Algorithms Homework 2

Aladdin Persson (aladdin.persson@hotmail.com)

1 Homework 1

Exercise 1

Suppose you have an external drive with capacity G, G integer. Given further a list of files $f_1 \dots f_n$ that you would like to save on the drive. Each file has a certain size, file f_i takes up g_i gigabytes. However, the total sum of all g_i is larger than G, so your goal is to find a largest subset of the files that fits on the drive with capacity G.

(a) How many different ways are there to choose a subset of the files that may or may not fit on the drive?
(b) Suggest a greedy algorithm that solves the problem. First explain your algorithm briefly in words, then present pseudo-code.
(c) What is the running time of your algorithm in terms of $O()$? Do you think there could be another algorithm with better time complexity in terms of $O()$? (No formal proof needed, just write down your thoughts.)
(d) Prove that your greedy algorithm will give you an optimal solution (this sub-problem is more difficult, you may skip it).
(e) Can you think of a special case of the problem, maybe a special property of the g_i so that you can come up with an algorithm that is faster (in terms of $O()$) than the one you suggested in (b)? If yes, write down the algorithm, give the running time in terms of $O()$ and explain why it is faster.

Solution:
a) For each file we consider storing it or not, and depending on this choice we move on to the next and do the same decision. Assuming we have n files this results in 2^n different ways.

b) I believe the first greedy algorithm that came to mind is sorting the files by gigabyte size in ascending order. Then simply picking the file that has smallest file size and continuining to pick the next one until the hard drive cannot store anything more.

Notation: Let S be the set of pairs (f_i, g_i) in any particular ordering. Let C denote the total capacity of the harddrive.

Algorithm 1 largestSubset(S, C)

```
Sort set S by second element in each pair, g_i in ascending order
Initialize empty set O for files to store in optimal solution
for f_i, g_i in S do
    if C − g_i ≥ 0 then
        Add f_i to set O
        C ← C − g_i
    else
        break loop
return O
```

c) The running time for the algorithm above is dominated by the time to sort which is $\mathcal{O}(nlog(n))$ since to go through all the files in the set S is linear time. My belief is that there doesn't exist a faster algorithm than $nlog(n)$ assuming that this suggested algorithm returns optimal solution. It couldn't be linear because there is a requirement that for each file we have to have some sort of ordering compared to other files, which implies sorting that can fastest be done in $\mathcal{O}(nlog(n))$.

Unsure how to prove this.
I don't think my proof by contradiction is a valid proof. I believe one should be argue for an "exchange argument" but not sure how this proof would look like.

e) As described in question *c)* the running time is dominated by the sorting which if we can assume that the g_i are already sorted then this removes the need to sort. In this special case this would result in a fast $\mathcal{O}(n)$.

Exercise 2: Suppose you're running a lightweight consulting business. Your clients are distributed between the East coast and the West Coast, and this leads to the following question. Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i, you'll incur an operating cost of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.) However, if you run the business out of one city in month i, and then out of the other city in month $i + 1$, then you incur a fixed moving cost of M to switch base offices. Given a sequence of n months, a plan is a sequence of n locations: each one equal to either NY or SF such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The cost of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city. The problem: Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost.

(a) How many different plans exist?
(b) Give an example to show that a greedy algorithm does not correctly solve this problem. Try to choose a small example, i.e., a small n.
(c) Give an efficient algorithm that takes values for n, M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the cost of an optimal plan. We recommend you use pseudo-code.
(d) What is the running time of your algorithm in terms of $O()$?

Solution:
a) In the beginning we can choose either NY or SF and from each of these choices we have the same choice again for a total of n choices. Similarly to the previous question we then have 2^n different plans.

b) Well we can have an algorithm that just looks which is cheapest for only one month forward. Let's pick $n = 2$ and let $M = 1000$, $S_1 = 10$, $N_1 = 5$, $S_2 = 1$, $N_2 = 100$. Now it will pick N_1 since $5 < 10$ as it's first month. The proposed greedy algorithm would then choose $N_2 = 100$ for a total cost of 105. The optimal solution in this case would be to pick S_1, S_2 for an optimal solution of 11. I believe this is just one instance of a greedy algorithm and one can propose many where it might be possible to solve this using a greedy algorithm.

c) I believe if one constructs a graph with a starting node with edges to the costs of N_1, S_1 and then the edges from N_1, S_1 respectively for the costs of N_2, S_2 and adding the cost of M if the edge goes from S to N or vice versa. This results then in a shortest path problem from starting node to the nodes either S_n or N_n and one could use an algorithm like Djikstra's algorithm to solve this which is a greedy algorithm so I don't think it would be necessary to use the dynamic programming approach here, please correct me if I'm wrong.

I'm having some difficulty thinking of an algorithm that works well as a dynamic programming. I feel the difficult part is that one has to keep track of where the current location is to have the cost of M added. The recurrence relationship differs depending on the location:

If current location is N:
OPT(i) = min{OPT($i-1$) + N_i, OPT(i-1) + S_i + M}.

Else if current location is S:
OPT(i) = min {OPT($i-1$) + N_i + M, OPT($i-1$) + +S_i}

This is the fundamental idea. So let's define this in an algorithm.

Algorithm 2 OPT(i, N)

```
if i == 0 then
    return 0
else:
    return min{OPT(i-1,N), OPT(i-1, S) + M} + N_i
```

Algorithm 3 OPT(i, S)

```
if i == 0 then
    return 0
else:
    return min{OPT(i-1, N) + M, OPT(i-1, S)} + S_i
```

Algorithm 4 find_optimum()

```
Define OPT as empty n x 2 matrix
for i = 1 to n do
    OPT[i,1] = OPT(i,N)
    OPT[i,2] = OPT(i,S)
return min{OPT[n,1], OPT[n,2]}
```

d) If one uses Djikstra which I believe should work on this problem since we know all the edges, nodes, and we wish to find the shortest path and also the graph is quite sparse and should then be quite fast on this problem. Djikstras algorithm with a heap implementation has $\mathcal{O}(|E| + |V|log(|V|))$ runtime. The proposed algorithm should be to fill the entire matrix which should be $\mathcal{O}(n * 2) = \mathcal{O}(n)$ operations.