|  |
| --- |
| MMU_MAIN PAGE.JPG |
| TCP2101 Algorithm Design & Analysis |
| Assignment |
|  |
|  |
| **29-Jan-16** |

|  |
| --- |
|  |

**Group Member :**

Oh Soon Kit 1132702874

Mahaseni Behzad 1141125333

**Table of content**

1. Introduction……………………………………………………..3
2. Implementation of algorithm………...……………………….....3
   1. Different array sizes..........................................................7
   2. Different array patterns………………………………….8

2.2.1 Random array pattern…………………………………8

2.2.2 Sorted array pattern…………………………………...11

* 1. Different cases…………………………………………..16

2.3.1 Worst case ……………………………………………16

2.3.2 Best case………………………………………...…….17

2.3.3 Average case………………………………………….17

* 1. Different pivot selection for quick-sort
     1. Last element…………………………………………18
     2. Random……………………………………………...18

1. Conclusion……………………………………………………..23
2. **Introduction**

Perform a comparative analysis between two sorting algorithms which are quick-sort and selection-sort.

1. **Implementation of algorithm**
   1. Quick-Sort
      1. Definition

Quicksort is a type comparison sort that can sort items of any type for a total order is defined. In efficient implementations it is not a stable sort because the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.

* + 1. Algorithm

|  |
| --- |
| Quick-Sort |
| void quickSort**(**int arr**[],** int l**,** int h**)** **{**  **if** **(**l **<** h**)** **{**  int p **=** partition**(**arr**,** l**,** h**);**  quickSort**(**arr**,** l**,** p **-** 1**);**  quickSort**(**arr**,** p **+** 1**,** h**);**  **}**  **}**  void swap**(**int**\*** a**,** int**\*** b**)** **{**  int t **=** **\***a**;**  **\***a **=** **\***b**;**  **\***b **=** t**;**  **}**  int partition**(**int arr**[],** int l**,** int h**)** **{**  int x **=** arr**[**h**];**  int i **=** **(**l **-** 1**);**  **for** **(**int j **=** l**;** j **<=** h **-** 1**;** j**++)** **{**  **if** **(**arr**[**j**]** **<=** x**)** **{**  i**++;**  swap**(&**arr**[**i**],** **&**arr**[**j**]);**  **}**  **}**  swap**(&**arr**[**i **+** 1**],** **&**arr**[**h**]);**  **return** **(**i **+** 1**);**  **}** |

* + 1. Code:

|  |
| --- |
| Quick-Sort |
| #include<stdio.h>  #include <ctime>  #include <iostream>  #include <cstdlib>  #include <iostream>  #include <fstream>  **using** **namespace** std**;**  long length **=** 10**;**  const long max\_length **=** 10**;**  int list**[**max\_length**];**  void quickSort**(**int arr**[],** int l**,** int h**);**  void printArray**(**int arr**[],** int size**);**  void random**()** **{**  srand**((**unsigned**)** time**(**0**));**  **for** **(**int i **=** 0**;** i **<** length**;** i**++)** **{**  list**[**i**]** **=** **(**rand**()** **%** 100**)** **+** 1**;**  **}**  **}**  void swap**(**int**\*** a**,** int**\*** b**)** **{**  int t **=** **\***a**;**  **\***a **=** **\***b**;**  **\***b **=** t**;**  **}**  int partition**(**int arr**[],** int l**,** int h**)** **{**  int x **=** arr**[**h**];**  int i **=** **(**l **-** 1**);**  **for** **(**int j **=** l**;** j **<=** h **-** 1**;** j**++)** **{**  **if** **(**arr**[**j**]** **<=** x**)** **{**  i**++;**  printArray**(**arr**,** length**);**  swap**(&**arr**[**i**],** **&**arr**[**j**]);**  **}**  **}**  swap**(&**arr**[**i **+** 1**],** **&**arr**[**h**]);**  **return** **(**i **+** 1**);**  **}**  void quickSort**(**int arr**[],** int l**,** int h**)** **{**  **if** **(**l **<** h**)** **{**  int p **=** partition**(**arr**,** l**,** h**);**  quickSort**(**arr**,** l**,** p **-** 1**);**  quickSort**(**arr**,** p **+** 1**,** h**);**  **}**  **}**  void printArray**(**int arr**[],** int size**)** **{**  int i**;**  **for** **(**i **=** 0**;** i **<** size**;** i**++)**  printf**(**"%d "**,** arr**[**i**]);**  printf**(**"\n"**);**  **}**  int main**()** **{**  double t1**,** t2**;**  cout **<<** "\nLength\t: " **<<** length **<<** '\n'**;**  random**();**  t1 **=** clock**();**  quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  **return** 0**;**  **}** |

* + 1. **Output**

Length :10

This is the array before sort : 18 78 78 52 42 6 91 73 95 41

18 78 78 52 42 6 91 73 95 41

18 6 41 52 42 78 91 73 95 78

6 18 41 52 42 78 91 73 95 78

6 18 41 52 42 78 73 78 95 91

6 18 41 52 42 73 78 78 91 95

6 18 41 42 52 73 78 78 91 95

* 1. Selection-Sort
     1. Definition

The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest or largest value is moved to its proper position in the array. The number of times the sortpasses through the array is one less than the number of items in the array.

* + 1. Algorithm

|  |
| --- |
| Selection-Sort |
| void selectionSort**(**int arr**[],** int n**)** **{**  int i**,** j**,** minIndex**,** tmp**;**  **for** **(**i **=** 0**;** i **<** n **-** 1**;** i**++)** **{**  minIndex **=** i**;**  **for** **(**j **=** i **+** 1**;** j **<** n**;** j**++)**  **if** **(**arr**[**j**]** **<** arr**[**minIndex**])**  minIndex **=** j**;**  **if** **(**minIndex **!=** i**)** **{**  tmp **=** arr**[**i**];**  arr**[**i**]** **=** arr**[**minIndex**];**  arr**[**minIndex**]** **=** tmp**;**  **}**  **}**  **}** |

* + 1. Code

|  |
| --- |
| Selection-Sort |
| #include<stdio.h>  #include <ctime>  #include <iostream>  #include <cstdlib>  #include <iostream>  #include <fstream>  **using** **namespace** std**;**  long length **=** 10**;**  const long max\_length **=** 10**;**  int list**[**max\_length**];**  void random**()** **{**  srand**((**unsigned**)** time**(**0**));**  **for** **(**int i **=** 0**;** i **<** length**;** i**++)** **{**  list**[**i**]** **=** **(**rand**()** **%** 100**)** **+** 1**;**  **}**  **}**  void selectionSort**(**int arr**[],** int n**)** **{**  int i**,** j**,** minIndex**,** tmp**;**  **for** **(**i **=** 0**;** i **<** n **-** 1**;** i**++)** **{**  minIndex **=** i**;**  **for** **(**j **=** i **+** 1**;** j **<** n**;** j**++)**  **if** **(**arr**[**j**]** **<** arr**[**minIndex**])**  minIndex **=** j**;**  **if** **(**minIndex **!=** i**)** **{**  printArray**(**arr**,** length**);**  tmp **=** arr**[**i**];**  arr**[**i**]** **=** arr**[**minIndex**];**  arr**[**minIndex**]** **=** tmp**;**  **}**  **}**  **}**  void printArray**(**int arr**[],** int size**)** **{**  int i**;**  **for** **(**i **=** 0**;** i **<** size**;** i**++)**  printf**(**"%d "**,** arr**[**i**]);**  printf**(**"\n"**);**  **}**  int main**()** **{**  double t1**,** t2**;**  cout **<<** "\nLength\t: " **<<** length **<<** '\n'**;**  random**();**  t1 **=** clock**();**  selectionSort **(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Selection Sort\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  **return** 0**;**  **}** |

* + 1. Output

Length :10

This is the array before sort : 6 79 97 72 57 90 76 20 93 1

6 79 97 72 57 90 76 20 93 1

1 79 97 72 57 90 76 20 93 6

1 6 97 72 57 90 76 20 93 79

1 6 72 57 76 20 79 90 93 97

1 6 20 57 76 72 79 90 93 97

1 6 20 57 72 76 79 90 93 97

1 6 20 57 72 76 79 90 93 97

* 1. **Different array sizes**

**2.2.1 Random Array Pattern**

* 1. Code:

|  |
| --- |
| Quick-Sort and Selection-Sort |
| #include<stdio.h>  #include <ctime>  #include <iostream>  #include <cstdlib>  #include <iostream>  #include <fstream>  **using** **namespace** std**;**  long length **=** 10**;**  const long max\_length **=** 300000**;**  int list**[**max\_length**];**  void quickSort**(**int arr**[],** int l**,** int h**);**  void read**()** **{**  srand**((**unsigned**)** time**(**0**));**  **for** **(**int i **=** 0**;** i **<** length**;** i**++)** **{**  list**[**i**]** **=** **(**rand**()** **%** 100**)** **+** 1**;**  **}**  **}**  void swap**(**int**\*** a**,** int**\*** b**)** **{**  int t **=** **\***a**;**  **\***a **=** **\***b**;**  **\***b **=** t**;**  **}**  int partition**(**int arr**[],** int l**,** int h**)** **{**  int x **=** arr**[**h**];**  int i **=** **(**l **-** 1**);**  **for** **(**int j **=** l**;** j **<=** h **-** 1**;** j**++)** **{**  **if** **(**arr**[**j**]** **<=** x**)** **{**  i**++;** // increment index of smaller element  swap**(&**arr**[**i**],** **&**arr**[**j**]);** **}**  **}**  swap**(&**arr**[**i **+** 1**],** **&**arr**[**h**]);**  **return** **(**i **+** 1**);**  **}**  void quickSort**(**int arr**[],** int l**,** int h**)** **{**  **if** **(**l **<** h**)** **{**  int p **=** partition**(**arr**,** l**,** h**);** /\* Partitioning index \*/  quickSort**(**arr**,** l**,** p **-** 1**);**  quickSort**(**arr**,** p **+** 1**,** h**);**  **}**  **}**  void printArray**(**int arr**[],** int size**)** **{**  **for** **(**int i **=** 0**;** i **<** size**;** i**++)**  printf**(**"%d "**,** arr**[**i**]);**  printf**(**"\n"**);**  **}**  void selectionSort**(**int arr**[],** int n**)** **{**  int minIndex**,** tmp**;**  **for** **(**int i **=** 0**;** i **<** n **-** 1**;** i**++)** **{**  minIndex **=** i**;**  **for** **(**int j **=** i **+** 1**;** j **<** n**;** j**++)**  **if** **(**arr**[**j**]** **<** arr**[**minIndex**])**  minIndex **=** j**;**  **if** **(**minIndex **!=** i**)** **{**  tmp **=** arr**[**i**];**  arr**[**i**]** **=** arr**[**minIndex**];**  arr**[**minIndex**]** **=** tmp**;**  **}**  **}**  **}**  int main**()** **{**  double t1**,** t2**;**  **for** **(**length **=** 10**;** length **<=** max\_length**;)** **{**  cout **<<** "\nLength\t: " **<<** length **<<** '\n'**;**  read**();**  t1 **=** clock**();**  quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  read**();**  t1 **=** clock**();**  selectionSort**(**list**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "selection Sort\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  **switch** **(**length**)** **{**  case 10:  length = 100;  break;  case 100:  length = 1000;  break;  case 1000:  length = 5000;  break;  case 5000:  length = 10000;  break;  case 10000:  length = 15000;  break;  case 15000:  length = 20000;  break;  case 20000:  length = 25000;  break;  case 25000:  length = 50000;  break;  case 50000:  length = 100000;  break;  case 100000:  length = 200000;  break;  case 200000:  length = 300000;  break;  case 300000:  length = 300001;  break;  }  }  return 0;  } |

* 1. Output
  2. **Plot**

Figure 3.1 shows the running time vs input size graph of random array pattern for both quick-sort

And selection sort

* 1. **Result Table**

|  |  |  |
| --- | --- | --- |
| Length | **Quick Sort** | **Selection Sort** |
| 10 | 0 | 0 |
| 100 | 0 | 0 |
| 1000 | 0 | 0 |
| 5000 | 0 | 0.031 |
| 10000 | 0.016 | 0.138 |
| 15000 | 0 | 0.285 |
| 20000 | 0.015 | 0.517 |
| 25000 | 0.016 | 0.802 |
| 50000 | 0.1 | 3.234 |
| 100000 | 0.442 | 13.743 |
| 200000 | 1.672 | 53.131 |
| 300000 | 3.882 | 119.495 |

**2.2.2 Sorted Array Pattern**

Code

|  |
| --- |
| Quick-Sort and Selection-Sort |
| #include<stdio.h>  #include <ctime>  #include <iostream>  #include <cstdlib>  #include <iostream>  #include <fstream>  **using** **namespace** std**;**  long length **=** 10**;**  const long max\_length **=** 40000**;**  int list**[**max\_length**];**  void quickSort**(**int arr**[],** int l**,** int h**);**  void random**()** **{**  srand**((**unsigned**)** time**(**0**));**  **for** **(**int i **=** 0**;** i **<** length**;** i**++)** **{**  list**[**i**]** **=** **(**rand**()** **%** 100**)** **+** 1**;**  **}**  **}**  void sorted**(){**  **for** **(**int i **=** 0**;** i **<** length**;** i**++)** **{**  list**[**i**]** **=** i**;**  **}}**  // A utility function to swap two elements  void swap**(**int**\*** a**,** int**\*** b**)** **{**  int t **=** **\***a**;**  **\***a **=** **\***b**;**  **\***b **=** t**;**  **}**  /\* This function takes last element as pivot, places the pivot element at its  correct position in sorted array, and places all smaller (smaller than pivot)  to left of pivot and all greater elements to right of pivot \*/  int partition**(**int arr**[],** int l**,** int h**)** **{**  int x **=** arr**[**h**];** // pivot  int i **=** **(**l **-** 1**);** // Index of smaller element  **for** **(**int j **=** l**;** j **<=** h **-** 1**;** j**++)** **{**  // If current element is smaller than or equal to pivot  **if** **(**arr**[**j**]** **<=** x**)** **{**  i**++;** // increment index of smaller element  swap**(&**arr**[**i**],** **&**arr**[**j**]);** // Swap current element with index  **}**  **}**  swap**(&**arr**[**i **+** 1**],** **&**arr**[**h**]);**  **return** **(**i **+** 1**);**  **}**  /\* arr[] --> Array to be sorted-quickSort, l --> Starting index, h --> Ending index \*/  void quickSort**(**int arr**[],** int l**,** int h**)** **{**  **if** **(**l **<** h**)** **{**  int p **=** partition**(**arr**,** l**,** h**);** /\* Partitioning index \*/  quickSort**(**arr**,** l**,** p **-** 1**);**  quickSort**(**arr**,** p **+** 1**,** h**);**  **}**  **}**  /\* Function to print an array \*/  void printArray**(**int arr**[],** int size**)** **{**  **for** **(**int i **=** 0**;** i **<** size**;** i**++)**  printf**(**"%d "**,** arr**[**i**]);**  printf**(**"\n"**);**  **}**  /\* arr[] --> Array to be sorted-selectionSort\*/  void selectionSort**(**int arr**[],** int n**)** **{**  int minIndex**,** tmp**;**  **for** **(**int i **=** 0**;** i **<** n **-** 1**;** i**++)** **{**  minIndex **=** i**;**  **for** **(**int j **=** i **+** 1**;** j **<** n**;** j**++)**  **if** **(**arr**[**j**]** **<** arr**[**minIndex**])**  minIndex **=** j**;**  **if** **(**minIndex **!=** i**)** **{**  tmp **=** arr**[**i**];**  arr**[**i**]** **=** arr**[**minIndex**];**  arr**[**minIndex**]** **=** tmp**;**  **}**  **}**  **}**  // Driver program to test above functions  int main**()** **{**  double t1**,** t2**;**  // printArray(list, length);  **for** **(**length **=** 10**;** length **<=** max\_length**;)** **{**  cout **<<** "\nLength\t: " **<<** length **<<** '\n'**;**  random**();**  t1 **=** clock**();**  quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort with Random List With Last Element As Pivot\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  random**();**  t1 **=** clock**();**  selectionSort**(**list**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Selection Sort with Random List\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  sorted**();**  t1 **=** clock**();**  quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort with Sorted List With Last Element As Pivot\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  sorted**();**  t1 **=** clock**();**  selectionSort**(**list**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Selection Sort with Sorted List\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  **switch** **(**length**)** **{**  **case** 10**:**  length **=** 100**;**  **break;**  **case** 100**:**  length **=** 1000**;**  **break;**  **case** 1000**:**  length **=** 5000**;**  **break;**  **case** 5000**:**  length **=** 10000**;**  **break;**  **case** 10000**:**  length **=** 15000**;**  **break;**  **case** 15000**:**  length **=** 20000**;**  **break;**  **case** 20000**:**  length **=** 25000**;**  **break;**  **case** 25000**:**  length **=** 30000**;**  **break;**  **case** 30000**:**  length **=** 40000**;**  **break;**  **case** 40000**:**  length **=** 40001**;**  **break;**  **}**  **}**  **return** 0**;**} |

* 1. Output

**i)Sorted Array Pattern**

**Plot**

Figure 3.2 shows the running time vs input size graph of sorted array pattern for both quick-sort

And selection sort

* 1. **Result Table**

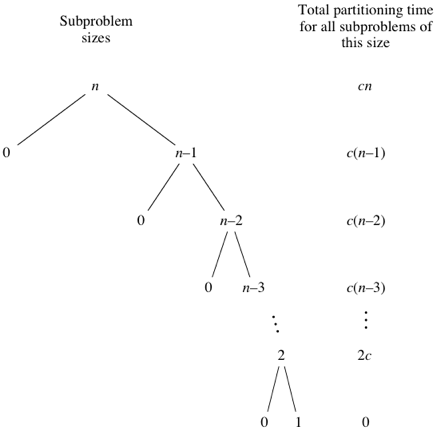
|  |  |  |
| --- | --- | --- |
| Length | **Quick Sort** | **Selection Sort** |
| 10 | 0 | 0 |
| 100 | 0 | 0 |
| 1000 | 0 | 0 |
| 5000 | 0 | 0.031 |
| 10000 | 0.016 | 0.138 |
| 15000 | 0 | 0.285 |
| 20000 | 0.015 | 0.517 |
| 25000 | 0.016 | 0.802 |
| 40000 | 0.1 | 3.234 |

* 1. **Different cases**

**i)Quick-Sort**

**Worst-case running time**

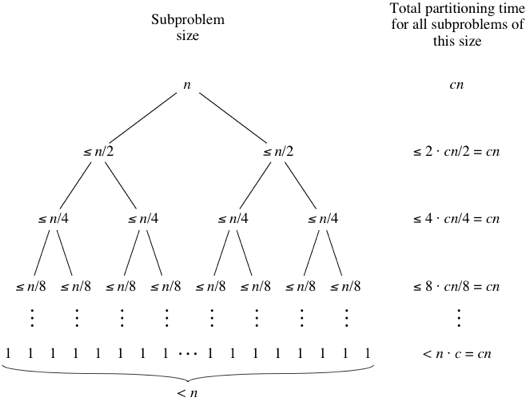
Quicksort always experience unbalanced partitions.For instance,the original call takes cn*cn* time for some constant c*c*, the recursive call on n-1*n*−1elements takes c(n-1)*c*(*n*−1) time, the recursive call on n-2*n*−2 elements takes c(n-2)*c*(*n*−2) time, and so on. Here's a tree of the subproblem sizes with their partitioning times:



when partitioning times for each level get combined,the result is as below  
 *cn*+*c*(*n*−1)+*c*(*n*−2)+⋯+2*c*=*c*(*n*+(*n*−1)+(*n*−2)+⋯+2)=*c*((*n*+1)(*n*/2)−1) .  
  
some low-order terms and constant coefficients might occur, but when big-Oh notation is used,they can be ignored. In big-Oh notation, quicksort's worst-case running time is O(n^2),it can be clearly seen from the sorted array pattern.

**Best-case running time**

Quicksort's best case occurs when the partitions are evenly balanced.The case below occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2(*n*−1)/2elements. The latter case occurs if the subarray has an even number n*n* of elements and one partition has n/2*n*/2 elements with the other having n/2-1*n*/2−1. In either of these cases, each partition has at most n/2*n*/2 elements, and the tree of subproblem sizes is similar to the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:



In big-Oh notation, quicksort's best-case running time is O(nlogn),it can be clearly seen from the random array pattern.

### Average-case running times

### the average-case running time of quick-sort is http://www.cs.nmsu.edu/~ipivkina/Spring08cs372/Cormen/chapter8Quicksort_files/bound.gif(*n* 1g *n*).For instance.if the split induced by randomized quick-sort puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth http://www.cs.nmsu.edu/~ipivkina/Spring08cs372/Cormen/chapter8Quicksort_files/bound.gif(1g *n*) and http://www.cs.nmsu.edu/~ipivkina/Spring08cs372/Cormen/chapter8Quicksort_files/bound.gif(*n*) work is performed at http://www.cs.nmsu.edu/~ipivkina/Spring08cs372/Cormen/chapter8Quicksort_files/bound.gif(1g *n*) of these levels. In big-Oh notation, quicksort's average-case running time is O(nlogn),it can be clearly seen from the random array pattern.

### ii)Selection-Sort

Selection sort is easy to analyze since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all *n* elements (this takes *n* − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining *n* − 1 elements and so on, for (*n* − 1) + (*n* − 2) + ... + 2 + 1 = *n*(*n* − 1) / 2 ∈ Θ(*n*2) comparisons (see [arithmetic progression](https://en.wikipedia.org/wiki/Arithmetic_progression)). Each of these scans requires one swap for *n* − 1 elements and the final element is already in place.In big-Oh notation, selection-sort's running time is O(n^2).

1. **Different pivot selection for quick-sort**

Choosing a random pivot minimizes the chance that you will encounter worst-case O(n2) performance.In general, frequent choosing first or last would cause worst-case performance for nearly-(sorted or reverse-sorted) data. Choosing the middle element would also be acceptable in the majority of cases.

* 1. Partition function can be easily replaced with a function that can choose pivot

randomly.

* 1. Code

|  |
| --- |
| Code for Random Pivot |
| int random\_partition**(**int**\*** arr**,** int start**,** int end**)**  **{**  srand**(**time**(NULL));**  int pivotIdx **=** start **+** rand**()** **%** **(**end**-**start**+**1**);**  int pivot **=** arr**[**pivotIdx**];**  swap**(**arr**[**pivotIdx**],** arr**[**end**]);** // move pivot element to the end  pivotIdx **=** end**;**  int i **=** start **-**1**;**  **for(**int j**=**start**;** j**<=**end**-**1**;** j**++)**  **{**  **if(**arr**[**j**]** **<=** pivot**)**  **{**  i **=** i**+**1**;**  swap**(**arr**[**i**],** arr**[**j**]);**  **}**  **}**  swap**(**arr**[**i**+**1**],** arr**[**pivotIdx**]);**  **return** i**+**1**;**  **}** |

* 1. Code

|  |
| --- |
| Code for Implementation Random Pivot |
| #include<stdio.h>  #include <ctime>  #include <iostream>  #include <cstdlib>  #include <iostream>  #include <fstream>  **using** **namespace** std**;**  long length **=** 10**;**  const long max\_length **=** 35000**;**  int list**[**max\_length**];**  void quickSort**(**int arr**[],** int l**,** int h**);**  void printArray**(**int arr**[],** int size**);**  void sorted**(){**  **for** **(**int i **=** 0**;** i **<** length**;** i**++)** **{**  list**[**i**]** **=** i**;**  **}}**  void random**()** **{**  srand**((**unsigned**)** time**(**0**));**  **for** **(**int i **=** 0**;** i **<** length**;** i**++)** **{**  list**[**i**]** **=** **(**rand**()** **%** 100**)** **+** 1**;**  **}**  **}**  void swap**(**int**\*** a**,** int**\*** b**)** **{**  int t **=** **\***a**;**  **\***a **=** **\***b**;**  **\***b **=** t**;**  **}**  int partition**(**int arr**[],** int l**,** int h**)** **{**  int x **=** arr**[**h**];**  int i **=** **(**l **-** 1**);**  **for** **(**int j **=** l**;** j **<=** h **-** 1**;** j**++)** **{**  **if** **(**arr**[**j**]** **<=** x**)** **{**  i**++;**  swap**(&**arr**[**i**],** **&**arr**[**j**]);**  **}**  **}**  swap**(&**arr**[**i **+** 1**],** **&**arr**[**h**]);**  **return** **(**i **+** 1**);**  **}**  int random\_partition**(**int**\*** arr**,** int start**,** int end**)**  **{**  srand**(**time**(NULL));**  int pivotIdx **=** start **+** rand**()** **%** **(**end**-**start**+**1**);**  int pivot **=** arr**[**pivotIdx**];**  swap**(**arr**[**pivotIdx**],** arr**[**end**]);** // move pivot element to the end  pivotIdx **=** end**;**  int i **=** start **-**1**;**  **for(**int j**=**start**;** j**<=**end**-**1**;** j**++)**  **{**  **if(**arr**[**j**]** **<=** pivot**)**  **{**  i **=** i**+**1**;**  swap**(**arr**[**i**],** arr**[**j**]);**  **}**  **}**  swap**(**arr**[**i**+**1**],** arr**[**pivotIdx**]);**  **return** i**+**1**;**  **}**  void quickSort**(**int arr**[],** int l**,** int h**)** **{**  **if** **(**l **<** h**)** **{**  int p **=** partition**(**arr**,** l**,** h**);** /\* Partitioning index \*/  quickSort**(**arr**,** l**,** p **-** 1**);**  quickSort**(**arr**,** p **+** 1**,** h**);**  **}**  **}**  void random\_quickSort**(**int arr**[],** int l**,** int h**)** **{**  **if** **(**l **<** h**)** **{**  int p **=** random\_partition**(**arr**,** l**,** h**);** /\* Partitioning index \*/  quickSort**(**arr**,** l**,** p **-** 1**);**  quickSort**(**arr**,** p **+** 1**,** h**);**  **}**  **}**  void printArray**(**int arr**[],** int size**)** **{**  int i**;**  **for** **(**i **=** 0**;** i **<** size**;** i**++)**  printf**(**"%d "**,** arr**[**i**]);**  printf**(**"\n"**);**  **}**  int main**()** **{**  double t1**,** t2**;**  **for** **(**length **=** 10**;** length **<=** max\_length**;)** **{**  cout **<<** "\nLength\t: " **<<** length **<<** '\n'**;**  random**();**  t1 **=** clock**();**  quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort with Random List With Last Element As Pivot\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  random**();**  t1 **=** clock**();**  random\_quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort with Random List With Random Pivot\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  sorted**();**  t1 **=** clock**();**  quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort with Sorted List With Last Element As Pivot\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  sorted**();**  t1 **=** clock**();**  random\_quickSort**(**list**,** 0**,** length **-** 1**);**  t2 **=** clock**();**  cout **<<** "Quick Sort with Sorted List With Random Pivot\t: " **<<** **(**t2 **-** t1**)** **/** CLK\_TCK **<<** " sec\n"**;**  **switch** **(**length**)** **{**  **case** 10**:**  length **=** 100**;**  **break;**  **case** 100**:**  length **=** 1000**;**  **break;**  **case** 1000**:**  length **=** 5000**;**  **break;**  **case** 5000**:**  length **=** 10000**;**  **break;**  **case** 10000**:**  length **=** 15000**;**  **break;**  **case** 15000**:**  length **=** 20000**;**  **break;**  **case** 20000**:**  length **=** 25000**;**  **break;**  **case** 25000**:**  length **=** 35000**;**  **break;**  **case** 35000**:**  length **=** 35001**;**  **break;**  **}**  **}**  **return** 0**;**  **}** |

* 1. **Output**

**Random Pivot selection for quick-sort in sorted array and random array pattern**

|  |  |  |
| --- | --- | --- |
| Length | **Quick**  **Sort(sorted)** | **Quick Sort(random)** |
| 10 | 0 | 0 |
| 100 | 0 | 0 |
| 1000 | 0.003 | 0 |
| 5000 | 0.053 | 0.002 |
| 10000 | 0.273 | 0.005 |
| 15000 | 0.597 | 0.01 |
| 20000 | 0.83 | 0.017 |
| 25000 | 1.309 | 0.027 |
| 35000 | 2.764 | 0.051 |

**Last Element Pivot selection for quick-sort in sorted array and random array pattern**

|  |  |  |
| --- | --- | --- |
| Length | **Quick**  **Sort(sorted)** | **Quick Sort(random)** |
| 10 | 0 | 0 |
| 100 | 0 | 0.002 |
| 1000 | 0.004 | 0 |
| 5000 | 0.098 | 0.002 |
| 10000 | 0.395 | 0.005 |
| 15000 | 0.909 | 0.01 |
| 20000 | 1.772 | 0.017 |
| 25000 | 2.555 | 0.027 |
| 35000 | 5.092 | 0.052 |

**Plot**

Figure above shows the random and last element pivot selection for quick-sort in random and sorted array pattern.

1. **Conclusion**

**Quick-sort Vs Selection-sort**

Quick sort is more efficient as compared to selection sort.The time complexity of

Quick sort is O(nlogn) while for selection is O(n^2).

**Sorted Quick-sort Vs Randomized Quick-sort**

Randomized quick-sort is more efficient as compared to sorted quick-sort it is because

based on the analysis,randomized quick sort can functions for array size which up to

300000 and etc while for sorted quick sort it can functions for array size which up to

around 40000.Aside from that, sorted quick sort has the worst case running time(O(n^2))

while randomized quick sort has both average and best case running time(O(nlogn)).

**Sorted quick-sort with last element as pivot Vs Random quick-sort with last element as pivot**

Apparently, random quick sort with last element as pivot is more efficient than sorted

Quick-sort with last element as pivot.

**Sorted quick-sort with random pivot Vs Random quick-sort random pivot**

Apparently, random quick sort random pivot is more efficient than sorted

Quick-sort with random pivot.