CSCI 570 HW3

Mounika Mukkamalla - 1001219359

October 1 2023

1

We can use min heap to store lengths of the rods.
Algorithm: Let input array be $input\_array = [l_1, l_2, ...l_n]$
1. Insert all the lengths into min heap, min_heap
2. Initialize $sum = 0$
3. if $n == 1$, return l_1;
4. if min_heap is empty, return sum
5. Get 2 minimum elements from min_heap, l_i, l_j
6. Insert $l_i + l_j$ into min_heap and increment sum by $l_i + l_j$.
7. Go to step 4.

Time complexity: $O(nlogn)$

2

a) Consider a case where there are 3 artists with deadlines and turnouts:

Artist	Deadlines	Turnouts
1	2	35
2	2	40
3	1	23

According to our algorithm, We first select artist 3 then either of artist 1 or 2.
Hence overall expected turnout according to algorithm is either 23+40 or 23+35. i.e., 63 or 58
But we can select Artist 1 and 2, scheduling artist 1 for 1^{st} day and artist 2 for the 2^{nd} day, we get overall turnout to be 35+40 = 75 greater than the algorithm's turnout
Hence this logic wouldn't work

b) Consider a case where there are 3 artists with deadlines and turnouts:

Artist	Deadlines	Turnouts
1	1	25
2	2	40
3	3	70

According to our algorithm, we first select artist 3 and then artist 2 with overall expected turnout resulting in 70 + 40 = 110
But we can select artist 1 for day1, artist 2 for day2 and artist 3 for day3 with overall expected turnout to be 25 + 40 + 70 = 135 > Algorithm's turnout
Hence this logic wouldn't work.

c) Let the given array be tuples of deadlines and turnouts: $music = [(D_1, A_1), (D_2, A_2), ..., (D_N, A_N)]$
Algorithm:
1. sort $music$ array based on expected turnouts, A_i, in decreasing order.
2. initialise $schedule$ of size m with all elements -1.
3. initialise i =0 // Used to traverse over $music$ array.
4. while i< N: // we try to find a day for every artist

a. j = m-1 // used to traverse over number of days

i. while j >=0 : // we find the maximum day satisfying artist's deadline

ii. if $schedule[j]! = -1$ and $D_i > j + 1$: // check if day j+1 is free and i^{th} artist's deadline

isn't exceeding the current day
1. $schedule[j] = i$ and break // Mark j+1 day as occupied
iii. decrement j by one
b. increment i by one
5. return *schedule*

3

Given ALG recurrence, $T(n) = 7T(n/2) + n^2$
$c = log_b a = log_2 7 = n^{2.8074}$
$f(n) = n^2$
$\implies f(n) < n^c$, hence $\exists \epsilon > 0 \ni f(n) = n^{c-\epsilon}$
According to Master's Theorem case 1,
if for some $\epsilon > 0$ $f(n) = O(n^{c-\epsilon})$ then $T(n) = \theta(n^c)$
Hence, complexity of $ALG = \theta(n^{2.8074})$

For ALG', recurrence, $T`(n) = aT`(n/4) + n^2 logn$
$c' = log_4 a = log_2 \sqrt{a}$
$f'(n) = n^2 logn$ and $n^{c'} = n^{log_2 \sqrt{a}}$

For ALG' to be asymptotically faster than ALG, complexity of ALG' should be less than complexity of ALG.
$\implies T'(n) < T(n) \implies T'(n) < n^{log_2 7}$
We need to find the largest value of a for ALG', hence let's take the highest complexity we can obtain from T`(n) and find a such that $T`(n) < n^{log_2 7}$
Highest complexity we can obtain from ALG' is $n^{c'}$ when $n^{c'} > f'(n)$
$\implies n^{c'} < n^{log_2 7}$
$\implies c' < log_2 7 \implies log_2 \sqrt{a} < log_2 7 \implies \sqrt{a} < 7 \implies a < 49$

Hence largest value of a for which ALG' performs asymptotically faster than ALG is **48**.

4

Let's write down complexities of each step:
Step a) : $O(1)$
Step b) : $O(n^2)$
Step c) and d) : $O(n)$
Step f) : $O(n)$
Each recursive function has additional $f(n) = O(n^2) + O(n) + O(n) = O(n^2)$
In step e), after dividing the array into 2 halves, we apply the sort recursively on the 2 lists.
Hence, recursive function becomes, $T(n) = 2T(n/2) + f(n) \implies T(n) = 2T(n/2) + O(n^2)$
Here, $a = 2, b = 2; c = log_b a = log_2 2 = 1$
for $\epsilon > 0, f(n) = \Omega(n^{c+\epsilon} = O(n^{1+1})$
hence by Master's Theorem, case 3, $T(n) = \theta(f(n)) = \theta(n^2)$

5

a) $T(n) = T(n/2) + 2^n$
$a = 1, b = 2 \implies c = log_b a = log_2 1 = 0$
$f(n) = 2^n > n^c \implies f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$
In Master's Theorem case 3, $T(n) = \theta(2^n)$

b) $T(n) = 5T(n/5) + nlogn - 1000n$
$a = 5, b = 5; c = log_b a = log_5 5 = 1$

$f(n) = nlogn - 1000n$ which is positive for higher values of n, $n >= n_0$
Hence we can ignore $1000n$ after a point, $f(n)$ becomes $\theta(nlogn)$
$f(n) = \theta(n^c log^k n)$ for k = 1, c = 1
Hence by Masters theorem case2, $T(n) = \theta(nlog^2 n)$

c) $T(n) = 2T(n/2) + log^2 n$
$a = 2, b = 2; c = log_b a = log_2 2 = 1;$
$f(n) = log^2 n = O(n^{c-\epsilon})$ for some $\epsilon > 0$
By applying Master's Theorem case 1, $T(n) = \theta(n^c) = \theta(n)$

d) $T(n) = 49T(n/7) - n^2 logn^2$
$f(n) = -n^2 logn^2$ which is a negative function
Hence Master's Theorem can't be applied.

e) $T(n) = 3T(n/4) + nlogn$
$a = 3, b = 4; c = log_b a = log_4 3$
$f(n) = nlogn > n^c \implies f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$
By applying Master's Theorem case 3, $T(n) = \theta(f(n)) = \theta(nlogn)$

6

Algorithm: Let's define a function to find mid element of the linked list using tortoise-hare approach .
Maintain 2 pointers and increment one pointer 2 times more than the other pointer.
Hence the 2^{nd} pointer would reach mid when 1^{st} pointer reaches end.

func findMid($startPointer$, $endPointer$):
1. Define $tempPointer$ to NULL
1. if $startPointer$ is NULL: // if linkedlist is empty return NULL
a. return NULL
2. $tempPointer = startPointer->next$
3. if($tempPointer == endPointer$) // if we reach the end of the list return list
a. return $startPointer$
4. increment $tempPointer$ to it's next element 1st increment for temp
5. if $tempPointer$ is not equal to $endPointer$
a. increment $startPointer$ to it's next element
b. increment $tempPointer$ to it's next element 2nd increment for temp
6. Go to step 3.

func findKey(key**,** $list$**):**
1. Let given linked list be $list$ and the element to be found is key
2. Initialize $startPointer$ to first element of the $list$ and $endPointer$ to $NULL$
3. if $startPointer == endPointer$:
a. if value at $startPointer$ is equal than key:
i. return True.
b. else:
i. return False.
4. $midPointer = findMid(startPointer, endPointer)$
5. if $midPointer$ is NULL
a. return False
6. if the value at $midPointer$ is less than key:
a. Update $startPointer$ to $midPointer->next$ // binary search the right half of the list
7. if value at $midPointer$ is greater than key:
a. Update $endPointer$ to $midPointer$ //binary search the left half of the list.
8. if value at $midPointer$ is equal to key:
a. return True.
9. Go to step3.

Time complexity: In every iteration, we divide the list into half.
In the divide step, we traverse half of the array to find the mid element.
Hence, We can write recurrence relation as $T(n) = T(n/2) + O(n)$
$a = 1, b = 2; c = log_b a = log_2 1 = 0$
Hence by Master's theorem case 1, complexity = $O(n)$

7

Let's define a function to find mid element of the linked list using tortoise-hare approach .
Maintain 2 pointers and increment one pointer 2 times more than the other pointer.
Hence the 2^{nd} pointer would reach mid when 1^{st} pointer reaches end.

func findMid($startPointer$, $endPointer$):
1. Define $tempPointer$ to NULL
1. if $startPointer$ is NULL: // if linkedlist is empty return NULL
a. return NULL
2. $tempPointer = startPointer->next$
3. if($tempPointer == endPointer$)   // if we reach the end of the list return list
a. return $startPointer$
4. increment $tempPointer$ to it's next element   1st increment for temp
5. if $tempPointer$ is not equal to $endPointer$
a. increment $startPointer$ to it's next element
b. increment $tempPointer$ to it's next element   2nd increment for temp
6. Go to step 3.

func merge ($startPointer1, endPointer1, startPointer2, endPointer2$**):**
1. Initialise $resultList$ = NULL
2. Traverse through 2 linkedlists using $startPointer1$ and $startPointer2$.
3. Attach the node with the minimum value to the $resultList$ and increment the minimum node pointer to next.
4. Continue step 3 until $startPointer1$ and $startPointer2$ reach $endPointer1$, $endPointer2$
5. if one of the pointers, $startPointer1$, $startPointer2$ reach end pointers first, iterate over the other pointer till it reaches the end pointer and append to the $resultList$

Time complexity for merge : O(m+n) where is m and n are sizes of 2 linkedlists.

Algorithm: Let given linked list be $list$ and the element to be found is key
func mergeSort($startPointer$**,** $endPointer$**,** key**)**:
1. if $startPointer$ == NULL or $startPointer$->next == NULL then return
2. mid = findMid($startPointer$, $endPointer$)
3. mergeSort($startPointer$, mid)
4. mergeSort(mid->next, $endPointer$)
5. merge($startPointer$, mid, mid->next, $endPointer$)

Complexity:
Divide step takes O(n) time
Merge step takes O(n) time
Recurrence relation: T(n) = 2T(n/2) + O(n) + O(n)
$a = 2, b = 2; c = log_b a = log_2 2 = 1$
$f(n) = \theta(n^c log^k n)$ for c=1 and k = 0
Hence by Master's theorem case 2, $T(n) = \theta(nlogn)$

8

a) Let 2 skyline arrays be $skyline1 = [a_1, a_2, ...a_n]$ and $skyline2 = [b_1, b_2, ...b_m]$.
Maintain h_1 and h_2, current heights of the skylines $skyline1$ and $skyline2$.
Iterate through both the arrays and insert the element with lower x-coordinate and insert maximum current heights of both the skylines without consecutive duplicate heights.

Algorithm:

func merge($skyline1$**,** $skyline2$**)**:

1. Define $merged\_array$
2. Initiate i = 0, j = 0, k=0 // pointers for $skyline1$, $skyline2$ and merged_array respectively
3. initiate $h_1 = 0$ and $h_2 = 0$ // represent current heights of x coordinates
4. insert 0's at the end of $skyline1$ and $skyline2$ for making computation easier.
5. while i!=n+1 and j!=m+1:

a. if $skyline1$[i] < $skyline2$[j]: // if x-coordinate of $skyline1$ is less than $skyline2$
l. h_1 = $skyline1$[i+1]
m. max_h = max(h_1, h_2) // Gets the maximum height of 2 skylines
n. if max_h != merged_array[k] // checks for duplicate heights
1. insert $skyline1$[i] and max_h to merged_array
2. increment k by 2
o. increment i by 2
b. else:
i. h_2 = $skyline2$[j+1]
ii. max_h = max(h_1, h_2) // Gets the maximum height of 2 skylines
iii. if max_h != merged_array[k] // checks for duplicate heights
1. insert $skyline2$[j] and max_h to merged_array
2. increment k by 2
iv. increment j by 2

6. if i!=n+1:
i. loop through $skyline1$ and insert all elements of the $skyline1$ to the merged_array
7. if j!=m+1:
i. loop through $skyline2$ and insert all elements of the $skyline2$ to the merged_array
8. Delete the last 0 of merged_array //to maintain notation consistency

We iterate through both the arrays, hence time complexity is O(m+n)

b) Given list of stages: stages = $[(l_1, h_1, r_1), (l_2, h_2, r_2), ..., (l_n, h_n, r_n)]$
By divide and conquer, we divide the array into 2 halves and get skylines from each half and merge both skylines using above algorithm.
Base case: when there is only one stage, $[(l_i, h_i, r_i)]$ then return array: $[l_i, h_i, r_i]$
Let's define function getSkyline(stages, i, j) that returns skyline array

Algorithm:

func getSkyline($stages$**,** i**,** j**)**: // i and j are left and right pointers of stages array

1. if i==j:
a. return $[l_i, h_i, r_i]$
2. mid = (i+j)/2
3. skyline1 = getSkyline(stages, i, mid)
4. skyline2 = getSkyline(stages, mid+1, j)
5. return merge(skyline1, skyline2)

getSkyline(stages, 0, n-1) gives the result.

Time complexity:

Divide step takes: O(1) time
Merge step takes: O(n) time
We divide the array into 2 halves, hence recurrence relation becomes: T(n) = 2T(n/2) + O(n)
By Masters theorem case2, Complexity becomes O(nlogn)

9

a) Let opt[k] be the sum of distinct combinations possible to sum up to k dollars from 1-dollar and 2-dollars coins, where $0 <= k <= n$
At current point, with zero dollars in hand, we have 2 possible ways to go:
1. to select one 1-dollar coin and find the number of distinct combinations for k-1, opt[k-1]
2. to select one 2-dollar coin and find the number of distinct combinations for k-2, opt[k-2]

b) Hence the recurrence relation becomes: opt[k] = opt[k-1] + opt[k-2]
Base cases: opt[0] = 1; opt[1] = 1

c) Pseudo Code 1:

```
int possibleCombinations(int n) {
1.    int opt[n+1];
2.    opt[0] = 1;
3.    opt[1] = 1;
4.    for(int i=2;i<=n;i++) {
i.      opt[i] = opt[i-1] + opt[i-2];
5.    }
6.    return opt[n]
7. }
```

Here the space complexity is $O(n)$, can be reduced to constant space through the following algorithm

Pseudo Code 2:

```
int possibleCombinations(int n) {
1.    temp1 = 1;
2.    temp2 = 1;
3.    for(int i=2;i<=n;i++) {
a.      result = temp1 + temp2;
b.      temp1 = temp2;
c.      temp2 = result;
4.    }
5.    return result
6. }
```

d) base cases here are when amount is 0 and 1. that is, opt[0] = opt[1] = 1;
Also this problem can just be formulated as a standard Fibonacci series problem.

e) Time complexity:
Run time complexity, we iterate through the array for O(n) times.
Inside each loop, we perform addition operation which is constant time
Hence overall complexity is O(n)

10

This problem can almost be formulated as 0-1 knapsack problem with values 1 for all weights.
a) Let opt[k][x] be the minimum number of packages for k items with x total weight, where $0 <= k <= n$ and $0 <= x <= W$

b) At a given point, for a given package, k with w_k, we have 2 possible ways:
1. We take the weight and find the minimum number of packages in the remaining packages with weight $x - w_k$
i. $opt[k][x] = 1 + opt[k-1][x - w_k]$
2. We don't select w_k and find the minimum number of packages in the remaining packages with weight x

i. $opt[k][x] = opt[k-1][x]$
We need to find minimum of these 2 cases and hence,
$opt[k][x] = min(1 + opt[k-1][x - w_k], opt[k-1][x])$

c) Pseudo code:
Function returns -1 if there is no way to find max profit with exact W weight to load.

```
int minPackages(int n, int W, int w[] ) {
1.  int opt[n+1][W+1];
2.  for (int k=0; k<=n; k++) {
i.    for (int x =0;x<=W; x++) {
a.       if (x ==0) opt[k][x]=0;
b.       if (k==0) opt[k][x] = infinity
c.       if(w[k] > x) opt[k][x] = opt[k-1][x]
d.       else opt[k][x] = min(1+opt[k-1][x-w[k]], opt[k-1][x])
ii.    }
3.  }
4.  if opt[n][W] == infinity then return -1
5.  else return opt[n][W];
6. }
```

d) Base cases:
1. When x = 0, though we have packages available, with 0 possible weight to pick up, opt[k][0] would be 0;
2. when k = 0 and x>0, there are no enough packages to return the possible weight, and therefore no way to maximise profit. Hence opt[0][x] is initially assigned to infinity(maximum possible value).
3. opt[k][x] = opt[k-1][x] when $w_k > x$

e) Runtime Complexity:
There are 2 loops, inside the inner loop, all the operations are of constant time complexity.
outer loop runs O(n) times and inner loop runs O(W) times. hence time complexity = $O(nW)$.