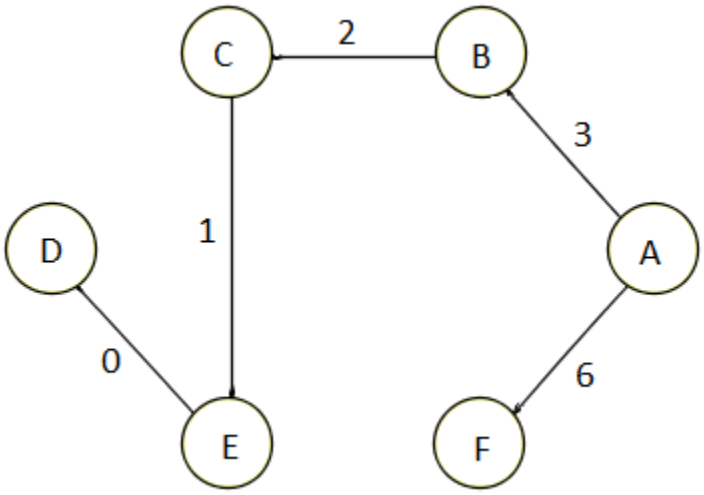
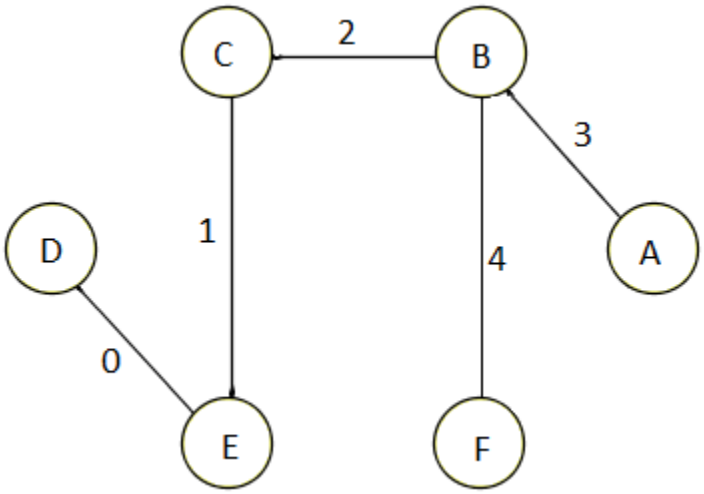
CSE 101 Winter 2013 Final Solutions (Version A)

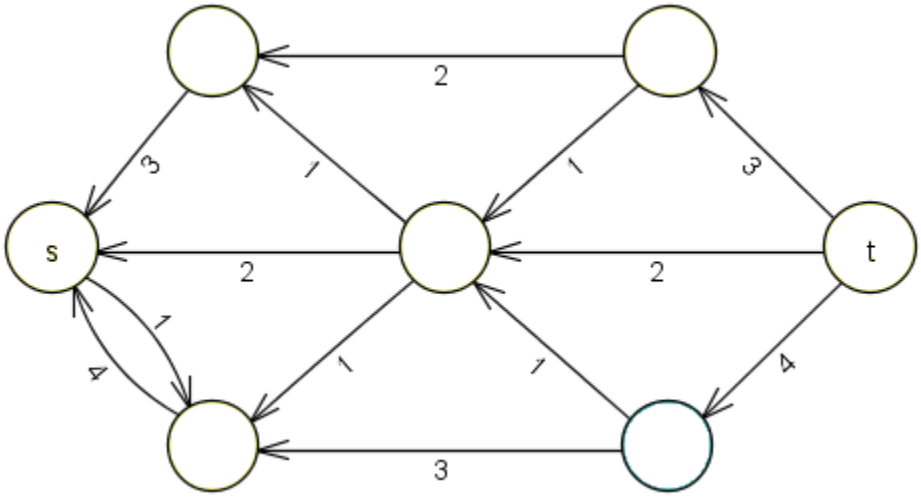
1. (a) The shortest-paths (Dijkstra's tree) is:



The MST is:



(b) The final residual graph is:



Maximum Flow = 9

2. (a) The recurrence relation is:

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

The 2 subproblems of size (n/2) are obtained by the two recursive calls on the left and right halves of array A, and O(n) work is done at each step for printing out the elements of the array.

Comparing this recurrence to the general form $T(n) = aT(n/b) + O(n^d)$, we see that:

$a = 2, b = 2, d = 1$

Therefore, $a = b^d = 2$.

By the Master Theorem, the solution to this recurrence is:

$T(n) = O(n^d \log n) = O(n\log(n))$

(b) There exists an **NP** problem which is also in **P**. → Neither. All problems in **P** are contained in **NP**.

There exists a problem in **P** which is **NP**-hard. → **P = NP**. If a problem is **NP**-hard, it indicates indicates that all **NP**-complete problems are reducible to it in polynomial time. Hence, if that **NP**-hard problem can be solved in polynomial time, so can all **NP**-complete problems.

There exists a problem in **NP** which is neither in **P** nor **NP**-complete. → **P** ≠ **NP**. A problem existing in **NP** but not in **P** directly implies that **P** is not equal to **NP**.

(c) This problem can be modeled as a matching problem.

Let each of the *n* tasks $t_i \in T$ and *m* servers $s_j \in S$ be represented by a distinct node in the flow network.

For each of the servers s' that a particular task t' is suited to, add an edge into the flow network from t' to s' with capacity 1.

Add a supersource node A to the flow network with exactly one edge of capacity 1 from node A to each task t_i. Similarly, add a supersink B to the flow network with exactly one edge of capacity 1 from each server s_j to B.

(We avoid naming the supersource and supersink S and T to prevent confusion with T, the set of tasks, and S, the set of servers.)

We can then find the max-flow in this network to solve the problem.

3. (a) Algorithm:

Sort all jobs $j_1, j_2, \ldots, j_n$ in ascending order of their durations $t_1, t_2, \ldots, t_n$.

Repeat the following until all jobs are finished:

Pick the first job in the list and execute it.

Remove that job from the list.

(b) Proof of correctness:

Assume towards a contradiction that there exists an optimal schedule of jobs called OPT, which results in waiting time $\leq$ that of the greedy schedule, S.

Let OPT_i and S_i indicate the first i terms of the schedules, respectively. Then, $OPT_0 = S_0$.

Let $OPT_i = S_i$ for some i. Assume that job i+1 of OPT, say j*, is not equal to job i+1 of S, say j'. We know that the duration of j', t' $\leq$ t*, the duration of j*, as the greedy algorithm picks the job with the least duration every time.

Swap the jobs j' and j* in OPT, such that j' now appears before j*. Then, the waiting times of all the jobs following j' in the new ordering have reduced by time (t* - t'). If t* < t', this improves the optimal schedule, which contradicts the definition of OPT. Therefore, t* = t', i.e., the optimal schedule picks the job with lowest duration every time.

(c) Running time:

Sorting all *n* jobs takes O(nlog(n)) time.

Picking and executing each job one by one takes O(n) time overall.

Therefore, the overall running time of the algorithm is O(nlog(n)).

4. (a) Subproblem definition:

Let subproblem R(i) stand for the result obtained from an optimal selection of "+" or "x" operators for the first i operands a_1, a_2, …, a_i.

(b) Base cases:

For consistency, R(0) = 0

R(1) = a_i, as no operand preceding a_i exists.

Recurrence relation:

$R(i) = \max\{R(i-1) + a_i, R(i-2) + (a_{i-1} \times a_i)\}$

The maximization is performed over two options:

$R(i-1) + a_i$ is the option of placing a "+" operator before the the i^{th} term, i.e., a_i is added to the optimal result from the first i-1 terms.

$R(i-2) + (a_{i-1} \times a_i)$ is the option of placing a "x" operator before the i^{th} term, i.e., a_i is multiplied with the $(i-1)^{th}$ term. As two multiplication operations cannot occur consecutively, we explicitly add a "+" operator between the optimal result of the first i-2 terms and a_{i-1}, and then a "x" operator between a_{i-1} and a_i.

(c) Running time:

Setting the base cases takes constant time, O(1).

To evaluate each of the *n-1* subproblems R(i), it takes two addition operations, one multiplication operations and one maximization operation. Together with the constant-time lookup of previous subsolutions, each subproblem is solved in constant time, O(1).

Therefore, the overall running time of the algorithm is O(n), linear time.

5. (a). Input: Undirected graph G, $k >= 0$.
Output: $S \subseteq V$, an IS of G, such that $|S| = k$

Idea: For each vertex $v \in V$, remove it if and only if the resulting graph still contains an IS of size k.

Algorithm: We are given an algorithm IS(G,k) that returns true if and only if G contains an IS of size k. We devise an iterative algorithm to solve the IS-SEARCH problem as follows.

```
IS-SEARCH(G,k):
    if !IS(G,k):
        return None  # no IS of size k in G
    end if
    for each vertex v in V:
        # remove v and all its incident edges
        V' = V - {v};  E' = E - { (u,v) : u in V }
        # check if there still exists an IS of size k
        if IS(G'=(V',E'),k):
            # This means v is not required for IS of size k
            V = V';  E = E'
        end if
     end for
     return V
```

(b). Running time analysis: Each vertex of G is examined exactly once and thus the inner for loop runs |V| times. Each iteration involves one invocation of IS(G,k) and linear amount of work to remove the incident edges.
Total time spent is $O(|V|*p(V,E) + |V|*(|V|+|E|))$ where $p(V,E)$ is runtime of IS on graphs with V vertices and E edges. The overall runtime is polynomial as long as $p(V,E)$ runs in polynomial time.

(c). Proof of Correctness: IS($G=(V,E),k$) remains true at every step so at the end, V contains every vertex to form IS of size k in G. At the same time, all other vertices will be removed as they are not required, so V contains only those vertices that form an IS of size k. Hence, the set V returned is an IS of size k in G.