COP 4531 - Assignment 6

Due Date - Monday, December 1st

Assigned - Friday, November 21st

Problem 1

We are given a list of jobs, along with the number of minutes it would take to complete each job. Our goal is to schedule these jobs to minimize the **average completion time**.

Example: Consider three jobs, job A that takes 50 minutes to complete, B that takes 20 minutes, and C that takes 35 minutes to complete.

If we schedule the jobs in the order A, B, C, then job A completes at time 50, job B completes at time 70, and job C completes at time 105. Then average completition time is $\frac{50+70+105}{3} = 75$. Note that this ordering is not optimal.

A. Give the pseudocode for a greedy algorithm that gives an optimal solution to this problem. Justify your answer.

B. Analyze the running time of your algorithm.

Problem 1 Solution

A. Give the pseudocode for a greedy algorithm that gives an optimal solution to this problem. Justify your answer.
Here we have the solution idea from the example that is given. As we want to minimize the average completion time we want the jobs to be as small as possible. Thus we would select the jobs starting with the smallest and so on. Let J , denote a list of jobs with 2 attributes *J.name* and *J.time*.

```
1. Sorted_Set = Sort(J) on attribute J.time
2. sum = 0;
3. for job in Sorted_Set:
4.   sum = sum + length(job.time)
5. end
6. return Sorted_Set,sum/#(J)
```

```
Simply print J.name for each J in Sorted_Set,
to get the order of execution
```

In order to prove that this is optimal we can assume that there is an optimal solution O and a greedy solution G. Suppose that they agree till the job i. Then for the $i+1$ job they are different. Then we can swap out the $i+1$ job from O and replace it with the $i+1$ job from G to get a better average. This contradicts the optimality of O. Hence we have the proof.

B. Analyze the running time of your algorithm.
The sort sub-routine will run in $\Theta(n \log n)$ (assuming merge sort) . The linear scan to find the sum is $O(n)$. But the running time will dominated by the sort and hence will be $\Theta(n \log n)$.

Problem 2

Consider the following problem:

We are given a rod of length m for some positive integer m, and positive integers $s_1, s_2, \ldots, s_k$ where $s_1 = 1$ and $s_i < m$ for all $2 \leq i \leq k$, representing valid sizes for smaller rods.

Our task is to cut the rod of length m into pieces, so that each piece is of length s_i for some $1 \leq i \leq k$, and **the number of pieces is minimized**.

Example: Given a rod of length 52, and positive integers 1, 5, and 10 for the valid sizes of smaller rods, the optimal solution is to divide the rod of length 52 into five pieces of length 10 and two pieces of length 1.

A. Write pseudocode for a greedy algorithm for the above problem. *Hint: Make the first cut as big a possible, followed by next largest possible cut, etc.*

B. Analyze the running time of the above algorithm.

C. Prove that this algorithm is not optimal if the valid sizes of smaller rods are 1, 3, and 4.

D. Fix $k = 3$, and come up with values of s_2 and s_3 so that this greedy algorithm would be optimal for this instance of the problem. Justify your answer.

E. Come up with a dynamic programming algorithm to this problem that would be all optimal for all instances of this problem. Justify your solution.

F. Analyze the running time of your dynamic programming solution.

Problem 2 Solution

A. Write pseudocode for a greedy algorithm for the above problem.
We note that the most obvious greedy strategy is to take the largest possible cut size and see how many of this size we can get from the original rod. If we assume that s_i is the largest sized cut allowed then the number of such cuts is $\lfloor \frac{m}{|s_i|} \rfloor$Then for the remainder of the rod of size $m - \lfloor \frac{m}{|s_i|} \rfloor \cdot |s_i|$ we consider the next largest size cut and so on till we are done. Let us denote the valid cuts by $S = \{s_1, ..., s_k\}$. Then the algorithm goes as follows:

```
1. Sorted_Set = Sort_Descending(S)
2. piece_counter = 0;
3. for s in Sorted_Set:
4.   if s > m:
5.     continue
6.   piece_counter = piece_counter + floor(m/length(s))
7.   m = m - floor(m/length(s)) * length(s)
8.   if (m == 0) // this will happen as min(S) = 1
9.      break;
10.end
11.return piece_counter
```

B. Analyze the running time of the above algorithm.
We note that the for loop runs through the entire set of the possible cut sizes and continues till we have a positive length of the original rod left. This is dominated by the time to sort and hence the time is $\Theta(n \log n)$.

C. Prove that this algorithm is not optimal if the valid sizes of smaller rods are 1, 3, and 4.
We fix $m = 10$. Then using the greedy algorithm we have $piece\_counter = 4$. This is because we will get 2 cuts of size 4 and two cuts of size 1. But we can do better because we could have taken two cuts of size 3 and one cut of size 4 making the total number of pieces 3 which is better than what we get using the greedy. Hence the greedy is not optimal.

D. Fix $k = 3$, and come up with values of s_2 and s_3 so that this greedy algorithm would be optimal for this instance of the problem. Justify your answer.
We can just fix $s_2 = 3$ and $s_3 = 6$. The greedy solution is always optimal for these values because s_2 divides s_3, and s_3 divides s_2.

By way of contradiction, assume that there is value of m so that the greedy algorithm is not optimal for these values of s_2 and s_3. Order the rods in the optimal solution and the greedy solution in descending order, and let i be the position in these orderings on which they differ. Let a be the i^{th} rod in the greedy solution, and b be the i^{th} rod in the optimal solution.

Then a is larger than b, and all rods after the i^{th} rod in the optimal solution are smaller than a. Since all smaller rods divide all larger rods, this means that there is a set of rods in the optimal solutions that add to a. Therefore, a can replace multiple rods in the optimal solution, reducing the number of rods in the optimal solution. This is a contradiction, and as such the greedy solution is optimal.

E. Come up with a dynamic programming algorithm to this problem that would be all optimal for all instances of this problem. Justify your solution. Let $C(m) =$ the optimal number of cuts using the set S of possible cut sizes and length m of the given rod. Note that if O is the optimal solution and the cut s_i is included in the optimal solution then we have that $C(m) = 1 + C(m - |s_i|)$. The problem is that we do not know s_i because we do not know O. Note that the way we are writing this the cut s_i can be included multiple times in O. So what do we do: we want to minimize and hence the most logical way to go about would be to select the s_i that gives us a minimum for $C(m) = 1 + C(m - |s_i|)$. Thus we have the following:

$$C(m) = 0$$

if $m = 0$. Otherwise

$$C(m) = min_{\{s_i \in S\}}\{1 + C(m - |s_i|)\}$$

Now we are done. The above recurrence gives the optimal number of rods that can be obtained in a bottom up manner. Note that in order to implement the solution we need to go through the values of m systematically. For each possible value of m we get a sub-problem. Thus if suppose we want to get the optimal number of rods for $m = M$ then we would want to compute $C(M)$ and we would have two for loops one from $m = 2 : M$ and the other from $i = 1 : k$. This would give us the hint at the running time of the algorithm. Looking at the pseudocode below we see that the for loop in line 1 goes on M times and the minimum in line 2 is computed for every iteration of the loop. This minimum computation takes $O(k)$ time and hence the running time is $O(Mk)$.

```
1. for m = 1 to M
2.   C(m) = min_{\{s_i \in S \}} \{1 + C(m - |s_i|)\}
3. return C(M)
```

Here is a more detailed pseudo-code for the problem that returns the selected cuts as well.

```
def Cut(s; k; M)
# s is the array of the rod cut sizes
# k is the number of allowed cut sizes
```

```
# M is the original rod length
1.  C[0]=0
2.  for m=1 to M
3.   min= \infinity
4.   for i= 1 to k
5.     if s[i] <= m then
6.       if 1 + C[m - s[i]) < min then
7.         min = 1 + C[m - s[i])
8.         cut = i
9.   C[m] = min
10.  S[m] = cut
11. return C and S
```

F. Analyze the running time of your dynamic programming solution.
By the comments above the running time would be $O(Mk)$ if $m = M$ is the length of the rod that we are cutting.

Note that this problem is a modified version of the problem of giving a change for d cents with coins of denominations $d_1, ..., d_k$. For most of the coin systems in the world including the US the greedy algorithm is actually the optimal.

Problem 3

An *ordered stack* $\mathcal{S}$ is a stack where the elements appear in increasing order (largest element on top). It has the following operations:

- INIT($\mathcal{S}$): Create an empty ordered stack.
- POP($\mathcal{S}$): Delete and return the top element from the ordered stack.
- PUSH($\mathcal{S}$, x): Remove all elements from the stack that are larger than x, and then push x on top of the stack.
- DESTROY($\mathcal{S}$): Delete all elements on the ordered stack.

We implement an ordered stack as a double linked list (maintaining a pointer to the top element).

A. What is the worst-case running time of each of the operations INIT, POP, PUSH, and DESTROY?

B. What is the amortized running time of m of these operations? Justify your answer.

Solution to Problem 3

A. INIT($\mathcal{S}$): $O(1)$
POP($\mathcal{S}$): $O(1)$
PUSH($\mathcal{S}$, x): $O(n)$, where n is the number of elements currently in stack.
DESTROY($\mathcal{S}$): $O(n)$, equivalent to popping all n elements from the stack

B. The amortized running time is constant, and considering for any sequence of m operations. This is because any element that is pushed can only be popped once, and creating a new stack takes constant time. More formally, we can say that we pay a cost of 2 for each push operation, one for the pushing, and one for popping, which may occur at any time in the future. This takes care of the cost of all push, pop, and destroy operations, and the init operations are constant, so they don't increase the asymptotic amortized running time.

Problem 4

A. What is the difference between the single-linkage algorithm we saw in class and Kruskle's algorithm?

B. Modify Kruskle's algorithm to a single-linkage algorithm, and give the resulting pseudocode.

C. What is the running time of the above single-linkage algorithm?

D. **Bonus:** Given the minimum spanning tree of a graph G on n nodes (with positive edge weights) as well as an integer $1 < k < n$, how would you efficiently find the single-linkage k-clustering of G? Analyze the running time of your solution.

Solution to Problem 4

A. Kruskle's algorithm returns a MST while single-linkage returns a clustering (a partitioning of the data into k groups.

B.
```
1. Let C be a union-find data structure
2. num_clusters = 0
3. for each vertex v  G.V
4.     Make-Set(v)  // Add set {v} to C
5. sort the edges of G.E into ascending order by cost
6.    for each edge (u, v)  G.E, considered in ascending order:
7.         if Find-Set (u)  Find-Set(v) //vertices not in the same set or "cluster"
8.            Union(u, v)    //union the two clusters
9.            num_clusters++
```

```
10.            if (num_clusters = k)
11.               return C
12. return C
```

C. The running time is the same as Kruskal's algorithm, which is $O(mlogn)$, where m is the number of edges and n is the number of nodes in the graph. Since the graph is typically complete for clustering problems, $m = \binom{n}{2}$, and so the running time is $O(n^2logn)$

D. Cut the minimum spanning tree at the $k-1$ shortest edges. The resulting trees correspond to the desired clusters. The running time is $O(nlogn)$, because it takes $O(nlogn)$ operations to sort the edges of the minimum spanning tree. The rest of the algorithm is linear in the number of nodes.