IEEEXtreme Türkiye Camp 24' Program

Day 2

1 Search Algorithms

It may be necessary to determine if an array or solution set contains a specific data, and we call this finding proccess **searching**. In this article, three most common search algorithms will be discussed: linear search, binary search, and ternary search.

This visualization may help you understand how the search algorithms work: Link.

1.1 Linear Search

Simplest search algorithm is *linear search*, also know as *sequential search*. In this technique, all elements in the collection of the data is checked one by one, if any element matches, algorithm returns the index; otherwise, it returns -1.

ity is $O(N)$

0	1	2	3	4	5	6	7	8	9	Search Key: 70
7	29	48	53	63	70	76	89	94	96	7 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	29 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	48 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	53 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	63 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	70 = 70, return 5

Figure 1: Example for linear search

```
int linearSearch(int *array, int size, int key){
    for (int i=0; i < size; i++)
        if (array[i] == key)
            return i;
    return -1;
}
```

1.2 Binary Search

We know linear search is quite a slow algorithm because it compares each element of the set with search key, and there is a high-speed searching technique for **sorted** data instead of linear search, which is **binary search**. After each comparison, the algorithm eliminates half of the data using the sorting property.

We can also use binary search on increasing functions in the same way.

Procedure:

- Compare the key with the middle element of the array,
- If it is a match, return the index of middle.
- If the key is bigger than the middle, it means that the key must be in the right side of the middle. We can eliminate the left side.
- If the key is smaller, it should be on the left side. The right side can be ignored.

Complexity:

$$\begin{aligned} T(N) &= T(N/2) + O(1) \\ T(N) &= O(logN) \end{aligned}$$

L				M					R	Search Key: 70
7	29	48	53	63	70	76	89	94	96	63 < 70, shift L

					L		M		R	Search Key: 70
7	29	48	53	63	70	76	89	94	96	89 > 70, shift R

					L M	R				Search Key: 70
7	29	48	53	63	70	76	89	94	96	70 = 70, return 5

Figure 2: Example for binary search

```
int binarySearch(int *array, int size, int key){
    int left = 0, right = size, mid;

    while (left < right){
        mid = (left + right) / 2;

        if (array[mid] >= key)
            right = mid;
        else
            left = mid + 1;
    }
    return array[left] == key ? left : -1 ;
}
```

1.3 Ternary Search

Suppose that we have a unimodal function, $f(x)$, on an interval $[l, r]$, and we are asked to find the local minimum or the local maximum value of the function according to the behavior of it.

There are two types of unimodal functions:

1. The function, $f(x)$ strictly increases for $x \leq m$, reaches a global maximum at $x = m$, and then strictly decreases for $m \leq x$. There are no other local maxima.

2. The function, $f(x)$ strictly decreases for $x \leq m$, reaches a global minimum at $x = m$, and then strictly increases for $m \leq x$. There are no other local minima.

In this document, we will implement the first type of unimodal function, and the second one can be solved using the same logic.

Procedure:

1. Choose any two points m_1, and m_2 on the interval $[l, r]$, where $l < m_1 < m_2 < r$.

2. If $f(m_1) < f(m_2)$, it means the maxima should be in the interval $[m_1, r]$, so we can ignore the interval $[l, m_1]$, move l to m_1

3. Otherwise, $f(m_1) \geq f(m_2)$, the maxima have to be in the interval $[l, m_2]$, move r to m_2

4. If $r - l < \epsilon$, where ϵ is a negligible value, stop the algorithm, return l. Otherwise turn to the step 1.

m_1 and m_2 can be selected by $m_1 = l + (r - l)/3$ and $m_2 = r - (r - l)/3$ to avoid increasing the time complexity.

Complexity:

$$\begin{aligned} T(N) &= T(2 \cdot N/3) + O(1) \\ T(N) &= O(logN) \end{aligned}$$

```
double f(double x);

double ternarySearch(double left, double right, double eps=1e-7){
    while (right - left > eps){
        double mid1 = left + (right - left) / 3;
        double mid2 = right - (right - left) / 3;

        if (f(mid1) < f(mid2))
            left = mid1;
        else
            right = mid2;
    }
    return f(left);
}
```

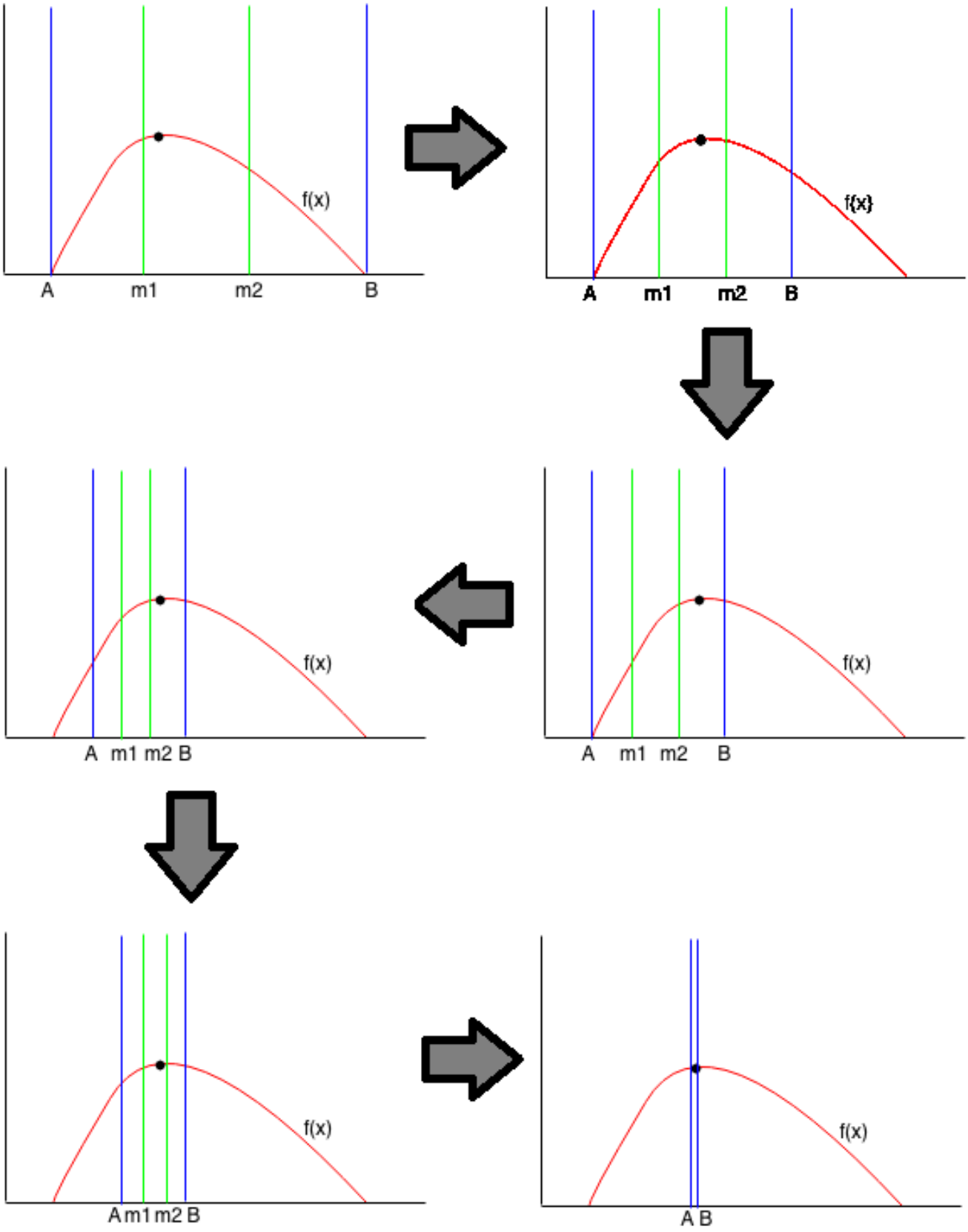


Figure 3: Example for ternary search

IEEEXtreme Türkiye Camp 24' Program

Day 4

1 Introduction

Next section is about the `Greedy Algorithms` and `Dynamic Programming`. It will be quite a generous introduction to the concepts and will be followed by some common problems.

2 Greedy Algorithms

A **greedy algorithm** is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.
The difficulty in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It is often difficult to argue that a greedy algorithm works.

2.1 Coin Problem

We are given a value V, if we want to make change for V cents, and we have an infinite supply of each of the coins = { C1, C2, .. , Cm } valued coins (sorted in descending order), what is the minimum number of coins to make the change?

2.1.1 Solution

Approach:

```
1- Initialize the result as empty.
2- Find the largest denomination that is
smaller than amount.
3- Add found denomination to the result. Subtract
the value of found denomination from amount.
4 - If amount becomes 0, then print the result.
Else repeat the steps 2 and 3 for the new value of amount
```

```
1    def min_coins(coins, amount):
2        n = len(coins)
3        for (i in range(0, n)):
4            while (amount >= coins[i]):
5                #while loop is needed since one coin can be used multiple times
6                amount = amount - coins[i]
7                print(coins[i])
```

For example, if the coins are the euro coins (in cents) 200,100,50,20,10,5,2,1 and the amount is 548. Then the optimal solution is to select coins 200+200+100+20+20+5+2+1 whose sum is 548.

200	100	50	20	10	5	2	1	remaining	current_total
X								348	200
X								148	400
	X							48	500
			X					28	520
			X					8	540
					X			3	545
						X		1	547
							X	0	548

In the general case, the coin set can contain any kind of coins and the greedy algorithm does not necessarily produce an optimal solution.
We can prove that a greedy algorithm does not work by showing a counterexample where the algorithm gives a wrong answer.
In this problem we can easily find a counterexample: if the coins are 6,5,2 and the target sum is 10, the greedy algorithm produces the solution 6+2+2 while the optimal solution is 5+5.

2.2 Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: We are given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, thus the minimum possible deadline for any job is 1. How do we maximize total profit if only one job can be scheduled at a time.

2.2.1 Solution

A Simple Solution is to generate all subsets of the given set of jobs and check an individual subset for the feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential. This is a standard Greedy Algorithm problem. Following is the algorithm:

```
1- Sort all jobs in decreasing order of profit.
2- Initialize the result sequence as the first job in sorted jobs.
3- Do the following for the remaining n-1 jobs
      If the current job can fit in the current result sequence without missing the
         deadline, add the current job to the result.
      Else ignore the current job.
```

```
# sample job : ['x', 4, 25] -> [job_id, deadline, profit]
# jobs: array of 'job's
def print_job_scheduling(jobs, t):
    n = len(jobs)

    # Sort all jobs according to decreasing order of profit
    for (i in range(n)):
        for (j in range(n - 1 - i)):
            if (jobs[j][2] < jobs[j + 1][2]):
                jobs[j], jobs[j + 1] = jobs[j + 1], jobs[j]

    # To keep track of free time slots
    result = [False] * t
    # To store result (Sequence of jobs)
    job = ['-1'] * t

    # Iterate through all given jobs
    for (i in range(len(jobs))):

        # Find a free slot for this job
        # (Note that we start from the last possible slot)
        for (j in range(min(t - 1, jobs[i][1] - 1), -1, -1)):

            # Free slot found
            if (result[j] is False):
                result[j] = True
                job[j] = jobs[i][0]
                break
    print(job)
```

2.3 Tasks and Deadlines

Let us now consider a problem where we are given n tasks with the durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn d - x points where d is the task's deadline and x is the moment when we finish the task. What is the largest possible total score we can obtain?
For example, suppose that the tasks are as follows:

$$\{task, duration, deadline\}$$
$$\{A, 4, 2\}, \{B, 3, 5\}, \{C, 2, 7\}, \{D, 4, 5\}$$

In this case, an optimal schedule for the tasks is C, B, A, D.
In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10.
Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks sorted by their durations in increasing order.

The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks.

2.3.1 Solution

```
def order_tasks(task):
    n = len(tasks)

    # Sort all task according to increasing order of duration
    for (i in range(n)):
        for (j in range(n - 1 - i)):
            if (tasks[j][1] > tasks[j + 1][1]):
                tasks[j], tasks[j + 1] = tasks[j + 1], tasks[j]

    point = 0
    current_time = 0
    # Iterate through all given tasks and calculate point
    for (i in range(len(tasks))):
        current_time = current_time + tasks[i][1]
        point = point + (tasks[i][2] - current_time)

    print(point)
```

2.4 Minimizing Sums

Where we are given n numbers and our task is to find a value x that minimizes the sum:

$$|a_1 - x|^c + |a_2 - x|^c + ... + |a_n - x|^c.$$

We focus on the cases c = 1 and c = 2.

- **Case c = 1**

In this case, we should minimize the sum

$$|a_1 - x| + |a_2 - x| + ... + |a_n - x|$$

For example, if the numbers are [1,2,9,2,6], the best solution is to select x = 2 which produces the sum

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

In the general case, the best choice for x is the median of the numbers, i.e., the middle number after sorting. For example, the list [1,2,9,2,6] becomes [1,2,2,6,9] after sorting, so the median is 2.
The median is an optimal choice, because if x is smaller than the median, the sum becomes smaller by increasing x, and if x is larger then the median, the sum becomes smaller by decreasing x. Hence, the optimal solution is that x is the median. If n is even and there are two medians, both medians and all values between them are optimal choices.

- **Case c = 2**

In this case, we should minimize the sum:

$$(a_1 - x)^2 + (a_2 - x)^2 + ... + (a_n - x)^2.$$

For example, if the numbers are [1,2,9,2,6], the best solution is to select x = 4 which produces the sum:

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46$$

In the general case, the best choice for x is the average of the numbers. In the example the average is (1 + 2 + 9 + 2 + 6)/5 = 4.
This result can be derived by presenting the sum as follows:

$$nx^2 - 2x(a_1 + a_2 + ... + a_n) + ({a_1}^2 + {a_2}^2 + ... + {a_n}^2)$$

The last part does not depend on x, so we can ignore it. The remaining parts form a function

$$nx^2 2xs$$

$$s = a_1 + a_2 + ... + a_n.$$

This is a parabola opening upwards with roots x = 0 and x = 2s/n, and the minimum value is the average of the roots x = s/n, i.e., the average of the numbers.

3 Dynamic Programming

Dynamic programming is a technique used to avoid computing multiple times the same sub-solution in a recursive algorithm. A sub-solution of the problem is constructed from the previously found ones. DP solutions have a polynomial complexity which assures a much faster running time than other techniques like backtracking, brute-force etc.

3.1 Memoization - Top Down

Memoization ensures that a method doesn't run for the same inputs more than once by keeping a record of the results for the given inputs (usually in a hash map).

To avoid the duplicate work caused by the branching, we can wrap the method in a class that stores an instance variable, cache, that maps inputs to outputs. Then we simply,

- check cache to see if we can avoid computing the answer for any given input, and
- save the results of any calculations to cache.

Memoization is a common strategy for dynamic programming problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the Fibonacci problem, above).
The other common strategy for dynamic programming problems is going bottom-up, which is usually cleaner and often more efficient.

3.2 Bottom Up

Going bottom-up is a way to avoid recursion, saving the memory cost that recursion incurs when it builds up the call stack.

Simply a bottom-up algorithm "starts from the beginning," while a recursive algorithm often "starts from the end and works backwards."

3.3 An example - Fibonacci

Let's start with an example which most of us are familiar, fibonacci numbers: finding the n^{th} fibonacci number defined by

$$F_n = F_{n-1} + F_{n-2} \text{ and } F_0 = 0, F_1 = 1$$

There are a few approaches and all of them work. Let's go through the codes and see the cons and pros:

3.3.1 Recursion

```
def fibonacci(n):
    if (n == 0):
        return 0
    if (n == 1):
        return 1

    return fibonacci(n - 1) + fibonacci(n - 2)
```

3.3.2 Dynamic Programming

- **Top Down - Memoization**

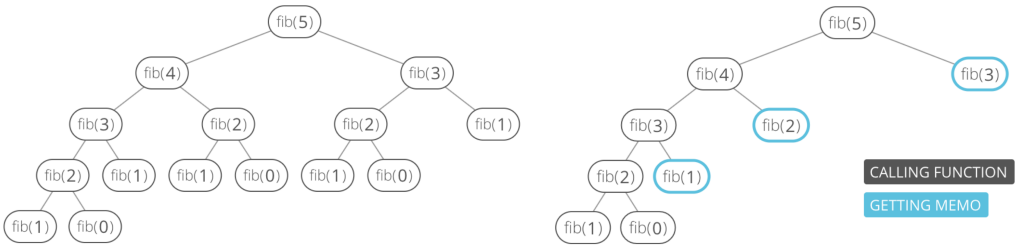
  The recursion does a lot of unnecessary calculation because a given fibonacci number will be calculated multiple times. An easy way to improve this is to cache the results (memoization):

```
cache = {}

def fibonacci(n):
    if (n == 0):
        return 0
    if (n == 1):
        return 1
    if (n in cache):
        return cache[n]

    cache[n] = fibonacci(n - 1) + fibonacci(n - 2)

    return cache[n]
```



Recursive Memoization

- **Bottom-Up**

  A better way to do this is to get rid of the recursion all-together by evaluating the results in the right order:

```
1       cache = {}
2
3       def fibonacci(n):
4          cache[0] = 0
5          cache[1] = 1
6
7          for (i in range(2, n + 1)):
8             cache[i] = cache[i - 1] + cache[i - 2]
9
10         return cache[n]
```

  We can even use constant space, and store only the necessary partial results along the way:

```
1       def fibonacci(n):
2         fib_minus_2 = 0
3         fib_minus_1 = 1
4
5         for (i in range(2, n + 1)):
6            fib = fib_minus_1 + fib_minus_2
7            fib_minus_1, fib_minus_2 = fib, fib_minus_1
8
9         return fib
```

3.4 How to Apply Dynamic Programming?

- Find the recursion in the problem.
- **Top-down:** store the answer for each subproblem in a table to avoid having to recompute them.
- **Bottom-up:** Find the right order to evaluate the results so that partial results are available when needed.

Dynamic programming generally works for problems that have an inherent left to right order such as strings, trees or integer sequences. If the naive recursive algorithm does not compute the same subproblem multiple time, dynamic programming won't help.

4 Common DP Problems

4.1 Coin Problem

As we discussed above Greedy approach doesn't work all the time for the coin problem.
For example: if the coins are 4,3,1 and the target sum is 6, the greedy algorithm produces the solution 4+1+1 while the optimal solution is 3+3.
This is where Dynamic Programming helps us:

4.1.1 Solution

Approach:

```
1- If V == 0, then 0 coins required.
2- If V > 0, minCoins(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}
           where i varies from 0 to m-1
           and coins[i] <= V
```

```
1   def minCoins(coins, target):
2       # base case
3       if (V == 0):
4           return 0
5
6       n = len(coins)
7       # Initialize result
8       res = sys.maxsize
9
10      # Try every coin that has smaller value than V
11      for i in range(0, n):
12          if (coins[i] <= target):
13              sub_res = minCoins(coins, target-coins[i])
14
15              # Check for INT_MAX to avoid overflow and see if
16              # result can minimized
17              if (sub_res != sys.maxsize and sub_res + 1 < res):
18                  res = sub_res + 1
19
20      return res
```

4.2 Knapsack Problem

We are given the weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent the values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item; or don't pick it (0-1 property).
Approach:

```
To consider all subsets of items, there can be two cases for every item: (1) the item
    is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of the
   following two values.
1- Maximum value obtained by n-1 items and W weight (excluding nth item).
2- Value of the nth item plus the maximum value obtained by n-1 items and W minus the
    weight of the nth item (including the nth item).

If the weight of nth item is greater than W, then the nth item cannot be included and
    case 1 is the only possibility.
```

For example:

```
   Let,
   Knapsack Max weight: W = 8 (units)
   Weight of items: wt = {3, 1, 4, 5}
   Values of items: val = {10, 40, 30, 50}
   Total items: n = 4

   Then,
   The following sums are possible:
   1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13

   The sum 8 is possible with 2 combinations
   {3, 5} and {1, 3, 4}

   for {3, 5} total value is 60
   and for {1, 3, 4} total value is 80

   But,
   There is a better solution with less weight:
   {1, 5} has total weight of 6 which is less than 8 and has total value 90.

   Hence,
   Our answer for maximum total value in the knapsack with given items and knapsack
      will be 80.
```

4.2.1 Recursion

```
def knapSack(W , wt , val , n):

    # Base Case
    if (n == 0 or W == 0):
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n - 1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W - wt[n - 1] , wt , val , n - 1), knapSack(W
            , wt , val , n - 1))
```

4.2.2 DP

It should be noted that the above function computes the same subproblems again and again. Time complexity of this naive recursive solution is exponential (2^n).
Since suproblems are evaluated again, this problem has Overlapping Subproblems property. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
    for (i in range(n + 1)):
        for (w in range(W + 1)):
            if (i == 0 or w == 0):
                K[i][w] = 0
            elif (wt[i - 1] <= w):
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w])
            else:
                K[i][w] = K[i - 1][w]

    return K[n][W]
```

6 Two Pointers Technique

Two pointers is really an easy and effective technique that is typically used for searching pairs in a sorted array.
Given a sorted array A (sorted in ascending order), having N integers, find if there exists any pair of elements (A[i], A[j]) such that their sum is equal to X.

Illustration :

A[] = {10, 20, 35, 50, 75, 80}
X = =70
i = 0
j = 5
A[i] + A[j] = 10 + 80 = 90
Since A[i] + A[j] > X, j--
i = 0
j = 4
A[i] + A[j] = 10 + 75 = 85
Since A[i] + A[j] > X, j--
i = 0
j = 3
A[i] + A[j] = 10 + 50 = 60
Since A[i] + A[j] < X, i++
i = 1
j = 3
m
A[i] + A[j] = 20 + 50 = 70
Thus this signifies that Pair is Found.

The algorithm basically uses the fact that the input array is sorted. We start the sum of extreme values (smallest and largest) and conditionally move both pointers. We move left pointer 'i' when the sum of A[i] and A[j] is less than X. We do not miss any pair because the sum is already smaller than X. Same logic applies for right pointer j.

Methods:

Here we will be proposing a two-pointer algorithm by starting off with the naïve approach only in order to showcase the execution of operations going on in both methods and secondary to justify how two-pointer algorithm optimizes code via time complexities across all dynamic programming languages such as Python, JavaScript, PHP etc and all Statically typed languages such as C++, Java, C# etc.

Naive Approach using loops
Optimal approach using two pointer algorithm
Method 1: Naïve Approach

Below is the implementation:

```
// C++ Program Illustrating Naive Approach to
// Find if There is a Pair in A[0..N-1] with Given Sum

// Importing all libraries
#include <bits/stdc++.h>

using namespace std;

bool isPairSum(int A[], int N, int X)
{
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         // as equal i and j means same element
         if (i == j)
            continue;

         // pair exists
         if (A[i] + A[j] == X)
            return true;

         // as the array is sorted
         if (A[i] + A[j] > X)
            break;
      }
   }

   // No pair found with given sum.
   return false;
}

// Driver code
int main()
{
   int arr[] = { 2, 3, 5, 8, 9, 10, 11 };
   int val = 17;
   int arrSize = *(&arr + 1) - arr;
   sort(arr, arr + arrSize); // Sort the array
   // Function call
   cout << isPairSum(arr, arrSize, val);

   return 0;
}
```

Output

1

Time Complexity: O(n2).
Auxiliary Space: O(1)

Method 2: Two Pointers Technique

Now let's see how the two-pointer technique works. We take two pointers, one representing the first element and other representing the last element of the array, and then we add the values kept at both the pointers. If their sum is smaller than X then we shift the left pointer to right or if their sum is greater than X then we shift the right pointer to left, in order to get closer to the sum. We keep moving the pointers until we get the sum as X.
Below is the implementation:

```
// C++ Program Illustrating Naive Approach to
// Find if There is a Pair in A[0..N-1] with Given Sum
// Using Two-pointers Technique

// Importing required libraries
#include <bits/stdc++.h>
using namespace std;

// Two pointer technique based solution to find
// if there is a pair in A[0..N-1] with a given sum.
int isPairSum(vector<int>& A, int N, int X)
{
   // represents first pointer
   int i = 0;

   // represents second pointer
   int j = N - 1;

   while (i < j) {

      // If we find a pair
      if (A[i] + A[j] == X)
         return 1;

      // If sum of elements at current
      // pointers is less, we move towards
      // higher values by doing i++
      else if (A[i] + A[j] < X)
         i++;

      // If sum of elements at current
      // pointers is more, we move towards
      // lower values by doing j--
      else
         j--;
   }
   return 0;
}
```

```
// Driver code
int main()
{
    // array declaration
    vector<int> arr = { 2, 3, 5, 8, 9, 10, 11 };

    // value to search
    int val = 17;

    // size of the array
    int arrSize = arr.size();

    // array should be sorted before using two-pointer
    // technique
    sort(arr.begin(), arr.end());

    // Function call
    cout << (isPairSum(arr, arrSize, val) ? "True"
                                : "False");

    return 0;
}
```

Output
True

Time Complexity: O(n log n) (As sort function is used)
Auxiliary Space: O(1), since no extra space has been taken.

7 Sliding Window Technique

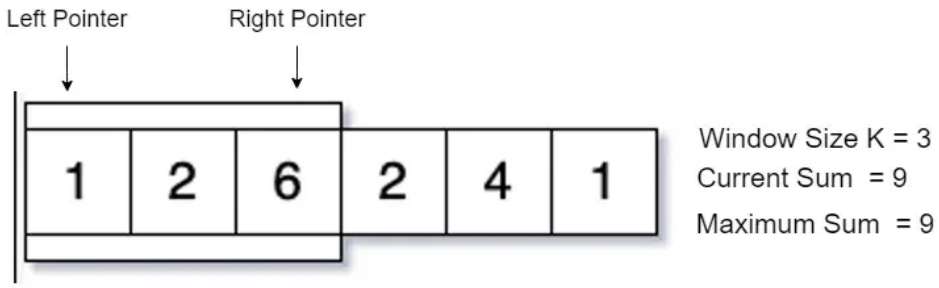
Sliding Window problems are problems in which a fixed or variable-size window is moved through a data structure, typically an array or string, to solve problems efficiently based on continuous subsets of elements. This technique is used when we need to find subarrays or substrings according to a given set of conditions.

Sliding Window Technique is a method used to efficiently solve problems that involve defining a window or range in the input data (arrays or strings) and then moving that window across the data to perform some operation within the window. This technique is commonly used in algorithms like finding subarrays with a specific sum, finding the longest substring with unique characters, or solving problems that require a fixed-size window to process elements efficiently.

Let's take an example to understand this properly, say we have an array of size **N** and also an integer **K**. Now, we have to calculate the maximum sum of a subarray having size exactly **K**. Now how should we approach this problem?

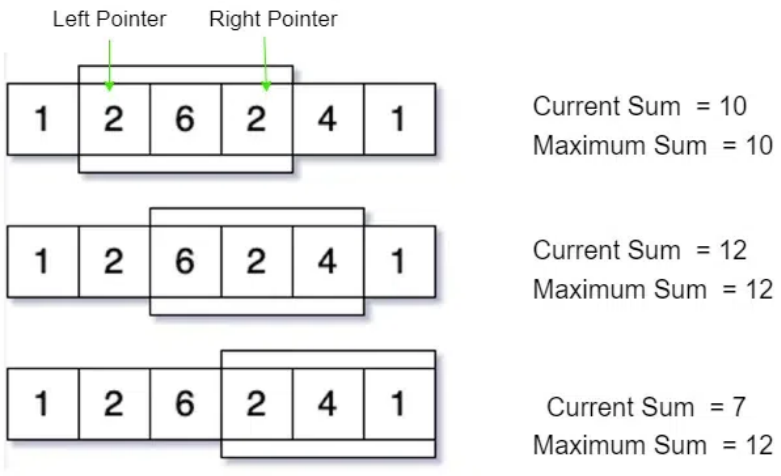
One way to do this by taking each subarray of size K from the array and find out the maximum sum of these subarrays. This can be done using Nested loops which will result into O(N2) Time Complexity.

But can we optimize this approach?
The answer is Yes, instead of taking each K sized subarray and calculating its sum, we can just take one K size subarray from 0 to K-1 index and calculate its sum now shift our range one by one along with the iterations and update the result, like in next iteration increase the left and right pointer and update the previous sum as shown in the below image:



Sliding window Technique

Now follow this method for each iteration till we reach the end of the array:



Sliding window Technique

So, we can see that instead of recalculating the sum of each K sized subarray we are using previous window of size K and using its results we update the sum and shift the window right by moving left and right pointers, this operation is optimal because it take O(1) time to shift the range instead of recalculating.

This approach of shifting the pointers and calculating the results accordingly is known as **Sliding window Technique.**

How to Identify Sliding Window Problems:

- These problems generally require Finding Maximum/Minimum **Subarray, Substrings** which satisfy some specific condition.
- The size of the subarray or substring **'K'** will be given in some of the problems.
- These problems can easily be solved in O(N2) time complexity using nested loops, using sliding window we can solve these in **O(n)** Time Complexity.
- **Required Time Complexity:** O(N) or O(Nlog(N))
- **Constraints:** N <= 106 , If N is the size of the Array/String.

To find the maximum sum of all subarrays of size K:

Given an array of integers of size **'n',** Our aim is to calculate the maximum sum of **'k'** consecutive elements in the array.

Input : arr[] = {100, 200, 300, 400}, k = 2
Output : 700

Input : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}, k = 4
Output : 39
We get maximum sum by adding subarray {4, 2, 10, 23} of size 4.

Input : arr[] = {2, 3}, k = 3
Output : Invalid
There is no subarray of size 3 as size of whole array is 2.

Naive Approach: So, let's analyze the problem with **Brute Force Approach.** We start with the first index and sum till the kth element. We do it for all possible consecutive blocks or groups of k elements. This method requires a nested for loop, the outer for loop starts with the starting element of the block of k elements, and the inner or the nested loop will add up till the kth element.

Below is the implementation of the above approach:

```
// a subarray of size k
#include <bits/stdc++.h>
using namespace std;

// Returns maximum sum in a subarray of size k.
int maxSum(int arr[], int n, int k)
{
   // Initialize result
   int max_sum = INT_MIN;

   // Consider all blocks starting with i.
   for (int i = 0; i < n - k + 1; i++) {
      int current_sum = 0;
      for (int j = 0; j < k; j++)
         current_sum = current_sum + arr[i + j];

      // Update result if required.
      max_sum = max(current_sum, max_sum);
   }

   return max_sum;
}

// Driver code
int main()
{
   int arr[] = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };
   int k = 4;
   int n = sizeof(arr) / sizeof(arr[0]);
   cout << maxSum(arr, n, k);
   return 0;
}
```

Output
24

Time complexity: O(k*n) as it contains two nested loops.
Auxiliary Space: O(1)

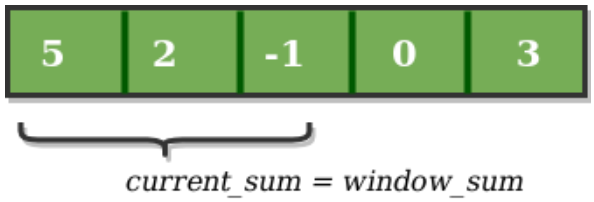
Applying the sliding window technique:

- We compute the sum of the first k elements out of n terms using a linear loop and store the sum in variable **window_sum.**
- Then we will traverse linearly over the array till it reaches the end and simultaneously keep track of the maximum sum.
- To get the current sum of a block of k elements just subtract the first element from the previous block and add the last element of the current block.

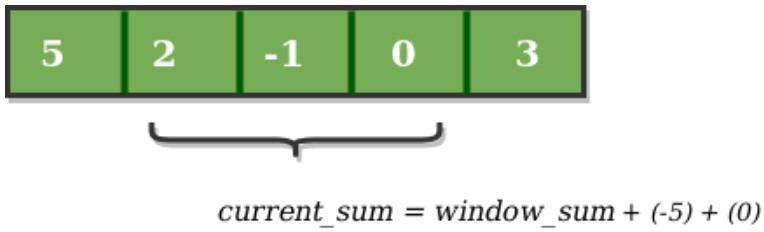
The below representation will make it clear how the window slides over the array.

Consider an array **arr**[] = {5, 2, -1, 0, 3} and value of **k** = 3 and **n** = 5

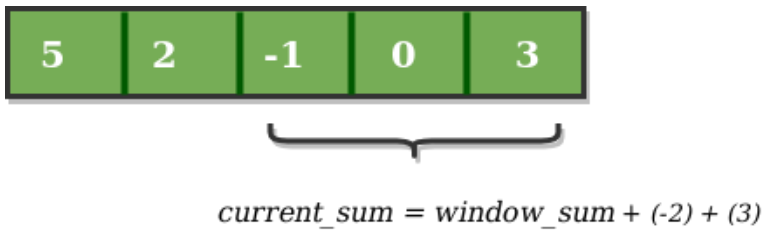
This is the initial phase where we have calculated the initial window sum starting from index 0 . At this stage the window sum is 6. Now, we set the maximum_sum as current_window i.e 6.



Now, we slide our window by a unit index. Therefore, now it discards 5 from the window and adds 0 to the window. Hence, we will get our new window sum by subtracting 5 and then adding 0 to it. So, our window sum now becomes 1. Now, we will compare this window sum with the maximum_sum. As it is smaller, we won't change the maximum_sum.



Similarly, now once again we slide our window by a unit index and obtain the new window sum to be 2. Again we check if this current window sum is greater than the maximum_sum till now. Once, again it is smaller so we don't change the maximum_sum.
Therefore, for the above array our maximum_sum is 6.



Below is the code for above approach:

```
// O(n) solution for finding maximum sum of
// a subarray of size k
#include <iostream>
using namespace std;

// Returns maximum sum in a subarray of size k.
int maxSum(int arr[], int n, int k)
{
   // n must be greater
   if (n <= k) {
      cout << "Invalid";
      return -1;
   }

   // Compute sum of first window of size k
   int max_sum = 0;
   for (int i = 0; i < k; i++)
      max_sum += arr[i];

   // Compute sums of remaining windows by
   // removing first element of previous
   // window and adding last element of
   // current window.
   int window_sum = max_sum;
   for (int i = k; i < n; i++) {
      window_sum += arr[i] - arr[i - k];
      max_sum = max(max_sum, window_sum);
   }

   return max_sum;
}

// Driver code
int main()
{
   int arr[] = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };
   int k = 4;
   int n = sizeof(arr) / sizeof(arr[0]);
   cout << maxSum(arr, n, k);
   return 0;
}
```

Output

24

Time Complexity: O(n), where n is the size of input array arr[].

Auxiliary Space: O(1)