

COT5405 Analysis of Algorithms

Programming Project-2

TEAM MEMBERS

Sharvani Gouni (541416974)

Sai Varun Reddy Gangasani (37227539)

Anvesh Gupta (89277005)

Contribution of Team members:

Sharvani Gouni

- Worked on implementation of task 4, task 7, task 8
- Worked on report for above strategies
- Executed the files on thunder server

Sai Varun Reddy Gangasani

- worked on implementation of task5, task6a, task6b
- Worked on report for above strategies
- Contributed to creating makefile

Anvesh Gupta

- Worked on implementation of task 1 , task 2 a ,task 2b, task 3,
- Worked on report for the above strategies
- Created random input files for comparative study

ALGORITHM DESIGN TASKS

Problem definition:

Consider a river that you need to cross. There are n platforms to help you cross. You start at platform 0 and must pay a cost each time you jump from a platform. The cost associated with platform i is $\text{cost}[i]$ for $i = 0, \dots, n - 1$. At each jump, you may skip $k - 1$ platforms. From platform i , you can jump up to platform $i + k$. If $i + k \geq n$, you can jump to the other side of the river. Your task is to find the minimum cost to reach the other side of the river. For this you shall solve the following two problems.

Problem1: Given an integer n representing the number of platforms on the river, an array cost of n integers representing the cost to jump from a platform, and an integer k representing the maximum jump length, find the minimum cost to cross the river.

Problem2: Given an integer n representing the number of platforms on the river, an array cost of n integers representing the cost to jump from a platform, an integer k representing the maximum jump length, and an integer m representing the exact number of jumps you must take, find the minimum cost to cross the river.

TASK1: Algorithm1: Design a $\Theta(k^n)$ time brute force algorithm for solving Problem1

Design:

This algorithm uses brute force approach using recursive formula to find minimum cost to cross the river by exploring all the possible combinations

Algorithm:

- The given brute force `brute_force` function is a recursive function, which explores all possible combinations to reach the end of the platforms with minimum cost.
- The base cases check if the platform number is 0 ($sn == 0$) or negative ($sn < 0$), if yes returns the empty path
- The given function iterates from 1 to k and make recursive calls for remaining $sn - i$ platforms
- the minimum cost and the path associated with it will be updated, if there is combination with lower cost

Pseudo code:

```
def brute_force(sn, cost, k):  
    # Base case: If sn (sn is platform number) is 0, return 0 cost and an empty path.  
    if sn == 0:  
        return 0, []  
  
    # Base case: If sn is negative, return positive infinity cost and an empty path.  
    if sn < 0:  
        return float("inf"), []  
  
    # Initialize minimum cost (mi) to positive infinity and an empty path (mp).
```

```

mi = float("inf")
mp = []
# Iterate over possible steps from 1 to k.
for i in range(1, k + 1):
    # Recursively call the function for the remaining sum (sn - i).
    pc, pp = brute_force(sn - i, cost, k)
    # Check if the recursive call is valid and the remaining sum is non-negative.
    if pc is not None and (sn - i) >= 0:
        # Calculate the current cost by adding the cost of the current step.
        cc = pc + cost[sn - i]
        # Update minimum cost and path if the current cost is less than the current minimum.
        if cc < mi:
            mi = cc
            mp = pp + [sn - i]
# Return the minimum cost and corresponding path.
return mi, mp

```

Algorithm Analysis:

Proof by Loop invariant:

Initialization:

At the starting of the algorithm, the function called `brute_force` is called with parameters `sn`(platform number), `cost`, and `k`. The base cases are checked to handle situations where `sn` is already 0 or negative. In these cases, the function returns values accordingly. This ensures that the loop invariant holds for these cases.

Maintenance:

Assume that at the start of iteration '*i*', the loop invariant holds true. During this '*i*'-th iteration, the function makes recursive calls for all possible steps from 1 to `k`. For each recursive call, it calculates the cost (`cc`) and the path (`pp`) based on the recursive result. If the recursive call is valid and the remaining sum after taking the step (`sn - i`) is non-negative, it updates the minimum cost (`mi`) and the corresponding path (`mp`).

Termination:

The loop terminates when `k` iterations are completed. At this point, the function has explored all possible combinations of steps to reach the target sum `n`. The loop invariant continues to hold true because the minimum cost and path have been updated throughout the iterations, ensuring that the minimum cost and path are correctly calculated.

For this task1, we can conclude that loop invariant holds true, ensuring we can find minimum cost to reach the end of platforms and cross the river.

Time complexity:

This algorithm considers all possible combinations of jumps from all platforms to jump the river from one side to the other. Since, it considers every possible combination, this algorithm will have exponential time complexity. As, k is the maximum jump length, in each jump it can jump up to k and there are n platforms in total, hence the final time complexity is $O(k^n)$.

Space complexity:

The space complexity of this algorithm is $O(n)$ due to the recursion stack. Here, the function calls are pushed on to the stack and the space required is directly proportional to the input length. Since, the input is of n platforms, hence the space complexity is $O(n)$.

Conclusion:

It is straightforward approach to implement using the recursive helper function. But its time complexity is high, which makes it inefficient for larger input.

Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task1.py
5 2
1 2 3 4 5
0 1 3
```

Algorithm2: Design a $O(n * k)$ time dynamic programming algorithm for solving Problem1

Optimal substructure property:

Def: $Opt(i)$ – to find the minimum cost of crossing i platforms, with a max jump length of k

Goal: $Opt(n)$: to find the min cost of crossing the n platforms containing from $0, \dots, n-1$, with a max jump length of k

Base condition: $opt(i) = 0$, if $i = 0$

For the i platforms, with maximum jump length k , we will check where can we achieve minimum cost among k jumps

$$opt(i) = \begin{cases} 0 & \text{if } i=0 \\ \min_k \{ opt(i-k) + cost(i-k) \} & i > 0, i \geq k \end{cases}$$

Task 2a : Give a recursive implementation of Alg2 using Memoization

Design:

In this approach we have used dynamic programming with memoization to find the minimum cost to reach the end of a platform sequence. It uses a recursive approach to store and retrieve previously calculated results.

Algorithm:

- The function called find is a recursive dynamic programming function which uses memoization to avoid redundant computations.
- The base cases handles situations when sn is already 0 or negative.
- The function checks if the result for the current state (sn, k) is already memoized. If so, it returns the memoized result.
- It iterates over possible steps from 1 to k and makes recursive calls for the remaining platforms (sn - i).
- The minimum cost and corresponding path are updated based on the recursive results.
- The result for the current state (sn, k) is memoized to avoid recomputing the same values.

Pseudo code:

```
def find(sn, k, cost, memo):
```

```
    # Base case: If sn is platform number is 0, return zero cost and an empty path.
```

```
    if sn == 0:
```

```
        return 0, []
```

```
    # Base case: If sn is negative, return positive infinity cost and an empty path.
```

```
    if sn < 0:
```

```
        return float("inf"), []
```

```
    # Check if the result for the current state (sn, k) is already memoized.
```

```
    if (sn, k) in memo:
```

```
        return memo[(sn, k)]
```

```
    # Initialize an empty path and minimum cost to positive infinity.
```

```
    path = []
```

```
    mi = float("inf")
```

```
    mp = []
```

```
    # Iterate over possible steps from 1 to k.
```

```
    for i in range(1, k + 1):
```

```
        # Recursively call the function for the remaining sum (sn - i).
```

```
        pc, pp = find(sn - i, k, cost, memo)
```

```
        # Check if the recursive call is valid and the remaining sum is non-negative.
```

```

if pc != float("inf") and (sn - i) >= 0:
    # Calculate the current cost by adding the cost of the current step.
    cc = pc + cost[sn - i]
    # Update minimum cost and path if the current cost is less than the current minimum.
    if cc < mi:
        mi = cc
        mp = pp + [sn - i]
# Memoize the result for the current state (sn, k).
memo[(sn, k)] = mi, mp
return memo[(sn, k)]

```

Algorithm Analysis:

Proof by Loop invariant:

Initialization: At the beginning of the loop, mi is initialized to positive infinity, and mp is an empty list. The loop invariant holds because there are no valid paths considered yet and the minimum cost and path are undefined.

Maintenance:

During each iteration of the loop, the algorithm explores all possible steps and calculates the cost and path for the remaining sum. The loop invariant holds because mi is continuously updated to be the minimum cost, and mp is the corresponding path.

Termination:

The loop terminates when all possible steps have been considered. At this point, the algorithm has explored all valid combinations of steps, and mi is the minimum cost to reach sn using at most k steps, and mp is the corresponding path.

In summary, the loop invariant holds true for this approach and ensures that mi is the minimum cost, and mp is the corresponding path to reach the last platform n using at most k steps. The loop terminates when all valid combinations are explored, and the memoization further ensures that previously computed results are reused to avoid redundant computations.

Time complexity:

The time complexity of this algorithm is $\Theta(n \cdot k)$. It iterates through the platforms and jumps up to k for reaching the next platform until it reaches the edge of the river. In, the number of iterations is directly proportional to the n and k .

Space complexity:

The space complexity of this approach is $\Theta(n)$. Since, we are using memoization approach to store the minimum cost for each platform and there are n platforms.

Conclusion:

The recursive implementation of this dynamic programming approach using memoization is relatively easy to implement. However, it may suffer from stack overflow issues for larger input sizes due to the recursion depth.

Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task2a.py
8 4
5 2 6 1 7 2 9 2
0 3 7
```

Task2b:

Design:

This approach implements an iterative (bottoms up) dynamic programming approach to find the minimum cost of reaching the other side of river crossing n platforms, considering a maximum jump length k .

Algorithm:

- Initialize a 2d array called `dp` to store the minimum cost and the list of chosen elements in each state
- base case, when the cost is 0 and the list of chosen elements is empty, if 0 platforms are chosen
- the dynamic programming loop iterates from 1 to n . for each platform, it considers the previous computed values and updates the minimum cost and list of platforms that are associated with minimum cost
- Finally, prints the path associated with minimum cost

Pseudo code:

```
# Initialize a 2D array dp with n+1 rows and 2 columns
dp = [[float("inf"), []] for i in range(n+1)]

# Set the base case: cost is 0 and the list of chosen elements is empty
dp[0] = [0, []]

# Iterate over each element from 1 to n
for i in range(1, n+1):

    # Iterate over the range of indices that can contribute to the current element
    for j in range(max(0, (i-k)), i):

        # Check if the cost of the current path is less than the current minimum cost
        if dp[j][0] + cost[j] < dp[i][0]:

            # Update the minimum cost
            dp[i][0] = dp[j][0] + cost[j]
```



```

# Update the list of chosen elements for the current minimum cost
dp[i][1] = dp[j][1] + [j]
# Print the list of chosen elements for the minimum cost path
print(dp[-1][1])

```

Algorithm Analysis:

Proof of correctness:

Initialization:

At the starting of the algorithm, the base case $i=0$ ensures that $dp[0]$ is correctly initialized. For the base case, the loop invariant holds true.

Maintenance:

For each i , the inner loop iterates over the valid indices j that contribute to the state i . It checks whether the cost of the path through j is less than the current minimum cost at i . If true, it updates $dp[i]$ with the minimum cost and the list of chosen elements.

Termination:

After the loop completes ($i=n$), $dp[n]$ represents the minimum cost to reach the state n . The list of chosen elements for the minimum cost path is stored in $dp[n][1]$. After finding all combinations and reaching the end of platforms sequence, the code ensures to find the path with minimum cost.

For this approach, loop invariant holds true and ensures that it finds path with minimum cost.

Time complexity:

The time complexity of this algorithm is $\Theta(n*k)$. It iterates through n platforms in the reverse order and jumps up to k for reaching the next platform until it reaches the edge of the river. The double loop iterates over each platform and considers up to ' k ' possible jumps for each platform.

Space complexity:

The space complexity of this approach is $O(n)$, as it maintains an array called dp to maintain the minimum cost for each platform and there are n platforms.

Conclusion:

The iterative bottom-up implementation of this algorithm is more challenging than the recursive approach. But it avoids stack overflow issues and is more efficient for larger input files.

Results:

```

PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task2b.py
10 3
4 8 2 9 3 3 6 2 1 9
0 2 5 8

```

Algorithm3: Design a $\Theta(n * \log(n))$ time dynamic programming algorithm for solving Problem1

Design:

This approach implements a dynamic programming solution to find the minimum cost of reaching the end of a platform sequence with a constraint on the maximum jump length k . It uses a priority queue (heap) to efficiently compute the minimum cost to cross the river.

Optimal substructure property:

Def: $\text{Opt}(i)$ – to find the minimum cost of crossing i platforms, with a max jump length of k

Goal: $\text{Opt}(n)$: to find the min cost of crossing the n platforms containing from $0, \dots, n-1$, with a max jump length of k

Base condition: $\text{opt}(i) = 0$, if $i = 0$

For the i platforms, with maximum jump length k , we will check where can we achieve minimum cost among k jumps

$$\text{opt}(i) = \begin{cases} 0 & \text{if } i=0 \\ \min_k \{ \text{opt}(i-k) + \text{cost}(i-k) \} & i > 0, i > k \end{cases}$$

Algorithm:

- The code starts with initializing a min-heap (h) with the costs, indices, and chosen platforms of the last k elements.
- The code iterates over the elements in reverse order from $n-k-1$ to 0 .
- For each element, it updates the h by removing elements that are outside the valid range of $i+k$.
- It calculates the minimum cost and the list of chosen platforms and updates the dp array.
- The code prints the sorted list of chosen elements for the minimum cost path.

Pseudo code:

```
import heapq

# Initialize a 2D array dp to store the minimum cost and the list of chosen elements for each state
dp = [[0, []] for i in range(n)]

# Initialize a min-heap h to efficiently find the minimum cost in the last k elements
h = []

# Initialize the heap with the costs of the last k elements along with their indices and chosen elements
for i in range(k):
```

```

    heapq.heappush(h, [cost[n-1-i], n-1-i, [n-1-i]])
# Iterate over the elements from n-k-1 to 0 in reverse order
for i in range(n-k-1, -1, -1):
    # Remove elements from the heap that are outside the valid range
    while h[0][1] > i + k:
        heapq.heappop(h)
    # Update dp with the minimum cost and the list of chosen elements
    dp[i][0] = h[0][0] + cost[i]
    dp[i][1] = [i] + h[0][2]
    # Push the updated information into the heap
    heapq.heappush(h, [dp[i][0], i, dp[i][1]])
# Print the sorted list of chosen elements for the minimum cost path
print(sorted(dp[0][1]))

```

Algorithm Analysis:

Proof by Loop invariant:

Initialization:

Before the starting of the loop $i = n-k-1$, the heap h is initialized with the costs, indices, and platforms of the last k elements $n-1$ to $n-k$. The dp array is correctly initialized up to index $n-k-1$ with minimum costs and platforms.

Maintenance:

Let, the invariant holds true before the i -th iteration of the loop. The inner loop removes elements from the heap that are outside the valid range $i + k$, maintaining the invariant. The code calculates the minimum cost and the list of chosen platforms for the current state i and updates the dp array. The heap is then updated with this new information. After the i th iteration, the invariant is maintained. The heap h correctly maintains information about the minimum cost paths for the last k elements ending at index $i + k$. The dp array is correctly updated up to index i .

Termination:

After the entire loop completes $i=0$, the heap h contains information about the minimum cost paths for the last k elements ending at index $k-1$. The dp array is correctly updated up to index 0 . The code prints the path of minimum cost, which is stored in $dp[0][1]$.

Time complexity:

In this approach the min heap performs the operations in $\log(n)$, and they are performed for each input which is a size of n . Hence, the time complexity of this algorithm is $\Theta(n\log(n))$.

Space complexity:

The space of this algorithm is $O(n)$. In this `dp[]` stores the minimum cost for each platform and min heap keep tracks of the platform and the costs of them.

Conclusion:

This approach of logarithmic time complexity is more complex than the above algorithms. It requires the use of priority queue to maintain the minimum cost efficiently.

results

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task3.py
10 4
2 8 5 6 9 3 9 1 6 2
0 3 7
```

Algorithm4: Design a $O(n)$ time dynamic programming algorithm for solving Problem1

Design:

This approach implements a dynamic programming solution to efficiently find the minimum cost of reaching the end of a sequence of platforms, given a constraint on the maximum jump length k . It uses a DP array to keep track of the minimum cost of reaching each platform. Additionally, a Parent array is employed to record the preceding platform for each step, enabling the reconstruction of the optimal path. The key aspect of this approach is its efficient computation of minimum costs by considering all feasible jumps within the jump range k , allowing for an effective determination of the least expensive route across the platforms.

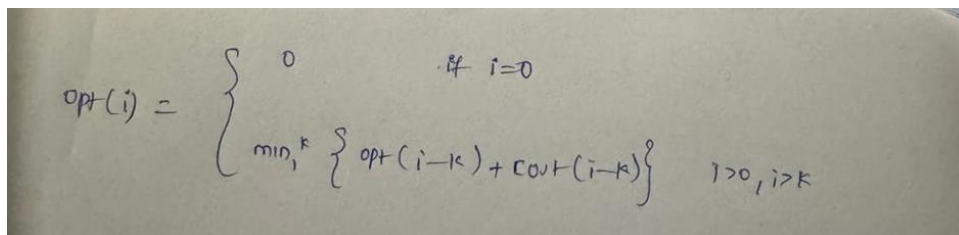
Optimal substructure property:

Def: $Opt(i)$ – to find the minimum cost of crossing i platforms, with a max jump length of k

Goal: $Opt(n)$: to find the min cost of crossing the n platforms containing from $0, \dots, n-1$, with a max jump length of k

Base condition: $opt(i) = 0$, if $i = 0$

For the i platforms, with maximum jump length k , we will check where can we achieve minimum cost among k jumps



$$opt(i) = \begin{cases} 0 & \text{if } i=0 \\ \min_{j \in [i-k, i-1]} \{ opt(j) + cost(j+1, i) \} & i > 0, i > k \end{cases}$$

Algorithm:

- `dp` array and parent array are used to store the minimum cost and path achieved
- We have used double ended queue in this algorithm, for constant operations and efficient tracking

- Pop and append operations are used to remove the platform that is not in the path and add the path that is contributed to minimum cost
- The loop, iterates from 2nd platform, and in backward direction

Pseudo code:

```
def min_cost_to_cross_river(n, cost, k):

    # Initialize arrays to store minimum cost and parent indices
    dp = [0] * n      # Minimum cost to reach each platform
    parent = [-1] * n  # Parent indices to store the path
    dp[0] = cost[0]    # Initial cost for the first platform

    # Initialize deque for efficient window tracking
    dq = deque()

    dq.append(0)       # Start with the first platform

    # Iterate over platforms starting from the second one
    for i in range(1, n):

        # Remove platforms that are out of the valid window
        while dq and dq[0] < i - k:
            dq.popleft()

        # Calculate the minimum cost to reach the current platform
        dp[i] = cost[i] + dp[dq[0]]

        parent[i] = dq[0] # Record the parent index

        # Remove platforms with higher cost from the window
        while dq and dp[i] <= dp[dq[-1]]:
            dq.pop()

        # Add the current platform to the window
        dq.append(i)

    # Find the last platform with the minimum overall cost
    min_cost_to_cross = float('inf')
    last_platform = -1

    for j in range(n-1, max(n-k, 0)-1, -1):
        if dp[j] < min_cost_to_cross:
            min_cost_to_cross = dp[j]
            last_platform = j
```

```

# Reconstruct the path from the last platform to the first

path = []

while last_platform != -1:

    path.append(last_platform)

    last_platform = parent[last_platform]

# Return the minimum cost and the final path achieved

return min_cost_to_cross, path[::-1]

```

Analysis:

Proof of correctness:

Initialization:

Before the loop starts, for the base case, when when $i=0$, there is only one path $dp[0]$. Loop invariant holds true for this case, as there is a single path.

Maintenance:

Lets, assume loop invaraint holds true at ith iteration In the loop, Deque contains platform numbers with valid k range. Here the dp array and parent array are updated with the valid minimum cost and current platform. At the $i+k$ iteration it dp and parent both return the valid minimum costs and paths.

Termination:

When the loop terminates $i=n$, all possible combinations are explored and returns the minimum cost to reach platform n . Loop invariant holds true, as the dp array now contains the minimum cost to reach each platform.

Time complexity:

In this approach, the minimum cost for each platform is given by deque. The operations on the deque have constat time complexity and they are performed for each platform, which results in linear time complexity $\Theta(n)$.

Space complexity:

The space complexity of the algorithm is $O(n)$, primarily due to the use of two arrays of size n : the Dynamic Programming array dp and the Parent array $parent$. Each of these arrays requires linear space relative to the number of platforms, n .

Conclusion:

This approach is most efficient than the above algorithms for problem 1. It requires the use of double ended queue to maintain the minimum cost efficiently. The implementation for this approach is more complex than the above algorithms but provides better complexities of all.

Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task4.py
10 4
34 53 28 31 27 14 26 42 19 10
0 4 8
```

Algorithm5: Design a $\Theta(k^n)$ time brute force algorithm for solving Problem2

Design:

This algorithm is used to implement a brute-force approach to find the minimum cost path to reach the end of a sequence of platforms with constraints on the maximum jump length (k) and the maximum number of jumps (m).

Algorithm:

- Here, the brute force(`brute_force`) function is a recursive function, which explores all possible combinations of steps to reach the end of the platforms upto `sn`(platform number) with `m` jumps and `k-1` maximum jump length.
- The base cases handle, when `m` or `sn` is negative and when both are 0, which returns empty path
- The `brute_force` function iterates over 1 to `k` jumps and makes recursive calls for remaining `sn-i` platforms decrementing `m` in each iteration
- The minimum cost and corresponding path are updated there is a combination with minimum cost of all based on recursive formula

Pseudocode:

```
def brute_force(sn, k, cost, m):
```

```
    # Base case: If m or sn(sn is platform number) is negative, return positive infinity cost and an empty path.
```

```
    if m < 0 or sn < 0:
```

```
        return float("inf"), []
```

```
    #Base case: If both sn and m are zero, return zero cost and an empty path.
```

```
    if sn == 0 and m == 0:
```

```
        return 0, []
```

```
    # Initialize minimum cost (mi) to positive infinity and an empty path (mp).
```

```
    mi = float("inf")
```

```
    mp = []
```

```
    # Iterate over possible steps from 1 to k.
```

```
    for i in range (1, k + 1):
```

```
        # Recursively call the function for the remaining sum (sn - i) and decrement m.
```

```
        cc, pp = brute_force(sn - i, k, cost, m - 1)
```

```

# Check if the recursive call is valid and the remaining sum is non-negative.

if cc is not None and (sn - i) >= 0:
    # Calculate the current cost by adding the cost of the current step.
    cc += cost [sn - i]

    # Update minimum cost and path if the current cost is less than the current minimum.
    if cc < mi:
        mi = cc
        mp = pp + [sn - i]

# Return the minimum cost and corresponding path.
return mi, mp

```

Algorithm Analysis

Proof by Loop invariant:

Initialization:

At the starting of the algorithm, the function called `brute_force` is called with parameters `sn(platform numner)`, `cost`, `k`, `m`. The base cases are checked to handle situations where `sn` is already 0 and `m` is 0 and `sn` and `m` are negative. In these cases, the function returns values accordingly. This ensures that the loop invariant holds for these cases.

Maintenance:

During each iteration of the loop, the algorithm explores all possible steps from 1 to `k` and calculates the cost and path for the remaining platforms. The loop invariant holds true because `mi` is continuously updated to be the minimum cost, and `mp` is the corresponding path.

Termination:

The loop terminates when `m` becomes negative or `sn` becomes negative. At this point, the algorithm has explored all valid combinations of steps, and `mi` is the minimum cost to reach `sn` using at most `m` steps, and `mp` is the corresponding path.

For this task5, we can conclude that loop invariant holds true, ensuring we can find minimum cost to reach the end of platforms and cross the river.

Time complexity:

The time complexity of this algorithm is exponential $O(k^n)$, where ' n ' is the number of platforms. This is because the algorithm explores all possible combinations of jumps, leading to an exponential number of recursive calls.

Space complexity:

The overall space complexity is dominated by the call stack, and the space complexity is $O(n)$, where n is the number of platforms. In this the recursive calls are responsible for the space complexity, the space used by the variables and other constants is relatively small compared to the call stack's contribution.

Conclusion:

The brute force algorithm for Problem2 is like Task1, with the addition of the constraint on the exact number of jumps. It is straightforward to implement but has a high time complexity.

Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task5.py
10 3 5
12 5 18 22 29 16 9 12 32 8
0 1 3 6 9
```

Algorithm6: Design a $\Theta(n * k * m)$ time dynamic programming algorithm for solving Problem2

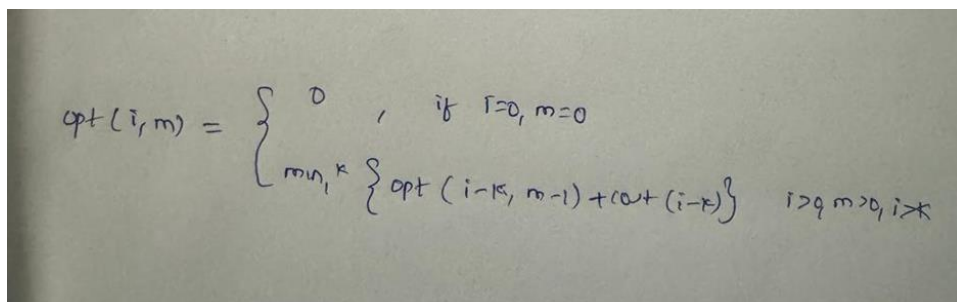
Optimal substructure property:

Def: $\text{Opt}(i, m)$ – to find the minimum cost of crossing i platforms, with a max jump length of k and exact number of jumps m

Goal: $\text{Opt}(n, m)$: to find the min cost of crossing the n platforms containing from $0, \dots, n-1$, with a max jump length of k and exact number of jumps m

Base condition: $\text{opt}(i, m) = 0$, if $i = 0$ and $m = 0$

For the i platforms, with a max jump of k and number of jumps need to take m , we will check where can we achieve minimum cost among k jumps and each, we take a jump we decrease m by 1 and add the associated cost of that platform to minimum cost



$$\text{opt}(i, m) = \begin{cases} 0 & \text{if } i=0, m=0 \\ \min_{1 \leq k \leq i} \{ \text{opt}(i-k, m-1) + \text{cost}(i-k) \} & \text{if } i > 0, m > 0, i \geq k \end{cases}$$

Task 6A: Give a recursive implementation of Algorithm6 using Memoization.

Design:

This algorithm is implemented using dynamic programming approach using memoization, with time complexity of $\Theta(n * k * m)$

Algorithm:

- The function called find is recursive function which explores all possible paths to find the minimum cost with the given constraints k and m
- Base case, returns empty path in if sn and m both are negative and 0
- This approach uses memoization, to store the previously computed values
- The dynamic loop iterates over 1 to k , with recursive exploring the next platform

Pseudocode:

```
def find(sn, k, cost, m, memo): # sn is the platform number
    # Base cases: if m or sn becomes negative, return None
```

```

if m < 0 or sn < 0
    return None, None

# If the result for the current state is already computed, return it
if (sn, m) in memo:
    return memo[(sn, m)]

# Base case: if no steps left and no budget left, return 0 cost and an empty path
if sn == 0 and m == 0:
    return 0, []

path = []
mi = float("inf")
mp = []

# Iterate over possible steps (1 to k)
for i in range(1, k + 1):
    # Recursively find the result for the next state
    pc, pp = find(sn - i, k, cost, m - 1, memo)

    # Check if a valid result is obtained
    if pc is not None and (sn - i) >= 0:
        # Calculate the cost for the current step
        cc = pc + cost[sn - i]

        # Update minimum cost and the corresponding path
        if cc < mi:
            mi = cc
            mp = pp + [sn - i]

# Memoize the result for the current state
memo[(sn, m)] = mi, mp
return memo[(sn, m)]

```

Algorithm Analysis

Proof of correctness:

Initialization:

Base cases: when sn (platform number) and m both are zero and infinity, it returns empty path. In this case, loop variant holds true

Maintenance:

We will assume Invariant holds true before a Recursive Call sn . For a given recursive call with sn , the function explores all possible paths by iteratively considering different jump sizes from 1 to

The recursive calls explore the next states, and the invariant holds for these states according to the assumption. The minimum cost and corresponding path for the current state sn, m is correctly updated based on the results of the recursive calls.

Termination:

The function has recursively explored all possible paths from the starting state $(n, k, cost, m)$ to reach the final state $(0, 0)$. The memo dictionary contains the minimum cost and corresponding path for all states (i, j) within the specified ranges. The result is obtained by retrieving the minimum cost and path from memo $[(0, 0)]$.

Time complexity:

The time complexity of this algorithm is $\Theta(n * k * m)$ since, it uses recursion and explores all possible combinations of platforms, jumps and max jumps must take.

Space complexity:

The space complexity of this approach is $O(n)$ since the memo table stores the minimum cost for n platforms.

Conclusion:

The recursive implementation of the dynamic programming algorithm for Problem2 using memoization is relatively easy to implement. However, it may suffer from stack overflow issues for large input sizes due to the recursion depth.

Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task6a.py
10 3 4
12 34 28 26 19 14 16 32 29 10
0 3 6 9
```

Task 6B : Give an iterative BottomUp implementation of Algorithm6.

Design:

This algorithm uses dynamic programming with bottomup approach to find the minimum cost for reaching the end of platforms with a time complexity of $\Theta(n * k * m)$.

Algorithm:

- In this approach a 3d array is initialized, to store the minimum cost and platform indices among the path associated with minimum cost.
- Base case, when platforms and m are 0, returns empty path
- Nested loops iterate over the maximum jump 1 to m , the platforms (i) , and the valid indices (j) that can contribute to the current element.
- Updates the minimum cost and the list of chosen elements for the current state based on the previous state.

Pseudocode:

Initialize a 3D array dp to store the minimum cost and the list of chosen elements for each state

```

dp = [[float("inf"), []] for i in range(m + 1)] for j in range(n + 1)]

# Set the base case: cost is 0 and the list of chosen elements is empty for the initial state (0 elements chosen)
dp[0][0] = [0, []]

# Iterate over the budget (l) from 1 to m
for l in range(1, m + 1):
    # Iterate over each element (i) from 1 to n
    for i in range(1, n + 1):
        # Iterate over the valid indices (j) that can contribute to the current element
        for j in range(max(0, i - k), i):
            # Update the minimum cost and the list of chosen elements for the current state
            if dp[j][l - 1][0] + cost[j] < dp[i][l][0]:
                dp[i][l][0] = dp[j][l - 1][0] + cost[j]
                dp[i][l][1] = dp[j][l - 1][1] + [j]

# Print the list of chosen elements for the minimum cost path with the given budget
print(dp[n][m][1])

```

Algorithm Analysis:

Proof of correctness:

Initialization:

The base case $dp[0][0]$ returns empty path. SO, it loop invariant holds true for this case

Maintenance:

We assume loop Invariant Holds Before an Iteration 1. For a given iteration with 1, the outer loop iterates over each element i and considers valid indices j . The inner loops correctly update the minimum cost and the list of platforms on that path for the current state (i, l) based on the results of the previous states $(j, l-1)$. The minimum cost and corresponding path are updated based on the optimal choice among different paths.

Termination:

The outer loop iterates over i from 1 to m , ensuring that the dp array correctly stores the minimum cost and path for all states $(i, 1)$ within the specified ranges. The final result is obtained by retrieving the minimum cost and path from $dp[n][m]$.

Time complexity:

The time complexity of this approach is $\Theta(n * k * m)$ as the implementation uses two loops, outer loop in reverse order from $n-1$ to 0 and inner loop from 1 to k and explores all possible combinations considering m .

Space complexity:

The space complexity of this approach is $O(n*m)$, as the dp is 3d array with dimensions $m*n*2$ to store and in each entry, it stores a specific combination of the current platform.

Conclusion:

The iterative bottom-up implementation of the dynamic programming algorithm for Problem2 is more challenging to implement than the recursive version. However, it avoids stack overflow issues and is more efficient for large input sizes.

Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task6b.py
10 2 6
12 6 17 29 25 32 17 21 10 16
0 1 2 4 6 8
```

Algorithm7: Design a $\Theta(n * m * \log(n))$ time dynamic programming algorithm for solving Problem2

Design:

The algorithm uses dynamic programming to find the minimum cost to cross a river, given constraints on jump length and the number of jumps. It employs a 2D table for calculating minimum costs and tracking the optimal path. The algorithm iteratively updates this table considering possible jumps within the given range and reconstructs the path after determining the minimum cost. This approach efficiently addresses the problem by systematically exploring and recording cost-effective routes.

Optimal substructure property:

Def: $\text{Opt}(i, m)$ – to find the minimum cost of crossing i platforms, with a max jump length of k and exact number of jumps m

Goal: $\text{Opt}(n, m)$: to find the min cost of crossing the n platforms containing from $0, \dots, n-1$, with a max jump length of k and exact number of jumps m

Base condition: $\text{opt}(i, m) = 0$, if $i = 0$ and $m = 0$

For the i platforms, with a max jump of k and number of jumps need to take m , we will check where can we achieve minimum cost among k jumps and each we take a jump we decrease m by 1 and add the associated cost of that platform to minimum cost

$$\text{opt}(i, m) = \begin{cases} 0 & \text{if } i=0, m=0 \\ \min_{1 \leq k \leq i} \{ \text{opt}(i-k, m-1) + \text{cost}(i-k) \} & \text{if } i > 0, m > 0, i \geq k \end{cases}$$

Algorithm:

- The code 2d array `dp[i][j]` represents the minimum cost to reach platform `i` using exactly `j` jumps.
- The min-heap is used to efficiently find the minimum cost among the possible previous platforms within the jump limit.
- The path array is updated to keep track of the chosen platforms for each state, to keep track of optimal path.

Pseudocode:

```
import heapq
```

```
def min_cost_to_cross_river(n, cost, k, m):
```

```
    # Initialize a 2D array dp to store the minimum cost for each state
```

```
    dp = [[float('inf')] * (m+1) for _ in range(n+1)]
```

```
    dp[0][0] = 0 # Base case: zero cost for zero jumps
```

```
    # Initialize a 2D array path to store the chosen platforms for each state
```

```
    path = [[[[] for _ in range(m+1)] for _ in range(n+1)]
```

```
    # Iterate over the platforms
```

```
    for i in range(1, n):
```

```
        # Iterate over the number of jumps
```

```
        for j in range(1, m+1):
```

```
            min_heap = [] # Use a min-heap to keep track of minimum cost
```

```
            # Iterate over the possible previous platforms within the jump limit
```

```
            for p in range(max(0, i-k), i):
```

```
                heapq.heappush(min_heap, (dp[p][j-1] + cost[i], p))
```

```
            min_cost, min_index = heapq.heappop(min_heap)
```

```
            dp[i][j] = min_cost # Update minimum cost
```

```
            path[i][j] = path[min_index][j-1] + [min_index] # Update chosen platforms
```

```
    return path[n-1][m]
```

Algorithm Analysis

Proof of correctness:

Initialization

At the beginning, the `dp` array is initialized such that `dp[0][0] = 0`, meaning the minimum cost to reach the initial platform (0) using 0 jumps is 0. Loop invariant holds true for this case.

Maintenance:

Assuming that the loop invariant holds before the current iteration, we need to show that it holds after the iteration. The code iterates through each platform i and each jump count j , and for each, it considers the cost of reaching that platform from the valid platforms using k jumps. The minimum cost to reach platform i using exactly j jumps is updated based on the minimum cost from the valid previous platforms. The path array is also updated accordingly. The loop invariant is maintained because, after the current iteration, $dp[i][j]$ correctly represents the minimum cost to reach platform i using exactly j jumps, and $path[i][j]$ records the corresponding path.

Termination:

After the termination of the loop, the dp array represents the minimum cost to reach any platform using any valid number of jumps. The final result is obtained from the path array, which contains the path corresponding to the minimum cost to cross the river using exactly m jumps.

Hence, loop invariant holds true for this case.

Time Complexity:

The time complexity of this approach is $\Theta(n * m * \log(n))$.

Space Complexity:

The space complexity of this approach $O(n*m)$, as this approach used 2d array with $n*m$ dimensions to find the minimum cost.

Conclusion:

The implementation of the dynamic programming algorithm for Problem2 with a logarithmic time complexity is more complex than the previous algorithms. It requires the use of a priority queue (heap) to maintain the minimum cost efficiently.

Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python task7.py
8 3 4
12 34 24 18 26 22 27 19
[0, 3, 5, 7]
```

Algorithm8: Design a $\Theta(n * m)$ time dynamic programming algorithm for solving Problem2

Design: This algorithm uses dynamic programming approach to reduce the time complexity to $\Theta(n * m)$

Optimal substructure property:

Def: $Opt(i,m)$ – to find the minimum cost of crossing i platforms, with a max jump length of k and exact number of jumps m

Goal: $Opt(n, m)$:to find the min cost of crossing the n platforms containing from $0, \dots, n-1$, with a max jump length of k and exact number of jumps m

Base condition: $opt(i,m) = 0$, if $i = 0$ and $m=0$

For the i platforms, with a max jump of k and number of jumps need to take m , we will check where can we achieve minimum cost among k jumps and each we take a jump we decrease m by 1 and add the associated cost of that platform to minimum cost

$$opt(i, m) = \begin{cases} 0 & \text{if } i=0, m=0 \\ \min_{1 \leq k \leq m} \{ opt(i-k, m-1) + cost(i-k) \} & \text{if } i>0, m>0, i \geq k \end{cases}$$

Algorithm:

- In this approach, we Initialize two 2D arrays, dp for minimum costs and path for the corresponding paths.
- Base Case: Set the base case where the cost is 0 and the list of platforms is empty for the initial state
- Loops are used based on number of platforms, m and maximum number of jumps k
- Return the minimum cost and the corresponding path to reach the end with the given maximum jumps.

Pseudocode:

```
def min_cost_to_cross_river(n, cost, k, m):
```

```
    # Initialize a 2D array to store minimum costs
```

```
    dp = [[float('inf')] * (m+1) for _ in range(n)]
```

```
    # Initialize a 2D array to store the minimum path
```

```
    path = [[[ ] for _ in range(m+1)] for _ in range(n)]
```

```
    # Set the base case: cost is 0 and the list of chosen elements is empty for the initial state (0 elements chosen)
```

```
    dp[0][0] = 0
```

```
    # Iterate over the platforms
```

```
    for i in range(n):
```

```
        # Iterate over the jumps
```

```
        for j in range(1, m+1):
```

```
            # Iterate over the valid indices (p) that can contribute to the current element
```

```
            for p in range(max(0, i-k), i):
```

```
                # Update the minimum cost and the list of chosen elements for the current state
```

```
                if dp[i][j] > dp[p][j-1] + cost[i]:
```

```
                    dp[i][j] = dp[p][j-1] + cost[i]
```

```
                # Update the path array
```

```
                path[i][j] = path[p][j-1] + [p]
```



```
# Return the minimum cost and path to reach the end with the given maximum jumps
```

```
return dp[n-1][m], path[n-1][m]
```

Analysis:**Proof of correctness:**

Initialization:

At the beginning of the code, when $i = 0$, the base case is set to $dp[0][0] = 0$. This ensures that the minimum cost for reaching the initial platform with zero and the path is an empty list. The loop invariant holds true at the beginning of the loop.

Maintenance:

Assume that the loop invariant holds true at the start of an iteration $dp[p][j-1]$ and $path[p][j-1]$ represent the minimum cost and path to reach platform p using exactly $j-1$ jumps for some p . During the iteration, the algorithm explores valid platforms p that could contribute to the current state $dp[i][j]$ and $path[i][j]$. For each such p , it updates the minimum cost and the path if it finds a better solution. The loop invariant is maintained as the minimum cost and corresponding path for the current state are correctly computed.

Termination:

After completing all iterations of the loop, the algorithm has filled the dp and $path$ arrays for all valid states platforms and m . The final result is stored in $dp[n-1][m]$ and $path[n-1][m]$, representing the minimum cost and path to reach the end platform using exactly m jumps.

Hence, the loop invariant at the end of the loop ensures the correctness of the solution.

Time complexity:

The time complexity of this approach is $\Theta(n * m)$ since, the code iterates through two loops jumps and platforms with constant time operations.

Space complexity:

The space complexity of this approach is $O(n*m)$, since it uses 2d array called dp to store the minimum cost to reach with maximum jumps constraint m

Conclusion:

The linear time complexity dynamic programming this algorithm is the most efficient among the algorithms for Problem2. It requires the use of a double-ended queue (deque) to maintain the minimum cost efficiently. The implementation is more complex than the previous algorithms but offers the best performance

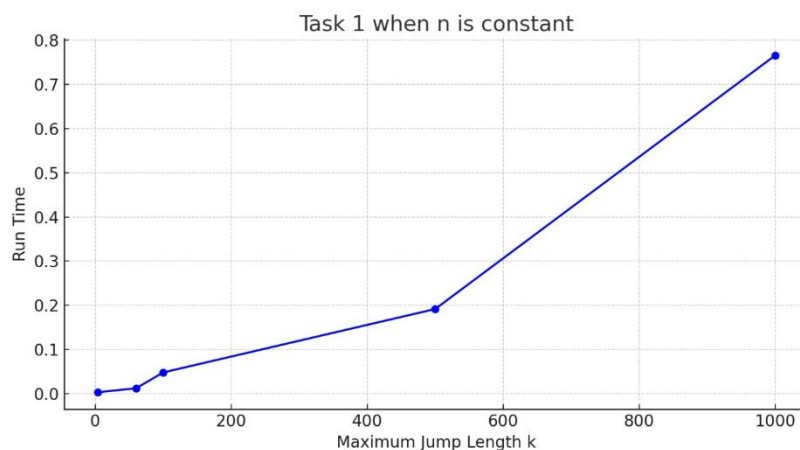
Results:

```
PS C:\Users\SHRAVANI\OneDrive\Desktop\aoaproject2> python
task8.py
10 4 4
28 22 12 29 35 16 20 18 9 30
0 2 5 8
```

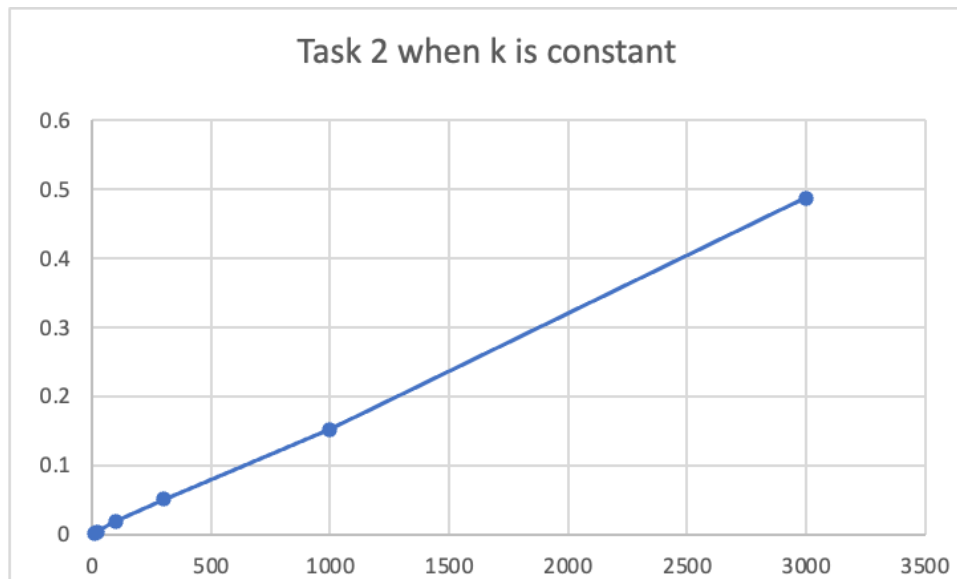
Experimental comparative study

For the algorithms we have experimented with large input sizes and randomly generated inputs, the algorithm's execution time is measured that produced comparison graphs, below are the performance graphs.

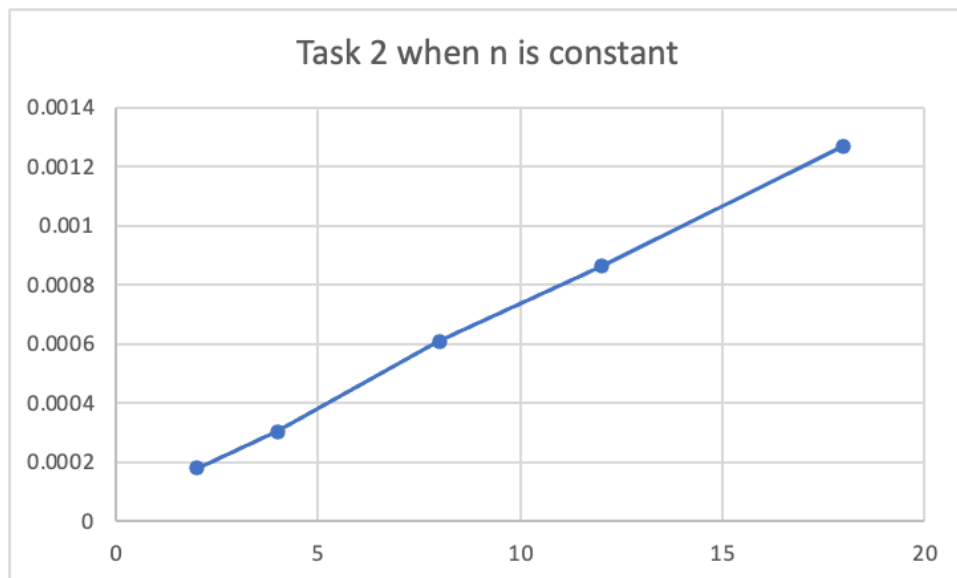
In the provided graphs x-axis represents number of platforms and y-axis represents the execution time

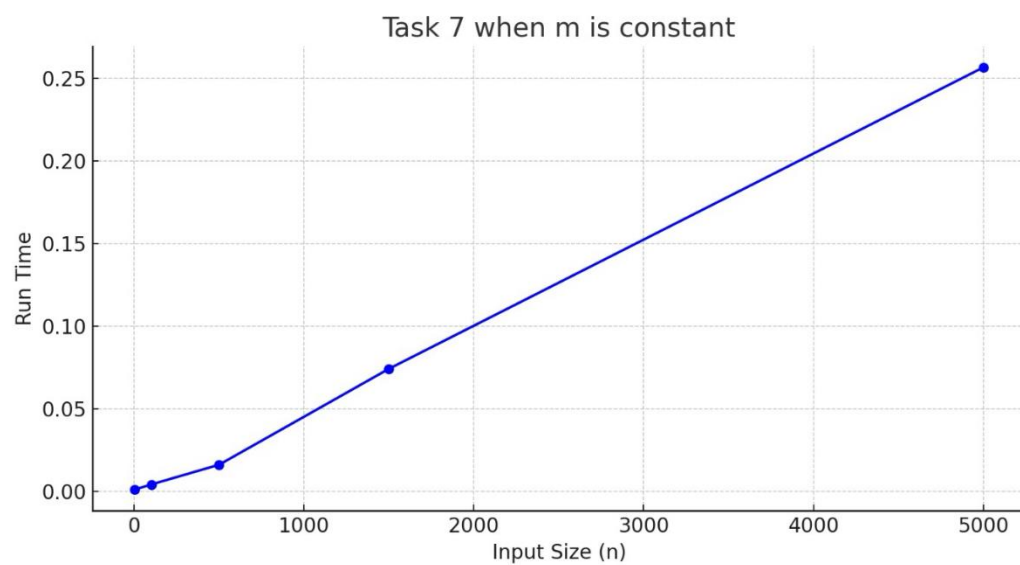
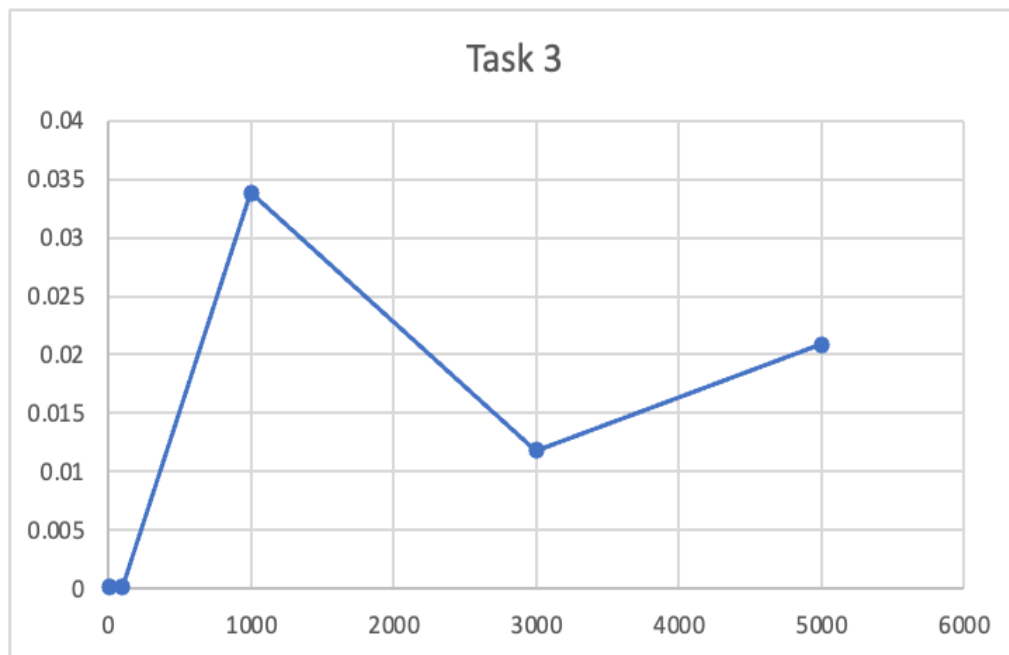


Below is the graph for task2a

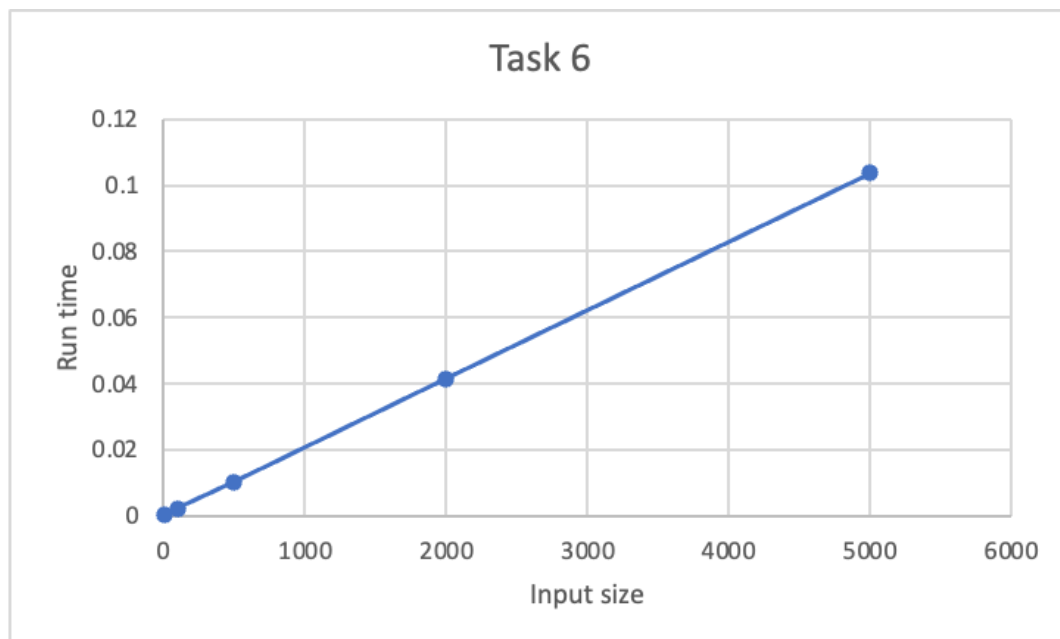
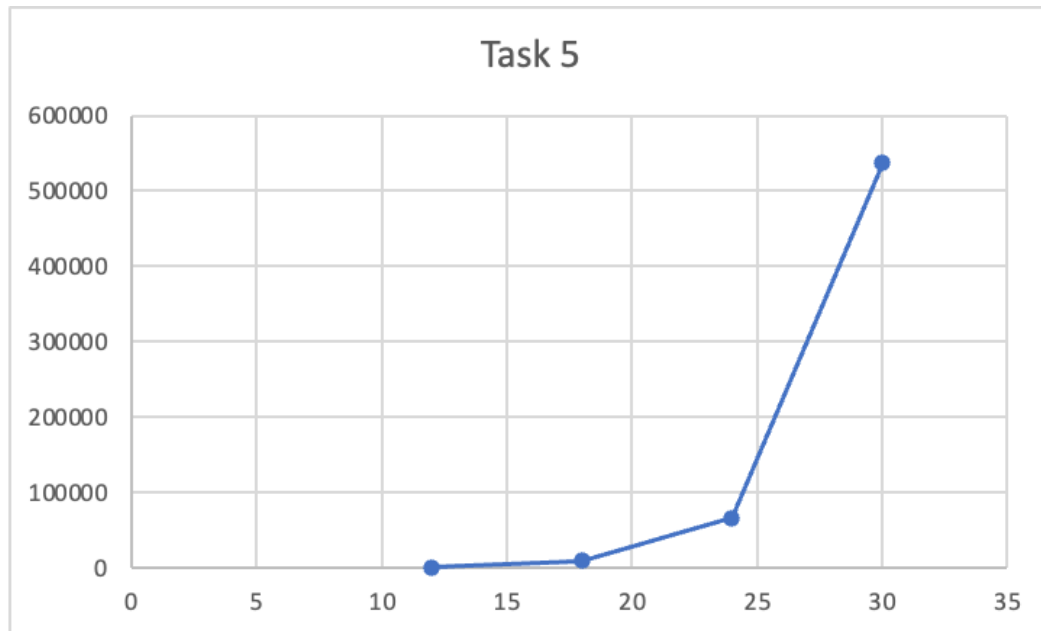


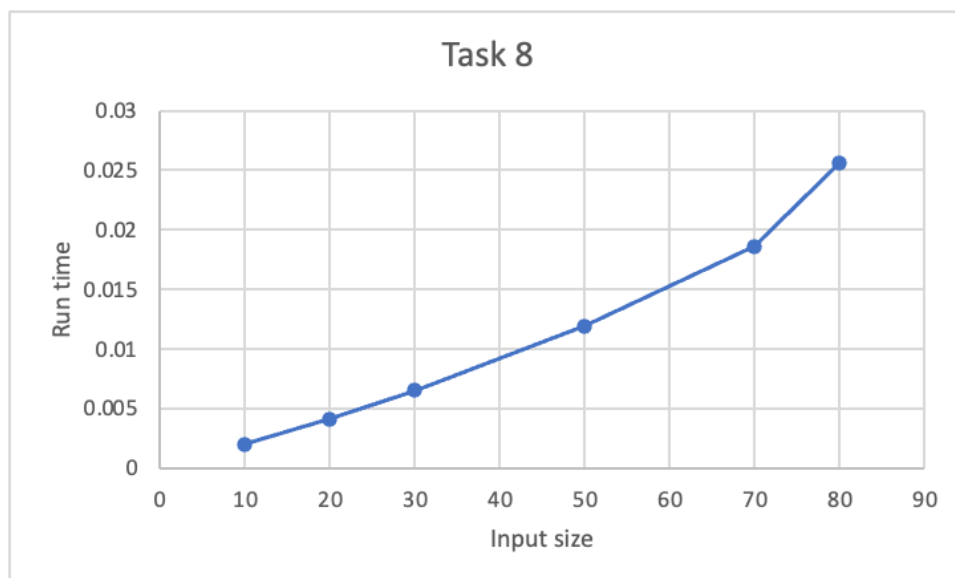
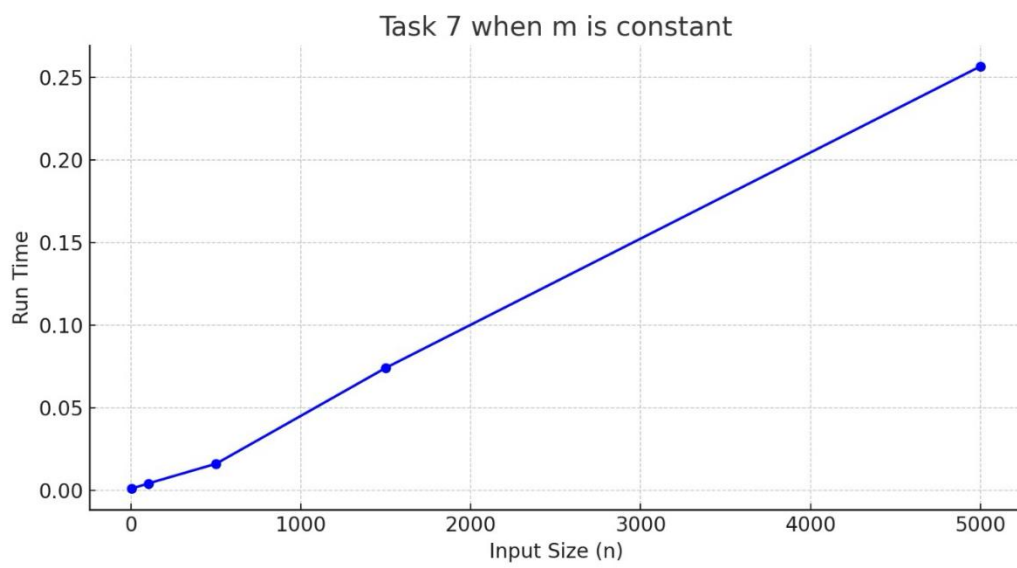
Below graph represents number of platforms used vs execution time of task2b





For the task5 graph, x axis is execution time and y axis is input size





Conclusion:

Analysis of Problem 1 Tasks (Tasks 1 to 4)

Task 1 ($\Theta(kn)$)

Complexity: Linear time complexity but with a multiplier 'k', which significantly influences the runtime.

Behavior: The runtime increases linearly with the size of 'n'. However, different values of 'k' can change the slope of this increase, leading to variations in efficiency.

Task 2 ($\Theta(nk)$)

Complexity: Similar to Task 1 but with different implementation details.

Behavior: Shows a linear relationship between runtime and input size. The factor 'k' affects the slope of the runtime curve, similar to Task 1.

Task 3 ($\Theta(n\log(n))$)

Complexity: Logarithmic-linear, more efficient than a purely linear complexity.

Behavior: Exhibits a slower growth rate in runtime compared to linear complexities. Suitable for larger datasets as it scales better.

Task 4 ($\Theta(n)$)

Complexity: Purely linear, one of the most efficient for large-scale problems.

Behavior: Direct and predictable increase in runtime with the size of 'n'. It's efficient and manageable for practical applications.

Analysis of Problem 2 Tasks (Tasks 5 to 8)

Task 5 ($\Theta(kn)$)

Complexity: Similar to Task 1, linear with a multiplier 'k'.

Behavior: Runtime increases linearly with 'n', heavily influenced by 'k'. The higher 'k' is, the steeper the increase in runtime.

Task 6 ($\Theta(nkm)$)

Complexity: Linear but influenced by three factors: 'n', 'k', and 'm'.

Behavior: Shows a more complex interplay between the factors. Each factor contributes to increasing the runtime, making it more sensitive to changes in any of the parameters.

Task 7 ($\Theta(nm\log(n))$)

Complexity: Combines linear and logarithmic factors, leading to a more complex behavior.

Behavior: The logarithmic component slows down the rate of increase, making it more efficient than purely linear algorithms for large 'n'.

Task 8 ($\Theta(nm)$)

Complexity: Linear, affected by both 'n' and 'm'.

Behavior: Exhibits a direct proportionality to 'n' and 'm'. The runtime increases linearly, making it predictable and scalable.

Individual Task Analysis

Each task demonstrates unique characteristics based on its time complexity and the factors influencing it. Tasks with linear complexities (like Tasks 4 and 8) are generally more scalable and predictable, making them suitable for larger datasets. Tasks with exponential or multi-factor linear complexities (like Tasks 1, 5, and 6) can quickly become inefficient as the size of the input or the complexity factors increase. Tasks that incorporate logarithmic components (like Tasks 3 and 7) offer a balance between efficiency and scalability, often performing well with large datasets.

In conclusion, the selection of an appropriate algorithm for a given problem should consider not only the time complexity but also the specific characteristics of the dataset and the computational resources available. Understanding these nuances is crucial for effective algorithm implementation in practical scenarios.