**AOA Lab 3**

Name: Vivian Vijay Ludrick

Roll No: 9914

Branch: SE Comps A

Batch: C

**Linear Search:**

**CODE:**

#include <iostream>

int main(){

// predefined array

int arr[] = {12, 11, 13, 5, 6, 6, 7};

int n = 7;

int element = 6;

for(int i = 0; i < n; i ++){

if(arr[i] == element){

std::cout<<"The element is at index:"<< i<<std::endl;

break;

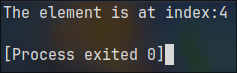
}

}

return 0;

}

**OUTPUT:**

****

**TIME COMPLEXITY:**

The best case time complexity of linear search is = Ω(1)

The worst case time complexity of linear search is = O(n)

The best case time complexity of linear search is = θ(n)

**Binary Search(non-rec):**

**CODE:**

#include <iostream>

int main(){

// predefined array

int arr[] = {5, 6, 6, 7, 11, 12, 13};

int n = 7;

int element = 11;

int low = 0, high = n;

for(int i = low; i < high; i++){

int mid = (low + high) / 2;

if(arr[mid] == element){

std::cout<<"The element is at index"<<mid<<std::endl;

break;

}else if(arr[mid] > element){

high = mid - 1;

}

else{

low = mid + 1;

}

}

if (low > high) {

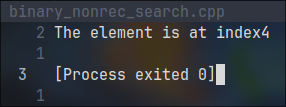
std::cout << "Element not found";

}

return 0;

}

**OUTPUT:**

****

**TIME COMPLEXITY:**

The best case time complexity of binary search is = Ω(1)

The worst case time complexity of binary search is = O(log2n)

The best case time complexity of binary search is = θ(log2n)

**Binary Search(rec):**

**CODE:**

#include <cstdio>

int binary\_rec(int a[], int low, int high, int element) {

if (low > high)

return -1; // Element not found

else {

int mid = (low + high) / 2;

if (a[mid] == element)

return mid + 1; // Element found

else if (a[mid] > element)

return binary\_rec(a, low, mid - 1, element);

else

return binary\_rec(a, mid + 1, high, element);

}

}

int main() {

// predefined array

int arr[] = {5, 6, 6, 7, 11, 12, 13};

int n = 7;

int low = 0;

int high = n;

int element = 11;

int answer = binary\_rec(arr, low, high, element);

if (answer != -1) {

printf("The data %d is at position %d\n", element, answer);

} else {

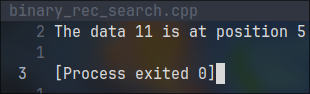
printf("The data %d is not present in the array\n", element);

}

return 0;

}

**OUTPUT:**

****

**TIME COMPLEXITY:**

The best case time complexity of binary search is = Ω(1)

The worst case time complexity of binary search is = O(log2n)

The best case time complexity of binary search is = θ(log2n)

**Ternary Search(non-rec):**

**CODE:**

#include <stdio.h>

int ternarySearch(int arr[], int low, int high, int element) {

while(low <= high) {

int mid1 = low + (high - low)/3; //mid1 calculation

int mid2 = high - (high - low)/3; //mid2 calculation

if(arr[mid1] == element) //if element == element at mid1

return mid1;

else if(arr[mid2] == element) //if element == element at mid2

return mid2;

else if(arr[mid1] > element) //part 1 of array

high = mid1 - 1;

else if(element > arr[mid2]) //part 3

low = mid2;

else { //part 2, middle part

low = mid1;

high = mid2-1;

}

}

return -1;

}

int main() {

//precondition, that the array should already be sorted

int arr[] = {5, 6, 6, 7, 11, 12, 13};

int n = 7;

int low = 0;

int high = n;

int element = 11;

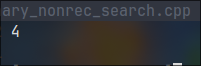
int index = ternarySearch(arr, low, high, element);

printf("%d\n", index);

return 0;

}

**OUTPUT:**

****

**TIME COMPLEXITY:**

The best case time complexity of ternary search is = Ω(1)

The worst case time complexity of ternary search is = O(log3n)

The best case time complexity of ternary search is = θ(log3n)

**Ternary Search(rec):**

**CODE:**

//ternary search recursive

#include <stdio.h>

int ternarySearchRec(int arr[], int low, int high, int element) {

int mid1 = low + (high - low)/3; //mid1 calculation

int mid2 = high - (high - low)/3; //mid2 calculation

if(arr[mid1] == element) //if element == element at mid1

return mid1;

else if(arr[mid2] == element) //if element == element at mid2

return mid2;

else if(low > high) //if element is not found

return -1;

else if(arr[mid1] > element) //part 1 of array

return ternarySearchRec(arr, low, mid1-1, element);

else if(element > arr[mid2]) //part 3

return ternarySearchRec(arr, mid2+1, high, element);

else //part 2, middle part

return ternarySearchRec(arr, mid1+1, mid2-1, element);

}

int main() {

//precondition, that the array should already be sorted

int arr[] = {5, 6, 6, 7, 11, 12, 13};

int n = 7;

int low = 0;

int high = n;

int element = 11;

int index = ternarySearchRec(arr, low, high, element);

printf("%d\n", index);

return 0;

}

**OUTPUT:**

****

**TIME COMPLEXITY:**

The best case time complexity of ternary search is = Ω(1)

The worst case time complexity of ternary search is = O(log3n)

The best case time complexity of ternary search is = θ(log3n)

**MIN MAX:**

**CODE:**

#include <iostream>

#include <vector>

void minmax(std::vector<int>& arr, int low, int high, int& pmin, int& pmax) {

int mid, min1, max1, min2, max2;

if (low == high) {

// Base case: only one element in the array

pmin = arr[low];

pmax = arr[low];

} else if (high == low + 1) {

// Base case: only two elements in the array

if (arr[low] < arr[high]) {

pmin = arr[low];

pmax = arr[high];

} else {

pmin = arr[high];

pmax = arr[low];

}

} else {

// Recursive case: more than two elements in the array

mid = (low + high) / 2;

minmax(arr, low, mid, min1, max1);

minmax(arr, mid + 1, high, min2, max2);

pmin = std::min(min1, min2);

pmax = std::max(max1, max2);

}

}

int main() {

std::vector<int> arr = {4, 7, 10, 8, 1, 9, 3, 5, 6, 6};

int min, max;

minmax(arr, 0, arr.size() - 1, min, max);

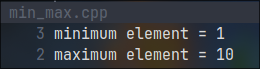
std::cout << "minimum element = " << min << std::endl;

std::cout << "maximum element = " << max << std::endl;

return 0;

}

**OUTPUT:**

****

**TIME COMPLEXITY:**

The best case time complexity of min max is = Ω(1)

The worst case time complexity of min max search is = O(n)

The best case time complexity of min max search is = θ(n)

**Comment on "Ternary Search is not that better than Binary Search".**

**⇒**

1. Time Complexity: While both ternary and binary search have logarithmic time complexities, ternary search may perform more comparisons per iteration, leading to potentially slower performance compared to binary search, especially in scenarios with a large search space.
2. Space Complexity: Ternary search may require more memory due to additional recursive calls and comparisons, leading to increased stack space usage compared to binary search, which has a simpler recursive structure.
3. Practical Considerations: In practice, binary search may often outperform ternary search due to its simplicity, ease of implementation, and potential for better cache utilization, making it a preferred choice in many scenarios despite ternary search's theoretical advantages.

**Discuss fake coin problem in detail.**

⇒ The fake coin problem is like a puzzle where you have a bunch of coins, and one of them is fake, but you don't know which one. The fake coin looks exactly like the real ones, but it's either lighter or heavier.

1. Brute Force Approach: Imagine you have a big pile of coins, and you pick two coins at a time. You weigh them on a balance scale. If they weigh the same, you move on to the next two coins. If one coin is lighter or heavier than the other, you know that one of those two coins is fake. You keep doing this until you find the fake coin.
2. Divide and Conquer Approach: Instead of comparing two coins at a time, you divide the pile of coins into smaller groups. You weigh equal numbers of coins from each group. If one group is lighter or heavier, you know the fake coin is in that group. Then, you divide that group further and repeat the process until you find the fake coin.

Brute Force Approach:

#include <iostream>

// Function to find the fake coin using the decrease and conquer technique

int findFakeCoin(int coins[], int n) {

if (coins[0] < coins[1]) {

std::cout << "The fake coin is at index 1" << std::endl;

return 0;

}

for (int i = 1; i < n; i++) {

if (coins[0] > coins[i]) {

std::cout << "The fake coin is at index " << i << std::endl;

return i;

}

}

std::cout << "No fake coin was found" << std::endl;

return -1;

}

// Driver code

int main() {

int coins[] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 0};

int n = sizeof(coins) / sizeof(coins[0]);

findFakeCoin(coins, n);

return 0;

}

