Name: Nikhil Sathyanarayana Shastry
USC_ID: 8464475401
Email: nsathyan@usc.edu

CSCI570 - Analysis of Algorithms (HW3)

Q1. (Greedy) Given n rods of lengths $L_1, L_2, \ldots, L_n$, respectively, the goal is to connect all the rods to form a single rod. The length and the cost of connecting two rods are equal to the sum of their lengths. Devise a greedy algorithm to minimize the cost of forming a single rod.

Solution:

- Firstly, we need to sort the input array of different lengths in the increasing order since it costs the least to join rods of smaller lengths
- Now we need to keep track of the total cost of joining the rods in a variable called ***total_cost*** which is initialized to **0**.
- We follow the following steps until the sorted array consists of only one element i.e. the completely joined rod.
    - Select the two shortest rods from the sorted array.
    - Merge the lengths of the 2 shortest rods chosen and combine it into one length.
    - Update the cost of merging the two rods in the ***total_cost*** variable.
    - Remove the 2 shortest rods from the sorted array
    - Insert the merged rod back into the sorted array without spoiling the sorted order
- Finally, we will have the minimum cost for merging n rods of different lengths stored in the ***total_cost*** variable, which will be returned.

Q2. (Greedy) During a summer music festival that spans m days, a music organizer wants to allocate time slots in a concert venue to various artists. However, each time slot can accommodate only one performance, and each performance lasts for one day.

There are N artists interested in performing at the festival. Each artist has a specific deadline, D_i, indicating the last day on which they can perform, and an expected audience turnout, A_i, denoting the number of attendees they expect to draw if they perform on or before their deadline. It is not possible to schedule an artist's performance after their deadline, meaning an artist can only be scheduled on days 1 through D_i.

The goal is to create a performance schedule that maximizes the overall audience turnout. The schedule can assign performances for n artists over the course of m days.

Note: The number of performances (n) is not greater than the total number of artists (N) and the available days (m), i.e., $n \leq N$ and $n \leq m$. It may not be feasible to schedule all artists before their deadlines, so some performances may need to be skipped.

(a) Let's explore a situation where a greedy algorithm is used to allocate n performance to m days consecutively, based on their increasing deadlines D_i. If, due to this approach, a task ends up being scheduled after its specified deadline D_i, it is excluded (not scheduled). Provide an counterexample to demonstrate that this algorithm does not consistently result in the best possible solution.

(b) Let's examine a situation where a greedy algorithm is employed to distribute n performance across m days without any gaps, prioritizing performances based on their expected turnouts A_i in decreasing order.

If, as a result of this approach, a performance ends up being scheduled after its specified deadline D_i, it is omitted from the schedule (not scheduled). Provide a counterexample to illustrate that this algorithm does not consistently produce the most advantageous solution.

(c) Provide an efficient greedy algorithm that guarantees an optimal solution to this problem without requiring formal proof of its correctness.

Solution:

Total number of days of the summer festival = m
Total number of artists = N
1 performance = 1 day and there are n performances
Last Deadline $-> D_i$
Audience turnout $-> A_i$

Goal: Maximize Audience Turnout
Note: $n \leq N \; and \; n \leq m$

(a) let us consider the following information:

- Artist A : Deadline D1 = 1, Audience Turnout(A1) = 200
- Artist B : Deadline D2 = 2, Audience Turnout(A2) = 75
- Artist C : Deadline D3 = 3, Audience Turnout(A3) = 300
- Artist D : Deadline D4 = 4, Audience Turnout(A4) = 125

Here we have 4 artists & 4 available days.
We want to schedule 3 performances i.e. n = 3

Let us use the greedy algorithm that schedules performances based on increasing deadlines. Then we get the following schedule:

Day 1: Artist A : Total turnout = 200 (within deadline)
Day 2: Artist B : Total turnout = 200 + 75 = 275 (within deadline)
Day 3: Artist C : Total turnout = 275 + 300 = 575 (within deadline)

Here we see that the given greedy algorithm gives a turnout of **575**.

However, we can prove with a counterexample that this solution does not give us the best possible solution in all scenarios. Let us consider the following scheduling order which acts as a counterexample:

Day 1: Artist A : Total turnout = 200 (within deadline)
Day 2: Artist D : Total turnout = 200 + 125 = 325 (within deadline)
Day 3: Artist C : Total turnout = 325+ 300 = **625** (within deadline)

Here in this counterexample we see that a different scheduling order can lead to a higher total audience turnout while still adhering to the deadlines of the artists.

This counterexample demonstrates that scheduling performances based solely on increasing deadlines may not consistently result in the best possible solution, as a different scheduling order can lead to a higher total audience turnout while still adhering to the artists' deadlines.

(b) let us consider the following information:

- Artist A : Deadline D1 = 3, Audience Turnout(A1) = 150
- Artist B : Deadline D2 = 2, Audience Turnout(A2) = 100
- Artist C : Deadline D3 = 4, Audience Turnout(A3) = 300
- Artist D : Deadline D4 = 1, Audience Turnout(A4) = 75

Here we have 4 artists & 4 available days.
We want to schedule 3 performances i.e. $n = 3$

Let us use the greedy algorithm that schedules performances based on decreasing expected turnout. Then we get the following schedule:

Day 1: Artist C : Total turnout = 300 (within deadline)
Day 2: Artist A : Total turnout = 300 + 150 = **450** (within deadline)
On Day 3 we try to schedule Artist B since it has the next highest turnout but according to the given rules for scheduling, as the deadline (D2 = 2) is exceeded it will be *omitted.*

However, we can prove with a counterexample that this solution does not give us the best possible solution in all scenarios. Let us consider the following scheduling order which acts as a counterexample:

Day 1: Artist A : Total turnout = 150 (within deadline)
Day 2: Artist B : Total turnout = 150 + 100 = 250 (within deadline)
Day 3: Artist C : Total turnout = 250 + 300 = **550** (within deadline)

Here in this counterexample we see that a different scheduling order can lead to a higher total audience turnout while still adhering to the expected turnout of the artists.

This counterexample demonstrates that scheduling performances based solely on decreasing expected turnout may not consistently result in the best possible solution, as a different scheduling order can lead to a higher total audience turnout while still adhering to the expected turnout of an artist.

(c) The greedy algorithm that guarantees an optimal solution to this problem is as follows:

- Create a list of artists, where each artist is represented by a tuple ($deadline$, $audience\_turnout$).
- First, sort this list in descending order of $audience\_turnout$.
- Then sort this array in the ascending order of deadlines. This ensures that if there exists two performances with the same number of $audience\_turnout$ we can prioritize the performance with the earliest first and then the one with a later deadline.
- Now, we initialize a $schedule\ array$ of size m to represent the time slots available at concert venue at the summer music festival. Initialize an empty list to store the $scheduled\ performances$.
- We also initialize a variable $total\_turnout$ to **zero**.
- We iterate through the sorted list of artists and do the following repeatedly:
  - **For each artist:** we find the latest available time slot on or before their deadline (from day 1 to D_i). We can achieve this by starting from D_i and then going backwards in the $schedule\ array$ until we find an empty slot.
  - On finding a suitable slot, we assign the artist to that slot, and update the $schedule\ array$, and add the $audience\_turnout$ value to the current value of the $\boldsymbol{total\_turnout}$ variable and update it. We also add the scheduled artist to the list of $scheduled\ performances$
  - If a performance cannot be scheduled within its deadline, it is skipped, and the algorithm proceeds to the next artist.

- We continue the above iteration until we have scheduled the required number of performances to be scheduled (n) or until we have taken into account the possibility of scheduling for all the artists.
- Finally, we return the $\boldsymbol{total\_turnout}$ for the festival and the list of $scheduled\ performances$, which will represent a schedule that maximizes the overall audience turnout within the specified constraints.

Q3. (Master Theorem) The recurrence $T(n) = 7T(n/2) + n^2$ describes the running time of an algorithm ALG. A competing algorithm ALG′ has a running time of $T'(n) = aT'(n/4) + n^2 \log n$ What is the largest value of a such that ALG′ is asymptotically faster than ALG?

Solution:

In ALG where $T(n) = 7T(n/2) + n^2$,
$a = 7$
$b = 2$
$f(n) = n^2$
$c = log_b(a) = log_2(7) = 2.807$

Therefore, ALG will fall under the $case$ 1 of the Master theorem.

Hence, $\boldsymbol{T(n) = \theta(n^{2.807})}$

In ALG' where $T'(n) = aT'(n/4) + n^2 * log(n)$,
$a = a$
$b = 4$
$f(n) = n^2 * log(n)$
$c = log_b(a) = log_4(a)$

Now, for ALG' to be asymptotically faster than ALG,

$T'(n)$ must be lesser than $T(n)$, Let us compare the value of c of both $T'(n)$ and $T(n)$:

$log_4(a) \leq log_2(7)$
$a \leq 4^{log_2(7)}$
$a \leq 2^{2*log_2(7)}$
$a \leq 2^{log_2(7^2)}$

From the log property $a^{log_a x} = x$ we have,

$a \leq 49$

Therefore, the largest value of a for ALG' to be asymptotically faster than ALG should be $\boldsymbol{a \leq 49.}$

Q4. (Master Theorem) Consider the following algorithm StrangeSort which sorts n distinct items in a list A.

(a) If n ≤ 1, return A unchanged.

(b) For each item x ∈ A, scan A and count how many other items in A are less than x.

(c) Put the items with count less than n/2 in a list B.

(d) Put the other items in a list C.

(e) Recursively sort lists B and C using StrangeSort.

(f) Append the sorted list C to the sorted list B and return the result.

Formulate a recurrence relation for the running time T (n) of StrangeSort on an input list of size n. Solve this recurrence to get the best possible O(·) bound on T(n).

Solution:

a) If $n \leq 1$, return A unchanged takes $T(n) = O(1)$.
b) For each item in A we scan and count how many other items in A are less than x. This takes $O(n^2)$
c) The worst case time complexity for partitioning the list and putting items into less than $n/2$ into list B takes $O(n)$ time.
d) The worst case time complexity for partitioning the other half of the list and putting the rest of the items into list C takes $O(n)$ time.
e) Sorting the lists B and C recursively using strangeSort takes $T(n/2)$ each time we divide the list into 2 sublists at each step.
f) Appending the sorted list C to the sorted list B takes $O(n)$.

The recurrence relation got after the above analysis is $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$.

The $O(n)$ for partitioning and appending are ignored in the above recurrence relation as the term $O(n^2)$ taken by $step\ b$ dominates them. Hence we upper bound it to $O(n^2)$.

Steps for solving the above equation with Masters theorem:

$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$

$a = 2, b = 2$

$c = log_2 2 = 1$

$n^c = n^1 = n$

$f(n) = n^2$

$Applying\ Masters\ Theorem, we\ know\ that$

$O(n) < O(n^2)$

$Hence, f(n) = \Omega(n^{c+\varepsilon})\ which\ is\ \boldsymbol{case\ 3}\ of\ the\ Masters\ theorem$

$Therefore, \boldsymbol{T(n) = \theta(n^2)}$

Q5. (Master Theorem) For the given recurrence equations, solve for T (n) if it can be found using the Master Method. Else, indicate that the Master Method does not apply.

(a) $T(n) = T(n/2) + 2^n$
(b) $T(n) = 5T(n/5) + n\log n - 1000n$

(c) $T(n) = 2T(n/2) + \log^2 n$
(d) $T(n) = 49T(n/7) - n^2 \log n^2$

(e) $T(n) = 3T(n/4) + n\log n$

Solution:

a. $T(n) = T(n/2) + 2^n$

- $a = 1, b = 2$
- $c = \log_2 1 = 0$
- $n^c = n^0 = 1$
- $f(n) = 2^n$
- Since $f(n)$ is very large compared to $n^{c+\varepsilon}$, following the master theorem the given recurrence function will fall under **case 3.**
- Therefore, $T(n) = \boldsymbol{\theta(2^n)}$.

b. $T(n) = 5T\left(\frac{n}{5}\right) + n\log n - 1000n$

- $a = 5, b = 5$
- $c = \log_5 5 = 1$
- $n^c = n^1 = n$
- $f(n) = nlogn - 1000n$
- In $f(n)$ since $nlogn$ is greater than $1000n$, let us consider for only $nlogn$
- This will fall under the **case 2** of the master theorem.
- Therefore, $\boldsymbol{\theta(nlog^2 n)}$. This is because k is already equal to 1 it becomes $\theta(nlog^{1+1}n)$

c. $T(n) = 2T(n/2) + \log^2 n$

- $a = 2, b = 2$
- $c = \log_2 2 = 1$
- $n^c = n^1 = n$
- $f(n) = \log^2 n$
- Since $f(n)$ is very small compared to $n^{c-\varepsilon}$, following the master theorem the given recurrence function will fall under **case 1.**
- Therefore, $T(n) = \boldsymbol{\theta(n)}$.

d. $T(n) = 49T(n/7) - n^2 \, logn^2$

- $a = 49, b = 7$
- $c = log_7 49 = 2$
- $n^c = n^2$
- $f(n) = -n^2 \, logn^2$
- Since $f(n)$ is a negative function, **we will not be able to apply master theorem** to the given recurrence equation.

e. $T(n) = 3T(n/4) + n \, log \, n$

- $a = 3, b = 4$
- $c = log_4 3 \approx 0.8$
- $n^c = n^{0.8}$
- $f(n) = nlogn$
- Since $f(n)$ is greater than $n^{c+\varepsilon}$**,** following the master theorem, the given recurrence function will fall under **case 3.**
- Therefore, $T(n) = \boldsymbol{\theta(nlogn)}$.

Q6. (Divide-and-Conquer) We know that binary search on a sorted array of size n takes Θ(log n) time. Design a similar divide-and-conquer algorithm for searching in a sorted singly linked list of size n. Discuss its worst-case runtime complexity.

Solution:

The initial intuition to solve the given problem is to split the linked list just like in the array version of the binary search. The algorithm using divide-and-conquer for searching in a sorted linked list of size n is as follows:

1. Start with the entire linked list.

2. Next, we make use of the fast and slow pointer strategy to find the middle node of the current linked list. In this strategy we have a fast pointer which moves 2 nodes at a time and the slow pointer moves one node at a time. By the time the fast pointer reaches the end of the linked list the slow pointer would have reached the middle node which will be used for the next steps of our algorithm.

3. We need to compare the value at the middle node with the target value.

- If they are equal, we return the middle node (element found).
- If the value of the middle node is greater than the target, discard the right half of the list (inclusive of the middle node) and repeat the search in the left half.
- If the value of the middle node is less than the target, discard the left half of the list (inclusive of the middle node) and repeat the search in the right half.

4. Finally, we need to repeat steps 2 and 3 until the target is found, or the sub list has no more nodes which will mean that the target does not exist in the given list, and we return False.

Since we are required to devise an algorithm for a linked list the process to find the middle element is an extra overhead. In a linked list we will have to make use of the fast and slow pointer method to find the middle element which will take $O(n/2)$ time in the worst case which is a little inefficient compared to the constant time that we will require to access the middle element in an array. Here n is the number of nodes in the current sub list. Hence the divide step in each iteration takes $O(n/2)$ time.

Now, let's us consider the worst-case runtime complexity of the whole algorithm:

We may have to perform $O(log\ n)$ recursive calls to halve the search space (similar to binary search) in the worst case. We also know that each divide step takes $O(n/2)$ time to find the middle element. Therefore, the overall worst-case runtime complexity of our algorithm is $O(n/2\ *\ log\ n)$, which simplifies to $\boldsymbol{O(n\ log\ n)}$.

Some of the worst-case complexity scenarios where the recursive calls are made the maximum number of times include the following:

- The target element is not present in the linked list: In this scenario the maximum number of recursive calls are made to find out that our target element does not exist in the list.
- The target element is present in the leftmost or the rightmost node of the linked list: This is one other scenario where maximum number of recursive calls are made to find the target element in the beginning or the end of the list.

Q7. (Divide-and-Conquer) We know that mergesort takes Θ(n log n) time to sort an array of size n. Design a divide-and-conquer mergesort algorithm for sorting a singly linked list. Discuss its worst-case runtime complexity.

Solution:

The initial intuition to solve the given problem is to split the linked list into half at each divide call and then recursively sort each half and then merge the sorted halves of the linked list together. The algorithm using divide-and-conquer for implementing a mergesort algorithm in a singly linked list of size n is as follows:

1. First, we use the fast and slow pointer strategy to divide the given linked list into two equal or almost equal halves. In this strategy, we have a fast pointer that moves 2 nodes at a time, and the slow pointer moves one node at a time. By the time the fast pointer reaches the end of the linked list, the slow pointer would have reached the middle of the list, which will be used to divide the linked list into 2 halves. We repeat this dividing process recursively on the left half and the right half sublists until we are left with n sublists, each containing only one element.

2. Once we are left with only one element in each sublist, we merge the 2 halves back together. We do this by comparing the values of the elements in the two lists at the current position. We select the smaller element as the next element in the merged list, advance the pointer in the corresponding sublist, and repeat this process until all the sublists are merge-sorted into one sorted linked list.

3. Return the merged, sorted linked list.

Now, let's us consider the worst-case runtime complexity of the whole algorithm:

The merge sort algorithm for a singly linked list operates in three main steps. Firstly, in the divide phase, the list is split into two halves, which takes $O(n)$ time since it requires traversing the entire list once to find the mid-point for dividing. This division occurs recursively, with $log(n)$ levels of recursion in total this is because the list is being divided at each step. Second, in the recursively sort phase, there are $log(n)$ levels of recursion, and at each level, $O(n)$ work is performed to merge the sorted halves. Finally, in the merge phase, which also occurs at each level of recursion, it takes $O(n)$ time to compare and merge each element from both halves. Overall, this algorithm's time complexity is $O(n\ log\ n)$ due to the repeated splitting and merging, making it an efficient method for sorting singly linked lists.

Some of the worst-case complexity scenarios where the recursive calls are made the maximum number of times include the following:

- The given linked list is already sorted in the reverse order.
- Each element in the given linked list has the same value.

Q8. (Divide-and-Conquer) Imagine you are responsible for organizing a music festival in a large field, and you need to create a visual representation of the stage setup, accounting for the various stage structures. These stages come in different shapes and sizes, and they are positioned on a flat surface. Each stage is represented as a tuple (L, H, R), where L and R are the left and right boundaries of the stage, and H is the height of the stage.

Your task is to create a skyline of these stages, which represents the outline of all the stages as seen from a distance. The skyline is essentially a list of positions (x-coordinates) and heights, ordered from left to right, showing the varying heights of the stages.

Take Fig. 1 as an example: Consider the festival setup with the following stages: (2, 5, 10), (8, 3, 16), (5, 9, 12), (14, 7, 19). The skyline for this festival setup would be represented as: (2, 5, 5, 9, 12, 3, 14, 7, 19), with the x-coordinates sorted in ascending order.

(a) Given the skyline information of n stages for one part of the festival and the skyline information of m stages for another part of the festival, demonstrate how to compute the combined skyline for all m+n stages efficiently, in O(m + n) steps.

(b) Assuming you've successfully solved part (a), propose a Divide-and- Conquer algorithm for computing the skyline of a given set of n stages in the festival. Your algorithm should run in O(n log n) steps.

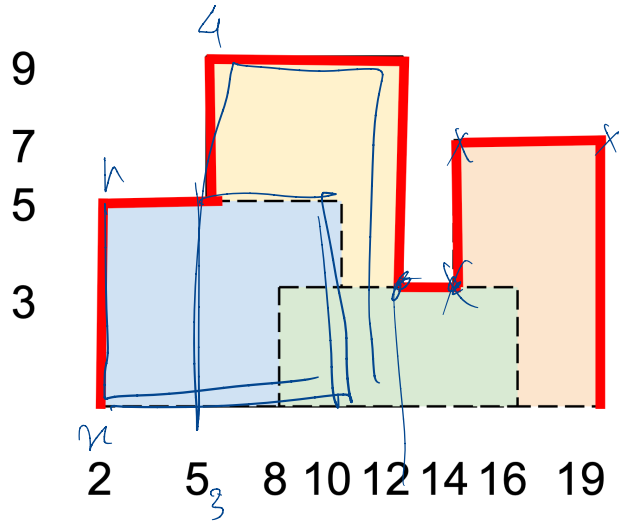

Figure 1: Example of festival stage setup.

Solution:

a)

1. Initialize two pointers namely *left_pointer* and *right_pointer*. Set both their initial values to 0.
2. Then, Initialize two variables namely *left_height* and *right_height*. Set both their initial values to 0.
3. Initialize an empty list *combined_skyline* to store the resulting combined skyline.
4. While *left_pointer* < *len*(*n)* and *right_pointer* < *len(m)*, do the following:
    I. Retrieve the x-coordinate and height values from *n* at the position indicated by *left_pointer* and store them as *left_x* and *left_h*, respectively.
    II. Retrieve the x-coordinate and height values from *m* at the position indicated by *right_pointer* and store them as *right_x* and *right_h*, respectively.
    III. If *left_x* < *right_x*, do as follows:
        i. Update the value to *left_height* = *left_h*.
        ii. *combined_skyline.**append**((left_x,**max**(left_height,right_height)))*. Here the maximum height is chosen between the *left_height* and *right_height*.
        iii. Increment the *left_pointer* by 1 to switch to the next stage on the left side.
    IV. If *right_x* < *left_x*, do as follows:
        i. Update the value to *right_height* = *right_h*.
        ii. *combined_skyline.**append**((right_x,**max**(left_height,right_height)))*. Here the maximum height is chosen between the *left_height* and *right_height*.
        iii. Increment the *right_pointer* by 1 to switch to the next stage on the right side.
    V. If *right_x* = *left_x*, do as follows:
        i. Update the value to *right_height* = *right_h and left_height* = *left_h*.
        ii. *combined_skyline.**append**((left_x,**max**(left_height,right_height)))*. Here the maximum height is chosen between the *left_height* and *right_height*.
        iii. Increment both the *left_pointer* and the *right_pointer* by 1 to switch to the next stages on both the left and the right sides.
5. After the end of the loop above, append the remaining stages in *n* and *m* to the list *combined_skyline*.
6. Finally we return the list *combined_skyline* as the final output containing the merged form of both the skylines.

The above algorithm effectively combines the skylines in a time complexity of O(m + n), where 'm' represents the number of stages in the left skyline and 'n' represents the number of stages in the right skyline. This efficiency is achieved by simply iterating through both the left and right skylines once and merging them to create a unified skyline.

b)

1. If the input *stages* has only 1 stage then we do as follows:
   I. We need to create an empty list *final_skyline* and append the points (L,H) and (R,H) to it. Here L and R are the left and right boundaries respectively and H is the height of the only stage in the list *stages.*
   II. Return *final_skyline*.
2. Else we split the list *stages* into two almost equal sublists by finding the middle point of the list.
3. Now, we recursively compute the skyline for the left and the right half of the stages i.e. *stages[:mid]* and *stages[mid:]*
4. Then, we merge the two skylines obtained from the recursive calls above by using the efficient merging approach explained in **part a** which takes O(m+n) steps.
5. Finally, we return the *final_skyline* obtained in the previous step.

The recurrence relation for the above algorithm is T(n) = 2T(n/2) + O(n), where the O(n) comes from merging the skylines. By using the Master Theorem can conclude that the overall time complexity is $O(n\ log\ n)$ as is falls under *case 2*.

Q9. (Dynamic Programming) Imagine you are organizing a charity event to raise funds for a cause you care deeply about. To reach your fundraising goal, you plan to sell two types of tickets.

You have a total fundraising target of n dollars. Each time someone contributes, they can choose to buy either a 1-dollar ticket or a 2-dollar ticket. Use Dynamic Programming to find the number of distinct combinations of ticket sales you can use to reach your fundraising goal of n dollars?

For example, if your fundraising target is 2 dollars, there are two ways to reach it: 1) sell two 1-dollar tickets; 2) sell one 2-dollar ticket.

(a) Define (in plain English) subproblems to be solved.

(b) Write a recurrence relation for the subproblems

(c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective.

(d) Make sure you specify base cases and their values; where the final answer can be found.

(e) What is the runtime complexity of your solution? Explain your answer.

Solution:

a. Let $OPT[n]$ be the number of distinct combinations required to reach the required fundraising goal of n dollars.

Our choices:

1. Buy a $1 ticket to contribute – OPT[M] = OPT[M-1]
2. Buy a $2 ticket to contribute – OPT[M] = OPT[M-2]

b. The recurrence relation for the subproblems is as follows:

OPT[M] = OPT[M-1] + OPT[M-2]

c. Pseudocode using iteration:

```
function combinationCount(n):
    → Initialize count[n + 1] with all values 0.
    if n == 0:
      return 1
     if n = 1:
      return 1

    count = [0] * (n + 1)
    count[0] = 1
    count[1] = 1

    for i = 2 to n + 1:
       count[i] = count[i − 1] + count[i − 2]

    return count[n]
```

d. Base cases:

1. If n (The target funding) = 0

   There is just one way to make 0 or lesser dollars. i.e. by not picking any dollar. Hence $OPT[0] = 1$

2. If n (The target funding) = 1

   There is just one way to make one dollar. i.e. by not picking the one dollar ticket. Hence $OPT[1] = 1$

   The FINAL answer can be found at $OPT[n]$ (The last element of the array)

e. Our algorithm has a worst case time complexity of $O(n)$. Our algorithm traverses an array of size $n + 1$ which takes $O(n)$ and it takes constant time to add the values from the previous 2 array elements which are nothing but the number of combinations to make the previous 2 totals.

Q10. (Dynamic Programming) Assume a truck with capacity W is loading. There are n packages with different weights, i.e. (w_1, w_2, . . . w_n), and all the weights are integers. The company's rule requires that the truck needs to take packages with exactly weight W to maximize profit, but the workers like to save their energies for after work activities and want to load as few packages as possible. Assuming that there are combinations of packages that add up to weight W, design an algorithm to find out the minimum number of packages the workers need to load.

(a) Define (in plain English) subproblems to be solved.

(b) Write a recurrence relation for the subproblems

(c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of packages to meet the objective.

(d) Make sure you specify base cases and their values; where the final answer can be found.

(e) What is the worst case runtime complexity? Explain your answer.

Solution:

a. Let $OPT[K, W]$ be the minimum number of packages that the workers would need to load to fill a truck of weight W with K ($0 \leq K \leq N$) items.

   Our choices:

   1. Do not pick the current package with weight w_k:
      $OPT[K, W] = 0 + OPT[K - 1, W]$
   2. Pick up the current package with weight w_k :
      $$OPT[K, W] = 1 + OPT[K - 1, W - w_k]$$

b. The recurrence relation for the subproblems is as follows:

$$OPT[K, W] = MIN(\, OPT[K - 1, W], 1 + OPT[K - 1, W - w_k])$$

c. Pseudocode using iteration:

```
function minimumPackages(packages, W, n):

1. Initialize an array Opt[n+1][W+1]
2. Opt[0][0] = 0
   For j from 0 to n:
        Opt[j][0] = 0
   For w from 1 to W:
        Opt[0][w] = ∞
   For i from 1 to n:
      For w from 1 to W:
           Choose = ∞
           Not_choose = Opt[i-1][w]
           If w >= packages[i-1]:
                    Choose = 1+Opt[i-1][w-packages[i-1]]

           Opt[i][w] = min(Choose, Not_choose)
   Return Opt[n][W]
```

d. Base cases:

1. OPT[k,0] = 0, here $0 \leq k \leq n$

2. OPT[0,0] = 0

3. OPT[0,k] = ∞, here $1 \leq k \leq W$

4. if $(w_n > W)$ then $OPT[n][W] = OPT[n-1][W]$

Here W is the total capacity of the truck and K is the list containing the weights of n packages.

The final answer can be found in the last cell of OPT. I.e. OPT[n][W]

e. Our algorithm has a worst case time complexity of $O(n * W)$ where W is the total capacity of the truck and K is the list containing the weights of n packages. As W is not the input size and is just a value , the runtime looks polynomial but since it is a *pseudo polynomial* it should be taken as an exponential runtime. Hence the actual runtime of our algorithm is $\underline{\boldsymbol{O(n * 2^w)}}$.