CSE 417
Algorithms and Computational Complexity
Midterm solutions Winter 2000

1. (25 points) Prove by induction that when $n \geq 2$ is a power of 2, the recurrence $T(n) = 2T(n/2) + 2$ with the initial condition $T(2) = 1$ has the *precise* solution $T(n) = 3n/2 - 2$.

Proof by induction on n:

Base case: $T(2) = 1$ by definition. $\frac{3}{2}2 - 2 = 3 - 2 = 1$, so it is correct in this case.

Inductive Hypothesis: Assume that $T(m) = \frac{3}{2}m - 2$ for all $m < n$.

Inductive Step:

$$\begin{aligned} T(n) &= 2T(n/2) + 2 && \text{by definition} \\ &= 2(\frac{3}{2}(n/2) - 2) + 2 && \text{by the inductive hypothesis} \\ &= 2(3n/4 - 2) + 2 \\ &= 3n/2 - 2 \end{aligned}$$

Therefore by induction it holds for all $n \geq 2$.

2. (25 points) Recall the *Quicksort* algorithm discussed in class and assume that we are running it on *distinct* elements.

   (a) What it the *worst-case* running time of *Quicksort* on an array of length n if the pivot is always chosen to be the first element of the array?

   $O(n^2)$

   (b) What is the *average-case* running time of *Quicksort* on an array of length n if the pivot is always chosen to be the first element of the array?

   $O(n \log n)$

   (c) The *median* of a set is the middle element of the set in sorted order. There is an algorithm that computes the median of a set of size n in *worst-case* $O(n)$ time. Suppose that in *Quicksort*, instead of using the first element of the array as a pivot, we computed the median element of the array and used it as the pivot.
   Write a recurrence describing the *worst-case* running time of this modified *Quicksort* algorithm on an array of length n.

   $T(n) = 2T(n/2) + cn$ for some constant c where the cn term reflects the costs of both finding the median and partitioning the array based on the pivot.

   (d) Solve the recurrence for part (c).

   $T(n) = O(n \log n)$ by the standard divide and conquer recurrence.

3. (25 points) Suppose you are given a *directed acyclic graph* $G = (V, E)$ which has an integer *reward* $r(v)$ associated with each vertex $v \in V$. For every node $v \in V$, we'd like to compute `best-reward`(v) which is the largest value of $r(w)$ among all nodes w reachable from v in G. Describe an algorithm running in $O(|V| + |E|)$ time that computes `best-reward`(v) for all nodes $v \in V$ and argue that is both correct and has the desired running time.
(Hint: use topological sort and consider the vertices in the reverse of this order.)

Run topological sort on the graph to obtain vertices numbered $v_1, \dots, v_n$. The important property is that there are no edges going from a vertex v_j to a vertex v_i for $i < j$.

```
For$j = n$ to 1 do
    best-reward$(v_j) = r(v_j)$
    Forall $w$ with $(v_j, w) \in E$ do
        if best-reward$(w)$ $>$ best-reward$(v_j)$ then
          best-reward$(v_j)$=best-reward$(w)$
    end for
end for
```

In total topological sort uses $O(|V| + |E|)$ time and we look at each vertex and each edge exactly once so we also take $O(|V| + |E|)$ time. Note that the algorithm relies on the key property to ensure that in the inner loop, all the best-rewards are already calculated.

Other solutions: For the subgraph reachable from a node v one can do DFS and in the postwork on a node set each nodes best-reward value to be the largest of the best-rewards in its children. However, to do this for all nodes v would seem to require $O(|V|(|V| + |E|))$ work. However, one could modify DFS so that when we repeat DFS on another node, if that node is alread visited then we don't visit it or the nodes reachable from it and instead we just take its best-reward value that's already been computed.

Common problems: assuming all larger numbered vertices are reachable from v_j and assuming you only needed to compute this for one vertex v.

4. (25 points) Consider the following recursive function designed for integers m, $n \geq 1$ (don't worry about its meaning).

```
Function Best-Match($m$,$n$)
    if ($m = 1$ or $n = 1$) then
        return 1
    else
        if $n$ is odd then
            $k$←(n+1)/2
        else
            $k$←n/2
        end if
        return max{Best-Match(m,k)+Best-Match(m-1,n), Best-Match(m,n-1)}
    end if
end
```

(a) Use *Dynamic Programming* to rewrite this function to be much more efficient.

Here is the Dynamic Programming solution with the simplest change to the code I can think of. For efficiency one would get rid of the large *if* inside the for loops and instead have two extra single initialization loops.

```
Function Best-Match($m$,$n$)
    for $i = 1$ to $m$ Do
        for $j = 1$ to $n$ Do
            if ($i = 1$ or $j = 1$) then
                Table[i,j]←1
            else
                if $j$ is odd then
                    $k$←(j+1)/2
                else
                    $k$←j/2
                end if
                return max{Table[i,k]+Table[i-1,j], Table[i,j-1]}
            end if
        end for
    end for
    Return Table[m,n]
end
```

(b) What is the *Time* and *Storage Space* used by your new algorithm?

$O(mn)$ for both, although space can be reduced to $O(n)$ by storing only the table for the current and previous rows.