CSOR W4246—Fall 2018

Brief solutions to hw2t

1. This algorithm is a modified BFS, starting at node s: for each node u, we *also* store the number of shortest $s-u$ paths in $u.num$. Return $t.num$. Running time: $O(n+m)$

Algorithm 1

`BFS`(G, s)

```
1: ...
2: while Q ≠ ∅ do
3:     u = dequeue(Q)
4:     for each (u, v) ∈ E do
5:         if discovered[v] = 0 then
6:             ...
7:             v.num = u.num
8:         else if dist[v] = dist[u] + 1 then
9:             v.num = v.num + u.num
10:        end if
11:    end for
12: end while
13: return t.num
```

2. Slightly modify Dijkstra's algorithm to also maintain the number of edges $best[u]$ on current shortest s-u path, for every node u. Initialize $best[s] = 0$, $best[v] = \infty$ for $v \neq s$.

Algorithm 2

`Update`(u, v)

```
1: if dist[v] > dist[u] + w(u, v) OR (dist[v] = dist[u] + w(u, v) AND (best[v] > best[u] + 1)) then
2:     dist[v] = dist[u] + w(u, v)
3:     prev[v] = u
4:     best[v] = best[u] + 1
5: end if
```

Correctness and running time follow from Dijkstra's algorithm.

3. Let $OPT(i)$ be the minimum total penalty to get to hotel i. Then

$$OPT(i) = \min_{0 \le j \le i-1} \left\{ OPT(j) + (200 - (a_i - a_j))^2 \right\}$$

We want $OPT(n)$. We can compute this by filling in a DP array with $n+1$ entries in a bottom-up fashion, starting with $OPT(0) = 0$. Running time: $O(n^2)$.

4. A greedy algorithm that sorts the customers by increasing order of service times and services them in this order is correct.

   Running time: $O(n \log n)$.

   Correctness: by an exchange argument; for any ordering of the customers, let c_j denote the j-th customer in the ordering. Then the total time is given by

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{c_j} = \sum_{i=1}^{n} (n-i) t_{c_i}$$

   One can observe that if $t_{c_i} > t_{c_j}$ for $i < j$, then swapping the positions of the two customers gives a better ordering. Then the ordering $t_{c_1} \leq t_{c_2} \leq \ldots \leq t_{c_n}$ provided by the greedy algorithm above is optimal.

5. Maintain a matrix $M[0...n][0...k+1]$ where $M[i][j]$ is the temporary imbalance for the first i elements that already tokened inot j parts. Then we can get

$$M[i][1] = |Sum[1][i] - \frac{Sum[1][n]}{k+1}|$$
$$M[i][j] = \min_{l \in [1,i-1]} \max\{M[l][j-1], |Sum[l+1][i] - \frac{Sum[1][n]}{k+1}|\}$$

   Then $M[n][k+1]$ would be the minimal imbalance. Still, need to record where to cut off to backtrack the partition. Time complexity is $O(n^2 k)$.

Solutions to recommended exercises

1. Solution to exercise 1

   (a) For this part, greedy algorithm is enough. Take as much as quarters as possible, then dimes, nickels, and rest pennies. Actually this could take only $O(1)$ by using a division:

   $$Q = n/25$$
   $$D = (n - 25 \times Q)/10$$
   $$N = (n - 25 \times Q - 10 \times D)/5$$
   $$P = n - 25 \times Q - 10 \times D - 5 \times N$$

   To Prove that this algorithm is correct, consider n in 4 cases:

   i. $n \geq 25$: We are taking as much quarters as we can. If the optimal solution takes less quarters, then it must contain some other coins with total value at least equal to 25. If there are 3 dimes, we can replace them with one quarter and one nickel; if there are at least 2 dimes and 1 nickel, we can replace them with one quarter; if there are some nickels and pennies, we can replace some of them with one quarter. In any case the replacement will bring less coins.

   Please be careful enough to notice that when n is greater than 25, there are still so many cases. You can't say that there should definitely be 2 dimes and 1 nickel.

   ii. n in $[10, 25)$: We are now taking as much dimes as we can. If the optimal takes less dimes, then it must contain some other coins with total value at least equal to 10. Replacing that combination will always bring less coins.

   iii. n in $[5, 10)$: We are now taking as much nickels as we can. If the optimal takes less nickels, then it must contain some other coins with total value at least equal to 5. Replacing that combination will always bring less coins.

   iv. n in $[0, 4)$: The only solution is using pennies.

   (b) Similar to (a), take as many c^k coins as possible, and then $c^k - 1$, and so on.
   To prove that this greedy algorithm is right, simply prove that except c^k coins, there can't be more than $(c^k - 1)$ coins of a kind (otherwise they would be a higher coin). Then prove that $(c-1)c^{k-1} + (c-1)c^{k-2} + ... + (c-1) < c^k$, which means there should be as many c^k coins as possible.

   (c) Any reasonable counterexample is acceptable. For example, with the coins set $\{1, 3, 4\}$, and the target value 6, greedy comes up with $\{4, 1, 1\}$, but $\{3, 3\}$ is a better choice.

(d) Use dynamic programming.

Let Cnt be a new array $[0...n]$

Let $Taken$ be a new array $[0...n]$

$Cnt[0] = 0$

For $i = 1$ To n Do

$Cnt[i] = MAX$

For $j = 1$ To k Do

If $Cnt[i] \geq Cnt[i - Coin[j]] + 1Do$

If $Cnt[i] = Cnt[i - Coin[j]]$

$Taken[i] = j$

Return $Cnt[n]$

Method takes $O(nk)$ time. Use Taken to backtrack the coin we need.

2. Solution to exercise 2.

Use a greedy algorithm for this problem. Define the position of any point on the road to be its distance from the western end of the road (which is at distance 0). Place the first base station at the largest position s_1 so that all houses between 0 and s_1 are covered by s_1. Remove all houses covered by s_1 and repeat on the remaining houses.

Running time: $O(n)$, where n is the number of houses (assuming positions of houses are given sorted).

3. Solutions to exercise 3

Use dynamic programming. Use array M to record the length of the sequence and S for backtracking. For every i from range 1 to n, find j from from range 1 to $i - 1$ such that $A[i] >= A[j]$ with maximum $M[j]$. Let $M[i] = M[j] + 1$ and $S[i] = j$. Find the maximum $M[i]$ and backtrack the sequence. As there are n elements in A and each take $O(n)$ time to compute, it totally costs $O(n^2)$.

4. Solution to exercise 4

   Let $OPT(j) =$ max sum of subarray **ending at** j. Then

   $$OPT(n) = \max\Big\{OPT(n-1) + A[n], 0\Big\}$$

   and

   $$OPT(j) = \max\Big\{OPT(j-1) + A[j], 0\Big\}.$$

   Boundary condition: $OPT(0) = 0$. Then the maximum length L of a contiguous subarray of A is given by

   $$L = \max_{1 \le j \le n} OPT(j)$$

   There are $n+1$ subproblems, each requiring $O(1)$ time in a bottom-up computation. Thus computing L requires $O(n)$ time.

5. Solution to exercise 5

   Let $OPT(i,j) =$ longest common substring of x, y **terminating at** x_i, y_j. Then

   $$OPT(i,j) = \begin{cases} 1 + OPT(i-1, j-1) & \text{, if } x_i = y_j \\ 0, \text{o}therwise \end{cases}$$

   Boundary conditions: $OPT(m,0) = 0$, $OPT(0,n) = 0$, $OPT(0,0) = 0$.

   We can construct an $(m+1) \times (n+1)$ DP table to compute each $OPT(i,j)$. The longest common substring of x, y is given by the max entry in the table.

   Running time: $O(mn)$