

TCP2101 Algorithm Design & Analysis

Trimester 2 2019/2020

TC02/TT06

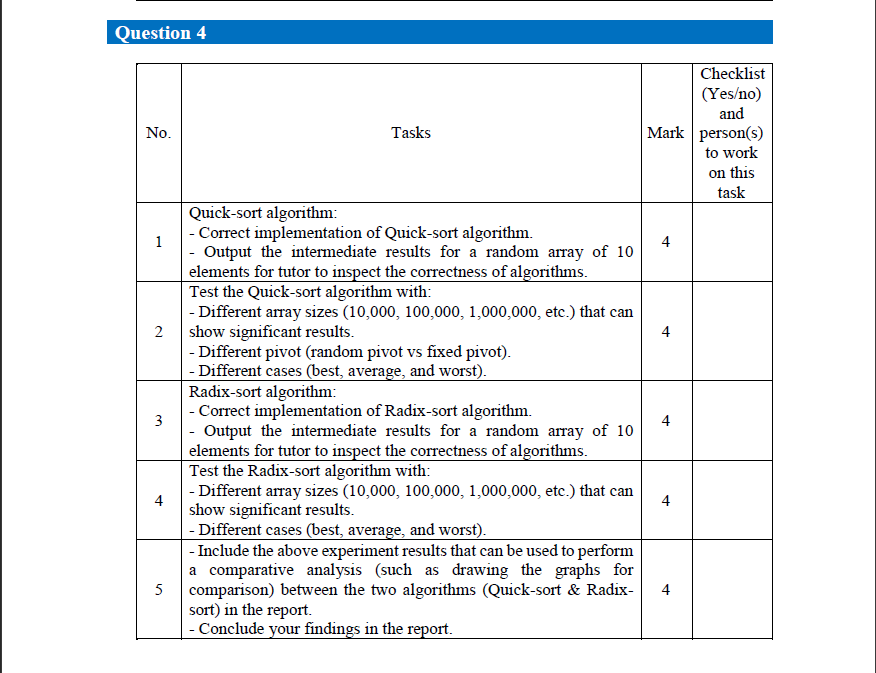
|  |  |  |  |
| --- | --- | --- | --- |
| Name | Student ID | Email | Contact |
| AMIERA NUR HAFIDZAH BINTI MUZAFFAR | 1181302678 | 1181302678@student.mmu.edu.my | +60 14-925 1921 |
| NIZAM, MUSHK | 1181300693 | 1181300693@student.mmu.edu.my | +60 11-2804 0455 |
| SARAH BATRISYIA BINTI AHMAD SALMAN (LEADER) | 1181301724 | 1181301724@student.mmu.edu.my | +60 11-6558 3004 |

**Table of Contents**

Page

|  |  |
| --- | --- |
| **Checklist ---------------------------------------------------------------------------------------------------------** | 2 |
| **Introduction -----------------------------------------------------------------------------------------------------** | 3 |
| **Quick-sort** |  |
| Question 1 : Quick-sort algorithm  1.1 Correct implementation of Quick-sort algorithm.--------------------------------  1.2 Output the intermediate results for a random array of 10 elements  for tutor to inspect the correctness of algorithms.------------------------------------- | 4  5 |
| Question 2 : Test the Quick-sort algorithm with:  2.1 Different array sizes (10,000, 100,000, 1,000,000, etc.) that can  show significant results.------------------------------------------------------------------  2.2 Different pivot (random pivot vs fixed pivot).-----------------------------------  2.3 Different cases (best, average, and worst).--------------------------------------- | 5  6-7  8 |
| **Radix-Sort** |  |
| Question 3 : Radix-sort algorithm  3.1 Correct implementation of Radix-sort algorithm.---------------------------------  3.2 Output the intermediate results for a random array of 10 elements  for tutor to inspect the correctness of algorithms.-------------------------------------- | 9-10  10 |
| Question 4: Test the Radix-sort algorithm with  4.1 Different array sizes (10,000, 100,000, 1,000,000, etc.) that can  show significant results.-------------------------------------------------------------------  4.2 Different cases (best, average, and worst).----------------------------------------- | 11  11 |
| **Analysis**  Question 5:  5.1 Include the above experiment results that can be used to perform a  comparative analysis (such as drawing the graphs for comparison)  between the two algorithms (Quick-sort & Radix-sort) in the report.------------------- | 12-14 |
| **Conclusion**--------------------------------------------------------------------------------------------------------  **Citation**------------------------------------------------------------------------------------------------------------ | 15  16 |

**Checklist**



|  |  |  |  |
| --- | --- | --- | --- |
| No. | Tasks | Mark | Checklist  (Yes/no) and person(s) to work on this task |
| 1. | Quick-sort algorithm:  1.1 Correct implementation of Quick-sort algorithm.  1.2 Output the intermediate results for a random array of 10 elements for tutor to inspect the correctness of algorithms. | 4 | Yes.  (Sarah) |
| 2. | Test the Quick-sort algorithm with:  2.1 Different array sizes (10,000, 100,000, 1,000,000, etc.) that can show significant results.  2.2 Different pivot (random pivot vs fixed pivot).  2.3 Different cases (best, average, and worst). | 4 | Yes.  (Mushk, Sarah) |
| 3. | Radix-sort algorithm:  3.1 Correct implementation of Radix-sort algorithm.  3.2 Output the intermediate results for a random array of 10 elements for tutor to inspect the correctness of algorithms. | 4 | Yes.  (Amiera) |
| 4. | Test the Radix-sort algorithm with:  4.1 Different array sizes (10,000, 100,000, 1,000,000, etc.) that can show significant results.  4.2 Different cases (best, average, and worst). | 4 | Yes.  (Amiera, Sarah) |
| 5. | 5.1 Include the above experiment results that can be used to perform a comparative analysis (such as drawing the graphs for comparison) between the two algorithms (Quick-sort & Radix-sort) in the report.  5.2 Conclude your findings in the report. | 4 | Yes.  (Mushk |

**Introduction**

For this assignment our team were assigned to analyze two sorting algorithms, Quick-sort algorithm and Radix-sort algorithm. Quick-sort algorithm is a sorting algorithm based on the divide-and-conquer paradigm. On the other hand, Radix-sort is a specialization of lexicographic-sort that uses bucket-sort (sorting algorithm that works by distributing the elements of an array into a number of buckets and uses the keys as indices into an auxiliary array of sequences (buckets) as the stable sorting algorithm in each dimension. The objective of this assignment is to implement the Quick-sort algorithm and Radix-sort algorithm correctly. The implemented algorithm will be tested with different size of array and will be analyzed based on the time complexity. For Quick-sort, a random pivot and fixed pivot will be compared. Then, for both Quick-sort and Radix-sort will be tested with the best, average and worst case. Lastly, our team will analyze the comparison between the two algorithms and make a conclusion.

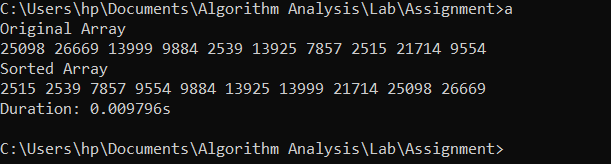
**Quick Sort**

**Question 1:**

1.1 Correct implementation of Quick-sort algorithm.

|  |  |
| --- | --- |
| #include <iostream>  #include <vector>  #include <ctime>  #include<cstdlib>  #include<chrono>  using namespace std;  int partition(int S[], int p,int r)  {  int x= S[p];  int i=p;  int j;  for(j=p+1; j<r; j++)  {  if(S[j]<=x)  {  i=i+1;  swap(S[i],S[j]);  }  }  swap(S[i],S[p]);  return i;  }  void quickSort(int S[], int p,int r)  {  int m;  if(p<r)  {  m=partition(S, p,r);  quickSort(S,p,m);  quickSort(S,m+1,r);  }  } | int main()  {  //const int size = 10;  //const int size = 100;  //const int size = 1000;  //const int size = 10000  const int size=100000;  int \*array = new int[size];  int p=0;  int r=size;  srand((unsigned)time(0));  for(int i=0; i<size; i++){  array[i] = (rand()%RAND\_MAX)+1;    }  cout<<"Original Array"<<endl;  for(int i=0;i<size;i++)  cout<< array[i] <<" ";  cout<< endl;  //Take start time.  auto start = chrono::system\_clock::now();  quickSort(array,p,r-1);  //Take end time.  auto end = chrono::system\_clock::now();  //Calculate duration.  chrono::duration<double> duration = end - start;  cout<<"Sorted Array "<<endl;  for(int i=0;i<size;i++)  cout<< array[i] <<" ";  cout<< endl;  //Display duration.  cout << "Duration: " << duration.count() << "s\n";  delete[] array;  } |

1.2 Intermediate results for a random array of 10 elements.



*Figure 1 shows the output of the random array of 10 elements for Quick-sort*

**Question 2 : Test the Quick-sort algorithm with**

2.1 Different array sizes (10,000, 100,000, 1,000,000, etc.) that can

show significant results.

Process Time (Seconds)

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | First round | Second Round | Third Round |
| 100 | 0s | 0s | 0s |
| 1000 | 0s | 0s | 0.000497s |
| 10000 | 0.000992s | 0.001173s | 0.001081s |
| 100000 | 0.017891s | 0.017793s | 0.019152s |
| 250000 | 0.04594s | 0.045847s | 0.045857s |
| 1000000 | 0.27961s | 0.284575s | 0.288686s |

*Table 1 shows the time taken for Quick-sort algorithm to run with different size of array*

2.2 Different pivot (random pivot vs fixed pivot).

Random Pivot for Quick Sort

Process Time (Seconds)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | First round | Second Round | Third Round | Average |
| 100 | 0s | 0s | 0s | 0s |
| 1000 | 0s | 0s | 0.000496s | 0.0001653s |
| 10000 | 0.001629s | 0.001507s | 0.00152s | 0.001552‬s |
| 100000 | 0.018816s | 0.019153s | 0.019402s | 0.019124s |
| 250000 | 0.049905s | 0.048915s | 0.04891s | 0.049243s |
| 1000000 | 0.264815s | 0.260155s | 0.267842s | 0.264271s |

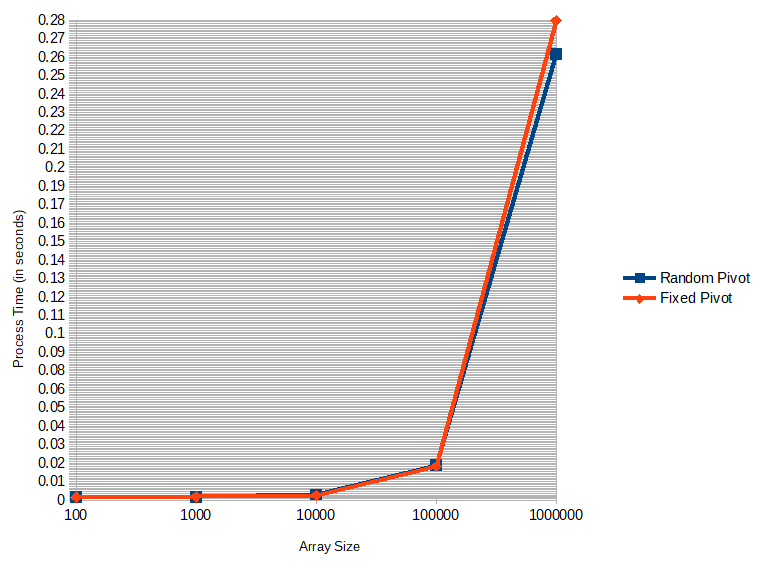
*Table 2 shows the time taken for Quick-sort to run by using a random pivot*

Fixed Pivot for Quick Sort

Process Time (Seconds)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | First round | Second Round | Third Round | Average |
| 100 | 0s | 0s | 0s | 0s |
| 1000 | 0s | 0s | 0.000497s | 0.000167s |
| 10000 | 0.000992s | 0.001173s | 0.001081s | 0.001082s |
| 100000 | 0.017891s | 0.017793s | 0.019152s | 0.018279s |
| 250000 | 0.04594s | 0.045847s | 0.045857s | 0.045881s |
| 1000000 | 0.27961s | 0.284575s | 0.288686s | 0.284290s |

*Table 3 shows the time taken for Quick-sort to run by using a fixed pivot*



*Figure 2 shows the comparison between Quick Sort with fixed Pivot and Random Pivot with*

*different array size.*

2.3 Different cases (best, average, and worst).

**Best case:**

* The best case usually occurs during the partition process where it picks the middle element as pivot.

**Average case:**

* It depends on the pivot that has been chosen. If the element inside the array is randomized, it will lead to a 50-50 chance to get an equal partitioning or not equal.

**Worst case:**

* Worst case happens during the partition process,where it picks the last element as pivot, the worst case would happen if the array is already sorted in increasing or decreasing order. In our example, the worst case happens because the array is sorted in ascending order.

Time Complexity(seconds)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best Case O(n log n) | Average Case O(n log n) | Worst CaseO(n2) |
| Quick Sort | 0.00364s | 0.00364s | 0.135533s |

*Table 4 shows the time taken for Quick-sort to run for Best, Average and Worst case (with size array of 100000).*

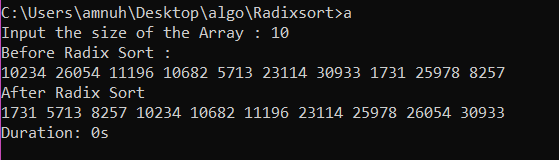
**Radix Sort**

**Question 3:**

3.1 Correct implementation of Radix-sort algorithm.

|  |  |
| --- | --- |
| #include<iostream>  #include <ctime>  #include <chrono>  using namespace std;    // Function to get maximum value in arr[]  int getMax(int arr[], int n)  {  int mx = arr[0];  for (int i = 1; i < n; i++)  if (arr[i] > mx)  mx = arr[i];  return mx;  }  // Function counting sort of arr[]  void countSort(int arr[], int n, int exp)  {  int output[n];  int i, count[10] = {0};    for (i = 0; i < n; i++)  count[ (arr[i]/exp)%10 ]++;    for (i = 1; i < 10; i++)  count[i] += count[i - 1];  swap(S[i],S[p]);  return i;  }  // Driver program  int main()  {  srand (time(0));  int i = 1;  int size;  cout << "Input the size of the Array : ";  cin>>size;  int arr[size];  srand((unsigned)time(NULL));  cout << "Before Radix Sort : " << endl;  for (int i = 0; i < size; i++){  arr[i] = 1+ rand() % 1000001;  show (arr[i]);  } | for (i = n - 1; i >= 0; i--)  {  output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];  count[ (arr[i]/exp)%10 ]--;  }  for (i = 0; i < n; i++)  arr[i] = output[i];  }  // main function Radix Sort  void radixsort(int arr[], int n)  {  int m = getMax(arr, n);    for (int exp = 1; m/exp > 0; exp \*= 10)  countSort(arr, n, exp);  }    // Print an array  void print(int arr[], int n)  {  for (int i = 0; i < n; i++)  cout << arr[i] << " ";  }  int show(int random)  {  cout << random << " ";  return 0;  }    cout << endl;  cout << "After Radix Sort" << endl;  auto start = chrono::system\_clock::now();  int n = sizeof(arr)/sizeof(arr[0]);  radixsort(arr, n);  auto end = chrono::system\_clock::now();  print(arr, n);  chrono::duration<double> duration = end - start;  cout<<endl;  cout << "Duration: " << duration.count() << "s\n";  return 0;  } |

3.2 Intermediate results for a random array of 10 elements.



*Figure 3 shows the output of the random array of 10 elements for Radix-sort*

**Question 4: Test the Radix-sort algorithm with**

4.1 Different array sizes (10,000, 100,000, 1,000,000, etc.) that can show significant results.

(Seconds)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | First round | Second Round | Third Round | Average |
| 100 | 0s | 0s | 0s | 0s |
| 1000 | 0s | 0s | 0s | 0s |
| 10000 | 0s | 0s | 0.00378s | 0.001260s |
| 100000 | 0.015643s | 0.015992s | 0.015629s | 0.015755s |
| 250000 | 0.031244s | 0.03125s | 0.037003s | 0.033166s |

*Table 5 shows the time taken for Radix-sort algorithm to run with different size of array*

4.2 Different cases (best, average, and worst).

Overall time complexity for the radix sort is **O(d(n+N))** where n = d-digits number, N = number that can take up to N possible values.

Best case and Average case

* Radix sort is best alternatives provided word size, w is expected to be less than log n compared to comparison sorts which cannot perform better than O(n log n).

Worst case

* While the word size is large, the key range is large. This situation will make radix sort become slower. It will essentially become a counting sort with a large memory footprint associated with it.

Time Complexity(seconds)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Best Case *O(wn)* | Average Case *O(wn)* | Worst Case *O(n+r) auxiliary* |
| Radix Sort | 0.003999s | 0.00355s | 0.015623s |

*Table 6 shows the time taken for Radix-sort to run for Best, Average and Worst case.*

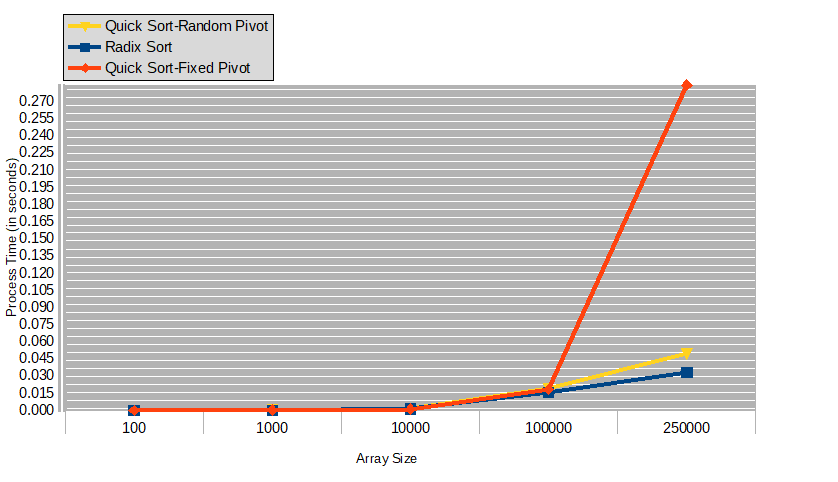
**Question 5 : Perform a comparative analysis (such as drawing the graphs for comparison) between the two algorithms (Quick-sort & Radix-sort) in the report.**

|  |  |
| --- | --- |
| **Quick Sort** | **Radix Sort** |
| Time Complexity is:  O(n log n)  O(n2) for worst case if not randomized | Time Complexity is:  O(d(n+N)) |
| Quick sort compares the item values against each other | Radix sort simply sorts by the binary representation of data. The items(as a whole) are never compared |
| It is massively recursive | It has very little recursion. |
| Quicksort gets slower for big array size | Radix sort is faster with big array size |

*Table 7 shows the general comparison between Quick-sort(fixed and random pivot) and radix sort.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Size of** | **Quick-Sort** |  | **Radix-Sort** |
| **array** | **Fixed pivot** | **Random pivot** |  |
| 100 | 0s | 0s | 0s |
| 1000 | 0.000167s | 0.0001653s | 0s |
| 10000 | 0.001082s | 0.001552‬s | 0.001260s |
| 100000 | 0.018279s | 0.019124s | 0.015755s |
| 250000 | 0.045881s | 0.049243s | 0.033166s |

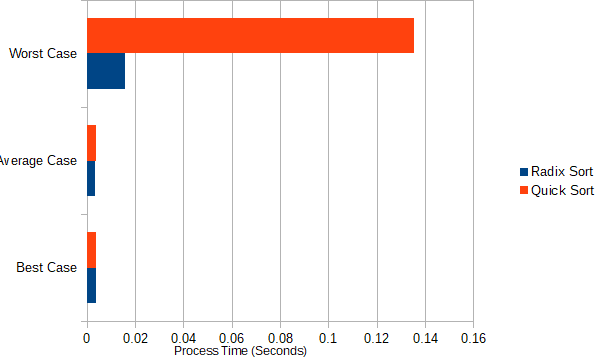
*Table 8 shows the time comparison between Quick-sort(fixed and random pivot) and radix sort.*

****

*Figure 4 Shows comparison between Quick Sort and Radix Sort with different array size.*

|  |  |  |
| --- | --- | --- |
|  | **Quick-Sort** | **Radix-Sort** |
| Best case | 0.00364s | 0.003999s |
| Average case | 0.00364s | 0.00355s |
| Worst case | 0.135533s | 0.015623s |

*Table 9 shows the time comparison for best case, average case and worst case between Quick-sort(fixed and random pivot) and radix sort.*

**

*Figure 5 Comparison between Best, Average and Worst case of Radix and Quick Sort.*

**Conclusion**

After experimenting with both Quicksort and Radix Sort algorithms, we can conclude that Radix sort is faster than Quick Sort. When the input size is smaller e.g 100, 1000 the difference between the process time is less but as the array size increases with significant number then the radix sort shows major increase than the quick sort.

For quick sort, using random pivot rather than fixed pivot can increase the process time as there is an overhead of choosing the random pivot whereas in fixed pivot quick sort, the last element is considered as the pivot.

Radix Sort is still considered faster than Quicksort because the difference between their best case,average case and worst case is very little while the difference between best case, average case and worst case of Quick Sort is huge.

**Citations**

* Radix Sort
* Radix sort code <https://www.geeksforgeeks.org/radix-sort/>
* Best, Average, Worst <https://www.growingwiththeweb.com/sorting/radix-sort-lsd/>
* Time complexity <https://books.google.com.my/books?id=NLngYyWFl_YC&pg=PA172&lpg=PA172&dq=O(d(n%2BN))+radix&source=bl&ots=BzVmHG3fDd&sig=ACfU3U300DjBRMEdrCXhv6BftQcPBHzPqA&hl=en&sa=X&ved=2ahUKEwj57cqz77bnAhWi6XMBHSNoCFkQ6AEwCnoECA0QAQ#v=onepage&q=O(d(n%2BN))%20radix&f=false>
* Time counter in second - Lab 1 (TCP2101 Algorithm Design and Analysis)
* Quick-sort pseudocode - Lecture 6b (TCP2101 Algorithm Design and Analysis)
* Quick-sort-<https://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/>
* Quick-sort-<https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm>