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

The MST is:
(b) The final residual graph is:

```
Maximum Flow = 12
```
2. (a) The recurrence relation is:

\[ T(n) = 2T(n/4) + O(1) \]

The 2 subproblems of size \((n/4)\) are obtained by the two recursive calls on the left-most and right-most quarters of array \(A\), and \(O(1)\) work is done at each recursive step for calculating the value of \(i\).

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

\(a = 2,\ b = 4,\ d = 0\)

Therefore, \(a > b^d\).

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

\[ T(n) = O(n^{\log(a)/\log(b)}) = O(n^{1/2}) \]

(b) There exists a problem in \(NP\) which is neither in \(P\) nor \(NP\)-complete. \(\rightarrow P \neq NP\). A problem existing in \(NP\) but not in \(P\) directly implies that \(P\) is not equal to \(NP\).

There exists a problem in \(NP\) with a polynomial-time reduction to the known \(NP\)-complete problem \(SAT\). \(\rightarrow\) Neither. \(NP\)-hard problems exist that can be reduced to \(NP\)-complete problems in polynomial time, but this does not provide any indication of the relation between \(P\) and \(NP\).

There exists a polynomial-time reduction from the known \(NP\)-Complete problem \(SAT\) to the s-t shortest path problem in an undirected graph. \(\rightarrow P = NP\). The s-t shortest path problem in an undirected graph can be solved in polynomial time. If \(SAT\) can be reduced in polynomial time to this problem, \(SAT\) can be solved in polynomial time.
3. (a) **Algorithm:**

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

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 $O(n\log(n))$ time.

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

Therefore, the overall running time of the algorithm is $O(n\log(n))$. 

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 a₁, a₂, ..., aᵢ.

(b) Base cases:

For consistency, R(0) = 0
R(1) = aᵢ, as no operand preceding aᵢ exists.

Recurrence relation:

\[ R(i) = \max\{R(i-1) + aᵢ, R(i-2) + (aᵢ-1 \times aᵢ)\} \]

The maximization is performed over two options:

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

R(i-2) + (aᵢ-1 \times aᵢ) is the option of placing a “x” operator before the \( i^{th} \) term, i.e., aᵢ is multiplied with the (i-1)\textsuperscript{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 aᵢ-1, and then a “x” operator between aᵢ-1 and aᵢ.

(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 $G$, $k \geq 0$.
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 $O(|V| \cdot p(V,E) + |V| \cdot (|V|+|E|))$ 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 $G$. 