|  |  |
| --- | --- |
| **NAME:** | MIHIKA CHACHAD |
| **UID:** | 2021300017 |
| **SUBJECT** | Design and Analysis of Algorithms |
| **EXPERIMENT NO:** | 2 |
| **DATE OF PERFORMANCE** | 09.02.2023 |
| **DATE OF SUBMISSION** | 22.02.2023 |
| **AIM:** | Experiment on finding the running time of an algorithm based on the divide and conquer strategy. |
| **THEORY:** | Merge sort: Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.  Quick sort: It picks an element as a pivot and partitions the given array around the picked pivot. The key process in quicksort is the partition. The partition puts x at its correct position in a sorted array and puts elements smaller than x before it and elements larger than x after it. |
| **ALGORITHM:** | 1. Quicksort   quickSort(arr[], low, high) {  if (low < high) {  /\* pi is partitioning index, arr[pi] is now at right place \*/  pi = partition(arr, low, high);  quickSort(arr, low, pi – 1); // Before pi  quickSort(arr, pi + 1, high); // After pi  }  }  partition (arr[], low, high){  // pivot (Element to be placed at right position)  pivot = arr[high]; //low for starting element or middle element  i = (low – 1) // Index of smaller element and indicates the  // right position of pivot found so far  for (j = low; j <= high- 1; j++){  // If the current element is smaller than the pivot  if (arr[j] < pivot){  i++; // increment index of the smaller element  swap arr[i] and arr[j]  }  }  swap arr[i + 1] and arr[high])  return (i + 1)  }   1. Merge sort   MergeSort(arr[], l, r)  If r > l  //Find the middle point to divide the array into two halves:  middle m = l + (r – l)/2  Call mergeSort for first half:  Call mergeSort(arr, l, m)  Call mergeSort for second half:  Call mergeSort(arr, m + 1, r)  Merge the two halves sorted in steps 2 and 3:  Call merge(arr, l, m, r) |
| **PROGRAM:** | 1. Mergesort:   #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void merge(int arr[], int p, int q, int r) {    int n1 = q - p + 1;    int n2 = r - q;    int L[n1], M[n2];    for (int i = 0; i < n1; i++){      L[i] = arr[p + i];    }    for (int j = 0; j < n2; j++){      M[j] = arr[q + 1 + j];    }    int i, j, k;    i = 0;    j = 0;    k = p;    // Until we reach either end of either L or M, pick larger among    // elements L and M and place them in the correct position at A[p..r]    while (i < n1 && j < n2) {      if (L[i] <= M[j]) {        arr[k] = L[i];        i++;      }      else {        arr[k] = M[j];        j++;      }      k++;    }    while (i < n1) {      arr[k] = L[i];      i++;      k++;    }    while (j < n2) {      arr[k] = M[j];      j++;      k++;    }  }  void mergeSort(int arr[], int l, int r) {    if (l < r) {      // m is the point where the array is divided into two subarrays      int m = l + (r - l) / 2;      mergeSort(arr, l, m);      mergeSort(arr, m + 1, r);      merge(arr, l, m, r);    }  }  int main(){      FILE \*fptr, \*sPtr;      int index=99;      int arrNums[100000];      clock\_t t;      fptr = fopen("Random.txt", "r");      sPtr = fopen("iTimes.txt", "w");      for(int i=0; i<=999; i++){          for(int j=0; j<=index; j++){              fscanf(fptr, "%d", &arrNums[j]);          }          t = clock();          mergeSort(arrNums, 0, index);          t = clock() - t;          double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;          fprintf(sPtr, "%lf\n", time\_taken);          printf("\t%lf\n", time\_taken);          index = index + 100;          fseek(fptr, 0, SEEK\_SET);      }      fclose(sPtr);      fclose(fptr);      return 0;  }   1. Quicksort   #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void swap(int \*a, int \*b) {    int t = \*a;    \*a = \*b;    \*b = t;  }  // function to find the partition position  int partition(int array[], int low, int high) {    // select the rightmost element as pivot    int pivot = array[high];    int i = (low - 1);    // traverse each element of the array    // compare them with the pivot    for (int j = low; j < high; j++) {      if (array[j] <= pivot) {        // if element smaller than pivot is found        // swap it with the greater element pointed by i        i++;        swap(&array[i], &array[j]);      }    }    swap(&array[i + 1], &array[high]);    return (i + 1);  }  void quickSort(int array[], int low, int high) {    if (low < high) {      int pi = partition(array, low, high);      quickSort(array, low, pi - 1);      quickSort(array, pi + 1, high);    }  }  int main(){      FILE \*fptr, \*sPtr;      int index=99;      int arrNums[100000];      clock\_t t;      fptr = fopen("Random.txt", "r");      sPtr = fopen("iTimes.txt", "w");      for(int i=0; i<=999; i++){          for(int j=0; j<=index; j++){              fscanf(fptr, "%d", &arrNums[j]);          }          t = clock();          quickSort(arrNums, 1, index);          t = clock() - t;          double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;          fprintf(sPtr, "%lf\n", time\_taken);          printf("\t%lf\n", time\_taken);          index = index + 100;          fseek(fptr, 0, SEEK\_SET);      }      fclose(sPtr);      fclose(fptr);      return 0;  } |
| RESULT:    From the graph, it can be seen that as the size of the input increases the time taken for sorting increases in a kind of linear manner. Merge sort is efficient for high input sizes.    From the graph, it can be seen that as the size of the input increases the time taken for sorting increases in a kind of linear manner. It is efficient on large data sets. It has a low overhead, as it only requires a small amount of memory to function.     * When we compare the time taken for sorting by both algorithms, we find that quick sort takes less time. * Hence, it can be appropriate to say that quick sort is more efficient than merge sort. | |
| **CONCLUSION:** | In this experiment, we learned about merge and quick sort algorithms. We performed sorting 1000 times and compared the time complexities for both algorithms using charts. |