Illinois Institute of Technology
Department of Computer Science

Solutions to Homework Assignment 4

CS 430 Introduction to Algorithms
Spring Semester, 2017

Solution:

1. (a) The greedy algorithm: each time, we pick the coin with the largest denomination that is smaller or equal to the change. The remaining amount of change is reduced by the value of the coin. We do the previous two steps repeatedly until the remaining amount of change is zero.

   For a coin system consisting of quarters, dimes, nickels and pennies, this algorithm indicates that the number of quarters $a_q = \frac{n}{25}$, the number of dimes $a_d = \frac{n\%25}{10}$, the number of nickels $a_n = \frac{n\%10}{5}$, the number of pennies $a_p = n\%5$.

   Proof of Correctness: To prove the correctness of the greedy algorithm for the coin system with consisting of quarters, dimes, nickels and pennies. We need the support of the following lemma:

   Lemma 1 *In the optimal solution, we can have at most two dimes, one nickel and four cents; we also cannot have two dimes and one nickel at the same time.*

   The correctness of this lemma is easy to prove. We can always replace three dimes with a quarter and a nickel, replace two nickels with one dime, replace five cents with one nickel, and replace two dimes and one nickel with a quarter. All these replacements will result in smaller number of coins. This lemma indicates that in the optimal solution, the total value dimes, nickels and pennies is no greater than 24 cents; the total value of nickels and cents is no greater than 9 cents; the total value of pennies is no greater than 4 cents.

   Then we can prove the correctness of our algorithm for this coin system. Assume that there is an optimal solution OPT that use smaller number of coins to make the change. Suppose OPT uses $a_q^{'}$ quarters, $a_d^{'}$ dimes, $a_n^{'}$ nickels and $a_p^{'}$ pennies. Since our greedy algorithm use as many quarters as possible, $a_q^{'} \leq a_q$. If $a_q^{'} < a_q$, the total value of dimes, nickels and cents in OPT will be greater than 24 cents since both our greedy algorithm and OPT make change for the same amount. This goes against our lemma. Then $a_q^{'} = a_q$. Following the same procedure, we can prove $a_d^{'} = a_d$, $a_n^{'} = a_n$ and $a_p^{'} = a_p$. This means OPT will use the same number of coins as our algorithm, which goes against the assumption that OPT uses smaller number of coins.

   (b) Assume that there is an optimal solution OPT that use smaller number of coins to make the change. Suppose the greedy algorithm uses a_0 coins with denomination c^0, a_1 coins with denomination c^1, ..., a_k coins with denomination c^k. Suppose the corresponding number in OPT are $a_0^{'}$, $a_1^{'}$, ..., $a_k^{'}$ respectively. Since our greedy algorithm use as many coin with largest denomination as possible, $a_k^{'} \leq a_k$. If $a_k^{'} < a_k$, the total value of the rest of the coins in OPT will be larger than c^k. In this coin system, this means that there are some coins with total value equal to c^k in the remaining coins. Then we can always replace these coins with one coin with value c^k. Thus $a_k^{'} = a_k$. Following the same procedure, we can prove $a_i = a_i^{'}$ for $i = k-1, k-2, ..., 0$. This means OPT will use the same number of coins as our greedy algorithm, which goes against the assumption.

   (c) Suppose the coin denominations are $\{1, 7, 10\}$. If we want to make a change of 15 cents. The solution given by the greedy algorithm is one coin of 10 and five coins of 1. The optimal solution uses two coins of 7 and one coin of 1.

(d) Recursive form: let $f(n)$ denote the minimum number of coins we use to make change for n coins.Let d_i denote the i-th denomination. Then it will have the following form of recursion:

$$f(n) = \begin{cases} \infty, & n < 0 \\ 0, & n = 0 \\ \min\limits_{1 \le i \le k} f(n - d_i) + 1, & n > 0 \end{cases}$$

Algorithm 1 Make Change: Dynamic Programming

```
Let $M$ be an array of size $n+1$
Let $D$ be an array of size $n+1$
$M[0] \leftarrow 0$
for $i \leftarrow 1$ to n do
    $M[i] \leftarrow \infty$
    for $j \leftarrow 1$ to k do
        if $1 + M[i - d_j] < M[i]$ then
            $M[i] \leftarrow 1 + M[i - d_j]$
            $D[i] = d_j$
        end if
    end for
end for
```

The iterative form will be represented in algorithm 1. The array M stores the minimum number of coins to make change for n cents. D stores the coins used. This algorithm runs in $O(nk)$ time. There are n steps in the outer loop and each inner loops contains k steps.

(e) Let A be the set of denominations. Let $g(n, A)$ be the function that returns 1 if we can make change for n using denominations in A, and returns 0 otherwise. The recursive formation is

$$g(n, A) = \begin{cases} 1, & n = 0 \\ 0, & n < 0 \vee (n \neq 0 \wedge A = \Phi) \\ \bigvee\limits_{1 \le i \le k} g(n - d_i, A - d_i), & \text{otherwise} \end{cases}$$

2. (a) [1] *Recurrence and algorithm*:

From (i, j), the robot can only move to $(i, j+1)$ or $(i+1, j)$. After making a movement, the robot is faced with a sub-problem with a slightly smaller board where the starting point is either $(i, j+1)$ or $(i+1, j)$ respectively. This shows a recurrence between the problem and subproblems. If we define that $C(i, j)$ is the maximum values a robot can collect when it starts from (i, j), we can construct the following recurrence:

$$C(i, j) = \begin{cases} \max\big(C(i, j+1), C(i+1, j)\big) + c_{ij}, & 1 \le i < n \text{ and } 1 \le j < m \\ C(i, j+1) + c_{ij}, & i = n \\ C(i+1, j) + c_{ij}, & j = m \\ c_{ij}, & i = n \text{ and } j = m \end{cases}$$

[1] Let us assume the robot will start at (1,1).

Then, the dynamic programming algorithm **without memoization** will use the recurrence to find out the optimum path which yields the maximum value. The algorithm (which will be eventually similar to Algorithm 1) is omitted since everything except the recurrence is details in the programming.

Time complexity:

Without memoization, we are essentially doing the brute-force search, and we need to explore all different paths to find out the optimum one among them. That is, the time complexity will be proportional to the number of the possible paths that can be taken by the robot. Now, this becomes a counting problem – the robot will make $n+m-2$ movements, $n-1$ of them will be the down-ward movements, and $m-1$ of them will be the right-ward movements. That implies there will be $\binom{n+m-2}{m-1}$ (or $\binom{n+m-2}{n-1}$, which is equal) different paths that can be taken. In conclusion, the time complexity will be $\Theta\left(\binom{n+m-2}{n-1}\right)$.

(b) If we are allowed to use the memoization, we can first construct a $n \times m$ memoization table to store the results of the subproblems. Then, to iteratively solve the problem, we can start filling the table from the cell (n, m), and reverse-engineer the above recurrence to fill up the table by iteratively filling up the cells that are adjacent to already-filled cells in the table. It takes $O(1)$ time to fill up any cell, therefore the final complexity of this iterative algorithm will be proportional to the number of cells in the table, which is $\Theta(nm)$.

3. Let us use $J_k(s_k, f_k, w_k)$ to denote the k-th job with the start time s_k, finish time f_k, and weight w_k.

(a) We can simply give a counter example to prove that the EDF (Earliest Deadline First) scheduling **does not always work**[2].
Scheduling between two jobs $J_1(1,3,1)$ and $J_2(2,4,100)$ will fail if we simply use EDF scheduling. The EDF scheduling suggests schedule J_1 with the total weight 1, but the optimum solution will schedule J_2 with the total weight 100.

(b) *Recurrence and algorithm*:

Let us say we will consider whether to include the jobs in our solution or not starting from the first job J_1, and we will linearly consider all remaining jobs (*i.e.,* $J_2, J_3, \cdots, J_n$).
Whenever we consider whether J_k should be included or not, we have to think about two cases:

- Include J_k: In this case, because we included J_k, any job among $J_{k+1}, J_{k+2}, \cdots$ overlapping with J_k should be immediately ruled out from our consideration. Then, we can find out the first non-overlapping job in the list $J_{k+1}, J_{k+2}, \cdots$ and consider whether that should be included, and this is the sub-problem.
- Do not include J_k: In this case, we will simply skip it and consider whether J_{k+1} should be included, and this is the sub-problem.

If we define that $W(k)$ is the maximum weight we can achieve when we schedule a list of jobs $J_k, J_{k+1}, \cdots, J_n$, we can construct the following recurrence from the above analysis.

$$W(k) = \begin{cases} \max\big(W(k+1), W(x) + w_k\big), & k \le n,\ J_x \text{ is the first non-overlapping job} \\ \max\big(W(k+1), w_k\big), & k \le n,\ \text{non-overlapping job does not exist} \\ -\infty, & k > n \end{cases}$$

[2] It means the EDF scheduling may work sometimes, but it may fail sometimes. That is why simply giving one counter example is sufficient.

Then, for the memoization purpose, we can construct a 1-dimensional table (*i.e.,* an array) to store all values of $W(k)$. To record which jobs should be included, we can construct another 1-dimensional table to store intermediate results whenever a new job is included at any stage by choosing $W(x) + w_k$ instead of $W(k+1)$ (or w_k instead of $W(k+1)$) in the max functions.

The dynamic programming algorithm using the above recurrence and the memoization table is somewhat standard (either recursively calculate $W(1)$ or iteratively fill up the memoization table starting from the n-th cell in the table).

Time complexity:

The time complexity is slightly complicated. To evaluate $\max\big(W(k+1), W(x) + w_k\big)$, one needs to find out the next non-overlapping job in the list $J_{k+1}, J_{k+2}, \cdots, J_n$, whose complexity will be $O(n-k)$. That means, the complexity of evaluating $W(1)$ will be $O(n-1)$, $W(2)$ will be $O(n-2)$, *etc.*

Therefore, the upper bound of the given algorithm's complexity is

$$O\Big(n-1+n-2+n-3+n-4+\cdots+2+1+0\Big) = O(n^2)$$