|  |  |
| --- | --- |
| **NAME:** | VIJESH HINGU |
| **UID:** | 2021300042 |
| **BATCH** | C |
| **SUBJECT** | DAA |
| **EXPERIMENT NO :** | 2 |
| **DATE OF PERFORMANCE** | 13-02-2023 |
| **DATE OF SUBMISSION** | 20-02-2023 |
| **AIM:** | To find the running time of merge sort and quick sort. |
| **ALGORITHM** | Merge sort –   1. Start 2. declare array and l, r and m. 3. Perform merge function. 4. if l > r         return     mid= 1+(r-l)/2     mergesort(array, l, m)     mergesort(array, m+1, r)     merge(array, l, m, r)   Quick sort –   1. Start 2. declare array and l, r and m. 3. Perform partition function. 4. If (low < high)   int pi = partition(arr, low, high);  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high); |
| **PROGRAM** | #include <stdio.h>  #include <stdlib.h>  #include <time.h>  const int limit = 100000;  const int block = 100;  void merge (int arr[], int l, int m, int r)  {  int i = 0, j = 0, k = l;  int n1 = m - l + 1;  int n2 = r - m;  int L[n1], R[n2];  for (i = 0; i < n1; i++)  L[i] = arr[l + i];  for (j = 0; j < n2; j++)  R[j] = arr[m + 1 + j];  while (i < n1 && j < n2)  {  if (L[i] <= R[j])  {  arr[k] = L[i];  i++;  }  else  {  arr[k] = R[j];  j++;  }  k++;  }  while (i < n1)  {  arr[k] = L[i];  i++;  k++;  }  while (j < n2)  {  arr[k] = R[j];  j++;  k++;  }  }  void mergeSort (int arr[], int l, int r)  {  if (l<r) {  int m = l+(r-l)/2;  mergeSort(arr, l, m);  mergeSort(arr, m + 1, r);  merge(arr, l, m, r);  }  }  void merge\_sort (FILE \*f)  {  printf("Block Size\tTime Taken\n");  int size = 0;  for (int times = 0; times<limit/block; times++)  {  size+=block;  int arr [size];  for (int i = 0; i<size; ++i)  fscanf(f,"%d",&arr[i]);  clock\_t t;  t = clock();  mergeSort(arr, 0, size-1);  t = clock()-t;  double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;  printf("%d\t%lf\n",size,time\_taken);  }  }  int partition (int arr[], int low, int high)  {  int pivot = arr[high];  int i = low-1;  for (int j = low; j <= high - 1; j++)  {  if (arr[j] < pivot)  {  i++;  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  int temp = arr[i+1];  arr[high] = arr[i+1];  arr[i+1] = temp;  return i+1;  }  void quickSort (int arr[], int low, int high)  {  if (low < high) {  int pi = partition(arr, low, high);  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  }  void quick\_sort (FILE \*f)  {  printf("Block Size\tTime Taken\n");  int size = 0;  for (int times = 0; times<limit/block; times++) {  size+=block;  int arr [size];  for (int i = 0; i<size; ++i)  fscanf(f,"%d",&arr[i]);  clock\_t t;  t = clock();  quickSort(arr, 0, size-1);  t = clock()-t;  double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;  printf("%d\t%lf\n",size,time\_taken);  }  }  int main ()  {  FILE \*f;  f = fopen("daa\_2\_random\_integers.txt", "w");  for (int i = 0; i<limit; ++i)  fprintf(f,"%d\n",rand());  merge\_sort(f);  quick\_sort(f);  fclose(f);  return 0;  } |
| **RESULT ( SNAPSHOT):**  Chart comparing merge and quick sort – | |

**CONCLUSION :**

With the help of this experiment, I was able to understand and implement merge sort and quick sort. I was able to differentiate between the runtimes of bot the algorithms for different number of input values.