# Solutions to Problem Set 3

Rashmi Dwaraka

October 10, 2016

## Contents

# 1 Stacking tiles

**Problem statement:**
Consider a set of n rectangular tiles where the $i_{th}$ tile is $l_i$ units long and $w_i$ units wide, $l_i \geq w_i \geq 1$. Assume that tile i can be stacked on top of tile j iff $l_i \leq l_j$ and $w_i \leq w_j$ . Give a polynomial-time algorithm for computing the maximum number of tiles that can be stacked together. Justify the correctness of your algorithm and analyze its asymptotic worst-case running time.
**Solution:**

- **Recurrence for optimal cost and Base case**
  Consider there are n tiles sorted in descending order of the area of the tiles to ensure better memoization. Using Dynamic Programming technique, the optimal solution can be written in the form of following recurrence:

$$OPT - STACKING(n) = \begin{cases} max(OPT - STACKING(j_{1..(n-1)}) + 1), & \textbf{if } n > 1 \\ 1, & \textbf{if n==1}. \\ 0, & \textbf{if n==0}. \end{cases}$$

  Given n tiles sorted in the descending order in terms of their area, we can find the optimal solution starting from the first tile. If the first tile which has the maximum area is part of the solution, then we find all the other tiles that can be stacked above it. If the first tile is not stackable, we find the next tile that can form the base of our stacked tiles. In this manner, we can break the problem into sub-problems and solve the recursion using dynamic programming technique.

- **Proof of correctness**
  Consider $t_i$ is part of the optimal solution $S^*$. Find all the tiles $t_{j:(2\rightarrow n)}$ that are stackable along with tile $t_i$. By contradiction, if there was any other optimal solution S' where $t_i$ is not part of the optimal solution, then we can consider S' as our optimal solution such that $|S'| \geq |S^*|$. Therefore the problem can be divided into subproblems and hence can be solved by dynamic programming technique.

- **Pseudo-code**
  In this algorithm, we store the number of tiles that can be stacked for each tile in an array max-optimal. The max count is returned as the optimal cost.

---

**Algorithm: OPT-STACK-COST($n$)**

---

```
1: procedure: OPT-STACK-COST(n)
2:    if n==0, return 0        #base case
3:    if n==1, return 1        #base case
4:    sort the tiles in descending order of (l · w)        #Sort requires O(n · logn)
5:    for i ← 1 to n
6:       max-optimal[i] = 1        #initialize the count for the tile itself
7:       for j ← 1 to (i − 1)
8:          if(isStackable(T[i],T[j])
9:             max-optimal[i] = max(max-optimal[j]+1,max-optimal[i])
11:   return max(max-optimal)        #returns maximum stackable tiles count
```

---

**Algorithm: isStackable(T[i],T[j])**

```
1: procedure: isStackable(T[i],T[j])
2:     if (l_{T[i]} ≤ l_{T[j]} and w_{T[i]} ≤ w_{T[j]})
3:         return True
4:     else return False
```

- **Optimal solution**
  We can keep track of the tile index and the cost can be calculated based on the length of the stacked array. The max-optimal[] hashmap stores the list of indexes.

**Algorithm: OPT-STACK-SOLN($T[1..n]$)**

```
1: procedure: OPT-STACK-SOLN(T[1..n])
2:     if n==0, return null        #base case
3:     if n==1, return T[1]        #base case
4:     sort the tiles in descending order of (l·w)
5:     for i ← 1 to n
6:         max-optimal[i] = [i]
7:         for j ← 1 to (i − 1)
8:             if(isStackable(T[i],T[j])
9:                 if (length(max-optimal[j])>length(max-optimal[i]))
10:                    Append max-optimal[i] with j
11:    return the list with max length from (max-optimal)
```

- **Analysis of worst-case running time**
  The Algorithm looks for all possible cases, and isStackable runs in a constant time operation. The sorting time of the tiles requires $O(n \cdot \log n)$. The worst-case runtime for OPT-STACK-SOLN, due to the loops from line 5 to 10 requires $O(n * (n-1))$. The worst-case running time is $O(nlogn + n^2)$ which is approximately $O(n^2)$.

## 2 Ordering disks

**Problem statement:**
Suppose you are the parts manager of a computer assembly firm. Your firm wants to buy n disks, one for each of the n computers the firm plans to assemble this year. There are m disk suppliers, each of which produces disks of the quality desired by your firm. For any supplier i and any integer j, you are given the cost of buying j disks from supplier i. (More formally, supplier i provides an array $A_i[1..n]$, where $A_i[j]$ is the cost of purchasing j disks from supplier i.)
Design an efficient algorithm to determine how many disks you would purchase from each supplier such that the total cost of purchasing the n disks is minimized. Justify the correctness of your algorithm and analyze its asymptotic worst-case running time.
**Collaborators: Nakul Camasamudram**
**Solution:**

- **Recurrence for optimal cost and Base Case**
  Given m suppliers of disks denoted by $A_i[1..n]$ and let n be the number of disks need to be purchased. We can calculate the minimised total cost of purchasing the n disks can be calculated with below recurrence.

$$OPT\_SOLN(n) = \begin{cases} 0, & \textbf{if n==0 or m==0} \\ min_{k \leftarrow (1\ to\ m)}(A_k[1]), & \textbf{if n==1 and } m > 0 \\ min_{k \leftarrow (1\ to\ m)}(CALC - COST(k, n)), & \textbf{otherwise} \end{cases}$$

$$CALC{-}COST(i, j) = \begin{cases} min_{l \leftarrow (0\ to\ (j-1))}(A_i[j-l] + min_{q \leftarrow ((i+1)\ to\ m)}(CALC - COST(q, l)) \\ 0, \quad \textbf{if } i > m \textbf{ or } j > n \textbf{ or } i < 1 \textbf{ or } j < 1 \end{cases}$$

  The recurrence above calculates for all possible combination of suppliers and quantity of disks, returns the minimum cost. For $i_{th}$ supplier, we calculate the combination of buying all n disks from the same supplier, or buying $(n - j)$ disks from $i_{th}$ supplier and the rest from the combination from other suppliers. This way, we can divide the problem into subproblems and find the minimum cost solution.

- **Proof of correctness: Optimal Substructure Property**
  Let's assume, that we already have an optimal solution S[n, all suppliers], and according to it we buy 'j' disks from supplier 'i', so in this case we need S[n-j, all suppliers - i] to be optimal. That makes subproblems independent from each other. If 'j' disks from supplier 'i' is not part of the optimal solution, then by contradiction, there might some optimal solution S', where S'[n, all suppliers] < S[n, all suppliers]

- **Pseudo-code**
  Let supplier i provides an array $A_i[1..n]$, where $A_i[j]$ is the cost of purchasing j disks from supplier i. The cost of the combination of the supplier and the number of disks is stored in C[i][j] array, so that we need not repeat the same calculation. If the requirement is to buy only 1 disk, then we can find the minimum of $A_i[1]$ array and this form our base case.
  Let us initialize the cost array c[i][j] to null initially which represents the cost of buying j disks from $i_{th}$ supplier.

**Algorithm: OPT-COST-N-DISK(n)**

```
1: procedure: OPT-COST-N-DISK(n)
2:    if (n == 0), return 0        # Base case
3:    if (n == 1), return min(A[1] for all suppliers)       # Base case
4:    min = ∞        # initialize min to a very high value
5:    for k ← 1 to m
6:       cost = CALC-COST(k, n)
7:       C[k][n] = cost        #stores the value for re-use
8:       if cost < min,
9:          min = cost
10:   return min        # returns the minimum cost based on all combinations
```

**Algorithm: CALC-COST(i,j)**

```
1: procedure: CALC-COST(i,j)
2:    if C[i][j] != null
3:       return C[i][j]       #If calculated earlier returns the value
4:    if (j > n)or(j < 1)or(i > m)or(i < 1), return 0
5:    min = ∞       # initialize min to a very high value
6:    for l ← 0 to (j − 1)
7:       sub-min = ∞       # initialize min to a very high value
8:       for q ← (i+1) to m
9:          cost = CALC-COST(q,l)
10:          C[q][l] = cost      #stores the value for re-use
11:          if cost < sub-min,
12:             sub-min = cost
13:      value = A_i[j − l] + sub-min
14:      if value < min
15:          min = value
16:          C[i][j] = min      #stores the value for re-use
17:   return min
```

- **Optimal Solution**
  By keeping track of the list of tuple (supplier,number of disks), while calulating the cost, we can manipulate the above algorithm to provide optimal solution. Here, the result stores the tuple with minimum cost.

**Algorithm: OPT-SOLN-N-DISK(n)**

```
1: procedure: OPT-SOLN-N-DISK(n)
2:    if (n == 0), return null
3:    if (n == 1), return {(supplier(min(A[1] for all suppliers)),1)}
4:    min = ∞       # initialize min to a very high value
5:    for k ← 1 to m
6:       cost = CALC-COST(k, n)
7:       C[k][n] = cost      #stores the value for re-use
8:       if cost < min,
9:          min = cost
10:         result = CALC-SOLN(k, n)
11:   return result    # returns the minimum cost (supplier,quantity) list
```

**Algorithm: CALC-SOLN(i,j)**

```
1: procedure: CALC-SOLN(i,j)
2:    if C[i][j] != null
3:       return C[i][j]
```

```
4:   if (j > n)or(j < 1)or(i > m)or(i < 1) return []
5:   min = ∞        # initialize min to a very high value
6:   for l ← 0 to (j − 1)
7:     sub-min = ∞        # initialize min to a very high value
8:     for q ← (i + 1) to m
9:       cost = CALC-COST(q, l)
10:        C[q][l] = cost       #stores the value for re-use
11:        if cost < sub-min,
12:          sub-min = cost
13:          result += [(q, l)]
14:    value = A_i[j − l] + sub-min
15:    if value < min,
16:        min = value
17:        C[i][j] = min       #stores the value for re-use
18:        result+ = [(i, (j − l))]
19:  return result
```

---

- **Analysis of worst-case running time**
  1. The OPT-COST-N-DISK runs for all the suppliers and hence the iteration takes $O(m)$ based on the lines 5 to 9
  2. The CALC-COST runs for all combination of suppliers and n value, and hence the iteration takes $O(nm)$ based on the lines 6 to 12

  From the above calculation we can say that the worst case running time of the algorithm is $O(n \cdot m^2)$

# 3 Weighted activity selection

**Problem statement:**
Consider a version of the activity selection problem, in which each activity has a weight, in addition to the start and finish times. (For example, the weight may signify the importance of the activity.) The goal is to select a maximum-weight set of mutually compatible activities, where the weight of a set of activities is the sum of the weights of the activities in the set.

(a) Give a counterexample to show that the greedy choice made for the activity selection problem will not work for the weighted activity selection problem.

**Solution:**
Consider an example with 4 activities with start and finish times as A = [(1,5),(5,10),(1,10)]. Let the weights of the activities assigned be W = [300,300,1000] respectively.

Using a greedy approach, where the selection of the first activity depends on the activity that finishes first, we can formulate the solution as $[A_1, A_2]$, whose total weight sums upto 600. Whereas, the single activity $A_3$ has weight of 1000. Even though the total number of non-overlapping activity returned is optimal in greedy approach, it will not work for weighted activity selection problem.

(b) Use dynamic programming to solve the weighted activity selection problem. Justify the correctness of your algorithm and analyze its asymptotic worst-case running time.

**Solution:**
Let us sort the list of activities based on the finish time of each activity. Let us consider $A_i$ is the very last activity in the optimal solution S. By this we can say that S includes all the activities that donot overlap with $A_i$. If $A_i$ is not part of the optimal solution, then we can consider some better solution S'.

Let the number of activities be $a_1, a_2, .., a_n$ in the ascending order thier finishing times. Let $w_1, w_2, .., w_n$ be thier corresponding wieghts. Let OPT(k) be the maximum weight of activities that can scheduled using first k activities.

- **Recurrence for optimal cost and Base Case**

$$OPT(k) = \begin{cases} 0, & \textbf{if k==0}. \\ w_1, & \textbf{if k==1}. \\ max(OPT(k-1), OPT(P(k)+w_k)), & \textbf{otherwise} \end{cases}$$

  Where, P(k) denotes the predecessor of activity $a_k$ where all activities of P(k) ends before $a_k$ starts. If there is no such activity then it computes to 0.

$$p(k) = \textit{largest index } i < k \textit{ such that } A_i \textit{ donot overlap with } A_k.$$

- **Proof of correctness: Optimal Substructure Property**
  Let us consider $A_i$ is the very last activity in the optimal solution S. By this we can say that S includes all the activities from 1 to i that donot overlap with $A_i$. The optimal solution S can be formulated as S = $W_i$ + S[P[i]].
  If $A_i$ is not part of the optimal solution, then the weighted actvity selection problem is set of non-overlapping activities from activities 1 to i-1.
- **Pseudo-code**

  The Algorithm starts with Sorting the Activities in terms the finish time and calculating the Predecessor of each avtivity. Let OPT-A[1..n] hold the optimal value for each activity

**Algorithm: findOptimalCost($n$)**

```
1: procedure: findOptimalCost(n)
2:    Sort the activities A in order of its finish time
3:    P = findPredecessor(A[1..n]) using Binary Search
#          where, P(j) = largest index (i < j) such that A_i donot overlap with A_j
4:    Initialize OPT-A[1..n] = null
5:    OPT-A[0] = 0        #base case
6:    OPT-A[1] = W_1      #base case
7:    for i ← 2 to n
8:       OPT-A[i] = max(W_i + OPT-A[P[i]], OPT-A[i − 1])
9:    return OPT-A[n]
```

– **Optimal solution**
After computing the optimal cost, based on the result stored, we can iterate through the cost and create the activity hashmap. A[i] will have the optimal activities set for activity 1 to i We can make the below changes to findOptimalCost algorithm .

**Algorithm: findOptimumSol(n)**

```
1: procedure: findOptimumSol(n)
2:    if n==0, A[0] = null      #base case
3:    if n==1, A[1] = {1}       #base case
4:    for i ← 2 to n
5:       if (W_i + OPT-A[P[i]]) > OPT-A[(i − 1)])
6:             A[i] = {i} union A[P[i]]
7:       else A[i] = A[i-1]
8:    return A[n]
```

– **Analysis of worst-case running time**
1. Ordering the activities in terms of finish time requires $(n \cdot \log n)$
2. Finding the Predecessor uses Binary search and time taken will be $(n \cdot \log n)$
3. Each recursive call of findOptimum runs in $O(1)$ time

Therefore overall running time can be calculated as $2 \cdot O(n \cdot \log n) + O(n)$ which can be approximated to $O(n \cdot \log n)$.

# 4 Making change

**Problem statement:**
You are given unlimited quantities of coins of each of the denominations $d_1, d_2, ..., d_m$ (all positive integers) and a positive integer n. You are asked to find the smallest number of coins that add up to n, or indicate that the problem has no solution. Design a dynamic programming algorithm to solve this problem. Justify its correctness and analyze its asymptotic worst-case running time.

**Solution:**

- **Recurrence for optimal cost and Base Case**
Let C(n) be the minimum number of coins of denominations $d_1, d_2, ..., d_m$ needed to add up to the sum n. In the optimal solution to making change for n, there must exist some first coin $d_i$, where $d_i \leq n$. Then, the remaining coins in the optimal solution must themselves be the optimal solution to sum up to $(n - d_i)$.

  Based on this we can write our base case and recursion as below:

$$C(n) = \begin{cases} 0, & \textbf{if n==0}. \\ min_{i \leftarrow d_i \leq n}(1 + C(n - d_i)), & \textbf{otherwise} \end{cases}$$

- **Proof of correctness: Optimal Substructure Property**
  Consider, the optimal solution in making change for a positive integer n be S. Let S include some first coin $d_i$ as part of the solution. Then, there exists some solution for (n - $d_i$). By Contradiction, suppose there was a better solution in making change for a positive integer n, then we replace that as our optimal solution. This contrdicts our earlier assumption.

- **Pseudo-code**
  Let D be an array of denominations $d_1, d_2, ..., d_m$ and n be the positive integer.

**Algorithm: make-change-cost**$(D, n)$

```
1: procedure: make-change-cost(D, n)
2:    C[0] = 0        #base case
3:    for i ← 1 to n
4:       min = ∞
5:       for k ← 1 to m
6:          if D[k] ≤ n
7:             if (1 + C[n − D[k]]) < min
8:                min = (1 + C[n − D[k]])
9:       C[i] = min
10:    return C
```

- **Optimal solution**
  In order find the optimal solution, we can keep track of the coins while calculating the optimum cost and the solution of denominations are printed in the order.

**Algorithm: make-change-soln**$(D, n)$

```
1: procedure: make-change-soln(D, n)
2:    C[0] = 0, S[0] = [ ]        #base case
3:    for i ← 1 to n
4:       min = ∞
5:       for k ← 1 to m
6:          if D[k] ≤ n
7:             if (1 + C[n − D[k]]) < min
8:                min = (1 + C[n − D[k]])
9:                d = k
10:      C[i] = min
11:      S[i] = d
12:   while (n > 0)
13:      Print S[n]
14:      n ← (n − D[S[n]])
```

- **Analysis of worst-case running time**
  1. The Loop from line 3 to 11 runs for $O(n)$ iterations.
  2. The inner loop from line 5 to 9 runs for $O(m)$ iterations.
  By the above calculation, we can say that the algorithm runs for $O(nm)$ times.