Theory Problem 4. Dynamic Programming

Moyuan Huang
Department of Computer Science
University of California, San Diego
moh008@eng.ucsd.edu

1 Planning a Trip

1.1 Algorithm

1.1.1 Pseudo code

Algorithm 1

Input:
A, a sorted sequence $a_1 < a_2 < ... < a_n$, hotels for stopping by. Here I expanded the array with $a_0 = 0$ to denote the starting point

Output:
ans, the optimal sequence of hotels at which to stop.

```
dp[n+1] ← 0, route[n+1] ← 0
for i from 1 to n do
    c ← INT_MAX, prev ← 0
    for prev from 0 to i-1 do
        if c > (dp[prev] + cost(a[prev], a[i])) then
            c ← (dp[prev] + cost(a[prev], a[i]))
            prev ← a[prev]
        end if
        dp[i] ← c, route[i] ← prev
    end for
end for
i = n + 1
while i ≥ 0 do
    ans.append(a[i])
    i = route[i]
end while
return ans.reverse()
```

Here I assume $reverse()$ is trivial and takes O(n) time.

1.1.2 Description

We first initialize two sets of arrays: dp where $dp[i]$ records the minimum cost when stopping at $a[i]$, and $route$ where $route[i]$ records the previous stop before $a[i]$ under the optimal strategy. Then we iterate from 1 to n to compute the minimum cost when stopping at $a[i]$. At each particular point, we compute what would be the cost if our previous stop is $a[prev]$, for $prev$ increases from 0 to i. The cost equals to, according to the problem description,

$$cost(p_1, p_2) = (p_1 - p_2 - 200)^2$$

Then we set the $prev$ as the one that gives the minimum cost at $a[i]$ and record it in $route[i]$, and set $dp[i]$ as exactly the minimum cost. In this end, we back track to get each optimal stops and return the sequence.

1.2 Correctness

Let's consider when we wanna stop at some hotel a_i. The cost at this particular point depends on its previous stop a_prev, which can be expressed by

$$total\_cost(a_i) = total\_cost(a_{prev}) + cost(a_{prev}, a_i)$$

By iterating from all the possible stop hotel, i.e.0 to $(i-1)$, we are assured to obtain the minimum cost at hotel i. After going through 1 to n, we are guaranteed to obtain the minimum cost at hotel n, i.e.the minimum cost of the whole trip.

1.3 Time Complexity

The loop a line2-11 runs at most n-1 times, which cost O(n) time, line5-10. runs at most n-2 time and takes O(n) time. The while loop in line 13-16 takesO(n) time, and reverse a array takes O(n) time.
In sum, the algorithm takes O(n^2) + 2O(n) = O(n^2) time complexity.

2 Partitioning Souvenirs

2.1 Algorithm

Algorithm 2

Input:
$x[n]$, A sequence of positive integers $x_1, x_2, ..., x_n$.
Output:
$paritionable$, 1 if possible to partition into three subsets with equal sums, and 0 otherwise.
1: $sort(x)$, $s \leftarrow sum(x)/3$,
2: $dp[sum(x)] \leftarrow False$
3: $dp[0] \leftarrow True$
4: **for** i from 0 to n **do**
5:     **for** j from $s - x[i]$ to 0 **do**
6:         **if** $dp[j]$ **then**
7:             $dp[j + x[i]] \leftarrow True$
8:         **end if**
9:     **end for**
10: **end for**
11: **if** !$dp[s]$ **then**
12:     **return** False
13: **end if**
14: $n \leftarrow choose\_a\_combination(x, s, [], n)$
15: repeat line $2 - 10$
16: **return** $dp[n, s]$

In Alg. 2, $dp[i]$ denoted whether we can sum up to i using the current numbers given. Suppose that we have already taken care of the first i elements of $x[]$. Now we take $x[i]$ and look at our table $dp[]$. It is set True in every location that corresponds to a sum that we can make from the first i numbers we have already processed. Now that we add the new number, $x[i]$, consider some location of $dp[j]$ that has True value. Now we can obtain a new value $j + x[i]$, and we immediately set the corresponding value in $dp[]$ to True.
Note one important detail in the j loop. It goes through the table from right to left. We have to do this in order to avoid double counting x[i].
After the first round, we may now check if $dp[s]$ is set to True and return False if it is still

False, because no combinations of numbers can add up to $sum(x)/3$. If it does, we call the choose_a_combination() function which is described in Alg. 3 to deleted from $x[]$ a valid combination of numbers that sum to s. Then we run the algorithm again to check if there is another group of numbers that sum up to s. If it does, this means that the $x[]$ can be split up to 3 equal groups, that we should return True. Otherwise, return False.

Algorithm 3 $choose\_a\_combination(x, target, ans, idx, found)$

Input:
x, A sequence to select from.
$taget$, the required sum value.
ans, the selected elements.
idx, the current index of selection.
$found$, if a valid combination have been found.
Output:
n, the length of x after deleting the selected element.
$found$, if a valid combination have been found.
1: **if** $found$ **then**
2:     **return** $x.length(), found$
3: **end if**
4: **if** $target == 0$ **then**
5:     **for** auto elem in ans **do**
6:         delete(x, elem)
7:     **end for**
8:     **return** $x.length(), True$
9: **end if**
10: **for** i from idx - 1 to 0 **do**
11:     **if** $x[i] > target$ **then**
12:         ans.append(x[i])
13:         $n, found \leftarrow choose\_a\_combination(x, target - x[i], ans, i, found)$
14:         ans.pop(x[i])
15:     **end if**
16: **end for**
17: **return** $x.length()$

The process of finding a combination of numbers that sum up to $target$ is described in Alg. 3. It is a recursive depth first search based algorithm: We keep trying putting elements in the combination until we finally find a valid one. Then we delete those element from the original set, i.e.$x[]$ since we can't reuse any of the elements. Finally, the global flag $found$ is set true and are returned together with the new length of $x[]$ after deletion.

2.2 Correctness

The logic of Alg. 2 has been illustrated right after the algorithm. One thing I try to make clear here is that, since the Alg. 3 may return any valid combination of numbers, we need to prove that any combination deleted by the Alg. 3 would result in an correct ans.
Note that the function would only delete a group of numbers that sum up to $target$, it doesn't matter which group is deleted, since if $x[]$ can be split into 3 equally summed groups, it makes no different that which group is selected first.

2.3 Time Complexity

Line 1 : O($n \log n$) + O(n)
Line 2 : O(n)
Loop 4-10: O(n)
Loop 5-8: O(X) ($X = s$ in our context)
Line 15: same as Line4-10

Line 14 In funtion choose_a_combination():
line 4-7: O(n)
line 10-16: T(n) = T(n-1) + O(n) = O(n^2)

In total:
O($n \log n$) + 2O(n) + 2O(nX) + O(n^2) = O(nX) + O(n^2)