CS170–Fall 2014 — Solutions to Homework 8

Quoc Thai Nguyen Truong, SID 24547327, `cs170-ig`

November 9, 2014

Collaborators: Shiv Sundram, Hriday Kemburu, Michael Ross.

1. Subsequence

Main idea.
It's obviously that if size of A is zero, it's going to return True, and if size of A is greater than B, it's going to return False.
We're going to do recursively.
The algorithm recursive reducing size of both A and B by one if the 1st element of A is same as the 1st element of B.
The algorithm recursive reducing size of B by one if the 1st element of A is NOT same as the 1st element of B.
Let $M[i, k] = A[1, \cdots, i]$ is the sub-sequence of $B[1, \cdots, k]$. We have

$$M[i,k] = \begin{cases} True & \text{if } i = 0 \\ False & \text{if } i > k \\ M(i-1, k-1] & \text{if } A[i] = B[k] \\ M(i, k-1] & \text{if } A[i] \neq B[k] \end{cases}$$

Pseudocode.

```
Line 0: Sub($A[1, \cdots, n], B[1, \cdots, m]$):
Line 1:     If $n == 0$: return True
Line 2:     Else If $n > m$: return False
Line 3:     Else If $A[1] = B[1]$: return Sub($A[2, \cdots, n], B[2, \cdots, m]$)
Line 4:     Else If $A[1] \neq B[1]$: return Sub($A[1, \cdots, n], B[2, \cdots, m]$)
```

Proof of correctness.
We know that there are only 2 possible values True or False.
The algorithm recursive at line 3 reducing size of both A and B by one if the 1st element of A is same as the 1st element of B.
The algorithm recursive at line 4 reducing size of B by one if the 1st element of A is NOT same as the 1st element of B.
We can see that, either recursive calls will reduce the size of either A or B or both A and B. And it will eventually hit either base case. If size of A is 0, we has been gone through all the element of A, so there's a sub sequence of B which is A, hence it return True.
If the size of A is greater than size of B, we know that it has been through $m - n - 1$ elements in B, so there's not enough element of B to compare with elements of A, hence it return False.

Therefore, the algorithm is correct to find whether A is sub-sequence of B
Running time.

$$\boxed{T(n,m) = \Theta(m)}$$

Justification of running time.
Let T(n,m) be the run time for the algorithm, with n is the size of array A, and m is the size of array B, $n < m$.
In the algorithm, if it returns True, we know that it's gonna recursive n time. If it returns False, we know that it's gonna recursive till the base case at $m = n$. We know that $m > n$, so the run time is:

$$\boxed{T(n,m) = \Theta(m)}$$

2. Another scheduling problem

Main idea. The idea is we create 2 hash Tables for A, and B, key is time and value is the optimal number of computations . We start from time n down to time 2, update optimal number of computations at time t for computer A and computer B. The last update for the hash Table is at the time 1. At the end, we return the maximum number of computations of table A and table B at time 1.
Let T(a,i) = the optimal number of computations of computer A at time i
Let T(b,i) = the optimal number of computations of computer B at time i
our sub-problems would be the optimal number of computations of a super computer at some time t.
The step to update hashTable of current computer at time t -1 to get the optimal number of computations is to take the maximum value of other computer at previous time t and sum(value of current Table at time t and value of current computer at time t-1).
If the value of other computer at previous time t is great than sum(value of current Table at time t and value of current computer at time t-1), we know that it has a move at time t, so that's why we don't add any value (mean adding 0).
Otherwise, get the sum(value of current Table at time t and value of current computer at time t-1) and update to hashTable of current computer at time t - 1.
We have the recurrence relations:

$$T(a,t-1) = max(A[t-1] + T_a[t], T_b[t])$$
$$T(b,t-1) = max(B[t-1] + T_b[t], T_a[t])$$

Pseudocode.

```
Line 0: Schedule($A[1,\cdots,n], B[1,\cdots,n]$):
Line 1:     $H_a$ = hashTable of A, $H_a[n]$ = A[n]
Line 2:     $H_b$ = hashTable of B, $H_b[n]$ = B[n]
Line 3:     For $i := n$ to $i = 2$:
Line 4:         $H_a[i-1] = \max(A[i-1] + H_a[i], H_b[i])$
Line 5:         $H_b[i-1] = \max(B[i-1] + H_b[i], H_a[i])$
Line 6:     return $\max(H_a[1], H_b[1])$
```

Proof of correctness.
We have the recurrence relations:
$T(a,t-1) = \max(A[t-1] + T_a[t], T_b[t])$
$T(b,t-1) = \max(B[t-1] + T_b[t], T_a[t])$
We can see that the Table of A and B will be update over time (decreasing). And each time it's updated, it get the most optimal value (best child) at that time $t-1$, by comparing the value of computations at time $t-1$ of current computer to the value at time t of Table of other computer (which is also a move). Eventually, it will get to time t = 1. By induction, if the current children have the best values of their parent, we know that the value of Table A at time 1 and Table B at time 1 has the highest value. Hence, return the maximum among these 2 values is also returning the most optimal solution. Therefore, the algorithm is correct to return the most optimal schedule.
Running time.

$$\boxed{T(n) = \Theta(n)}$$

Justification of running time.
Let T(n) be the run time for the algorithm, with n is the size of array A, and B.
From line 3 to 5, we just iterate through the list which is n time to update the hashTable for H_a and H_b, so the run time is $\Theta(n)$. Therefore, the run time is:

$$\boxed{T(n) = \Theta(n)}$$

3. Park Tours

Main idea.
The idea is to find the all possible shortest path list of k attractions in order, and return the minimum among them.
Let S(k,i): smallest cost path of length k ending at i.
Let d(i,j): the shortest distance from a_i to a_j for $i < j$
Our sub-problems would be: the shortest path list of k attractions in order with starting attractions are $a_1, \cdots, a_{m-k}$. Therefore, we have:
S(k,m) = $\Big[min_{j=k-1}^{m-1}\Big\{S(k-1,j)+d(j,m)\Big\}$
S(k,m-1) = $\Big[min_{j=k-1}^{m-2}\Big\{S(k-1,j)+d(j,m-1)\Big\}$
...
S(k,k) = $\Big[min_{j=k-1}^{k-1}\Big\{S(k-1,j)+d(j,k)\Big\}$
At the end we return the minimum value among these S's.

Pseudocode.
```
Line 0: ShorestTours(graph G, $a_1$ $a_2$ $\cdots$ $a_m$, k):
Line 1:     d = 2-D m x m Array
Line 2:     For $i := 0$ to $i = m-1$:
Line 3:         For $j := i+1$ to $j = m$:
Line 4:             d[i,j] = Dijkstra($a_i, a_j$)
Line 5:     shortestTemp = infinity
Line 6:     T = 2-D Array, length is m and height = k, and set every value to infinity
Line 7:     For $i := 1$ to $i = m$: T[1,i] = 0 , set the 1st row value to 0
Line 8:     For $i := 2$ to $i = k$:
Line 9:         For $j = 1$ to $j = m-i+1$:
Line 10:            For $z = j+1$ to $z = m-i+2$:
Line 11:                T[i, j] = min(T[i,j], $T[j,z]+d(j,z)$)
Line 12:     For $i := 1$ to $i = m-k-1$:
Line 13:         shortest = min(shortest, T[k,i])
Line 14:     return shortest
```

Proof of correctness.
We create every possible shortest distance in length of k attractions (in order). Hence, getting the minimum among these values is returning the shortest distance of k attractions. Therefore, the algorithm is correct to return the minimum distance of k attractions.
Running time.

$$\boxed{T(k,m,E,V) = \Theta\Big(km^2 + m(|V|+|E|)\log(V)\Big)}$$

Justification of running time.
Let T(k,m,E,V) be the run time for the algorithm, with V is number of vertices, E is number of edges, m is number of attractions , and k .
From line 2 to 4: running Dijkstra on every single attractions and there are m attractions, so the run time is $\Theta(m(|V|+|E|)\log(V))$

From line 8 to 11: there are 3 for loop, update the minimum for each row take $\Theta(m^2)$ and we doing this k time, so the run time is $\Theta(km^2)$

From line 12 to 13: we iterate m-k-1 time to find the shortest value which take $\Theta(m-k-1)$ Therefore, the total run time is :

$$\boxed{T(k,m,E,V) = \Theta\Big(km^2 + m(|V|+|E|)\log(V)\Big)}$$

4. Optimal binary search trees

Main idea.
The main idea is to iterate through n words, and making each word a root. When making each word a root, we will recursively split the words to the left into a sub- binary search trees and doing the same thing with the words to the right to other sub-binary search tree. It's essentially making every all the possible sub-tree (in alphabet order). Those sub-binary trees will be stored in case of use it again, and by doing this, we can reduce the run time.

Pseudocode.
```
Line 0: Table = Hash Table, key is the tree, value is the cost
Line 1: OptimalTree($W[w_1, w_2, \cdots, w_n]$, $p_1 \cdots p_n$):
Line 2:     For word in $W[w_1, w_2, \cdots, w_n]$:
Line 3:         making word be the root of the tree)
Line 4:         If right is in the Table:
Line 5:             right = OptimalTree(words on the right of the root,$p_1 \cdots p_n$)
Line 6:         Else : right = cost of right from the Table
Line 7:         If left is in the Table:
Line 8:             left = OptimalTree(words on the left of the root,$p_1 \cdots p_n$)
Line 9:         Else : left = cost of left from the Table
Line 10:         subtree = Build a Tree(left,root,right)
Line 11:         cost = Table(left) + Table(root) + frequencies
Line 12:         If $cost < Table(subtree)$ : Table(subTree) = cost
Line 13:    return minimum(all sub-trees in Table)
```

Proof of correctness.
The algorithm creates every possible tree and by memorizing the subtrees will help for the run time because it's only take constant time to look up to retrieve that. We also get the cost by adding cost of left sub- binary search tree + cost of right sub- binary search tree + all of elements in the list. We can see that this works because of update the depth of sub- binary search trees. To update the depths, the algorithm is moving to the left sub-trees and to right sub-trees down a depth (which we get by adding these elements) and adding the new root. At the end, we just return the least cost among these trees's costs. Therefore, the algorithm is correct.
Running time.

$$\boxed{T(n) = \Theta(n^2)}$$

Justification of running time.
Since we have n words, so there are T(i,j) possible sub-binary search tree or sub-problems for which $0 \leq i, j \leq n$. Hence, iterate through and fill in will take $\Theta(n^2)$. Then, it will take $\Theta(1)$ to retrieve them and update the cost. Therefore, the total run time is:

$$\boxed{T(n) = \Theta(n^2)}$$

5. Beat inference

Main idea.
Just like problem 3,the idea is to find the all possible shortest path list of k attractions in order, and return the minimum among them.
Our sub-problems would be: the shortest cost list of m indices in order with starting indices are $a_1, \cdots, a_{n-m}$.
At the end we return the minimum value among these values.
To do this, starting at level row = 1, we work out way until row = m. Calculator the cost for each column in row by adding the square of different of two different times and the value of level $row - 1$. Get the minimum cost among them, and update to current row,column. At the end, iterate through the column values of level m, and return the least cost among these values.
Pseudocode.

```
Line 0: Inference($A[1, \cdots, n]$, d, m):
Line 1:     shortest = infinity
Line 2:     T = 2-D Array, length is n and height = m, and set every value to infinity
Line 3:     For $i := 1$ to $i = n$: T[1,i] = A[i]
Line 4:     For $i := 2$ to $i = m$:
Line 5:         For $j = 1$ to $j = n - i + 1$:
Line 6:             For $z = j + 1$ to $z = n - i + 2$:
Line 7:                 temp = $(t_z - t_j - d)^2 - A[t_z]$
Line 8:                 T[i, j] = min(T[i,j], temp)
Line 9:     For $i := 1$ to $i = n - m - 1$:
Line 10:        shortest = min(shortest, T[m,i])
Line 11:     return shortest
```

Proof of correctness.
The algorithm is solving by adding the cost of current beat x, difference of beat x and beat y, and the difference accumulated in beat y which is least cost (this value is stored or memorized in the table). Storing the least cost of beat x with the beat that cause its cost to be the least cost. If beat b is least cost aggregate beat cost so far, we can see that beat x + beat y will be the least cost because we only choose a smaller. Therefore, the algorithm is correct to to use sub-problem to return the minimum cost solution.
Running time.

$$\boxed{T(m,n) = \Theta(m \times n^2)}$$

Justification of running time.
Let T(m,n) be the run time for the algorithm, n is size of array A , and m .
At line 2: Iterator n time to initialize all the values of columns in 1st row take $\Theta(n)$.
From line 4 to 8: there are 3 for loop, update the minimum for each row in the inner double loops take $\Theta(n^2)$ and we doing this m time (outer loop), so the run time is $\Theta(m \times n^2)$.
At line 9 and 10: iterate through the values of level (row) m would take $\Theta(n)$
Therefore, the total run time is :

$$\boxed{T(m,n) = \Theta(m \times n^2)}$$

6. Optional Bonus Problem: Image re-sizing

Main idea. YOUR ANSWER GOES HERE
Pseudocode. YOUR ANSWER GOES HERE
Proof of correctness. YOUR ANSWER GOES HERE
Running time. YOUR ANSWER GOES HERE
Justification of running time. YOUR ANSWER GOES HERE