HW3 Solutions
CSE 101, Spring 2002

Problem 1

Please refer to the example given in CLRS section 15.2.

Problem 2

The easiest way to solve this problem is to just use the DP algorithm for finding the LCS of two arrays as a black box. Applied to this problem, the first step of this approach would be to make a copy of the input array and then sort it in decreasing order. Next we would simply find the LCS of the two arrays and output its length. The running time of this would be

\[
O(\text{time for sorting + time for LCS}) = O(n \log n + n^2) = O(n^2)
\]

The alternative solution (that done by the cool students.) Was to solve this problem directly, with a unique DP algorithm. This required a fuller knowledge of DP. This approach involved the following steps:

1. **Define the problem:** Let \(LDS(i)\) be defined as the length of the longest decreasing subsequence of the sub-array starting at index \(i\) and going to the end of \(A[]\), where the size of \(A[]\) is \(n\). So the length of the longest decreasing subsequence would be in \(LDS(1)\).

2. **Define the recursive Solution.**

   \[
   LDS[i] = \begin{cases} 
   1 & \text{if } i = n \\
   \max(\text{LDS}[i + 1], 1 + \text{LDS}[k] \text{ s.t. } A[k] < A[i]) & \text{otherwise}
   \end{cases}
   \]

   This isn’t as bad as it looks. Basically at every point we look at the possibility of including the currently indexed element into the sequence, to do this we add the element and then look for the next element which could follow it in the sequence (hence the requirement for the \(A[k] < A[i]\), the next elements has to be less than the current one). We compare the length of this sequence and compare it to the best known sequence which doesn’t include this element, and we take the maximum.

3. **Build the DP algorithm** This is done by “flipping” the recursion and working from the bottom up. We basically work back from \(i = n\).
LONGESTDECREASINGSEQ(A[n])
1  LDS[n] ← 1
2  for i ← (n − 1) to 1
3     do
4         for k ← i to n
5             do
6                 if A[k] < A[i]
7                     then BREAK
8
9     if (LDS[i + 1] < 1 + LDS[k])
10        then LDS[i] ← 1 + LDS[k]
11      else LDS[i] ← LDS[i + 1]
12
13  return LDS[1]

The running time of this algorithm is bounded by the nested for loop, which is \(O(n^2)\). This gives you the same running time as the above approach.

Problem 3

Again there were two different approaches to this problem, but here these depended on the specifics of the problem definition. Assuming that you could invest two separate dollar amounts into the same enterprise, then it was possible to do a reduction to the 0-1 Knapsack problem. But if you didn’t make this assumption, then again you had to “roll your own” DP algorithm. We’ll deal with the cases in this same order.

Knapsack approach

Here we treat each element of the return table as an object which can be placed in the knapsack. The size of the knapsack is then the amount of money \(n\), which we have to invest. We’ll assume that we have a black-box algorithm for the 0-1 knapsack problem, though you probably should have been familiar with it to solve this problem. I say that because many people got the running-time for this approach incorrect, due to lack of understanding.

Runtime analysis: The runtime of the 0-1 Knapsack DP algorithm is \(O(NC)\), where \(N\) is the number of objects which can be placed in the bag, and \(C\) is the capacity of the bag. So how does this translate into the running time for our Algorithm? Well our \(n\) maps into \(C\), while the number of objects is \(n \times m\), which maps into \(N\). So replacing these terms with our own we get the runtime of \(O(mn^2)\). (The most common mistake people made with this problem was to assume the that running time of our algorithm was exactly the same as the running time of 0-1 Knapsack algorithm.)
General DP approach

Here we make the assumptions (Following the same outline as problem 2...)

1. Define the problem Define $Invest[n, e]$ to be the best investment return for $n$ dollars in the set of 1...$e$ enterprises of the total $E$.

2. Define the Recursive Solution (Note: Many of you got this part of DISCUS. It is a partial solution, but not one that is complete. To receive full credit you still needed to turn it into a DP algorithm with sub-exponential running time.)

$$Invest[m, E] = \begin{cases} 
0 & \text{if } m = 0 \text{ or } E = \emptyset \\
\max_{0 \leq i \leq m}(\text{return}[i, e] + Invest[m - i, E - \{e\}]) & \text{otherwise} 
\end{cases}$$

3. DP Algorithm Let $n$ be the dollar investment, $E$ be the set of companies, and $\text{return}[][]$ be the table of investments.

$\text{INVESTOR}(n, E, \text{return}[][])$
1. Populate the first row of the table
2. for $i \leftarrow 1$ to $n$
3. do
4. $\text{Invest}[i, 1] \leftarrow \text{return}[i, 1]$
5. Now fill the rest of the table
6. The outer loop increments the number of enterprises in the set.
7. for $e \leftarrow 2$ to $E$
8. do
9. The inner loop explores all possible investments within this set of enterprises.
10. for $i \leftarrow 1$ to $n$
11. do
12. $\text{Invest}[i, e] = \max_{0 \leq k \leq i}(\text{return}[i, e] + \text{Invest}[m - i, e - 1])$
13. return $\text{Invest}[E, n]$

Runtime Analysis This algorithm contains a max function, doubly nested within two for loops. The max must explore all possible combinations from 0 ≤ $k$ ≤ $i$, where $i$ is $n$ in the worst case. This leads to a runtime of $O(En^2)$, where $E$ is actually $m$ from the problem statement (sorry for the lack of consistency). So in terms of the original variables we have a runtime of $O(mn^2)$. Which is the same as the first algorithm.

Problem 4

Use a modified DFS to solve the maze. Let us think of the maze as a graph, where the rooms are the nodes in the graph $G$ and there is an edge between two nodes if the two rooms that
are represented by those two nodes are adjacent. To solve the maze problem we apply DFS with the following color coding:

\texttt{discovered} = "a penny placed heads up in the center of the room"

\texttt{finished} = "a penny placed tails up in the center of the room"

As soon as we enter a room that is outside of the maze we stop since we have found a way out.

\section*{Problem 5}

Use a modified BFS to determine whether a graph is bipartite.

\begin{algorithm}
\textbf{BP}(G, s)
  \begin{algorithmic}
    \State for each vertex \( v \)
    \State \hspace{1em} \texttt{color}[v] \leftarrow \texttt{WHITE}
    \State for each vertex \( v \)
    \State \hspace{1em} if \texttt{color}[v] = \texttt{WHITE}
    \State \hspace{2em} \texttt{color}[v] \leftarrow \texttt{BLUE}
    \State \hspace{2em} \texttt{EnQ}(Q,v)
    \State \hspace{2em} \textbf{while} \( Q \neq \emptyset \)
    \State \hspace{3em} \( u \leftarrow \texttt{DeQ}(Q); \)
    \State \hspace{3em} \textbf{for each} \( v \) \texttt{adj}[u]
    \State \hspace{4em} if \texttt{color}[u] = \texttt{color}[v]
    \State \hspace{5em} return \texttt{FALSE}
    \State \hspace{4em} if \texttt{color}[v] = \texttt{WHITE}
    \State \hspace{5em} \textbf{if} \texttt{color}[u] = \texttt{RED}
    \State \hspace{6em} \texttt{color}[v] \leftarrow \texttt{BLUE}
    \State \hspace{5em} \textbf{else}
    \State \hspace{6em} \texttt{color}[v] \leftarrow \texttt{RED}
    \State \hspace{5em} \texttt{EnQ}(Q,v)
    \State \hspace{4em} \textbf{end for each}
    \State \hspace{3em} \textbf{end while}
    \State \hspace{2em} \textbf{end if}
    \State \hspace{1em} \textbf{end for each}
    \State \hspace{1em} return \texttt{TRUE}
  \end{algorithmic}
\end{algorithm}

Runtime complexity of the algorithm is \( \Theta(V+E) \). Space complexity of the algorithm is \( \Theta(V) \).