + b'$

   (b) Let f and g be two polynomials of degree $< n$. Write $f = f_l + f_b x^{n/2}$ and $g = g_l + g_b x^{n/2}$ where f_l, f_b, g_l, g_b are polynomials of degree $< n/2$. Exactly as in (a) we get $fg = f_l g_l + ((f_l + f_b)(g_l + g_b) - f_l g_l - f_b g_b)x^{n/2} + f_b g_b x^n$, thus we only need to make 3 multiplications of polynomials of degree$< n/2$. If $T(n)$ is the running time, we obtain the recursion formula

$$T(n) = 3T(n/2) + O(n)$$

   Solving it gives $T(n) = O(n^{\log_2 3})$.

(c) Let $y = \sum_{i=0}^{n-1} a_i 2^i$ and $z = \sum_{i=0}^{n-1} b_i 2^i$ be two n-bit integers. We define polynomials f and g as follows: $f = \sum_{i=0}^{n-1} a_i x^i$ and $g = \sum_{i=0}^{n-1} b_i x^i$. Compute $h = fg$. Now $y = f(2)$ and $z = g(2)$, so $xy = h(2)$.

Input: $y = \sum_{i=0}^{n-1} a_i 2^i$, $z = \sum_{i=0}^{n-1} b_i 2^i$
$f = \sum_{i=0}^{n-1} a_i x^i$, $g = \sum_{i=0}^{n-1} b_i x^i$
Compute $h = fg$, with $h = \sum_{i=0}^{2n-2} c_i x^i$
We now want to compute $h(2)$
for $i = 0$ to $2n-1$ **do**
  $d_i = (r_{i-1} + c_i)\ mod\ 2$ the new digit
  $r_i = (r_{i-1} + c_i - d_i)/2$ the carrier
end for
Output $yz = \sum_i d_i 2^i$

Running time: The polynomial multiplication takes $O(n^{\log_2 3})$. Note that all the coefficients c_i are bounded by n, as $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. Thus r_i is also bounded by n. Every stage of the loop takes $O(\log n)$ time, so the total is $O(n^{\log_2 3} + n \log n) = O(n^{\log_2 3})$

4. Let f and g be two polynomials. The obvious approach to division would be to compute the values of f and g in roots of unity using FFT and then find a polynomial h such that $h(\omega^k) = \frac{f(\omega^k)}{g(\omega^k)}$. The problem is that even if $f = gp$ for some polynomial p, it may happen that $g(\omega^k) = 0$ and $p(\omega^k) \neq 0$. In this case, $f(\omega^k) = 0$ as well and we cannot compute the value of p in ω^k directly from f and g.

5. (a) $z^3 - iz^2 + 2z - 1$

(b) $p(z) = iz^3 - z^2 + 2z - i$. Divide into the odd and the even part: $p_e = -z - i$ and $p_o = iz + 2$. Each one of these has to be evaluated at the 4th roots of unity. Divide again: $p_{ee} = -i$,$p_{eo} = -1$, $p_{oe} = 2$ and $p_{oo} = i$. We evaluate each of these in 2nd roots of unity, 1 and -1.

$$p_{ee}(1) = p_e e(-1) = -i \quad p_{eo}(1) = p_e o(-1) = -1 \quad p_{oe}(1) = p_o e(-1) = 2 \quad p_{oo}(1) = p_o o(-1) = i$$

Next step

$$p_e(1) = p_{ee}(1^2) + 1p_{eo}(1^2) = -i - 1 \quad p_e(i) = p_{ee}(i^2) + ip_{eo}(i^2) = -2i$$

$$p_e(-1) = p_{ee}((-1)^2) - 1p_{eo}((-1)^2) = 1 - i \quad p_e(-i) = p_{ee}((-i)^2) - ip_{eo}((-i)^2) = 0$$

For the odd one

$$p_o(1) = p_{oe}(1^2) + 1p_{oo}(1^2) = 2 + i \quad p_o(i) = p_{oe}(i^2) + ip_{oo}(i^2) = 1$$

$$p_o(-1) = p_{oe}((-1)^2) - 1p_{oo}((-1)^2) = 2 - i \quad p_o(-i) = p_{oe}((-i)^2) - ip_{oo}((-i)^2) = 3$$

Finally

$$p(1) = p_e(1^2) + 1p_o(1^2) = 1 \quad p(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i) = p_e(i) + (\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i)p_o(i) = \frac{1}{\sqrt{2}} + (\frac{1}{\sqrt{2}} - 2)i$$

on so forth. Conclusion: some things are better left for computers.

Algorithms – Exercise 6

Due Wednesday 20/12/06 24:00

1. Compute the prefix function π defined in the string matching algorithm for the given pattern: [a,b,a,c,a,b,c,a,a,b]

2. Explain how to determine the occurrences of pattern *P* in the text *T* by examining the π function for the string *P^T* (the concatenation of *P* and *T*).

3.
   a. What is the maximum number of times that any character in the text will be looked at by the stringSearchKMP function?
   b. Is it possible that adjacent characters in the text will be considered this many times?

4.
   a. Find all the integers *a* such that:
      i. $a \bmod 3 = 6$
      ii. $a \bmod 11 = 20$
      iii. $a \bmod 20 = 5$
   b. Show that if n_1,n_2 are not relatively prime, then there exists a_1,a_2 such that the following set of equation does not have a solution *x:*
      i. $x \bmod n_1 = a_1$
      ii. $x \bmod n_2 = a_2$

5. You are given a number *N*. Your task is divided into two stages:
   a. Phase I: you are given arbitrarily large space and time to run any calculations you wish on *N*. You may store the results of your calculations.
   b. Phase II: the memory you used in phase I now turns into read-only memory. You are now given only $O(\log(\log(N)))$ bits of disk space, and a read-only array *A* composed of {0,1} of length *O(N)*.

   Give an algorithm that runs according to these two stages, and outputs "true" if the number of *1*'s in *A* is *N*, and "false" otherwise.

   Hints:

1. Use phase I to store $a_i=(N \bmod p_i)$, when p_i is the i-th prime number, for $i=1,..,k$ such that k is large enough to guarantee that the a_i determine N uniquely. (Show that $k=O(\log(N))$ is sufficient)
2. You may assume that $p_i=O(i \cdot \log(i))$, show how to determine if the number of *1*'s in A is N using only a space of $O(\log(p_k))$ bits.

Algorithms – Exercise 6 (Solution)

1. We can use a divide and conquer technique for the problem using the following observation:

$$\prod_{i=1}^{n}(x-x_i)=\prod_{i=1}^{\lfloor n/2\rfloor}(x-x_i)\prod_{i=\lfloor n/2\rfloor+1}^{n}(x-x_i)$$

Pseudo-Code:

Multiply(*i,j*)
//Multiplies the i-th until j-th terms of the polynomial ($j \geq i$)
If (*i == j)* return *($x-x_i$)*;
Return
Multiply_by_FFT(Multiply($i, i+\lfloor (j-i)/2 \rfloor$),Multiply($i+\lfloor (j-i)/2 \rfloor+1, j$));

Where Multiply_by_FFT is simply a sub-routine that multiplies two polynomials of degree bounded by *n* using FFT in time *O(nlog(n))*.
Running time: we'll denote the running time function *T(n)*, then
T(n)=2T(n/2)+O(nlog(n))
And we conclude that *$T(n)=O(n\log^2(n))$*. (e.g. by simple induction)

2. Pseudo-Code:

FFT4(a)
//a is a vector of size 4^n for some $n \in \mathbb{N}$
$n \leftarrow length(a)$;
If (n==1) return a;
$w_n \leftarrow e^{2\pi i/n}$;
$w \leftarrow 1$;
$a_0 \leftarrow (a_0,...,a_{n-4})$; $a_1 \leftarrow (a_0,...,a_{n-3})$; $a_2 \leftarrow (a_0,...,a_{n-2})$; $a_3 \leftarrow (a_0,...,a_{n-1})$;
$y_0 \leftarrow FFT4(a_0)$; $y_1 \leftarrow FFT4(a_1)$; $y_2 \leftarrow FFT4(a_2)$; $y_3 \leftarrow FFT4(a_3)$;
for k←0 to n/4-1
$A[k] \leftarrow y_0[k]+w\,y_1[k]+w^2\,y_2[k]+w^3\,y_3[k]$;
$A[n/4+k] \leftarrow y_0[k]+i\,w\,y_1[k]-w^2\,y_2[k]-i\,w^3\,y_3[k]$;
$A[n/2+k] \leftarrow y_0[k]-w\,y_1[k]+w^2\,y_2[k]-w^3\,y_3[k]$;
$A[3n/4+k] \leftarrow y_0[k]-i\,w\,y_1[k]-w^2\,y_2[k]+i\,w^3\,y_3[k]$;
return A;

The algorithm is justified through the following equation:

$$P(x)=\sum_{i=0}^{n-1}a_i\cdot x^i=\sum_{i=0}^{n/4-1}a_{4i}\cdot x^{4i}+x\sum_{i=0}^{n/4-1}a_{4i+1}\cdot x^{4i+1}+x^2\sum_{i=0}^{n/4-1}a_{4i+2}\cdot x^{4i+2}+x^3\sum_{i=0}^{n/4-1}a_{4i+3}\cdot x^{4i+3}=$$

$$P_0(x^4)+xP_1(x^4)+x^2P_2(x^4)+x^3P_3(x^4)$$

Now since computing the *FFT* of *a* is equivalent to evaluating the polynomial defined by *a* in the *n=length(a)* roots of unity of order *n,* and since we know that raising those *n* roots of unity to the fourth power, yields 4 equal and consecutive sets of the *n/4* roots of unity of order *n/4*, we see that we can compute the *FFT* of *a* through computing the *FFT* of the four polynomials

P_0, P_1, P_3, P_3 . The *for* loop calculates in each iteration the *k-th,k+n/4-th,k+n/2-th, k+3n/4-th* entry of the *FFT*. An exact proof for the correctness of this algorithm follows the same lines as that of the regular *FFT* (see Cormen ch. 30).
The running time of the algorithm is given by the following recursion formula: $T(n)=4T(n/4)+O(n)$
Which yields $T(n)=O(n\log(n))$. (Proof e.g. by induction)

3. We assume the alphabet is {-1,1} or change every 0 to -1.

   Define polynomials P_{SR} and P_T for *T* and *S-Reverse*:

   $$P_{SR} = \sum_{i=0}^{n} s_{n-i} \cdot x^i \ , \ P_T = \sum_{i=0}^{m} t_i \cdot x^i$$

   Multiply P_{SR} and P_T through FFT:

   $$P = P_{SR} \cdot P_T = \sum_{j=0}^{n+m} c_j \cdot x^j \text{ , where } c_j = \sum_{k=0}^{j} s_{n-k} \cdot t_{j-k} \text{ . (if } n<k, \ s_{n-k}=0 \text{).}$$

   c_j is the sum of at most $n+1$ non-zero products ($k=0,..,n$), and since $s_{n-k}=0$ and t_{j-k} can be either 1 or -1 for each *j*, we conclude that $c_j \le n+1$.

   $c_j = n+1 \Leftrightarrow$ for every $0 \le k \le n$, both s_{n-k}, t_{j-k} are 1 or -1 $\Leftrightarrow$

   $\Leftrightarrow$ for every $0 \le k \le n$, $s_{n-k} = t_{j-k}$, meaning a match was found.

   We therefore return every index *j*, such that $c_j=n+1$.

   Run time:

   a. Defining the polynomials is $O(m+n)=O(m)$ time.

   b. FFT: $O(m\log(m))$.

   c. Checking which $c_j=n+1$ is $O(m+n)=O(m)$

   and totally: $O(m\log(m))$.

4. We use the following observation:

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

Now, given a point value representation of $A(x)$ in the points $(x_1,x_2,..,x_n)$ (none of the points is 0), we can derive a point value representation of $A(x)$ in the points $(y_1,y_2,..,y_n)=(x_1^{-1},x_2^{-1},..,x_n^{-1})$ by:

$$A^{Rev}(y_i) = x_i^{1-n} A(x_i)$$

Since we consider arithmetical operations in this context (in FFT context) to be $O(1)$, evaluation at each point takes $O(1)$, and therefore the algorithm runs in $O(n)$ time in total.

Note: Pay attention that if the set of evaluation points is closed to inversion (i.e. $x \in A \Rightarrow x^{-1} \in A$), then this algorithm can be modified slightly to derive a point-value representation of A^{Rev} from the point-value representation of A in the same points of evaluation in $O(n)$ time. For instance, the roots of unity of a certain order are closed to inversion.

Algorithms - Exercise 7

Due Wednesday 13/12 24:00

1. For any $k > 0$ and any $n > 0$ let $p = \sum_{i=0}^{kn-1} a_i x^i$ be a polynomial where $a_i = 1$ if $k|i$ and $a_i = 0$ otherwise. Compute $FT_{kn}(p)$.

2. (reshut) Given a complex number z, the *z-chirp transform* of a vector $a = (a_0, a_1, \ldots, a_{n-1})$ is the vector $y = (y_0, y_1, \ldots, y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j z^{jk}$. Give a $O(n \log n)$-time algorithm to compute the transform for any z. Hint: Use the equation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} (a_j z^{j^2/2})(z^{-(k-j)^2/2})$$

   to view the chirp transform as a convolution.

3. Give a linear-time algorithm to determine if a text T is a cyclic rotation of another string T'. For example, *car* and *arc* are cyclic rotations of each other.

4. The binary gcd algorithm

   (a) Prove that if a and b are both even then $gcd(a, b) = 2gcd(a/2, b/2)$

   (b) Prove that if a is odd and b is even, then $gcd(a, b) = gcd(a, b/2)$

   (c) Prove that if a and b are both odd, then $gcd(a, b) = gcd((a - b)/2.b)$

   (d) Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in $O(\log(a))$ time. Assume that each subtraction, parity check and halving can be done in $O(1)$ time.

5. (a) Compute 10^{-1} mod 21 using the extended Euclid algorithm

   (b) What does $Ex - Euclid(f_{k+1}, f_k)$ return? Prove.

6. (a) Define $lcm(a_1, \ldots, a_n)$ to be the *least common multiple* of $a_1, \ldots, a_n$, that is, the smallest positive integer which is a multiple of each a_i. Give an efficient algorithm to compute $lcm(a_1, \ldots, a_n)$ using $Euclid(a, b)$ as a subroutine.

   (b) $a_1, \ldots, a_n$ are positive integers with $a_i \leq M$ for each i. Give an algorithm that checks whether $a_1, \ldots, a_n$ are pairwise relatively prime with runtime $O(n \log n \log M)$, assuming that operations on numbers take $O(1)$ time. ($a_1, \ldots, a_n$ are pairwise relatively prime if $gcd(a_i, a_j) = 1$ for every $i \neq j$.)

Algorithms – Exercise 7 (Solution)

1. The Fourier Transform of p, is the vector which results from assigning the polynomial p all the roots of unity of order kn. Let us denote a primitive kn-th root of unity by w, and then the roots of unity will be - $1,w,\ldots,w^{kn-1}$. The l-th coordinate of p is $\sum_{i=0}^{n-1} a_{k\cdot i}(w^l)^{i\cdot k} = \sum_{i=0}^{n-1} a_{k\cdot i}(w^{l\cdot k})^i$. Now if $w^{lk}=1$ (which happens iff $nk|lk$ iff $n|l$) then the coordinate is simply n, otherwise we get by the geometric series formula $\sum_{i=0}^{n-1} a_{k\cdot i}(w^{l\cdot k})^i = \frac{(w^{l\cdot k})^n - 1}{w^{l\cdot k} - 1} = \frac{w^{knl} - 1}{w^{l\cdot k} - 1} = 0$. Therefore, the Fourier transform of p equals n in all coordinates which n divides and 0 otherwise.

2. We will use the hint:

   $y_k = z^{k^2/2}\sum_{j=0}^{n-1}(a_j \cdot z^{j^2/2})(z^{-(k-j)^2/2})$ (This can be verified easily)

   We now define two polynomials:

   $p(x)=\sum_{i=0}^{n-1} a_i \cdot z^{i^2/2}\cdot x^i$, $q(x)=\sum_{i=-(n-1)}^{n-1} z^{-i^2/2}\cdot x^{i+(n-1)}$

   So that $p(x)q(x)=$*(terms of order < n-1)*$+\sum_{k=0}^{n-1}\sum_{j=0}^{n-1}(a_j \cdot z^{j^2/2})(z^{-(k-j)^2/2})\cdot x^{k+(n-1)}+$

   +(terms of order > 2n-2).

   And so the algorithm works as follows:

   1. Compute the vector $v=(z^{k^2/2})_{k=-(n-1)}^{n-1}$, and the vector $u=(a_k \cdot z^{k^2/2})_{k=0}^{n-1}$, and define the coefficient representation of $p(x),q(x)$.
   2. Calculate $p(x)q(x)$ using *FFT*, and store the coefficients between $n-1$ and $2n-2$ in $(m_1,\ldots,m_n)$
   3. Assign $y_k = v_k \cdot m_k$ for every k between 0 and $n-1$. And return the y_k-s.

   The above discussion proves the correctness of the algorithm.

   Running Time: In order to compute the vector v we require $O(n)$ operations, the same holds for u. Therefore stage 1 takes $O(n)$ time. Stage 2 takes $O(nlog(n))$ time for the *FFT*. And then step 3 takes some more $O(n)$ time. All together that makes $O(nlog(n))$ as required.

3. We recall from class that we can find all the occurrences of a pattern P of length n in a test T of length m in $O(m+n)$ time. First we check if the two strings are of equal lengths, if not we output "no". Then, we define $P=T$ and $T=T'T'$, and run the KMP algorithm. If the KMP found an occurrence of P in T, we return "yes", otherwise "no".
   Correctness: If the algorithm outputted "yes", this means that there is an occurrence of T inside $T'T'$, let's say the occurrence started in the m-th position, and the lengths of T,T' are n. Then the string which composes of the last m letters of T' and then the first n-m letters of T' equals T. Which means that T is indeed a cyclic rotation of T'.
   If the algorithm outputted "no", this means that either T and T' are not of the same length, or that T does not occur in $T'\hat{}T'$. Let us assume in contradiction that T is a cyclic rotation of T', then there exists an index m such that the last m letters of T' concatenated to the first n-m elements of T' equal T. But this means that T occurs in $T'\hat{}T'$ in the m-th position.
   Running Time: Since checking the lengths, and the KMP run in linear time, we receive a linear time algorithm.

4.
   a. By the fundamental theorem of algebra, a and b have the following representation $a=p_1^{l1}p_2^{l2}\ldots p_n^{ln}$, $b = p_1^{r1}p_2^{2}\ldots p_n^{rn}$ (we may assume that the prime factors are the same by allowing l_i to receive also the value 0). The $gcd(a,b)=p_1^{min(l1,r1)}\ldots p_n^{min(ln,rn)}$. In the case where a,b are even we get that one of the prime factors is 2 and that it appears in a positive multiplication in both the representation of a and that of b. Lets us assume without loss of generality that $p_1=2$. We therefore get - $a/2= p_1^{l1-1}p_2^{l2}\ldots p_n^{ln}$, $b/2 = p_1^{r1-1}p_2^{2}\ldots p_n^{rn}$ and $gcd(a/2,b/2)= p_1^{min(l1,r1)-1}\ldots p_n^{min(ln,rn)}=2gcd(a/2,b/2)$
   b. We will still assume $p_1=2$. In the case where a is odd and b is even, we get $l1=0$ and $r1>0$, therefore $min\{l1,r1\}=0=min(l1,r1-1)$, and $gcd(a,b/2)= p_1^{min(l1,r1-1)}\ldots p_n^{min(ln,rn)}= p_1^{min(l1,r1)}\ldots p_n^{min(ln,rn)}=gcd(a,b)$.
   c. If a,b are both odd then we get $a-b$ is even. We will use the fact that for every a,b there holds $gcd(a,b)=gcd(a-b,b)$ (see below), but from article b for $a-b,b$ we get $gcd(a,b)=gcd((a-b)/2,b)$.
      Lemma: for every a,b: $gcd(a-b,b)=gcd(a,b)$
      Proof: First, if d is a common divisor of a,b then it is a divisor of $a-b$, and therefore a common divisor of $a-b,b$. Second, if d is a common divisor of $a-b,b$ it is a divisor of $(a-b)+b=a$ and therefore a common divisor of a,b. Therefore the common divisors of a,b and of $a-b,b$ are the same, and in particular the greatest common divisor.
   d. *Pseudo-Code for binary_gcd(a,b)*
      i. *if (a is 0) return b*
      ii. *if (a is 1) return 1*
      iii. *if (b is 1) return 1*
      iv. *if (a is even & b is even) return 2*binary_gcd(a/2,b/2)*
      v. *if (a is odd & b is even) return gcd(a,b/2)*
      vi. *if (a is even & b is odd) return gcd(a/2,b)*
      vii. *if (a is odd & b is odd) return gcd($|a-b|$/2,b)*

The correctness of the algorithm is justified through articles *a-c*.
Running time: It is clear that at each recursive iteration, at least one of the coordinates becomes smaller by a factor of 2, therefore there can be no more then $\lceil \log(a) \rceil + \lceil \log(b) \rceil$ iterations, and since $a \geq b$ and each iteration takes *O(1)* time (since subtraction, parity check and halving require *O(1)* time), we get running time of *O(log(a))*.

5.

a. We'll run Ex-Eculid(21,10), then *y mod 21* will be $10^{-1} \bmod 21$.
   i. Ex-Euclid(21,10) calls …
   ii. Ex-Euclid(10, 21 mod 10) = Ex-Euclid(10,1) calls …
   iii. Ex-Euclid(1, 10 mod 1) = Ex-Euclid(1, 0). This is evauluated to be (1,1,0).

   Now we roll back the recursion:

   In the second stage, (x,y)=(0, $1-\left\lfloor \frac{10}{1} \right\rfloor * 0$)=(0,1). In the first stage, (x,y) = (1, $0-\left\lfloor \frac{21}{10} \right\rfloor * 1$)=(1,-2). And this is the result. We see that indeed $1=1*21-2*10$. We deduce that $10^{-1} \pmod{21}=-2 \pmod{21}=19 \pmod{21}$.

b. We will denote $Ex\text{-}Euclid(f_{k+1},f_k)=(d_k,x_k,y_k)$. We recall that the Fibonacci sequence is defined as – $f_0=1$ $f_1=1$, $f_{k+1}=f_k+f_{k-1}$ $(k>0)$

   We will prove by induction that $d_k=1$ for all $k>0$, and that $x_{k+1}=y_k=(-1)^{k+1}f_{k-1}$ for all $k>0$. The fact that $x_{k+1}=y_k$ is obvious from the pseudo-code of *Extended-Euclid.* We turn to the other parts of the claim: (For $k=0$, the answer is $d_0=1,x_0=0,y_0=1$)

   $k=1$: *Ex-Eculid(2,1)=(1,0,1),* and therefore it agrees with the formulas.

   $1,...,k \rightarrow k+1$: First, we notice that $f_k>f_{k-1}$ for all $k>1$. Now, by the induction formula for *f:* $f_{k+1}=f_k+f_{k-1}$, we see that $f_{k+1} \bmod f_k=f_{k-1}$. And so, we see that the recursion call in $Extended\text{-}Euclid(f_{k+1},f_k)$ is $Extended\text{-}Euclid(f_k, f_{k+1} \bmod f_k)=Extended\text{-}Euclid(f_k,f_{k-1})=(d_k,x_k,y_k)$. Now, by the induction hypothesis, we deduce that $(d_{k+1},x_{k+1},y_{k+1})=Extended\text{-}Euclid(f_{k+1},f_k)=$
   $(d_k,y_k,x_k-\left\lfloor \frac{f_{k+1}}{f_k} \right\rfloor * y_k)=(1,y_k,y_{k-1}-y_k)=(1,y_k,(-1)^k f_{k-2}-(-1)^{k+1}f_{k-1})=$
   $(1,y_k,(-1)^{k+2}(f_{k-2}+f_{k-1}))=(1,y_k,(-1)^{k+2}f_k)$. And the proof is complete.

6.

a. Lemma 1: $lcm(a_1,a_2,\ldots,a_n)=lcm(lcm(a_1,a_2),a_3\ldots,a_n)$ for every positive integers $a_1,a_2,\ldots,a_n$.

Proof: if m is a multiplier of a_1,a_2, then it is a multiplier of $l= lcm(a_1,a_2)$. This is true because we can divide m by l with remainder r so $m=x*l+r$ where $r<l$. And now, $a_1|l,m$ and therefore $a_1|r$. The same holds for a_2. And so we see that r is a common multiplier of a_1,a_2, but l is the least common multiplier of them, and so we arrive at a contradiction. We conclude that if m is a multiplier of $a_1,a_2,\ldots,a_n$, it is a multiplier of $lcm(a_1,a_2),a_3\ldots,a_n$. The other direction is obvious. And so, every multiplier of $a_1,a_2,\ldots,a_n$ is a multiplier of $lcm(a_1,a_2),a_3\ldots,a_n$ and vice versa, and so in particular their least common multiplier is the same. That completes the proof.

Lemma 2: For every positive integers a,b there holds: $lcm(a,b)=a*b/gcd(a,b)$

Proof: By the fundamental theorem of algebra, a and b have the following representation $a=p_1^{l1}p_2^{l2}\ldots p_n^{ln}$, $b = p_1^{r1}p_2^{2}\ldots p_n^{rn}$ (we may assume that the prime factors are the same by allowing l_i to receive also the value 0). The $gcd(a,b)=p_1^{min(l1,r1)}\ldots p_n^{min(ln,rn)}$. And also $lcm(a,b)= p_1^{max(l1,r1)}\ldots p_n^{max(ln,rn)}$. And so we conclude that $lcm(a,b)*gcd(a,b)= p_1^{max(l1,r1)}\ldots p_n^{max(ln,rn)} * p_1^{min(l1,r1)}\ldots p_n^{min(ln,rn)}= p_1^{(l1+r1)}\ldots p_n^{(ln+rn)}=a*b$. As required.

We arrive at the following algorithm:

Euclid-LCM($a_1,\ldots,a_n$)

1. *if (n=2) return a_1*a_2/Euclid(a_1,a_2)*
2. *return Euclid-LCM($lcm(a_1,a_2),a_3,\ldots,a_n$)*

The presented algorithm calls for the *Euclid* algorithm $n-1$ times.

b. Pseudo-Code:

Relatively-Prime($a_1,\ldots,a_n$)

return (*Relatively-Prime*($a_1,\ldots, a_{ceil(n/2)}$) AND ($a_{ceil(n/2)+1},\ldots,a_n$) AND $gcd(a_1*a_2*\ldots*a_{ceil(n/2)},a_{ceil(n/2)+1}*\ldots a_n)=1$)

Remark: We will assume that $n=2^k$ for simplicity for the analysis,

In order to prove the correctness of the algorithm, we will prove that $a_1,a_2,\ldots,a_n$ are pairwise relatively prime $\leftrightarrow$ $a_1,a_2,\ldots, a_{ceil(n/2)}$ are pairwise relatively prime, $a_{ceil(n/2)+1},\ldots,a_n$ are pairwise relatively prime and $a_1*a_2*\ldots*a_{ciel(n/2)},a_{ceil(n/2)}*\ldots a_n$ are relatively prime. The $\rightarrow$ side of the claim is obvious. For the other direction, we have to show that a_i and a_j are relatively prime when $i>ceil(n/2)$ and $j \leq ceil(n/2)$. If we

assume in contradiction that they both have a common divisor $d>1$ then d is also a common divisor of $a_1*a_2*\ldots*a_{ceil(n/2)}$, $a_{ceil(n/2+1)}*\ldots a_n$ and therefore we arrive at a contradiction to our assumption that $a_1*a_2*\ldots*a_{ceil(n/2)}$, $a_{ceil(n/2)}*\ldots a_n$ are relatively prime.

Running Time: We denote the running time by $T(n)$. Recalling that calculating $gcd(a,b)$ for $a,b<C$ requires $O(log(C))$ time, and that $a_i \leq M$ for every i, and so $a_1*a_2*\ldots*a_{ceil(n/2)},a_{ceil(n/2)}*\ldots a_n \leq M^{ceil(n/2)}$, we arrive at the following recursion formula:
$T(n)=2T(n/2)+O(log(M^{n/2}))=2T(n/2)+O(n*log(M))$
By solving the recursion we see that, $T(n)=O(n*log(n)*log(M))$. As required.

Algorithms – Exercise 8

Due Wednesday 20/12/06 24:00

1. Compute the prefix function π defined in the string matching algorithm for the given pattern: [a,b,a,c,a,b,c,a,a,b]

2. Explain how to determine the occurrences of pattern *P* in the text *T* by examining the π function for the string *P^T* (the concatenation of *P* and *T*).

3.
   a. Consider the string matching algorithm with a pattern *P* and a text *T*. What is the maximum number of times that any character in the text *T* will be looked at by the algorithm?
   b. Is it possible that adjacent characters in the text *T* will be considered this many times?

4.
   a. Find all the integers *a* such that:
      i. $a \bmod 3 = 2$
      ii. $a \bmod 11 = 9$
      iii. $a \bmod 20 = 5$
   b. Show that if n_1,n_2 are not relatively prime, then there exists a_1,a_2 such that the following set of equation does not have a solution *x:*
      i. $x \bmod n_1 = a_1$
      ii. $x \bmod n_2 = a_2$

5. You are given a number *N*. Your task is divided into two stages:
   a. Phase I: you are given arbitrarily large space and time to run any calculations you wish on *N*. You may store the results of your calculations.
   b. Phase II: the memory you used in phase I now turns into read-only memory. You are now given only $O(\log(\log(N)))$ bits of disk space, and a read-only array *A* composed of {0,1} of length $O(N)$.

   Give an algorithm that runs according to these two stages, and outputs "true" if the number of *1*'s in *A* is *N*, and "false" otherwise.

   Hints:

1. Use phase I to store $a_i=(N \bmod p_i)$, when p_i is the i-th prime number, for $i=1,..,k$ such that k is large enough to guarantee that the a_i determine N uniquely. (Show that $k=O(\log(N))$ is sufficient)
2. You may assume that $p_i=O(i \cdot \log(i))$, show how to determine if the number of *1*'s in A is N using only a space of $O(\log(p_k))$ bits.

Algorithms - Exercise 8 Solution

1. Compute the prefix function Π defined in the string matching algorithm for the given pattern: [a,b,a,c,a,b,c,a,a,b]

Answer:

i	1	2	3	4	5	6	7	8	9	10
P[i]	**a**	**b**	**a**	**c**	**a**	**b**	**c**	**a**	**a**	**b**
Π[i]	**0**	**0**	**1**	**0**	**1**	**2**	**0**	**1**	**1**	**2**

2. Explain how to determine the occurrences of pattern *P* in the text *T* by examining the Π function for the string *P^T* (the concatenation of *P* and *T*).

Answer:

Denote: $|P| = m$
$|T| = n$

For the non-trivial case where $n \geq m$ go over all $2m \leq i \leq n+m$ if:

- $\Pi(i) < m$: continue.
- $\Pi(i) = m$: mark T(i-2m+1) as the beginning of an occurrence of P in T.
- $\Pi(i) > m$: if there exists a j such that $\Pi^j(i) = m$ Mark T(i-2m+1) as the beginning of an occurrence of P in T.

Using the definition of $\Pi^*(i)$ (Cormen p927) an equivalent answer is:

For all $2m \leq i \leq n+m$ such that $m \in \Pi^*(i)$ mark T(i-2m+1) as the beginning of an occurrence of P in T.

Note: the index into T is "i-2m+1" as i is an index into P^T marking the and of an instance of P. This means we need to subtract m to mark the beginning and another m as we are looking for an index into T not P^T.

3. a. Consider the string matching algorithm with a pattern *P* and a text *T*. What is the maximum number of times that any character in the text *T* will be looked at by the algorithm?

Answer:

Claim: Any character in T can be considered at most |P| times.
Proof: consider the loop over the characters in T. Each such character can either be looked at once, or several times within the inner loop. The inner loop's stops once k == 0. Within the loop k is given the value of $\Pi(k)$ which by definition is smaller then k and as k is smaller or equal to |P|, at most |P| steps will take till k will be equal to zero and the inner loop will stop.

Example of |P| looks at each character in T:

Consider P = aaa
T = aab
The third character of T, 'b', will be looked at |P| times.

b. Is it possible that adjacent characters in the text *T* will be considered this many times?

Answer:
If |P| > 1, No.
The only why a character could be considered |P| times is if it was considered in the inner loop with the iterations ending when k == 0. The equality check outside the inner loop can enlarge k by at most 1, meaning the next iteration of the outer loop (adjacent character) will perform at most 1 inner loop iteration.

4. a. Find all the integers *a* such that:
i. $a \bmod 3 = 2$
ii. $a \bmod 11 = 9$
iii. $a \bmod 20 = 5$

Answer:
Let's start with a few definitions:
$n_1 = 3, n_2 = 11, n_3 = 20$
$n = n_1 \times n_2 \times n_3$
$a_i = a \bmod n_i, 1 \leq i \leq 3$
$m_i = n / n_i$
$c_i = m_i(m_i^{-1} \bmod n_i)$

Note that $\gcd(n_1, n_2) = \gcd(n_3, n_2) = \gcd(n_1, n_3) = 1$. Hence: n_1, n_2, n_3 are pairwise relatively prime.
As you have seen in the proof of the Chinese reminder theorem since m_i and n_i are pairwise relatively prime c_i is well defined and $(m_i^{-1} \bmod n_i)$ exists (see theorem 31.6 and corollary 31.26 in Cormen).

By the Chinese theorem we get that a is defined uniquely:

$a = (a_1c_1 + a_2c_2 + a_3c_3) \bmod n$

Now it is left to calculate the values:
$a_1 = 2$, $a_2 = 9$, $a_3 = 5$ (given in the question)
$n = 660$
$m_1 = 220, m_2 = 60, m_3 = 33$
$m_1^{-1} = 1$, $m_2^{-1} = 9$, $m_3^{-1} = 17$
$c_1 = 220$, $c_2 = 540$, $c_3 = 561$

=> $a = (2*220+9*540+5*561) \bmod 660 = 185$ => $a = 185 + k$, $k \epsilon Z$

b. Show that if n_1, n_2 are not relatively prime, then there exists a_1, a_2 such that the following set of equation does not have a solution x:
i. $x \bmod n_1 = a_1$
ii. $x \bmod n_2 = a_2$

Answer:

By Contradiction, lets assume that for all a_1, a_2 there exists a solution x.
i=> $x = a_1 + kn_1$
ii=> $x = a_2 + ln_2$
=> $a_1 + kn_1 = a_2 + ln_2$ => $kn_1 - ln_2 = a_2 - a_1$ (**)

From n_1 and n_2 being non relatively prime we get that $\gcd(n_1, n_2) = s > 1$.

Thus $s | (kn_1 - ln_2)$ from (**) we get $s | (a_2 - a_1)$.
Let us choose a1=s-1 and a2 = s we get s|(s-s+1)=>s|1 which contradicts s>1□

5. You are given a number *N*. Your task is divided into two stages:

a. Phase I: you are given arbitrarily large space and time to run any calculations you wish on *N*. You may store the results of your calculations.

b. Phase II: the memory you used in phase I now turns into read-only memory. You are now given only $O(\log(\log(N)))$ bits of disk space, and a read-only array *A* composed of {0,1} of length $O(N)$.

Give an algorithm that runs according to these two stages, and outputs "true" if the number of *1*'s in *A* is *N*, and "false" otherwise.

Hints:
1. Use phase I to store $a_i = (N \bmod p_i)$, when p_i is the *i*-th prime number, for $i=1,..,k$ such that *k* is large enough to guarantee that the a_i determine *N* uniquely. (Show that $k=O(\log(N))$ is sufficient)
2. You may assume that $p_i = O(i \log(i))$, show how to determine if the number of *1*'s in *A* is *N* using only a space of $O(\log(p_k))$ bits.

Answer:

Phase I:
Given N we will compute $a_i=(N \bmod p_i)$ for i=1,..,k.
Which k should we choose, such that k is large enough to guarantee that the a_i determine N uniquely?
According to the Chinese theorem $N \bmod(p_1p_2\dots p_k)$ is a unique solution for the equations defined by a_i, up to addition subtraction of $p_1p_2\dots p_k$. Thus, we have to choose k such that $p_1p_2\dots p_k>L$, and thus we will get a unique solution in the range [0,L-1].

It is obvious that for each i, $p_i \geq 2$. Thus, $L < 2^{ceil(\log(L))} \leq \Pi_{i=1}^{ceil(\log(L))} p_i$.

From the above claim we get that by choosing $k=O(\log(L))=O(\log(N))$, $L < p_1p_2\dots p_k$,we get that the a_i's determine N uniquely. We store k, $p_1,\dots,p_k$ and the a_i's.

Phase II:
Algorithm:

1. r := 0
2. for i=1,…,k
    a. Iterate over A sequentially
        i. If at the current position you see a '1' in A, update: $r = r + 1 \pmod{p_k}$
    b. if ($r != a_i$)
        i. return false
3. return true

As proved in phase I, a_i determine N uniquely, thus if S is a solution of the equations defined by a_i, we can conclude that S=N. Thus the algorithm will return true, iff the number of 1's, S, is N as required.

Now we will see that the algorithm uses only *O(log(log(N)))* extra space in the second phase:
Since the only additional memory we are using is r,i (we assume that we can iterate over A by request, without referring to the pointer sizes), we get:

1. For i we need O(log(k)) bits. Since k=O(log(N)), we get O(log(log(N)) space for i.
2. r will hold a maximal value of p_k, therefore we need $O(\log(p_k))$ bits for r. but since (by the hint) we know that $p_k=O(k*\log(k))$, we get that $O(\log(p_k))=O(\log(\log(N))$.

In total we used O(log(log(N))) additional space in the second phase.

Algorithms - Exercise 9

Due Wednesday 27/12 24:00

1. It is possible to strengthen Euler's theorem slightly to the form:

$$a^{\lambda(n)} \equiv 1(\text{mod } n) \text{ for all } a \in Z_n^*$$

where $n = p_1^{k_1} \cdots p_r^{k_r}$ and $\lambda(n)$ is defined by $\lambda(n) = lcm(\phi(p_1^{k_1}), \ldots, \phi(p_r^{k_r}))$. Prove:

   (a) $\lambda(n)|\phi(n)$

   (b) If $\lambda(n)|n-1$ then $a^{n-1} \equiv 1(\text{mod } n)$ for all $a \in Z_n^*$. Such an n is called a *Carmichael* number

   (c) If n is Carmichael then p^2 does *not* divide n for every p

   (d) If n is Carmichael then n is a product of at least 3 primes

2. Maximal independent set problem. Given an undirected graph $G = (V, E)$, where each vertex has d neighbors we want to find a large *independent* subset $S \subset V$. S is independent if for every $v, y \in S$ we have $(u, v) \notin E$. Consider the following decentralized algorithm for the problem. Each vertex v_i independently picks a random value x_i, where $x_i = 1$ with probability p and $x_i = 0$ with probability $1-p$. A vertex is in S iff it picked 1 and all its neighbors picked 0.

   (a) Prove that the resulting S is an independent set

   (b) Define y_i to be 1 if v_i is in S and 0 otherewise. Now $|S| = \sum y_i$. Find the expected size of S as a function of $n = |V|$,d and p.

   (c) Find p that maximizes the expected size of S. Give a formula for the expected size of S when p is set to this value.

3. Suppose you are trying to sell your house. Every day a different person comes, and offers you a price. You can either agree to his price or turn him away. You know that there are n people with distinct bids $b_1, \ldots, b_n$, but they come in random order. Give a strategy under which you sell the house for the highest price possible with probability at least $\frac{1}{4}$. For example, if the strategy is accepting the first buyer, the probability is $\frac{1}{n}$. You may assume that n is even.

4. Consider the following analogue of Karger's algorithm for finding minimum $s-t$ cuts. We will contract edges iteratively: in each iteration, let s and t denote the possibly contracted edges that contain the original nodes s and t, respectively. To make sure that s and t do not get contracted, at each iteration we delete any edges connecting s and t and select a random edge to contract among the remaining edges. Give an example to show that the probability that this method finds a minimum $s-t$ cut can be exponentially small.

Solution to Ex 9 in Algorithms

1. (a) As we have seen in class, $\phi(n) = \prod_i \phi(p_i^{k_i})$, so $\lambda(n)|\phi(n)$.

   (b) If $\lambda(n)|n-1$ then $n-1 = s\lambda(n)$ for some integer s, and then $a^{n-1} = (a^{\lambda(n)})^s \equiv 1 \bmod n$.

   (c) Let $n = \prod_i p_i^{k_i}$ be a Carmichael number. If $p^2|n$ for some p, we know that for some i we have $k_i \geq 2$. Then $\phi(p_i^{k_i}) = (p_i - 1)p_i^{k_i - 1}$, so $p|\phi(p_i^{k_i})$. In this case, $p|\lambda(n)$. But p does not divide $n-1$, a contradiction.

   (d) Assume that n is Carmichael and $n = pq$ for primes p, q with $p > q$. Then $lcm(p-1, q-1)|n-1$, in particular $p-1|n-1$. But $(p-1)q = pq - p < n-1$ and $(p-1)(q+1) = pq + p - q - 1 > n-1$. Contradiction.

2. (a) If $v \in S$, then x_i is zero for all the neighbors of v, thus none of these neighbors are in S. So if v is a neighbor of u, at lease one of them is not in S, so S is independent.

   (b)

   $$y_i = x_i \prod_{j \text{ neighbor of } i} (1 - x_j)$$

   So $Pr(y_i = 1) = p(1-p)^d$, and then $E(y_i) = p(1-p)^d$. As expectation is linear, we get $E(|S|) = E(\sum_i y_i) = np(1-p)^d$.

   (c) We compute the derivative of $E(|S|)$ with respect to p and look for its zeros.

   $$\frac{\partial E(|S|)}{\partial p} = n(1-p)^d + npd(1-p)^{d-1} = 0$$

   $$1 - p = pd \quad \Rightarrow \quad p = \frac{1}{d+1}$$

   So if we pick the optimal p, we get $E(|S|) = n\frac{d^d}{(d+1)^{d+1}}$

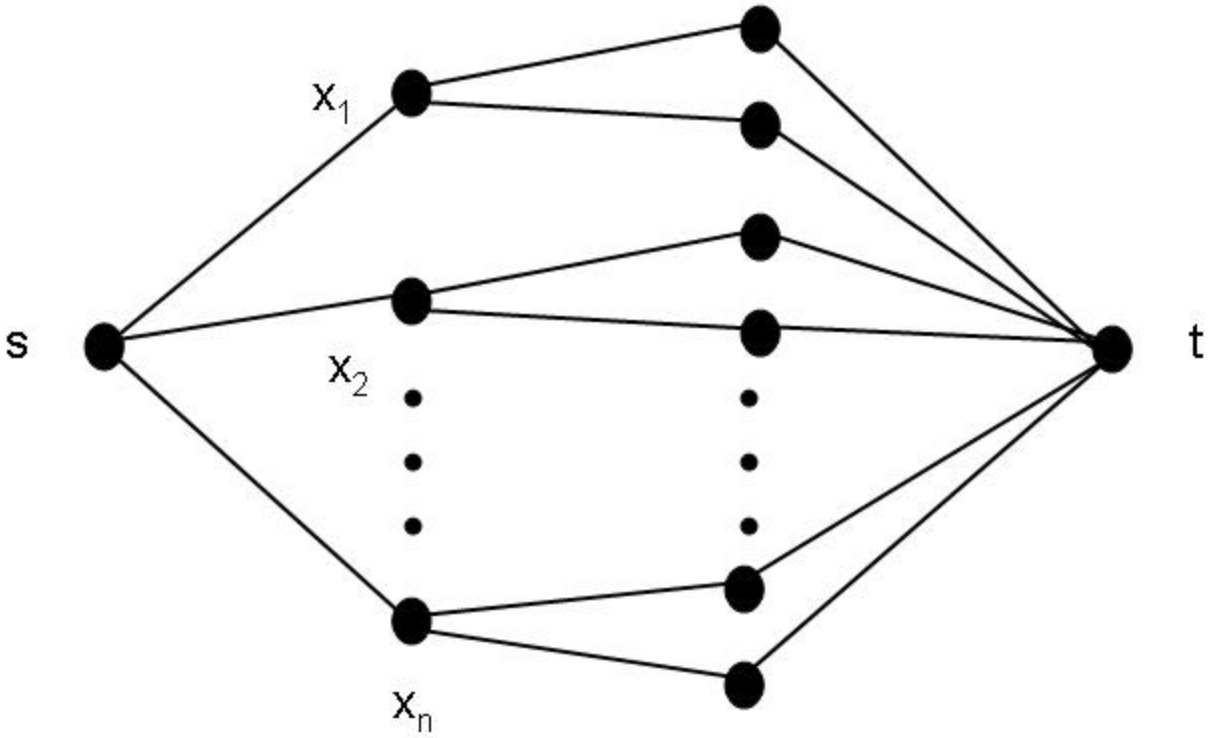
3. Algorithm: let $n/2$ buyers to come and go, and compute the maximum of their offers z. Now wait till somebody offers more than z, and except him (otherwise, except the last).

   Proof: let M be the maximal offer and m the second best one. With probability $\frac{1}{4}$ m will be among the first $n/2$ and M among the last $n/2$. In this case the algorithm will pick M, which is the right answer.

4. Let $G = (V, E)$ be the graph in the figure. It is easy to see that the minimal $s-t$ cut in G is $\{s\}, V \setminus \{s\}$. This means that in order to get the optimal cut, the algorithm cannon contract the edges between s and x_i for any i. We begin with $5n$ edges, and we must contract $4n$ out

of them. Every time the probability that we will contract a 'good' edge is at most 4/5. Thus the success probability p of the algorithm is

$$p \leq (\frac{4}{5})^{4n} = (\frac{4^4}{5^4})^n \leq 2^{-n}$$


s
x_1
x_2
x_n
t

Algorithms – Exercise 10

Due Wednesday 3/1/07 24:00

1. Let $n=pq$, where p,q are prime numbers. Show how you can find p,q given $n, \phi(n)$ (Euler's phi function of n) in polynomial time (The solution should be polynomial in the input size- O($log(n)$)).

2. Prove that if x is a nontrivial square root of *1* modulo n (i.e. $x^2 = 1 mod\ n, x \neq 1, -1 mod\ n$), then $gcd(x-1,n)$ and $gcd(x+1,n)$ are both nontrivial divisors of n.

3.
   a. Suppose we have a sequence of items passing by one at a time. We want to maintain a sample of one item with the property that it is uniformly distributed over all the items we have seen at each step. Moreover, we want to accomplish this without knowing the total number of items in advance or storing all the items that we see. Consider the following algorithm which stores just one item in memory at all times. When the first item appears, it is stored in the memory. When the k-th item appears, it replaces the item with probability $1/k$. Show this algorithm solves the problem.
   b. Suppose we modify the algorithm above, so that when the k-th item appears, it replaces the item in memory with probability 1/2. What is the probability that we have stored the i-th element after seeing n elements?

4. Consider the n-dimensional cube routing algorithm described in the tirgul.
   a. Show that for every edge e of the graph, the expected number of routing paths passing through e is $O(1)$ (as a function of n). (Hint: the number of routing paths passing through e can be written as a sum of random variables that may take $\{0,1\}$ values, bound their expectation and use linearity of expectation)
   b. Show that for every packet, the expected time before reaching its destination node is bounded from above by an $O(n)$ bound.

Note: (Diffie-Hellman key exchange)

Alice and Bob may communicate through a given channel. However, the channel is not safe since the malicious Eve is eavesdropping. Alice and Bob would like to agree on a certain value, they would both know but Eve wouldn't.

Consider the following protocol:

1. Alice and Bob agree on a prime number p, and a generating element in Z_p^*, g.
2. Alice picks a random natural number a and sends $(g^a \bmod p)$ to Bob.
3. Bob picks a random natural number b and sends $(g^b \bmod p)$ to Alice.
4. Alice computes $(g^b)^a \bmod p$.
5. Bob computes $(g^a)^b \bmod p$.

Now Alice and Bob know the number g^{ab}, which Eve supposedly does not know. This was the first practical method for establishing a shared secret over an unprotected communications channel (introduced at 76').

For more info: http://en.wikipedia.org/wiki/Diffie-Hellman

Algorithms – Exercise 10 (Solution)

1. We notice that since $n=pq$ for p,q different primes, then by the formula $\phi(n)=n\prod_{\substack{p/n \\ prime}}(1-\frac{1}{p})$, we conclude $\phi(n)=(p-1)(q-1)$, and therefore $n-\phi(n)+1=p+q$, and we know that $q=n/p$ and hence $n-\phi(n)+1=p+n/p$, or $p^2-p(n-\phi(n)+1)+n=0$ and so we can solve for p by solving a quadratic equations (the two roots would be p and q). We notice that this algorithm is indeed polynomial in $log(n)$. This is since the solution of a quadratic equation requires arithmetical operations (clearly polynomial), and finding the square root of a given natural number. We show a simple way to find m given m^2: we perform a binary search over all numbers smaller then m^2:

   Simple_sqrt(n, high, low)

   $mid = \left\lfloor \frac{high+low}{2} \right\rfloor$

   if $(mid^2==n)$ *return mid;*

   if $(mid^2 > n)$ *return Simple_sqrt(n,mid-1,low);*

   if $(mid^2 < n)$ *return Simple_sqrt(n,high,mid+1);*

   We see this is simply a binary search, it will perform in $O(log(n))$, i.e. polynomial in the input size. Note that this algorithm assumes that n indeed has an integer square root.

2. First by definition we notice that $gcd(x-1,n)$ and $gcd(x+1,n)$ are both divisors of n. Let's assume without loss of generality $x<n$, we must prove that the gcd's are not 1. We can write $n= p_1^{l_1} \cdot p_2^{l_2} \cdot \ldots \cdot p_k^{lk}$, and since $x^2 \underset{n}{\equiv} 1$ then $x^2 \underset{p_i}{\equiv} 1$ for every i, and since p_i is prime, then necessarily $x \underset{p_i}{\equiv} 1$ or -1. By the Chinese remainder theorem, we know that since x is not $1,-1 \pmod n$ then there is an index i such that $x \underset{p_i}{\equiv} 1$ and j such that $x \underset{p_j}{\equiv} -1$ (otherwise, all the remainders $x \bmod p_i^{li}$ would have been 1 or -1, and hence $x \bmod n$ would have been 1 or -1 respectively). So we conclude that there exists i,j such that $x-1 \underset{p_i}{\equiv} 0$ and $x+1 \underset{p_j}{\equiv} 0$ and so $gcd(x-1,n) \geq p_i > 1$ and $gcd(x+1,n) \geq p_j > 1$.

3.

a. We need to prove that if we received n items in total, then each item has a probability of 1/n to be chosen. Consider the *i-th* item. It will be chosen iff it is taken in the *i-th* stage, and then not replaced in all stages *i+1,…,n*. We'll denote A_i the event that *i* is taken in the *i-th* step. We notice that $A_1,..,A_n$ are independent events. Therefore:

$$Pr(\ i\_is\_chosen\ ) = Pr(\ A_i \cap \overline{A_{i+1}} \cap \ldots \cap \overline{A_n}\ ) = (\frac{1}{i}) \cdot (1 - \frac{1}{i+1}) \ldots (1 - \frac{1}{n}) = \frac{1}{n}$$

b. We'll denote A_i to be the same as above. And compute for *i>1:*

$$Pr(\ i\_is\_chosen\ ) = Pr(\ A_i \cap \overline{A_{i+1}} \cap \ldots \cap \overline{A_n}\ ) = (\frac{1}{2}) \cdot (1 - \frac{1}{2}) \ldots (1 - \frac{1}{2}) = \frac{1}{2^{n-i+1}}$$

And the probability that the first item is chosen is:

$$Pr(1\_is\_chosen\ ) = Pr(\ A_1 \cap \overline{A_2} \cap \ldots \cap \overline{A_n}\ ) = 1 \cdot (1 - \frac{1}{2}) \ldots (1 - \frac{1}{2}) = \frac{1}{2^{n-1}}$$

4.

a. Let e be an edge in the n-dimensional cube, say between the vertices $(x_1,..,x_i,\ldots,x_n)$ and $(x_1,...,x_i',\ldots,x_n)$. We'll define random variables 1_v for every $v \in V$ as follows – 1_v equals 1 iff the routing path between v and $g(v)$ passes through the edge e in the direction $(x_1,..,x_i,\ldots,x_n) \rightarrow (x_1,...,x_i',\ldots,x_n)$.

   i. $E[\,1_v\,] = 0$ for every $v \in V$ such that $(v_i,\ldots,v_n) \neq (x_i,\ldots,x_n)$ (because the routing algorithm routes by changing v bit by bit to $g(v)$, therefore if the above condition does not hold, the path will simply not go through e).

   ii. We now consider the case where $(v_i,\ldots,v_n)=(x_i,\ldots,x_n)$ -

   $$E[\,1_v\,] = Pr(\,1_v = 1\,) = Pr(\ prefix_i(\ g(\ v\ )) = (\ x_1,...,x_i'\,)) = \frac{1}{2^i}$$

   where $prefix_i$ denotes the first i bits of $g(v)$. The probability is 2^{-i} since $g(v)$ is chosen uniformly over all the vertices of V, there are 2^n vertices, and 2^{n-i} that satisfy the above condition.

Let us denote in X, the number of routing paths going through e then $X=X_1+X_2+X_3+X_4$ where X_1 denotes the number of routing paths between v and $g(v)$ going through e in the direction $(x_1,..,x_i,\ldots,x_n) \rightarrow (x_1,...,x_i',\ldots,x_n)$, and X_2 denotes the number of routing paths between v and $g(v)$ going through e in the direction $(x_1,..,x_i',\ldots,x_n) \rightarrow (x_1,...,x_i,\ldots,x_n)$. X_3 is the number of routing paths going through e through $g(v)$ and $\pi(\ v\ )$ (v's final destination) in the direction $(x_1,..,x_i,\ldots,x_n) \rightarrow (x_1,...,x_i',\ldots,x_n)$, and X_4 is the number of paths in the opposite direction between those vertices. We notice that X_1,X_2,X_3,X_4 are identically distributed, and therefore $E[X]=E[X_1+X_2+X_3+X_4]=$

$E[X_1]+E[X_2]+E[X_3]+E[X_4]=4E[X_1]$. We also notice that

$$E[X_1] = E[\sum_{v\in V} 1_v] = \sum_{v\in V} E[1_v] = 0 + \left|\{v : (v_i,\ldots,v_n)=(x_i,..,x_n)\}\right| \cdot \frac{1}{2^i} =$$

$$= 2^{i-1} \cdot \frac{1}{2^i} = 0.5$$

Therefore $E[X]=2$.

b. We'll denote the expected time for a certain packet v before reaching its destination $\pi(v)$ by Y. Then we notice that the routing path between v and $\pi(v)$ consists of at most $2n$ edges (since it will take him at most n edges to get from v to $g(v)$ and then at most n to get from $g(v)$ to $\pi(v)$). Let Y_i denote the random variable that specifies the time it took the packet v to get from its $(i-1)$-th stop to its i-th stop (and $Y_i=0$ for every i longer then the total length of v's routing path).

Then, $Y = \sum_{i=1}^{n} Y_i$ and we notice that $Y_i \le X_e$ for some X_e – the routing paths passing through the edge e (for some edge e). But from a we get $E[X_e]=2$, and hence $E[Y_i] \le 2$. By taking expectation we get:

$$E[Y] = E[\sum_{i=1}^{2n} Y_i] = \sum_{i=1}^{2n} E[Y_i] \le 4n .$$

Remark: We can also get a $2n$ bound by noticing that the constraint is that only one path in the same direction would go through e in any given turn. Therefore, as we have seen in a the expected number of edges passing through e in any one direction is 1 and therefore obtain a $2n$ bound.

Algorithms - Exercise 11

Due Wednesday 10/1 24:00

1. Broadcast attack on Rabin cryptosystem. Alice and Mary both use the Rabin cryptosystem. Alice's private key is (p_1, q_1) with public key $n_1 = p_1 q_1$ and Mary's is (p_1, q_2) and $n_2 = p_2 q_2$. Assume that $n_1 < n_2$. Bob wants to send the same message $0 < M < n_1$ both to Alice and Mary, thus he is sending $C_1 = M^2 \mod n_1$ and $C_2 = M^2 \mod n_2$. Show that if the evil eavesdropper Eve gets hold of C_1 and C_2, she can recover the original message M (Eve knows n_1 and n_2 as they are public). Hint: use the chinese remainder theorem for $N = n_1 n_2$.

2. Secret sharing with cheating parties. You have seen in class a secret sharing scheme in which a secret (a number s) is shared between n people such that each $k+1$ of them can recover s together, and each k of them know nothing about s. In the scheme we assumed that when a group of people try to recover the secret, everyone in the group tells the truth. What happens if that is not true?

   (a) Show that if f and g are two polynomials above F_p of degree $\leq k$ with $f \neq g$ and $A \subset F_p$ with $|A| = 3k+1$ then $|a \in A \ s.t. \ f(a) \neq g(a)| \geq 2k+1$.

   (b) Show that even if k people are telling lies, any group of $3k+1$ people can recover the secret (the algorithm does not have to be efficient. just show that the secret is uniquely defined).

3. RSA

   (a) Prove that RSA is multiplicative: if C_1 is the encryption of M_1 and C_2 is the encryption of M_2, then $C_1 C_2$ is the encryption of $M_1 M_2$.

   (b) Let (n, e) be a public key of RSA. Show that if we have an efficient algorithm A which decrypts the 1 percent of the messages, we can build an efficient randomized algorithm that decrypts *every* message with high probability. Hint: if you need to decrypt C, pick a random M', compute its encryption C' and try decrypting CC'.

4. Comparing Linear Programs. A linear program is a sequence of computations, where at each step the result is a product or a sum of the results of two of the previous steps or the inputs. More formally: a linear program with m inputs $x_1 \dots x_m$ is a sequence $\{(o_i, l_i, r_i)\}_{i=1}^{n}$, with $o_i \in \{+, -, *\}$ and $-m \leq l_i, r_i \in \leq i$. Now o_i is the operator used at the $i-th$ step, and l_i, r_i determine which previous results we use. If $l_i < 0$ or $r_i < 0$ we use x_{-l_i} or x_{-r_i}.

   Example: $m = 2$ and $n = 3$. the program is $(+, -1, -1); (+, 1, -2); (*, 1, 2)$ The result after the first step is $x_1 + x_1 = 2x_1$. After the second step $2x_1 + x_2$. Third step: $2x_1 * (2x_1 + x_2) = 2x_1^2 + x + 1x + 2$.

(a) Write the result obtained by the following program with $m = 2, n = 4$:

$$(+, -1, -2); (*, -1, -2); (+, 1, 2); (*, 3, 3)$$

(b) Show that the result of the $n - th$ step is a polynomial f in m variables with maximal degree bounded by 2^n.

(c) Show that all the coefficients of f are bounded by 2^{2^n}

(d) Our goal is to compare two given programs. The programs are identical if the corresponding polynomials f_1 and f_2 are equal. We cannot compute explicitly f_1 and f_2, as they might be too long. Let $g = f_1 - f_2$ and suppose $g \neq 0$. Let p be a random prime number with $2^{2n} < p < 2^{3n}$. Bound from above the probability that $g \equiv 0 \; mod \; p$.

(e) Let $h \neq 0$ be a polynomial over F_p with degree$\leq 2^n$. Bound from above the probability that $h(x) = 0$ for a random $x \in F_p$

(f) Give an efficient randomized algorithm that decided with high probability whether two given programs are identical.

Soluiton to Exercise 11 in Algorithms

1. The equations $0 \leq x < n_1 n_2$, $x \bmod n_1 = C_1$ and $x \bmod n_2 = C_2$ have a unique solution x due to the Chinese remainder theorem. x can be efficiently computed. Now $x \equiv M^2 (\bmod\ n_1)$ and $x \equiv M^2 (\bmod\ n_1)$, with $0 \leq x, M^2 < n_1 n_2$ thus according to the uniqueness part of the Chinese remainder theorem $x = M^2$. All that is left is to find the square root of x. This can be done, for example, by binary search:

   Input: $x < 2^n$
   $s = 0$, $b = 2^n$
   if $((s+b)/2)^2 < x$ **then**
   $s = (s+b)/2$
   else
   $b = (s+b)/2$
   end if
   Output s

2. (a) Let $X \subset F_p$ be defined as the set of points on which f and g agree: $X = \{x \in F_p | g(x) = f(x)\}$. If $f \neq g$ with degrees bounded by k then $|X| \leq k$. If $|A| = 3k+1$ then $|A \setminus X| \geq 2k+1$, as requested.

   (b) Let f be the original secret, $A \subset F_p$ with $|A| = 3k+1$ and $h : A \to F_p$ the values reported by people. We claim that given h, f can be uniquely defined as a polynomial which values agree with those of h for at least $2k+1$ points in A. Proof: as at most k people cheat, f and h agree on at least $2k+1$ points in A. On the other hand, if $g \neq f$, then g and f agree on at most k points, thus g and h agree on at most $k+k = 2k$ points.

3. RSA

   (a) By definition: $C_1 C_2 = (M_1^e \bmod n)(M_2^e \bmod n)$. Thus $C_1 C_2 \bmod n = (M_1 M_2)^e \bmod n$.

   (b) As hinted: given C to decrypt, we pick a random $M' \in Z_n^*$, compute its encryption C' and try decrypting CC'. If the decoding succeeds and we get X, then by (a) follows that $MM' = X$, thus $M = XM'^{-1} \bmod n$. Encryption is 1:1 and onto, so if we pick M' uniformly in Z_n^*, then C' is uniform in Z_n^* and thus CC' is uniform in Z_n^*. So every time we have probability of 0.01 to succeed, and the trials are independent. If we try k times, the probability of success is $1 - (1 - 0.01)^k$, and can be made arbitrarily close to 1.

4. Comparing Linear Programs.

   (a) $((x_1 + x_2) + x_1 x_2)^2$

(b) By induction: the polynomials obtained up to step n are of degree $\leq 2^{n-1}$, thus addition or multiplication of two of those results in a polynomial of degree $\leq 2^n$.

(c) We shall prove that the coefficients are bounded by 2^{3^n} (3 instead of 2!!). By induction, coefficients up to step n are smaller than $2^{3^{n-1}}$. As seen before, the degree of f is bounded by 2^n. There are at most 2^{2^n} different ways (much less, actually) to obtain a monomial of degree 2^n as a product of two monomials. Thus the coefficients are at most $2^{2^n}2^{3^{n-1}}2^{3^{n-1}} \leq 2^{3^n}$.

(d) Take any non-zero monomial of g with coefficient a. As $|a| < 2^{3^n}$ the number of primes dividing a is at most 3^n. On the other hand, the number of primes between 2^{2n} and 2^{3n} is $\Omega(\frac{2^{3n}}{3n})$, thus the probability that $g \equiv 0 \ mod \ p$ is bounded by $\frac{3^n 3n}{2^{3n}} \leq 2^{-n}$.

(e) By the Schwartz-Zippel lemma if $h \neq 0$ is a polynomial in m variables over F_p with degree $\leq 2^n$ then the probability of $h(x_1, \ldots, x_m) = 0$ when x_i are uniformly distributed in F_p is bounded by $\frac{2^n}{p}$. With our choice of p, $\frac{2^n}{p} < 2^{-n}$

(f) Combining the pieces: We draw a random prime p with $2^{2n} < p < 2^{3n}$, and draw random $x_1 ... x_m \in F_p$. We now run the linear programs on $x_1 ... x_n$, treating addition and multiplication as operations modulo p. If two results are equal, the algorithm claims that the programs are equal, otherwise it claims that they are different.

If the programs are equal, the algorithm will always output the right answer. What is the probability that the programs are different and the algorithm will fail? By the bound above, this probability is less than $2^{-n} + 2^{-n} = 2^{-n+1}$. The running time is polinomial in n and m, as all the calculations are done mod p, and the number of digits on p is at most 3^n.

Algorithms – Exercise 12

Due Wednesday 17/1/06 24:00

1. Remainder: given an undirected graph $G=(V,E)$ a matching is a set $M \subseteq E$ such that no vertex in V is adjacent to more then one edge in M. A maximal matching is a matching of the largest cardinality between all matchings in G.
   a. Give an efficient algorithm that given a graph G, finds a matching M in G such that if $M \subseteq M'$ where M' is a matching, then $M=M'$ (i.e. M is not a proper subset of any other matching). What is the running time of your algorithm?
   b. Show that the algorithm you've given in a. is a 2-approximation to the problem of finding the maximal matching in G.
   c. Show that the bound from b. is tight for every $n = |V|$, such that n is divisible by 4. In other words, for every k, give a graph G on $4k$ vertices, for which the algorithm you've given in a. may return a matching of exactly half the size of the maximal matching.

2. Run the Approx-Subset-Sum algorithm we have seen in the tirgul on the following input. Write down the L_i after each iteration of the *for* loop. $S=\{13,15,16,17,22,23,24,27,29,50\}$, $t=125$, $\varepsilon = 0.4$

3. (reshut, but recommended) Consider the following approximation algorithm for the *0-1* knapsack problem (see ex3, q2). Sort the objects by decreasing ratio of profit to size. Let the sorted order of objects be $a_1,a_2,\ldots,a_n$. Find the lowest k such that the size of the first k objects exceeds N. Now, pick one the two: either $\{a_1,\ldots,a_{k-1}\}$ or $\{a_k\}$ (pick the more profitable one of course). Show that this is a 2-approximation for the problem. (we assume all objects are of size not greater then N)

4. Weighted Set Cover. Give a $ln(|U|)$ approximation to the following problem: Given a universe U of n elements, a collection of subsets of U, $A=\{S_1,\ldots,S_k\}$, and a cost function $c: A \to R^+$, find a minimum cost sub-collection $B \subseteq A$ that covers all elements of U.
   Remarks:
   1. You may assume $U = \bigcup_{S \in A} S$.
   2. The cost of $B \subseteq A$ is defined as $\sum_{S \in B} c(S)$.

Hint: try to modify the algorithm you've seen in class for the non-weighted set cover.

Algorithms – Exercise 12 (Solution)

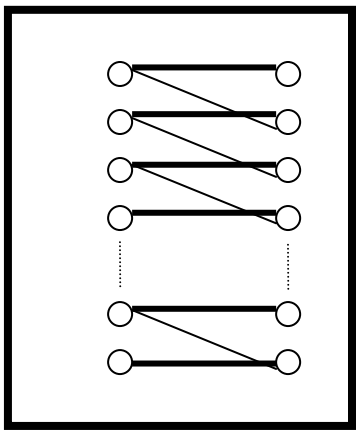
1.

a. Pseudo-Code:

Find-Max-Matching(G)
//G is a graph, its Vertex set is V, and edge set is E
1. Initialize A, an array of size V with the value "Not Covered"
2. Initialize M to be an empty list
3. For $e = \{u,v\} \in E$
    If (A[u]="Not Covered" and A[v]="Not covered")
        Append e to the end of M
        A[u] $\leftarrow$ *"Covered"*
        A[v] $\leftarrow$ *"Covered"*
4. return M

We claim that *M* is not a proper subset of any other matching in *G*. Otherwise, let $e=\{u,v\} \in E$ be such that $M \cup \{e\}$ is a matching. Then necessarily *u,v* are not covered by *M*. Therefore, when the *for* loop in 3 considered *e*, it would have been added to *M*. *M* is clearly a matching since if $\{u,v\},\{u,w\} \in M$ then let us assume (without loss of generality) that *{u,v}* was considered after *{u,w}*, then we arriving at the *if* in 3, we would have got *A[u]="Covered"*, and *{u,v}* would not have been added to *M*.
The running time of the algorithm is *O(V+E)*.

b. Let *M* be the output of the algorithm in *a*, and let *M'* be a maximal matching in *G*. We notice that if $|M| = k$, then the number of vertices covered by *M* is *2k* (follows directly from the definition of matching). We also notice that for every edge $\{u,v\} \in M'$, we get that either *u* is covered by *M* or *v* is (otherwise $M \cup \{u,v\}$ would be a strict superset of *M*, which is a matching in *G* in contradiction). We therefore conclude that $|M'| \leq 2k \leq 2|M|$. And therefore the algorithm in *a* is a 2-approximation.
c. For every integer *k*, we will show a graph of *4k* edges in which there is a matching that is not a subset of any other matching of size *k*, but the maximal matching is of size *2k*:



The algorithm may return the matching indicated by the thin line, while the maximal matching is that returned by the thin line.

2. The lists themselves appear in a separate file. It is important to notice how long the lists tend to get even for a relatively small input. This indicates that although the algorithm is polynomial, it may be in many cases too slow to use in practice.

3. A solution to the 0-1 knapsack problem is a sequence of $(s_1,s_2,\ldots,s_n)\in\{0,1\}^n$ that indicates for each item i whether it was taken or not. We recall the fractional knapsack problem which allowed s_i's to take values in [0,1] (still maximizing the quantity $q(s_1,\ldots,s_n)=\sum_{i=1}^{n} s_i x_i$ under the constraint $\sum_{i=1}^{n} s_i w_i \le N$). We saw that the solution to this problem was (assuming the objects are sorted in a decreasing order of profit to size, and k is defined as in the specification of the question):
$s_1^f=1,\ldots,\ s_{k-1}^f=1,\ s_k^f=N-\frac{N-\sum_{i=1}^{k-1} w_i}{w_k},\ s_{k+1}^f=0,\ldots,\ s_n^f=0.$ Since all allowed solutions of the *0-1* knapsack are allowed solutions for the fractional knapsack problem, we conclude that $q(s_1^{OPT},\ldots,s_n^{OPT})\le q(s_1^f,\ldots,s_n^f)$ (where $s_1^{OPT},\ldots,s_n^{OPT}$ denote the optimal solution for the 0-1 knapsack). Therefore:

$$q(s_1^{OPT},\ldots,s_n^{OPT})\le q(s_1^f,\ldots,s_n^f)=\sum_{i=1}^{n} s_i^f x_i=\sum_{i=1}^{k-1} x_i+s_k^f x_k\le\sum_{i=1}^{k-1} x_i+x_k$$

We therefore conclude that either $0.5\cdot q(s_1^{OPT},\ldots,s_n^{OPT})\le\sum_{i=1}^{k-1} x_i$, or $0.5\cdot q(s_1^{OPT},\ldots,s_n^{OPT})\le x_k$. Since our algorithm picks the higher of these two

options we see that the algorithm returns a solution which is at least half the quality of the optimal, and therefore is a 2-approximation to the problem.

4. Pseudo Code:

1. $C \leftarrow \varnothing$
2. $X \leftarrow U$
3. While $X \neq \varnothing$

Calculate f(S)= $\frac{c(S)}{|X \cap S|}$ *for every set S not in C*

$S_0 \leftarrow$ *The set not in C with the lowest f(S)*

$C \leftarrow C \cup S_0$

$X \leftarrow X - S_0$

4. return C

First of all it is obvious that *C* is a sub-collection of *A* that covers *U*. We'll denote $|U| = n$. We will denote the set *X* after the *j-th* iteration as X_j. We will assume without loss of generality that $S_1,\ldots,S_n$ is the order in which the sets in *A* were chosen by the algorithm. Observe that $|X_{j+1}| = |X_j| - |X_j \cap S_j|$, and also that $q(OPT) \geq f(S_j)|X_j| = \frac{c(S_j)|X_j|}{|S_j \cap X_j|}$ for every *j* (we will prove that later). We conclude therefore that $|S_j \cap X_j| \geq \frac{c(S_j)|X_j|}{f(S_j)}$, and hence for every iteration *j* we get-

$$|X_{j+1}| = |X_j| - |X_j \cap S_j| \leq |X_j|(1 - \frac{c(S_j)}{q(OPT)}) \leq \ldots$$

$$\leq |X_0|(1 - \frac{c(S_1)}{q(OPT)})\ldots(1 - \frac{c(S_j)}{q(OPT)}) \leq n(1 - \frac{c(S_1)}{q(OPT)})\ldots(1 - \frac{c(S_j)}{q(OPT)}) \leq$$

Now by the averages inequality:

$$\leq n\left(1 - \frac{\sum_{k=1}^{j} c(S_k)}{j \cdot q(OPT)}\right)^j < n \cdot exp\left(-\frac{\sum_{k=1}^{j} c(S_k)}{q(OPT)}\right)$$

. We conclude that once $\sum_{k=1}^{j} c(S_k) = ln(n) \cdot q(OPT)$, then $|X_{j+1}| < 1$, and therefore $X_{j+1} = \varnothing$, and the algorithm will exit. And therefore, the solution we obtain is a *ln(n)*-approximation.

We are left to prove that $c(OPT) \geq f(S_j)|X_j| = \frac{c(S_j)|X_j|}{|S_j \cap X_j|}$. Let us denote the price of each element $u \in X_j$ by $price(u) = min_{u \in S \notin C} f(S)$, that is the minimal price we may pay in order to cover *u* (the *price* may differ from iteration to

iteration of course). We notice that in order to cover all the remaining elements, i.e. all elements in X_j, we will have to pay at least $\sum_{u \in X_j} price(u)$. So we get $q(OPT) \leq \sum_{u \in X_j} price(u) \leq \sum_{u \in X_j} f(S_j) = |X_j| f(S_j)$, where the last inequality is justified by the choice of S_j as the element that minimizes $f(S)$.

```
L =

     0


L =

     0    13


L =

     0    13    15    28


L =

     0    13    15    16    28    29    31    44


L =

  Columns 1 through 10

     0    13    15    16    17    28    29    30    31    32

  Columns 11 through 16

    33    44    45    46    48    61


L =

  Columns 1 through 10

     0    13    15    16    17    22    28    29    30    31

  Columns 11 through 20

    32    33    35    37    38    39    44    45    46    48

  Columns 21 through 28
```

```
   50    52    54    61    66    68    70    83
```

L =

```
  Columns 1 through 10

    0    13    15    16    17    22    23    28    29    30

  Columns 11 through 20

   31    32    33    35    36    37    38    39    40    44

  Columns 21 through 30

   45    46    48    50    52    54    56    58    60    62

  Columns 31 through 40

   66    68    70    73    75    77    83    89    91    93

  Column 41

  106
```

L =

```
  Columns 1 through 10

    0    13    15    16    17    22    23    24    28    29

  Columns 11 through 20

   30    31    32    33    35    36    37    38    39    40

  Columns 21 through 30

   41    44    45    46    47    48    50    52    54    56

  Columns 31 through 40
```

```
   58    60    62    64    66    68    70    72    74    76

  Columns 41 through 50

   78    80    82    84    86    89    91    93    97    99

  Columns 51 through 54

  101   106   113   117


L =

  Columns 1 through 10

    0    13    15    16    17    22    23    24    27    28

  Columns 11 through 20

   29    30    31    32    33    35    36    37    38    39

  Columns 21 through 30

   40    41    42    43    44    45    46    47    48    49

  Columns 31 through 40

   50    52    54    56    58    60    62    64    66    68

  Columns 41 through 50

   70    72    74    76    78    80    82    84    86    89

  Columns 51 through 60

   91    93    95    97    99   101   105   109   113   116

  Columns 61 through 62

  120   124


L =
```

Columns 1 through 10

0 13 15 16 17 22 23 24 27 28

Columns 11 through 20

29 30 31 32 33 35 36 37 38 39

Columns 21 through 30

40 41 42 43 44 45 46 47 48 49

Columns 31 through 40

50 52 54 56 58 60 62 64 66 68

Columns 41 through 50

70 72 74 76 78 80 82 84 86 89

Columns 51 through 60

91 93 95 97 99 101 105 109 113 116

Columns 61 through 62

120 124

L =

Columns 1 through 10

0 13 15 16 17 22 23 24 27 28

Columns 11 through 20

29 30 31 32 33 35 36 37 38 39

Columns 21 through 30

40 41 42 43 44 45 46 47 48 49

Columns 31 through 40

50 52 54 56 58 60 62 64 66 68

Columns 41 through 50

70 72 74 76 78 80 82 84 86 88

Columns 51 through 60

90 92 94 96 98 100 104 108 112 116

Columns 61 through 62

120 124

Algorithms – Exercise 13

Due Wednesday 24/1/06 24:00

Definition: We say that a probabilistic algorithm is a C-approximation to a maximization problem if for every instance of the problem it returns an output O (which is a random variable) which satisfies $q(OPT) \leq C \cdot E[q(O)]$.
An example to such an algorithm is the 2-approximation to the problem of finding an assignment that satisfies a maximal number of linear equations over Z_2, which you have seen in class.

1. Weighted Max-Cut. Give a probabilistic 2-approximation algorithm to the weighted Max-Cut problem:
   You are given an undirected graph $G=(V,E)$, and a weight function $w: E \to R^+$. A cut in G is a subset $S \subseteq V$, the weight of the cut is defined as $w(S)= \sum_{e \in E(S)} w(e)$ where $E(S) = \{\{u,v\} \in E : u \in S, v \in \overline{S}\}$. The problem is to find the cut of maximal weight.

2. Verification of sorting networks.
   a. You are given a comparison network which receives n inputs, and you wish to verify that it indeed sorts any set of n inputs (i.e. that its output is the n inputs in a sorted order). Show it is enough to test the network on $n!$ certain inputs to verify its correctness.
   b. Show that if, given input sequence $a_1,a_2,\ldots,a_n$, a comparison network outputs the sequence $a_{\pi(1)},a_{\pi(2)},\ldots,a_{\pi(n)}$ for some permutation π, then for any weak-monotonically ascending function f, it returns the output $f(a_{\pi(1)}), f(a_{\pi(2)}),\ldots, f(a_{\pi(n)})$, given the input $f(a_1),f(a_2),\ldots,f(a_n)$.
   c. Show that if a comparison network which receives n inputs, sorts correctly every input composed of $0,1$ (i.e. all possible 2^n sequences of $\{0,1\}$) then it sorts correctly every input of size n.

3. Assume the *PRAM (Parallel Random Access Memory)* model with *CRCW (Concurrent Read Concurrent Write)*.
   Assume conditioning/comparison/arithmetic/memory operations are one operation and consume $O(1)$ time.
   a. Give an algorithm that sorts n different numbers in $T(n)=O(\log(n))$ time, $W(n)=O(n^2)$ operations.
   b. Give an algorithm that finds the maximum of n numbers in $T(n)=O(1)$ time, and $W(n)=O(n^{1.5})$ operations.

   Hint: Both algorithms are modifications on the $O(1)$ time algorithm for finding maximum that we've seen in the tirgul.

Algorithms – Exercise 13 (Solution)

1. We give the following algorithm: for every vertex $v \in V$, put it in the set S in probability *0.5*. Do so independently for each vertex. We will denote the maximum cut in the graph with S_{max}.
   Then clearly $w(S_{max}) = \sum_{e \in S_{max}} w(e) \leq \sum_{e \in E} w(e)$. Let 1_e be a random variable that takes the value *1* if $e \in E(S)$, and *0* otherwise. Then
   $E(w(S)) =$
   $$E[\sum_{e \in E} w(e) \cdot 1_e] = \sum_{e \in E} w(e) \cdot E[1_e] = \sum_{\{u,v\} \in E} w(e) \cdot Pr(u \in S, v \notin S \_OR\_ u \notin S, v \in S)$$
   $$= \sum_{\{u,v\} \in E} w(\{u,v\}) \cdot (0.5 * 0.5 + 0.5 * 0.5) = 0.5 \sum_{(u,v) \in E} w(e).$$

   We conclude: $w(S_{max}) \leq 2\, E(w(S))$, and therefore the algorithm is a probabilistic 2-approximation to *MAX-CUT*.

2.

   a. We will test the network on all the possible permutations of *{1,2,…,n}*. All together there are *n!* such permutations. Let us assume that it sorts all these inputs correctly. Let $(x_1,\ldots,x_n)$ be another set of <u>different</u> integers. We will prove that it sorts them correctly as well. Let π be a permutation such that $x_{\pi(1)} < x_{\pi(2)} \ldots < x_{\pi(n)}$. We will prove that for every level *i* of the sorting network, the ordering of the elements for the input $(x_1,x_2,..,x_n)$ is the same as that of the elements for the input $(\pi^{-1}(1),\ldots,\pi^{-1}(n))$:

      For *i=0*, it follows from the definition of π.

      $i \to i+1$. If we assume that the ordering of $(x_1,\ldots,x_n)$ is the same as that of $(\pi^{-1}(1),\ldots,\pi^{-1}(n))$ after the *i-th* level, then since they are ordered the same, then for every comparison operation the outcome would be the same, and so we see that if the permutation (ordering) before the *(i+1)-th* level was σ for both outputs, then the comparators on the *(i+1)-th* level performed the same permutation τ on these two inputs, and so the permutation after the *(i+1)* stage is $\tau \circ \sigma$ for both inputs.

      This shows that the network sorts well for a set of *n* different numbers. If $(x_1,\ldots,x_n)$ is a sequence of not necessarily different integers, then we may define a new order: $x_i \succ x_j$ *iff* $x_i > x_j$ *or* $x_i = x_j$ *and* $i > j$. Then we may assume that the comparators perform the $\succ$ rather then > operation (since if they encounter two equal numbers, it does not matter if they swap them or not). And so we get a reduction to the problem of sorting *n* different numbers, since $(x_1,\ldots,x_n)$ are considered different under the $\succ$ order relation.

b. See Lemma 27.1 in Cormen (p. 709)
c. See Lemma 27.2 in Cormen (p. 711)

3.

a. The algorithm:
   i. Stage 1: Build a two dimensional $n*n$ array of $\{0,1\}$, where in the (i,j) coordinate, there is a 1 iff $A(i)>A(j)$ or $A(i)=A(j)$ and $i \geq j$. All these operations can be done concurrently since we have *Concurrent Read* assumption. Denote this array by B. Since all these are done concurrently, this stage takes time $T(n)=O(1)$, and operations $W(n)=O(n^2)$.
   ii. Stage 2: Build an array M of n elements such that $M(i)=B(1,i)+B(2,i)+\ldots B(n,i)$. For every i. The calculation of separate $M(i)$-s is done concurrently, we will show below how to calculate the sum of n numbers in $W(n)=O(n)$ and $T(n)=O(\log(n))$. And so all together, this stage takes $W(n)=n*O(n)=O(n^2)$ and time $T(n)=O(\log(n))$.
   iii. Stage 3: Now for every i, $M(i)$ holds the number of cells which are smaller then $A(i)$ plus the number of elements that are equal to $A(i)$ but have a smaller index in A. So if we for every i, we write $A(i)$ into $C(M(i))$, then C will be a sorted array which holds the same elements as A. This can be done in $W(n)=O(n)$, and $T(n)=O(1)$, since all the cells of C can be computed concurrently.

   <u>*Lemma:*</u> In the *PRAM CRCW* model, it is possible to calculate the sum of n numbers in $O(\log(n))$ time, with $O(n)$ operations.

   <u>*Proof:*</u> We give an algorithm – Given $a_1,\ldots,a_n$, if $n=1$ return a_1, else calculate recursively $s_1=a_1+\ldots+a_{ceil(n/2)}$, and $s_2=a_{ceil(n/2)+1}+\ldots+a_n$, and return s_1+s_2. The running time of the algorithm satisfies the recursion: $T(n)=T(n/2)+O(1)$, and so $T(n)=O(\log(n))$, and the operations number satisfies $W(n)=2W(n/2)+O(1)$, and so $W(n)=O(n)$.

   <u>Running Time:</u> Summing up the running time from the three stages we reach $T(n)=O(1)+O(\log(n))+O(1)=O(\log(n))$. Also summing up the operations $W(n)=O(n^2)+O(n^2)+O(n)=O(n^2)$.

   <u>Remark:</u> In this algorithm we don't use the fact that it is also a *Concurrent Write* model, so it could be done also with *Exclusive Write*.

b. We have seen in the tirgul how to find the maximum of n elements in $O(1)$ time and $W(n)=O(n^2)$ using the *CRCW* model (see remark below). We perform the following algorithm:
   i. Divide the n given numbers into $n^{1/2}$ sets of $n^{1/2}$ numbers. Use n processors for each of the $n^{1/2}$ sets to find their maximum in $O(1)$ time. Save the maximum of each of these $n^{1/2}$ sets in $m_1,\ldots,m_{sqrt(n)}$.

ii. Use n processors to find the maximum of $m_1,\ldots,m_{sqrt(n)}$. Output that maximum.

Since stage i. can be done in $O(1)$ time for each of the sets, and the calculation in each set can be done concurrently, we see that it takes $O(1)$ time for the first stage. Since we use $n^{1/2}$ sets of n processors for $O(1)$ time, we get $W(n)=O(n^{1.5})$ for the first stage. The second stage uses n processors, and takes $O(1)$ time, and so $W(n)=O(n)$ operations. All in all, we get $O(1)$ time and $W(n)=O(n^{1.5})$, where we obviously did not use more then $n^{1.5}$ processors at each time phase.

Remark: Pay notice that if the inputted numbers are not different, we need a small modification of the algorithm shown in class for finding the maximum of n numbers. Since we want the array B to have exactly one row in which there are only 1s (and that would be the row with the index of the maximal element), we can modify it as follows (the modification is in the *if*):

Input: an array A holding n numbers.

1. for $1 \le i, j \le n$
   a. $If\ (A(i) > A(j)\ \_\ OR\ \_(A(i) = A(j)\ \_\ AND\ \_\ i \ge j))$
      i. $B(i,j) \leftarrow 1$
   b. $Else\ B(i,j) \leftarrow 0$
2. $for\ 1 \le i \le n$
   a. $M(i) \leftarrow B(i,1)\ \&\ B(i,2)\ \&\ \ldots\ \&\ B(i,n)$

Output: An array M of length n, with $(n-1)$ 0s, and 1 in the index of one of the maximal elements of A. We can easily write the maximum into another address in $O(1)$ time, and $O(n)$ operations, when asked to.