CP- EITC

Lecture 3

CP Cell Leads:
Idriss EL ABIDI
Souhail LOUZI

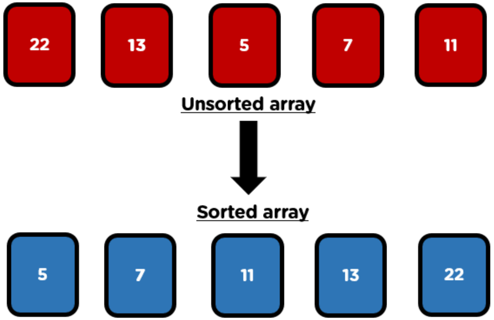
Sorting & Binary Search

January, 2025

Introduction to Sorting

Definiton:

Sorting is a fundamental algorithm design problem. Many efficient algorithms use sorting as **a subroutine**, because it is often easier to process data if the elements are in a sorted order.

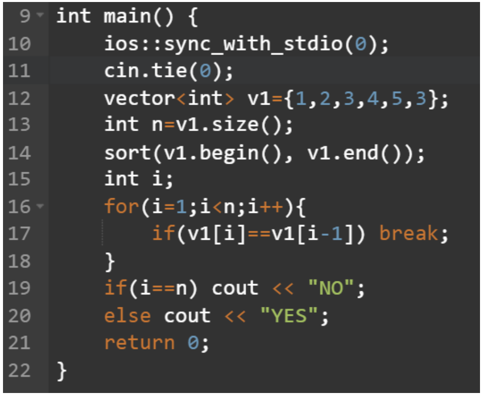


Introduction to Sorting

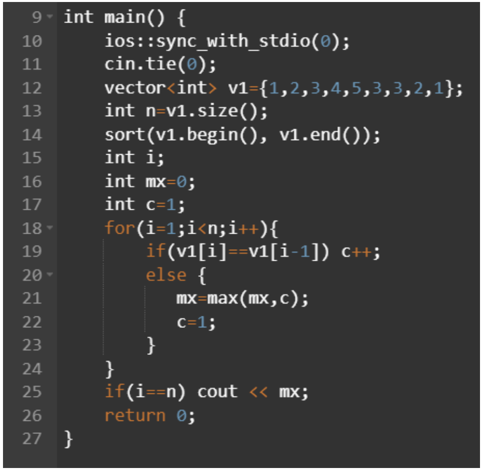
Example :

- For example, the problem does an array contain two equal elements? is easy to solve using sorting. If the array contains two equal elements, they will be next to each other after sorting, so it is easy to find them.
- Also, the problem what is the most frequent element in an array? can be solved similarly.

Solutions



```
int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    vector<int> v1={1,2,3,4,5,3};
    int n=v1.size();
    sort(v1.begin(), v1.end());
    int i;
    for(i=1;i<n;i++){
        if(v1[i]==v1[i-1]) break;
    }
    if(i==n) cout << "NO";
    else cout << "YES";
    return 0;
}
```



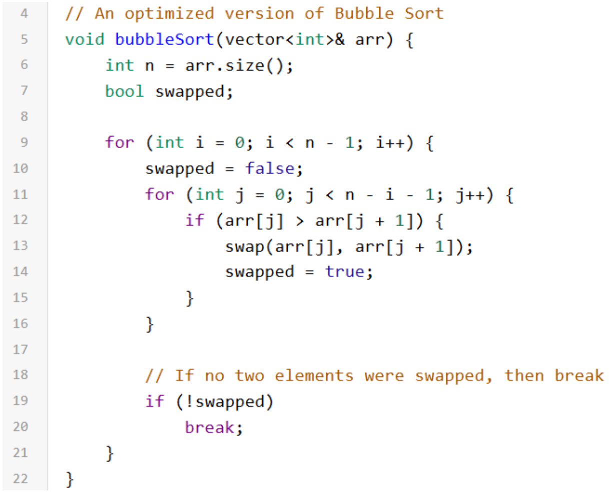
```
int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    vector<int> v1={1,2,3,4,5,3,3,2,1};
    int n=v1.size();
    sort(v1.begin(), v1.end());
    int i;
    int mx=0;
    int c=1;
    for(i=1;i<n;i++){
        if(v1[i]==v1[i-1]) c++;
        else {
            mx=max(mx,c);
            c=1;
        }
    }
    if(i==n) cout << mx;
    return 0;
}
```

O(n^2) *algorithms*

Bubble sort:

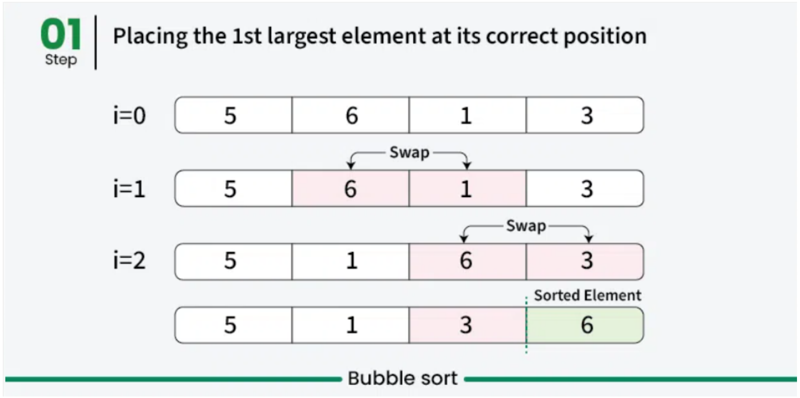
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large arrays as its average and worst-case time complexity is quite high.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j],array[j+1]);
        }
    }
}
```



```cpp
// An optimized version of Bubble Sort
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        // If no two elements were swapped, then break
        if (!swapped)
            break;
    }
}
```

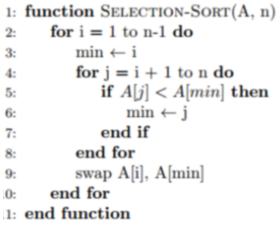
$O(n^2)$ *algorithms*



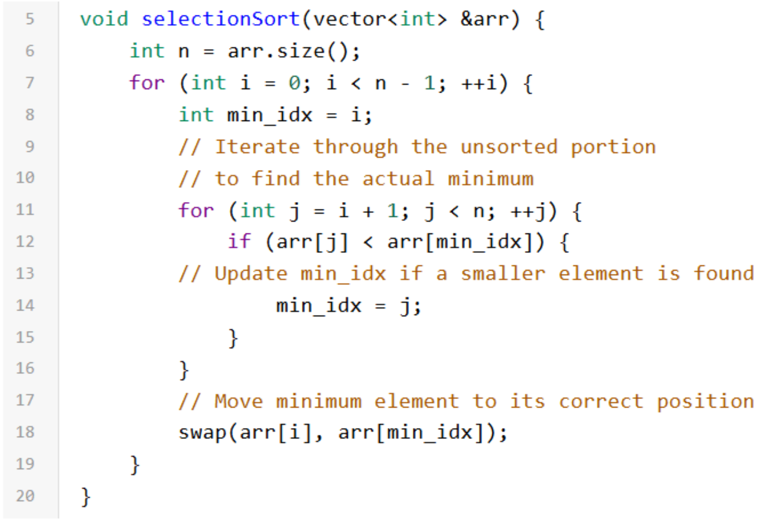
$O(n^2)$ *algorithms*

Selection sort:

It's based on building the sorted array from scratch: take the smallest element in every iteration and place it at the beginning. Since we need to find the minimum element N times, the number of operations is
N+(N-1)+(N-2)+...+1=N(N+1)/2=$O(N^2)$.

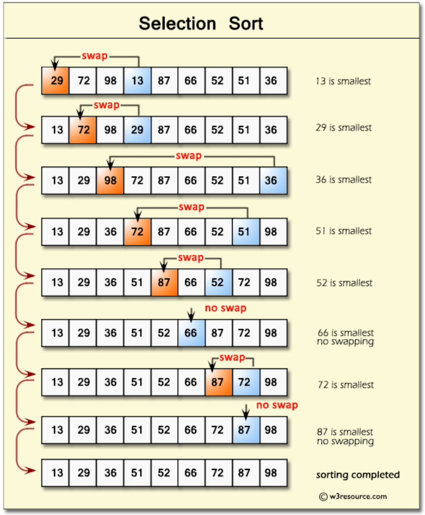


```
function SELECTION-SORT(A, n)
    for i = 1 to n-1 do
        min ← i
        for j = i + 1 to n do
            if $A[j] < A[min]$ then
                min ← j
            end if
        end for
        swap A[i], A[min]
    end for
end function
```



```cpp
void selectionSort(vector<int> &arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        int min_idx = i;
        // Iterate through the unsorted portion
        // to find the actual minimum
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[min_idx]) {
        // Update min_idx if a smaller element is found
                min_idx = j;
            }
        }
        // Move minimum element to its correct position
        swap(arr[i], arr[min_idx]);
    }
}
```

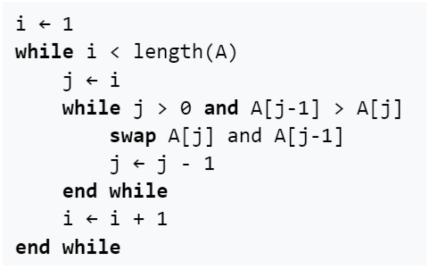
Selection sort:



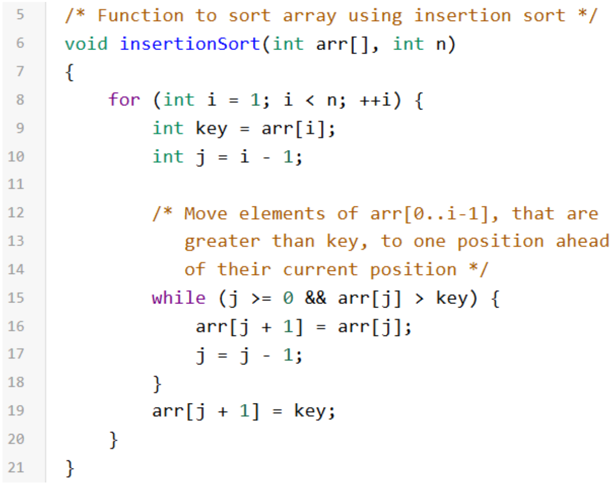
$O(n^2)$ *algorithms*

Insertion sort:

Insertion sort is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.



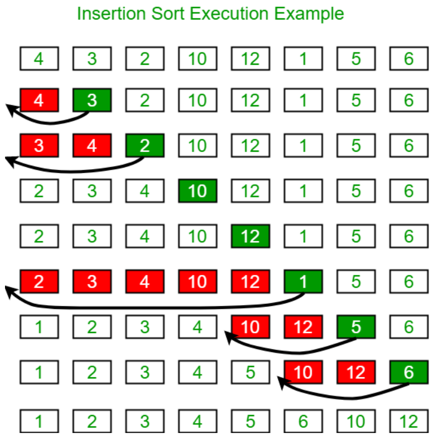
```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```



```cpp
/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

$O(n^2)$ *algorithms*

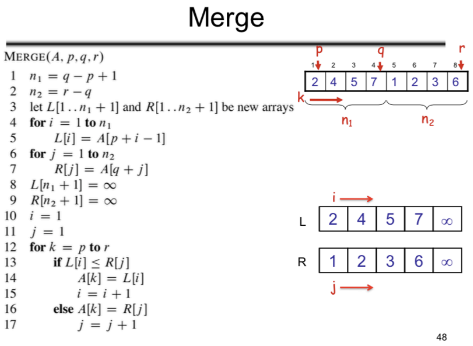
Insertion sort:



O(n log n) algorithms

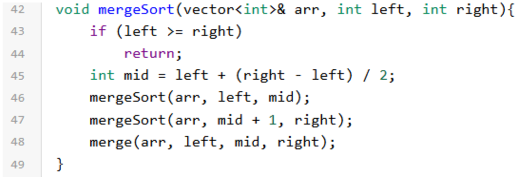
Merge Sort

The merge sort algorithm is based on a divide-and-conquer strategy. If we have the whole array, we can recursively sort the first and second halves of the array and then merge them.

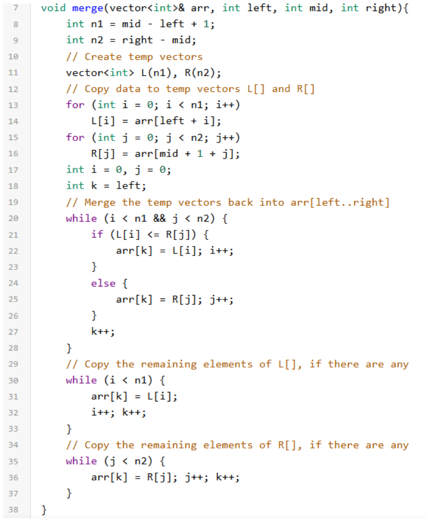


Merge

```
MERGE(A, p, q, r)
  $n_1 = q - p + 1$
  $n_2 = r - q$
  let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
  for $i = 1$ to $n_1$
      $L[i] = A[p + i - 1]$
  for $j = 1$ to $n_2$
      $R[j] = A[q + j]$
  $L[n_1 + 1] = \infty$
  $R[n_2 + 1] = \infty$
  $i = 1$
  $j = 1$
  for $k = p$ to $r$
      if $L[i] \le R[j]$
          $A[k] = L[i]$
          $i = i + 1$
      else $A[k] = R[j]$
          $j = j + 1$
```



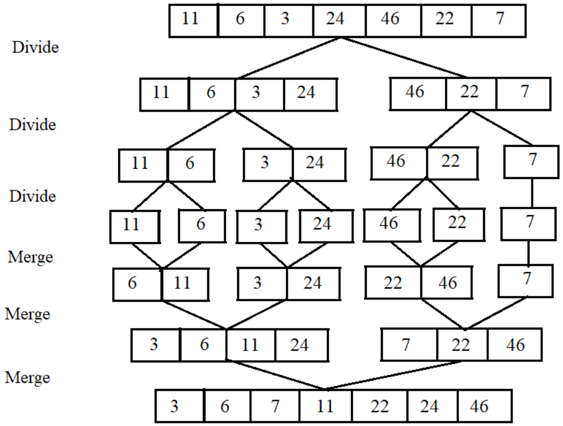
```cpp
void mergeSort(vector<int>& arr, int left, int right){
    if (left >= right)
        return;
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```



```cpp
void merge(vector<int>& arr, int left, int mid, int right){
    int n1 = mid - left + 1;
    int n2 = right - mid;
    // Create temp vectors
    vector<int> L(n1), R(n2);
    // Copy data to temp vectors L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0;
    int k = left;
    // Merge the temp vectors back into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i]; i++;
        }
        else {
            arr[k] = R[j]; j++;
        }
        k++;
    }
    // Copy the remaining elements of L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++; k++;
    }
    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j]; j++; k++;
    }
}
```

O(n log n) algorithms

Merge Sort



O(n log n) algorithms

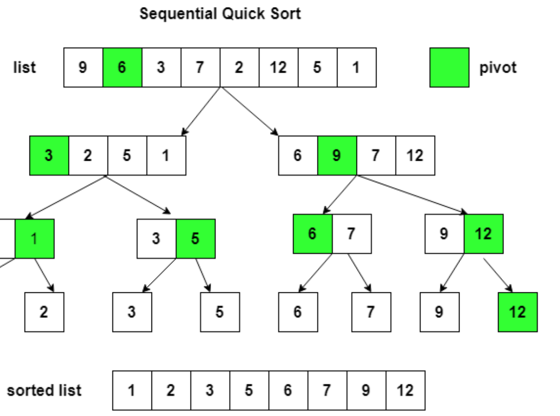
Quick Sort

QuickSort works on the principle of divide and conquer, breaking down the problem into smaller sub-problems.
There are mainly three steps in the algorithm:

- Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
- Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right.
- Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
- Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

O(n log n) algorithms

Quick Sort



Quick Sort vs Merge Sort

- Merge Sort and its variations are used in library methods of programming languages. For example its variation TimSort is used in Python, Java Android and Swift. The main reason why it is preferred to sort non-primitive types is stability which is not there in QuickSort. For example Arrays.sort in Java uses QuickSort while Collections.sort uses MergeSort.
- Quick Sort is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivots position (without considering their original positions).
- Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN) , which means it performs well even on large datasets.
- Stability : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.

Introduction

the C++ standard library contains the function sort that can be easily used for sorting arrays and other data structures.

Why use it ?

- First, it saves time because there is no need to implement the function.
- Second, the library implementation is certainly correct and efficient.
- It is not probable that a home-made sorting function would be better.
- The sort() function is implemented using the Intro Sort Algorithm. It is the combination of three standard sorting algorithms: insertion sort, quick sort and heap sort. It is chooses the best algorithm that fits the given case.

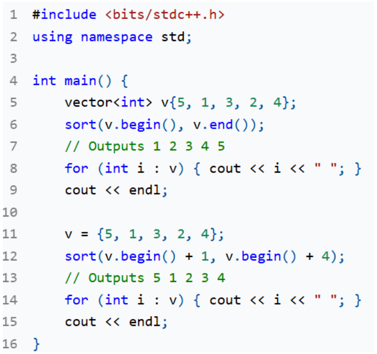
Static Arrays

To sort static arrays, use sort(arr, arr + N) where N is the number of elements to be sorted. The range can also be specified by replacing arr and arr + N with the intended range. For example, sort(arr + 1, arr + 4) sorts indices $[1, 4]$.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int arr[] = {5, 1, 3, 2, 4};
    int N = 5;
    sort(arr, arr + N);
    for (int i = 0; i < N; i++) cout << arr[i] << " ";  // 1 2 3 4 5
    cout << endl;

    int arr2[] = {5, 1, 3, 2, 4};
    sort(arr2 + 1, arr2 + 4);
    for (int i = 0; i < N; i++) cout << arr2[i] << " ";  // 5 1 2 3 4
}
```

Vectors

In order to sort a dynamic array, use sort(v.begin(), v.end()) (or sort(begin(v),end(v))). The default sort function sorts the array in ascending order. Similarly, we can specify the range. For example, sort(v.begin() + 1, v.begin() + 4) sorts indices $[1, 4]$.



```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v{5, 1, 3, 2, 4};
    sort(v.begin(), v.end());
    // Outputs 1 2 3 4 5
    for (int i : v) { cout << i << " "; }
    cout << endl;

    v = {5, 1, 3, 2, 4};
    sort(v.begin() + 1, v.begin() + 4);
    // Outputs 5 1 2 3 4
    for (int i : v) { cout << i << " "; }
    cout << endl;
}
```

Arrays of Pairs & Tuples

By default, C++ pairs are sorted by first element and then second element in case of a tie. Tuples are sorted similarly.

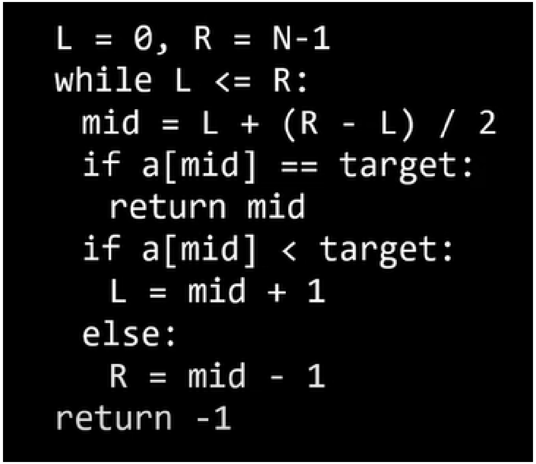
```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<pair<int, int>> v{{1, 5}, {2, 3}, {1, 2}};
    sort(v.begin(), v.end());

    /*
     * Outputs:
     * 1 2
     * 1 5
     * 2 3
     */
    for (pair<int, int> p : v) { cout << p.first << " " << p.second << endl; }
}
```

Binary search

When we binary search on an answer, we start with a search space of size N which we know the answer lies in. Then each iteration of the binary search cuts the search space in half, so the algorithm tests $\mathcal{O}(\log N)$ values. This is efficient and much better than testing each possible value in the search space.
Here's a step-by-step description of using binary search to play the guessing game:

- Let min = 1 and max = n.
- Guess the average of max and min, rounded down so that it is an integer.
- If you guessed the number, stop. You found it!
- If the guess was too low, set min to be one larger than the guess.
- If the guess was too high, set max to be one smaller than the guess.
- Go back to step two.

Target Value



```
L = 0, R = N-1
while L <= R:
  mid = L + (R - L) / 2
  if a[mid] == target:
    return mid
  if a[mid] < target:
    L = mid + 1
  else:
    R = mid - 1
return -1
```

Implementation

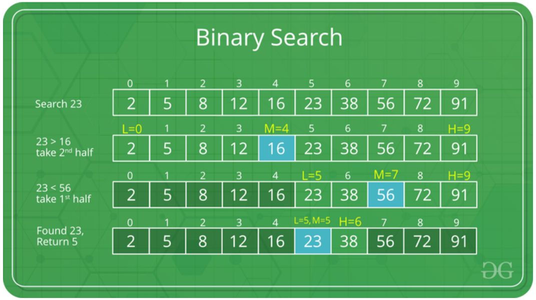
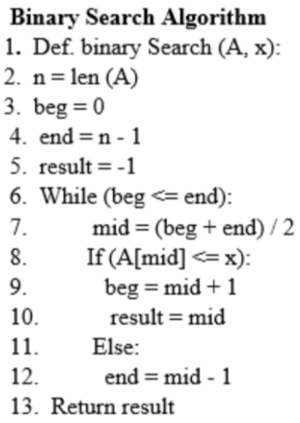


Figure 1: Example

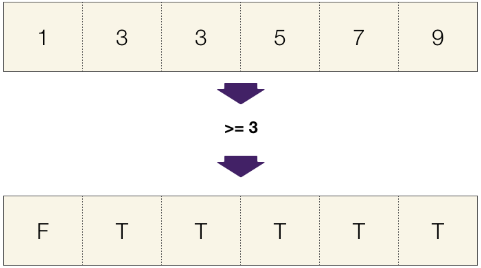


Binary Search Algorithm

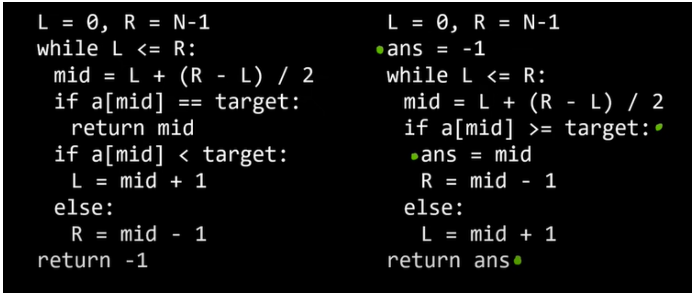
```
1.  Def. binary Search (A, x):
2.  n = len (A)
3.  beg = 0
4.  end = n - 1
5.  result = -1
6.  While (beg <= end):
7.          mid = (beg + end) / 2
8.          If (A[mid] <= x):
9.              beg = mid + 1
10.             result = mid
11.         Else:
12.             end = mid - 1
13. Return result
```

Figure 2: Algorithm

Find First Greater Value



Implementation



```
L = 0, R = N-1
while L <= R:
 mid = L + (R - L) / 2
 if a[mid] == target:
  return mid
 if a[mid] < target:
  L = mid + 1
 else:
  R = mid - 1
return -1
```

```
L = 0, R = N-1
ans = -1
while L <= R:
 mid = L + (R - L) / 2
 if a[mid] >= target:
  ans = mid
  R = mid - 1
 else:
  L = mid + 1
return ans
```

Functions in C++

The C++ standard library contains the following functions that are based on binary search and work in logarithmic time:

- **lower_bound** returns a pointer to the first array element whose value is at least x.
- **upper_bound** returns a pointer to the first array element whose value is larger than x.
- **equal_range** returns both above pointers.

The functions assume that the array is sorted. If there is no such element, the pointer points to the element after the last array element.

Functions in C++

```
auto k = lower_bound(array,array+n,x)-array;
if (k < n && array[k] == x) {
    // x found at index k
}
```

Then, the following code counts the number of elements whose value is x:

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";
```

Using equal_range, the code becomes shorter:

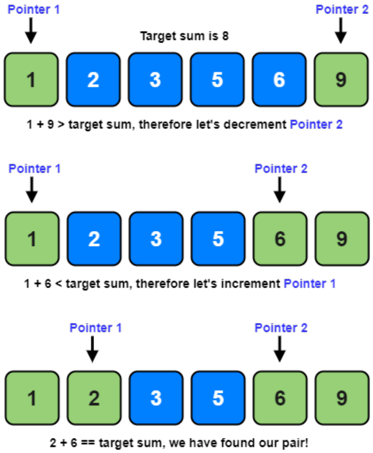
```
auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";
```

Two Pointers

Definition

- In the two pointers method, two pointers are used to iterate through the array values. Both pointers can move to one direction only, which ensures that the algorithm works efficiently.
- Iterating two monotonic pointers across an array to search for a pair of indices satisfying some condition in linear time.

Example


Pointer 1
Pointer 2
Target sum is 8
1 2 3 5 6 9
1 + 9 > target sum, therefore let's decrement Pointer 2
Pointer 1
Pointer 2
1 2 3 5 6 9
1 + 6 < target sum, therefore let's increment Pointer 1
Pointer 1
Pointer 2
1 2 3 5 6 9
2 + 6 == target sum, we have found our pair!

Problems

Thank you for listening !

Idriss El Abidi
Souhail Louzi

Special thanks to the former CP leads:
Hamza BA-MOHAMMED, Hachim Hassani Lahsinui and mohamed ghacham amrani
for their insightful review and precious remarks.