Suraj Iyer

2021300045

SE Comps A

Batch C

DAA EXPERIMENT 2

Aim - Experiment to find the running time of an algorithm.

Details - The understanding of running time of algorithms is explored by implementing two basic sorting algorithms namely Merge and Quick sorts. These algorithms work as follows.

MergeSort - 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. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

QuickSort - Like Merge Sort, **QuickSort** is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick the first element as a pivot.
* Always pick the last element as a pivot (implemented below)
* Pick a random element as a pivot.
* Pick the median as the pivot.

The key process in quicksortis a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Code -

| #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) {  FILE \*fp;  fp = fopen("daa\_2\_merge\_sort.txt", "w");  fprintf(fp,"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]);  // now our array is ready, we perform merge sort  clock\_t t;  t = clock();  mergeSort(arr, 0, size-1);  t = clock()-t;  double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;  // storing the result in a file  fprintf(fp,"%d\t%lf\n",size,time\_taken);  }  fclose(fp);  }  int partition (int arr[], int low, int high) {  int pivot = arr[high]; // pivot  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) {  FILE \*fp;  fp = fopen("daa\_2\_quick\_sort.txt", "w");  fprintf(fp,"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]);  // now our array is ready, we perform quick sort  clock\_t t;  t = clock();  quickSort(arr, 0, size-1);  t = clock()-t;  double time\_taken = ((double)t)/CLOCKS\_PER\_SEC;  // storing the result in a file  fprintf(fp,"%d\t%lf\n",size,time\_taken);  }  fclose(fp);  }  int main () {  // generating 1,00,000 integers and storing them in a file  FILE \*f;  f = fopen("daa\_2\_random\_integers.txt", "w");  for (int i = 0; i<limit; ++i)  fprintf(f,"%d\n",rand());  // merge sort  merge\_sort(f);  // quick sort  quick\_sort(f);  fclose(f);  return 0;  } |
| --- |

1,00,000 randomly generated numbers -

| 1804289383  846930886  1681692777  1714636915  1957747793  424238335  719885386  1649760492  596516649  1189641421  1025202362  1350490027  783368690  1102520059  2044897763  1967513926  1365180540  1540383426  304089172  1303455736  35005211  521595368  294702567  1726956429  336465782  861021530  278722862  233665123  2145174067  468703135  1101513929  1801979802  1315634022  635723058  1369133069  1125898167  1059961393  and so on… |
| --- |

Size of block along with time taken to sort using merge sort -

| Block Size Time Taken  100 0.000007  200 0.000024  300 0.000016  400 0.000020  500 0.000026  600 0.000031  700 0.000037  800 0.000042  900 0.000082  1000 0.000055  1100 0.000058  1200 0.000064  1300 0.000098  1400 0.000077  1500 0.000099  1600 0.000084  1700 0.000090  1800 0.000095  1900 0.000100  2000 0.000105  2100 0.000113  2200 0.000135  2300 0.000121  2400 0.000127  2500 0.000134  2600 0.000141  2700 0.000180  2800 0.000150  2900 0.000156  3000 0.000196  3100 0.000169  3200 0.000173  3300 0.000179  3400 0.000184  3500 0.000190  3600 0.000216  and so on… |
| --- |

Size of block along with time taken to sort using quick sort -

| Block Size Time Taken  100 0.000009  200 0.000049  300 0.000062  400 0.000093  500 0.000132  600 0.000199  700 0.000216  800 0.000256  900 0.000314  1000 0.000378  1100 0.000451  1200 0.000526  1300 0.000659  1400 0.000711  1500 0.000807  1600 0.000907  1700 0.001015  1800 0.001154  1900 0.001238  2000 0.001367  2100 0.001532  2200 0.001643  2300 0.001941  2400 0.001973  2500 0.002152  2600 0.002292  2700 0.002464  2800 0.002662  2900 0.002817  3000 0.003047  3100 0.003313  3200 0.003636  3300 0.003800  3400 0.004044  3500 0.004228  3600 0.004457  and so on… |
| --- |

Graph of time taken to sort using merge sort and quick sort against size of block to be sorted -

|  |
| --- |

Conclusion - After a hour or two of cumbersome coding, I have finally executed merge sort and quick sort of 1,00,000 integers, taken 100 at a time. It is with clear observation that merge sort is a much more efficient way of sorting your data when it comes in larger sizes, since what took quick sort around 30 minutes, took merge sort mere seconds. We notice that theoretically, the time complexity of both merge sort and quick sort is the same (O(nlogn)), however, in practice merge sort is astonishingly faster. In terms of space complexity, quick sort has higher efficiency (O(logn)) than merge sort (O(n)). Hence we can conclude that merge sort is faster than quick sort in terms of time complexities, however, quick sort is better in terms of space complexity.