H.W 5 Solutions

Problem 6.1 – Contiguous Subsequence of largest sum

Subproblems:
Define an array of subproblems $D(i)$ for $0 \leq i \leq n$. $D(i)$ will be the largest sum of a (possibly empty) contiguous subsequence ending exactly at position i.

Algorithm and Recursion:
The algorithm will initialize $D(0) = 0$ and update the $D(i)$'s in ascending order according to the rule:
$D(i) = max\{0, D(i-1) + a_i\}$
The largest sum is then given by the maximum element $D(i)^*$ in the array D. The contiguous subsequence of maximum sum will terminate at i^*. Its beginning will be at the first index $j \leq i^*$ such that $D(j-1) = 0$, as this implies that extending the sequence before j will only decrease its sum.

Correctness:
The contiguous subsequence of largest sum ending at i will either be empty or contain a_i. In the first case, the value of the sum will be 0. In the second case, it will be the sum of a_i and the best sum we can get ending at $i-1$, i.e. $D(i - 1) + a_i$. Since we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

Running Time:
The running time for this algorithm is $O(n)$, as we have n subproblems and the solution of each can be computed in constant time. Moreover, identification of the optimal subsequence requires a single $O(n)$ time pass through the array D.

Problem 6.7

Subproblems: Define variables $L(i, j)$ for all $1 \leq i \leq j \leq n$ so that, in the course of the algorithm, each $L(i, j)$ is assigned the length of the longest palindromic subsequence of string x[i, …., j].

Algorithm and Recursion:
The recursion will then be:
L(i, j) = max{L(i + 1, j), L(i, j−1), L(i+1, j − 1) + **equal**(x[i], x[j])*2}
where **equal(a, b)** is 1 if *a* and *b* are the same character and is 0 otherwise.

Pseudo Code:

```
For i=1 to n
    L(i,i) = 1

For i = 1 to n-1
    L(i,i+1) = 1 + equal(i,i+1)

For length = 3 to n                  // Subsequences of lengths 3 to n
   For i = 1 to n – length +1
     j = i + length -1
     L(i, j) = max (L(i+1, j), L(i,j−1), L(i+1,j−1) + equal(x[i],x[j])*2)

Return L(1,n)
```

Correctness and Running Time:
Consider the longest palindromic subsequence s of x[i,.... , j] and focus on the elements x[i] and x[j] . There are then three possible cases:

- If both x[i] and x[j] are in s then they must be equal and L(i, j) = L(i + 1, j − 1) + 2
- If x[i] is not a part of s, then L(i, j) = L(i + 1, j).
- If x[j] is not a part of s, then L(i, j) = L(i, j − 1).

Hence, the recursion handles all possible cases correctly. The running time of this algorithm is $O(n^2)$, as there are $O(n^2)$ subproblems and each takes O(1) time to evaluate according to our recursion.

Problem 6.10

Subproblems

Define $L(i, j)$ to be the probability of obtaining exactly j heads amongst the first i coin tosses.

Algorithm and Recursion

By the definition of L and the independence of the tosses, it is clear that:

$$L(i, j) = p_i L(i-1, j-1) + (1 - p_i) L(i-1, j) \qquad j = 0, 1, \ldots, i$$

We can then compute all $L(i, j)$ by initializing $L(0, 0) = 1$, $L(i, j) = 0$ for $j < 0$, and proceeding incrementally (in the order $i = 1, 2, \ldots, n$, with inner loop $j = 0, 1, \ldots, i$). The final answer is given by $L(n, k)$.

```
Algorithm FindProb (n,k)

L[0][0] = 1
L[0][-1] = 0
For i = 1 to n
    For j = 0 to i
          L[i][j] = p_i L[i-1][j-1] + (1-p_i )L[i-1][j]

Return L(n,k)
```

Correctness

The recursion is correct as we can get j heads in i coin tosses either by obtaining $j-1$ heads in the first $i-1$ coin tosses and throwing a head on the last coin, which takes place with probability $p_i L(i-1,j-1)$, or by having j heads after $i-1$ tosses and throwing a tail in the last toss, which has probability $(1-p_i)L(i-1, j)$. Besides, these two events are disjoint, so the sum of their probabilities equals $L(i, j)$.

Running Time

The 2 nested loops take $O(n^2)$ time.

Problem 6.17

This problem reduces to Knapsack with repetitions. The total capacity is v and there is an item i of value x_i and weight x_i for each coin denomination. It is possible to make change for value v if and only if the maximum value we can fit in weight v is v. The running time is $O(nv)$.

Problem 6.21

Subproblems
The subproblem V (u) will be defined to be the size of the minimum vertex cover for the subtree rooted at node u. We have V (u) = 0 if u is a leaf, as the subtree rooted at u has no edges to cover. The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes.

Hence for any internal node i

$$V(i) = min((\sum_{j:(i,j)\in E}(1+\sum_{k:(j,k)\in E}V(k)),\ \ (1+\sum_{(j:(i,j)\in E}V(j)))$$

The algorithm starts at the leaf nodes and fills up the array V(i) in the order of decreasing depth (bottom up order) until it reaches the root of the tree.

Correctness
The algorithm considers 2 cases:

If node i is included in the Minimum Cover Set, then all edges connecting node i to its descendants are covered and finding the Minimum Cover Set reduces to covering all edges beyond *i's* descendants (which involves finding the Minimum Cover set for each subtree rooted at *i's* descendants). The solution to this case is given by
$(1+\sum_{(j:(i,j)\in E}V(j))$, where the addition of 1 signifies the inclusion of i in the cover set.

If node *i* is not included in the Minimum Cover Set, then all descendants of *i* have to be included in the set (to cover edges connecting *i* to its descendants). Once the descendants of *i* are added to the set, we don't have to worry about covering edges connecting the descendants of *i* to their descendants in the tree, and the problem reduces to finding the Minimum Cover set for each subtree rooted at the descendants of the descendants of *i.* The solution to this case is given by:

$$\sum_{j:(i,j)\in E}(1+\sum_{k:(j,k)\in E}V(k))$$

Since the minimum cover set will select the minimum of the two possible scenarios, our recursion is correct.

Running time

There are |V| subproblems but for computing *V(i)* for all vertices *i* in the graph we look at atmost 2|E| edges in total. Hence the algorithm runs in O(|E|) time.