|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Analysis of Algorithms** | **Semester:** | **IV** |
| **Date of Performance:** | **12 / 02 / 2024** | **Batch No:** | **EXCP B1** |
| **Faculty Name:** | **Prof. Payal Varangoankar** | **Roll No:** | **16014022096** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** |  |

**Experiment No: 2**

**Title: Binary search Finding Minimum and Maximum.**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| To learn the divide and conquer strategy of solving the problems of different types |

|  |
| --- |
| **COs to be achieved:** |
| **CO2: Describe various algorithm design strategies to solve different problems.** |

## 

|  |
| --- |
| **Theory:** |
| **Historical Profile:**  Finding maximum and minimum or Binary search are a few problems that are solved with the divide-and-conquer technique. This is one the simplest strategies that works on dividing the problem to the smallest possible level.  Binary Search is an extremely well-known instance of the divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element, "probe" the middle element of the array. Then continue in either the lower or upper segment of the array, depending on the outcome of the probe until the required (given) element is reached.  **New Concepts to be learned:**   1. Number of comparisons 2. Application of algorithmic design strategy to any problem 3. Classical problem-solving vs. Divide-and-Conquer problem-solving. |
| **Code:**  **BINARY SEARCH ITERATIVE:**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  **int** binary\_search(**int** A**[]**, **int** key, **int** imin, **int** imax, **int** \*iterations)  {      \*iterations = 0;      while (imax >= imin) {          (\*iterations)++;  **int** imid = (imin + imax) / 2;          if (A[imid] == key) {              return imid;          } else if (A[imid] < key) {              imin = imid + 1;          } else {              imax = imid - 1;          }      }      return -1;  }  **int** main() {    **int** n, value, iterations;        printf("Enter value of n: ");      scanf("%d", &n);    **int** arr[n];      printf("Original Array: ");      for (**int** i = 0; i < n; i++) {          arr[i] = i;          printf("%d ", arr[i]);      }      printf("\n");      printf("Enter value you want to search for: ");      scanf("%d", &value);      clock\_t l1, l2;      l1 = clock();  **int** index = binary\_search(arr, value, 0, n - 1, &iterations);      l2 = clock();      if (index != -1) {          printf("Value found at index: %d\n", index);      } else {          printf("Value not found\n");      }    **double** t1 = 1000000 \* ((**double**)(l2 - l1)) / CLOCKS\_PER\_SEC;      printf("TIME (in micro seconds) : %f \n", t1);      printf("Number of iterations: %d\n", iterations);      return 0;  }  **BINARY SEARCH RECURSIVE:**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  **void** binary\_search(**int** A**[]**, **int** key, **int** imin, **int** imax, **int** \*iterations) {      if (imax < imin) {          printf("KEY NOT FOUND");      } else {  **int** imid = (imin + imax) / 2;          (\*iterations)++;          if (A[imid] < key) {              binary\_search(A, key, imid + 1, imax, iterations);          } else if (A[imid] > key) {              binary\_search(A, key, imin, imid - 1, iterations);          } else {              printf("\nKEY %d FOUND AT %d position", A[imid], imid);          }      }  }  **int** main() {  **int** n, value;      clock\_t l1, l2;      printf("Enter value of n: ");      scanf("%d", &n);  **int** arr[n];      printf("Original Array: ");      for (**int** i = 0; i < n; i++) {          arr[i] = i;          printf("%d ", arr[i]);      }      printf("\n");      printf("Enter value you want to search for: ");      scanf("%d", &value);      l1 = clock();  **int** iterations = 0;      binary\_search(arr, value, 0, n - 1, &iterations);      l2 = clock();  **double** t1 = ((**double**)(l2 - l1)) / CLOCKS\_PER\_SEC;      printf("TIME : %f \n", t1);      printf("Number of iterations: %d\n", iterations);      return 0;  }  **Max-Min Method:**  #include <stdio.h>  #include <time.h>  **void** MaxMin\_linear(**int** a**[]**, **int** n, **int**\* maxl, **int**\* minl, **int**\* iterations) {      \*minl = \*maxl = a[0];      \*iterations = 0;        for(**int** i = 1; i < n; i++) {          (\*iterations)++;          if (a[i] >= \*maxl) {              \*maxl = a[i];          } else if (a[i] <= \*minl) {              \*minl = a[i];          }      }  }  **void** MaxMin\_dac(**int** a**[]**, **int** i, **int** j, **int**\* max, **int**\* min, **int**\* iterations) {      if (i == j) {          \*min = \*max = a[i];      } else if (i == j - 1) {          (\*iterations)++;          if (a[i] < a[j]) {              \*max = a[j];              \*min = a[i];          } else {              \*max = a[i];              \*min = a[j];          }      } else {  **int** max1, min1;  **int** mid = (i + j) / 2;            MaxMin\_dac(a, i, mid, max, min, iterations);          MaxMin\_dac(a, mid + 1, j, &max1, &min1, iterations);            if (\*max < max1) {              \*max = max1;          }          if (\*min > min1) {              \*min = min1;          }          \*iterations += 2;      }  }  **int** main() {  **int** maxl, minl, max, min, n, iterations\_linear, iterations\_dac;      clock\_t l1, l2, d1, d2;        printf("Enter value of n: ");      scanf("%d", &n);  **int** arr[n];      printf("Original Array: ");      for (**int** i = 0; i < n; i++) {          arr[i] = rand() % 10;          printf("%d ", arr[i]);      }      printf("\n");        l1 = clock();      MaxMin\_linear(arr, n, &maxl, &minl, &iterations\_linear);      l2 = clock();        d1 = clock();      MaxMin\_dac(arr, 0, n - 1, &max, &min, &iterations\_dac);      d2 = clock();    **double** t1 = 1000000 \* ((**double**)(l2 - l1)) / CLOCKS\_PER\_SEC;  **double** t2 = 1000000 \* ((**double**)(d2 - d1)) / CLOCKS\_PER\_SEC;        printf("LINEAR : \n");      printf("MAX = %d, MIN = %d \n", maxl, minl);      printf("TIME (in microseconds) : %.2f \n", t1);      printf("Number of iterations: %d\n", iterations\_linear);        printf("DIVIDE AND CONQUER : \n");      printf("MAX = %d, MIN = %d \n", max, min);      printf("TIME (in microseconds): %.2f \n", t2);      printf("Number of iterations: %d\n", iterations\_dac);        return 0;  } |

|  |
| --- |
| **Stepwise-Procedure / Algorithm:** |
| **Algorithm Iterative Binary Search**  int binary\_search(int A[ ], int key, int imin, int imax)  //The algorithm takes as parameters an array A[1.. n] , the search key and lower-higher index pair of the array.  // Output- The algorithm returns index of the search key in the given array, if it’s present.  {  // continue searching while [imin, imax] is not empty  WHILE (imax >= imin)  {  // calculate the midpoint for roughly equal partition  int imid = midpoint(imin, imax);  IF(A[imid] == key)  // key found at index imid  return imid;  // determine which subarray to search  ELSE If (A[imid] < key)  // change min index to search upper subarray  imin = imid + 1;  ELSE  // change max index to search lower subarray  imax = imid - 1;  }  // key was not found  RETURN KEY\_NOT\_FOUND;  }  The space complexity of Iterative Binary Search:  **Algorithm Recursive Binary Search**  int binary\_search(int A[], int key, int imin, int imax)  //The algorithm takes as parameters an array A[1.. n] , the search key and lower-higher index pair of the array.  // Output- The algorithm returns index of the search key in the given array, if it’s present.  {  // test if array is empty  IF (imax < imin)  // set is empty, so return value showing not found  RETURN KEY\_NOT\_FOUND;  ELSE{  // calculate midpoint to cut set in half  int imid = midpoint(imin, imax);  // three-way comparison  IF (A[imid] > key)  // key is in 🡨 lower subset  RETURN binary\_search(A, key, imin, imid-1);  ELSE IF (A[imid] < key)  // key is in 🡪 higher subset  RETURN binary\_search(A, key, imid+1, imax);  ELSE  // key has been found  RETURN imid;  }  }  **Algorithm StraightMaxMin:**  VOID StraightMaxMin (Type a[], int n, Type& max, Type& min)  // Set max to the maximum and min to the minimum of a[1:n].  { max = min = a[1];  FOR (int i=2; i<=n; i++)  {  IF (a[i]>max) then max = a[i];  IF (a[i]<min) min = a[i];  }  }  **Algorithm: Recursive Max-Min**  VOID MaxMin(int i, int j, Type& max, Type& min)  // A[1:n] is a global array. Parameters i and j are integers, 1 <= i <= j <= n.  //The effect is to set max and min to the largest and smallest values in a[i:j], respectively.  {  IF (i == j) max = min = a[i]; // Small(P)  ELSE IF (i == j-1) { // Another case of Small(P)  IF (a[i] < a[j])  max = a[j]; min = a[i];  ELSE { max = a[i]; min = a[j];  }  ELSE { Type max1, min1;  // If P is not small divide P into subproblems. Find where to split the set.  int mid=(i+j)/2;  // solve the sub problems.  MaxMin(i, mid, max, min);  MaxMin(mid+1, j, max1, min1);  // Combine the solutions.  IF (max < max1) max = max1;  IF (min > min1) min = min1;  }  } |

|  |
| --- |
| **Output:** |
| **BINARY METHOD ITERATIVE :**    **BINARY METHOD RECURSIVE:**    **Max-Min METHOD:** |

|  |
| --- |
| **Observation Table:** |
| BINARY SEARCH METHOD:      MAX MIN METHOD: |

|  |
| --- |
| **Calculations:** |
|  |

|  |
| --- |
| **Calculations:** |
| **The space complexity of Recursive Binary Search:**  O(log2 n)  **The Time complexity of Binary Search:**  O(log2 n)  **The space complexity of Max-Min:**  O(log2 n)  **Time complexity for Max-Min:**  O(n) |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| Solve the problems theoretically which was implemented during practical |

|  |
| --- |
| **Conclusion:** |
| We have successfully implemented divide and conquer technique to binary search algorithm and to max min algorithm. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |