Greedy

Assignment Solutions

PW SKILLS

Q1. Minimum Cost of ropes

There are given N ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. The task is to connect the ropes with minimum cost. Given N size array arr[] contains the lengths of the ropes.

Input:

n = 5

arr[] = {4, 2, 7, 6, 9}

Output:

62

Explanation:

1. **First, connect ropes 4 and 2, which makes the array {6,7,6,9}. Cost of this operation 4+2 = 6.**
2. **Next, add ropes 6 and 6, which results in {12,7,9}. Cost of this operation 6+6 = 12.**
3. **Then, add 7 and 9, which makes the array {12,16}. Cost of this operation 7+9 = 16.**
4. **And finally, add these two which gives {28}.**
5. **Hence, the total cost is 6 + 12 + 16 + 28 = 62.**

Solution:

C++ Code: https://pastebin.com/bDcdc0tG

Approach:

1. Create a min-heap or priority queue to store the lengths of the ropes. This data structure will automatically maintain the smallest element at the top.
2. Insert all the rope lengths into the min-heap.
3. Initialize a variable "cost" to keep track of the total cost of connecting the ropes. Set it to 0.
4. While the min-heap contains more than one rope, perform the following steps:
   a. Extract the two smallest ropes from the min-heap.
   b. Calculate the cost of merging the two ropes by adding their lengths.
   c. Add the merged length to the cost variable.
   d. Insert the merged rope length back into the min-heap.
5. Once only one rope remains in the min-heap, return the final cost.
6. By repeatedly merging the two smallest ropes, we ensure that we always connect the ropes with the smallest lengths first. This approach guarantees the minimum cost to connect all the r**opes into one.**

Output:

```
Enter the number of ropes: 4
Enter the lengths of the ropes: 4 3 2 6

Minimum cost to connect the ropes: 29


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the number of ropes: 5
Enter the lengths of the ropes: 4 2 7 6 9

Minimum cost to connect the ropes: 62


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(N log N), Because heap operations like insert and extract take O(Logn) time in priority_queue and for all n elements it takes n*logn.

Space complexity: O(N) because we need to store the rope lengths in an array, and the priority queue (min-heap) requires additional space to store the heap structure. Therefore, the space complexity is proportional to the number of ropes, N.

Q2. Minimum Number of Platforms Required for a Railway/Bus Station
Given the arrival and departure times of all trains that reach a railway station, the task is to find the minimum number of platforms required for the railway station so that no train waits. We are given two arrays that represent the arrival and departure times of trains that stop.

Input:
arr[ ] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}
dep[ ] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}
Output: 3
Explanation: There are at-most three trains at a time (time between 9:40 to 12:00)

Solution:
C++ Code: https://pastebin.com/KvH47i9t

Approach:
In the given approach, the greedy strategy is used to find the minimum number of platforms required.

1. The algorithm iterates through the sorted arrays of arrival and departure times in a single pass, making locally optimal decisions at each step.
2. The greedy choice made is to increment the platforms count whenever a train arrives before or at the same time as the previous train departs. This indicates that an additional platform is needed to accommodate the arriving train without any overlap.
3. Conversely, if a train's departure time is earlier than the next train's arrival time, it implies that a platform can be freed up as the departing train no longer occupies it.
4. By making these greedy choices and updating the platforms count accordingly, the algorithm keeps track of the maximum number of platforms needed at any given point. The final result is the maximum count obtained throughout the traversal, representing the minimum number of platforms required to avoid any train waiting.

This greedy approach works because it exploits the fact that if a train arrives before another train departs, it will require a separate platform. By handling the trains in their sorted order, the algorithm ensures that the platforms are allocated optimally to minimize the total number needed.

Output:

```
Minimum number of platforms required: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n log n), where n is the number of trains. This is because the algorithm involves sorting the arrival and departure arrays, which take O(n log n) time, and then traversing the sorted arrays once, which takes O(n) time.

Space complexity: O(1) because it uses a constant amount of extra space to store the variables and does not require any additional data structures that grow with the input size

.Q3. Minimum Fibonacci terms with sum equal to K
Given a number k, find the required minimum number of Fibonacci terms whose sum is equal to k.

Input : k = 17
Output : 3
Explanation: terms are: 13 + 3 + 1 = 17

Input: k = 4
Output: 2
Explanation: Many alternatives are there:
2+2, 1+3, 1+1+2, 1+1+1+1
The minimum number of terms is 2 i.e. in 2+2 and 1+3. So, you can choose any of the two.

Input: k = 3
Output: 1
Explanation: It is possible as:
3, 1+2, 1+1+1
Here, 3 itself is a term in the Fibonacci sequence, hence minimum no. of terms = 1 i.e. 3 itself.

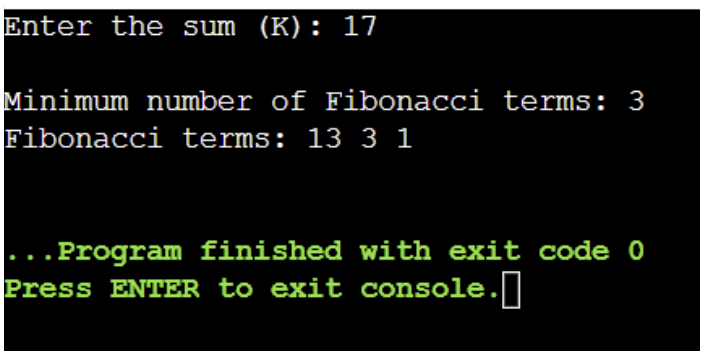
Solution:
C++ Code: https://pastebin.com/WwQDYUVF

Approach:

1. Generate the Fibonacci terms until the sum is reached using the getFibonacciTerms method. The terms are stored in a list.
2. Initialize an empty list to store the selected terms.
3. Initialize variables sum to keep track of the current sum and index to iterate through the Fibonacci terms in descending order.
4. Iterate through the Fibonacci terms in descending order:
   a. If the sum plus the current term is less than or equal to K, add the term to the selected terms list and update the sum.
   b. Decrement the index.
   c. Repeat until the sum equals K or all terms have been considered.
5. Print the minimum number of Fibonacci terms by getting the size of the selected terms list.
6. Print the selected Fibonacci terms using the printFibonacciTerms method.

This approach finds the minimum number of Fibonacci terms with a sum equal to K by starting from the largest term and gradually adding smaller terms until the sum is reached.

Output:



```
Enter the sum (K): 4

Minimum number of Fibonacci terms: 2
Fibonacci terms: 3 1


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the sum (K): 34

Minimum number of Fibonacci terms: 1
Fibonacci terms: 34


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n), where n is the number of Fibonacci terms generated until the sum is reached. This is because the code iterates through the Fibonacci terms once to generate them and then iterates through them again in reverse order to select the terms that contribute to the sum.

Space complexity: O(n), where n is the number of Fibonacci terms generated until the sum is reached. This is because the code stores the Fibonacci terms in a vector/list, which requires memory proportional to the number of terms.

Q4. Divide 1 to n into two groups with minimum sum difference
Given a positive integer n such that n > 2. Divide numbers from 1 to n in two groups such that absolute difference of sum of each group is minimum. Print any two groups with their size in first line and in next line print elements of that group.

Input: 5
Output: 2
5 2
3
4 3 1
Explanation: Here sum of group 1 is 7 and sum of group 2 is 8.
Their absolute difference is 1 which is minimum.
We can have multiple correct answers. (1, 2, 5) and (3, 4) is another such group.

Input: 6
Output: 2
6 4
4
5 3 2 1
Explanation: There can be multiple correct answers. Like [6,3,2] and [5,4,1] is also correct.

Solution:

C++ Code: https://pastebin.com/RaxjrGMg

Approach:

1. We can always divide sum of n integers in two groups such that their absolute difference of their sum is 0 or 1.
2. So sum of group at most differ by 1.
3. We define sum of group1 as half of n elements sum.
4. Now run a loop from n to 1 and insert i into group1 if inserting an element doesn't exceed group1 sum otherwise insert that i into group2.

Output:

```
Enter a positive integer greater than 2: 5

Group 1 (size = 3): 5 2 1
Group 2 (size = 2): 4 3

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter a positive integer greater than 2: 6

Group 1 (size = 3): 6 3 2
Group 2 (size = 3): 5 4 1

...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n log n) because the sorting operation takes O(n log n) time, and the subsequent iteration over the sorted array takes O(n) time.

Space complexity: O(n) because we need additional space to store the input numbers in the nums list, as well as the two groups group1 and group2. The space required grows linearly with the input size.

Q5. Divide cuboid into cubes such that sum of volumes is maximum

Given the length, breadth, height of a cuboid. The task is to divide the given cuboid in minimum number of cubes such that size of all cubes is same and sum of volumes of cubes is maximum. This output should be the side of cube and number of cubes.

Input : l = 2, b = 4, h = 6
Output : 2 6
A cuboid of length 2, breadth 4 and height 6 can be divided into 6 cube of side equal to 2.
Volume of cubes = 6*(2*2*2) = 6*8 = 48.
Volume of cuboid = 2*4*6 = 48.

Input : 1 2 3
Output : 1 6

Solution:

C++ Code: https://pastebin.com/jXQ987X8

Approach:

1. To maximize the volume of the cuboid, we need to ensure that each side is divided evenly among the cubes.
2. The side length of the cubes should be the greatest common divisor (GCD) of the length, breadth, and height of the cuboid.
3. The GCD represents the largest possible side length that can evenly divide all three dimensions.
4. To calculate the number of cubes, we divide the total volume of the cuboid by the volume of one cube.
5. The volume of one cube is given by the cube of the side length (x * x * x).
6. Therefore, the total number of cubes is obtained by dividing the cuboid's volume (l * b * h) by the cube of the side length (x * x * x).
7. This formula allows us to determine the maximum number of equally sized cubes that can fit inside the given cuboid.

Output:

```
Enter the length, width, and height of the cuboid: 2 4 6

Side of cube: 2
Number of cubes: 6


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the length, width, and height of the cuboid: 1 2 3

Side of cube: 1
Number of cubes: 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(1) because the algorithm performs a fixed number of operations regardless of the input size. It directly calculates the side length of the cube and the number of cubes without any loops or recursive calls.

Space complexity: O(1) as well. The algorithm only uses a constant amount of additional space to store the side length and count of the cube in the Cube object. It does not require any dynamically allocated memory that grows with the input size.

Q6. Find minimum number of currency notes and values that sum to given amount
Given an amount, find the minimum number of notes of different denominations that sum up to the given amount.
We may assume that we have infinite supply of notes of values {2000, 500, 200, 100, 50, 20, 10, 5, 1}

Input : 800
Output :

Currency		Count
500	:	1
200	:	1
100	:	1

Input : 2456
Output :

```
Currency   Count
    2000   :    1
    200    :    2
    50     :    1
    5      :    1
    1      :    1
```

Solution:
C++ Code: https://pastebin.com/HB3d5ynd

Approach:

1. We have a fixed set of denominations available: 2000, 500, 200, 100, 50, 20, 10, 5, & 1.
2. We start with the highest denomination and iterate through each denomination.
3. For each denomination, we check if it can be accommodated in the given amount.
4. If the current denomination can be accommodated, we calculate the count of that denomination and update the remaining amount.
5. We repeat this process for each denomination, moving from highest to lowest.
6. Finally, we print the denominations and their respective counts that make up the given amount.

The approach follows a greedy strategy by selecting the highest denomination first and trying to accommodate as many notes of that denomination as possible before moving on to the next lower denomination. This ensures that we use the minimum number of notes to represent the given amount.

Output:

```
Enter the amount: 2456

Currency  Count
2000 : 1
200 : 2
50 : 1
5 : 1
1 : 1


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the amount: 1882

Currency  Count
500 : 3
200 : 1
100 : 1
50 : 1
20 : 1
10 : 1
1 : 2


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the amount: 2019

Currency  Count
2000 : 1
10 : 1
5 : 1
1 : 4


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(1), as the algorithm has a fixed number of iterations (9) that does not depend on the size of the input.

Space complexity: O(1), as the algorithm only uses a fixed amount of space to store the notes and note counters, which does not depend on the size of the input.

Q7. Buy Maximum Stocks if i stocks can be bought on ith day
In a stock market, there is a product with its infinite stocks. The stock prices are given for N days, where arr[i] denotes the price of the stock on the ith day. There is a rule that a customer can buy at most i stock on the ith day. If the customer has k amount of money initially, find out the maximum number of stocks a customer can buy.

Input: price[ ] = { 10, 7, 19 },
k = 45
Output: 4
Explanation:
A customer purchases 1 stock on day 1 for 10 rs, 2 stocks on day 2 for 7 rs each and 1 stock on day 3 for 19 rs.Therefore total of 10, 7 * 2 = 14 and 19 respectively. Hence, total amount is 10 + 14 + 19 = 43 and number of stocks purchased is 4.

Input: price[ ] = { 7, 10, 4 },
k = 100
Output: 6

Solution:
C++ Code: https://pastebin.com/fiLgNPLL

Approach:
The algorithm follows a greedy approach by selecting the stocks with the lowest prices first, ensuring that the customer gets the maximum number of stocks within their available budget. By iteratively choosing the stocks with the lowest prices, the algorithm aims to maximize the total number of stocks bought while respecting the constraint of the available amount of money.
Steps:

1. Create a list of pairs to store the stock prices and corresponding day indices.
2. Sort the list of pairs in ascending order based on the stock prices.
3. Iterate over the sorted list and calculate the maximum number of stocks that can be bought on each day.

4. Keep track of the total number of stocks bought and update the available amount of money accordingly.
5. Return the total number of stocks bought as the result.

Output:

```
Enter the number of days: 3
Enter the stock prices for each day: 10 7 19
Enter the initial amount of money: 45

The maximum number of stocks that can be bought is: 4


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter the number of days: 3
Enter the stock prices for each day: 7 10 4
Enter the initial amount of money: 100

The maximum number of stocks that can be bought is: 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Time complexity: O(n log n), where n is the number of days. This is because the algorithm sorts the prices, which takes O(n log n) time, and then iterates over the sorted prices once, which takes O(n) time. Therefore, the dominant factor is the sorting step.

Space complexity: O(n), where n is the number of days. This is because the algorithm creates a list of pairs to store the prices and day indices, which requires O(n) space.