Report

30132195
MD Shaherier
CPSC 319
TA: XI Wang

Complex analysis

```
// Selection sort in Java

import java.util.Arrays;

class SelectionSort {
  void selectionSort(int array[]) {
    int size = array.length; = n (number of operations)

    for (int step = 0; step < size - 1; step++) { -> n-1 ops.
      int min_idx = step; -> 2 ops } (constant time (k))

      for (int i = step + 1; i < size; i++) {  -> n-1 ops

        // To sort in descending order, change > to < in this line.
        // Select the minimum element in each loop.
        if (array[i] < array[min_idx]) {
          min_idx = i;
        }
      }

      // put min at the correct position
      int temp = array[step];
      array[step] = array[min_idx];
      array[min_idx] = temp;
    }
}
```

Rule
-> ignore constants
-> take the highest power

So $T(n) = (n-1) \times k_1 \times (n-1) \times k_2 + k_3$

-> $(n-1)^2 [k_1 \times k_2] + k_3$

-> $n^2 - 2n + 1 + [k_1 \times k_2] + k_3$ (ignore 2, ignore 1)

-> n^2 so $O(n^2)$

constant time k_2

constant time k_3

Time complexity

Best $O(n^2)$

Worst $O(n^2)$

Average $O(n^2)$

K mean constant operation so constant time.

```
// Insertion sort in Java

import java.util.Arrays;

class InsertionSort {

  void insertionSort(int array[]) {
    int size = array.length;

    for (int step = 1; step < size; step++) {
      int key = array[step];
      int j = step - 1;

      // Compare key with each element on the left of it until an element
smaller than
      // it is found.
      // For descending order, change key<array[j] to key>array[j].
      while (j >= 0 && key < array[j]) {
        array[j + 1] = array[j];
        --j;
      }

      // Place key at after the element just smaller than it.
      array[j + 1] = key;
    }
  }

}
```

n = 3 (5) 1

Step: 1, 2

j = 0, 1

n (number of operation)

(n-1) ops

k1 ops

for ascending this is not here so it is skipped.

for descending it n steps so it n(n-1)/2

for average it n-2 steps so it (n-1)(n-2)/4

k2 ops

k3 ops.

Summary.

best case = $(n-1) \times k_1 + (n-1)k_3 = O(n)$

Average case = $\frac{(n-1) \times k_1 \times (n-2) \times k_2}{4} + (n-1) \times k_3 = O(n^2)$

Worst case = $(n-1)k_1 + \frac{n(n-1) \times k_2}{2} + n-1 \times k_3 = O(n^2)$

```
// Merge sort in Java

class MergeSort {

  // Merge two subarrays L and M into arr
  void merge(int arr[], int p, int q, int r) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[] = new int[n1];
    int M[] = new int[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];

    // Maintain current index of sub-arrays and main array
    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++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
    }

    while (j < n2) {
      arr[k] = M[j];
      j++;
      k++;
    }
  }

  // Divide the array into two subarrays, sort them and merge them
  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) / 2;

      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);

      // Merge the sorted subarrays
      merge(arr, l, m, r);
    }
  }

  // Print the array
  static void printArray(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n; ++i)
      System.out.print(arr[i] + " ");
    System.out.println();
  }
}
```

Here ascending, descending, random doesn't matter.

merge $k_1 + k_2 + n_1 k_3 + n_2 * k_4 + k_5 +$

$\{(n_1+n_2)k_6$ or $(n_1+n_2)k_7$ or $(n_1+n_2)k_8\} +$

$n_1(k_9) + n_2(k_{10})$ -> means ignored

so overall =

$n_1 + n_2 + (n_1+n_2) + n_1 + n_2 = 3n_1 + 3n_2$

$3(n_1+n_2) = O(n)$

So the overall merge sort takes

$k_1 + n\log(n) = n\log(n)$

= number of oper per level * no of levels

$n \times \log(n)$

merge x0 levels

```
// Quick sort in Java

import java.util.Arrays;

class Quicksort {

  // method to find the partition position
  static int partition(int array[], int low, int high) {

    // choose the rightmost element as pivot
    int pivot = array[high];

    // pointer for greater element
    int i = (low - 1);

    // traverse through all elements
    // compare each element with pivot
    for (int j = low; j < high; j++) {
      if (array[j] <= pivot) {

        // if element smaller than pivot is found
        // swap it with the greatr element pointed by i
        i++;

        // swapping element at i with element at j
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
      }

    }

    // swap the pivot element with the greater element specified by i
    int temp = array[i + 1];
    array[i + 1] = array[high];
    array[high] = temp;

    // return the position from where partition is done
    return (i + 1);
  }

  static void quickSort(int array[], int low, int high) {
    if (low < high) {

      // find pivot element such that
      // elements smaller than pivot are on the left
      // elements greater than pivot are on the right
      int pi = partition(array, low, high);

      // recursive call on the left of pivot
      quickSort(array, low, pi - 1);   [n]

      // recursive call on the right of pivot   [n] * log(n)
      quickSort(array, pi + 1, high);
    }                                   best and average
  }
}
```

Worst case

-> If pi is the smallest or largest element

-> so the first quick sort call n times

-> the second quick sort is called n times thus $O(n^2)$ Worst = n^2

Best and worst case

-> If pi is not smallest/largest or it the median

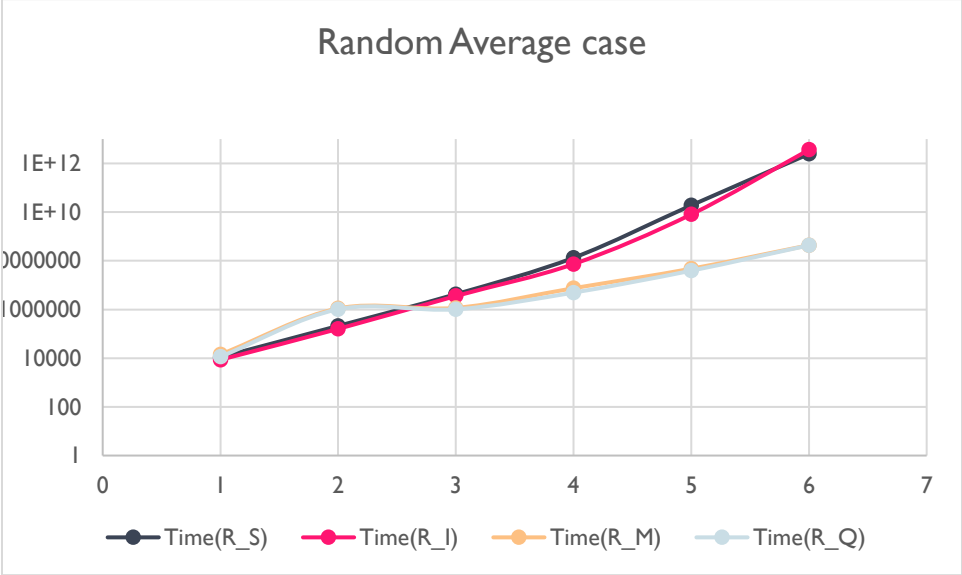
-> first quick sort call n times -> n best/average = $n \log(n)$

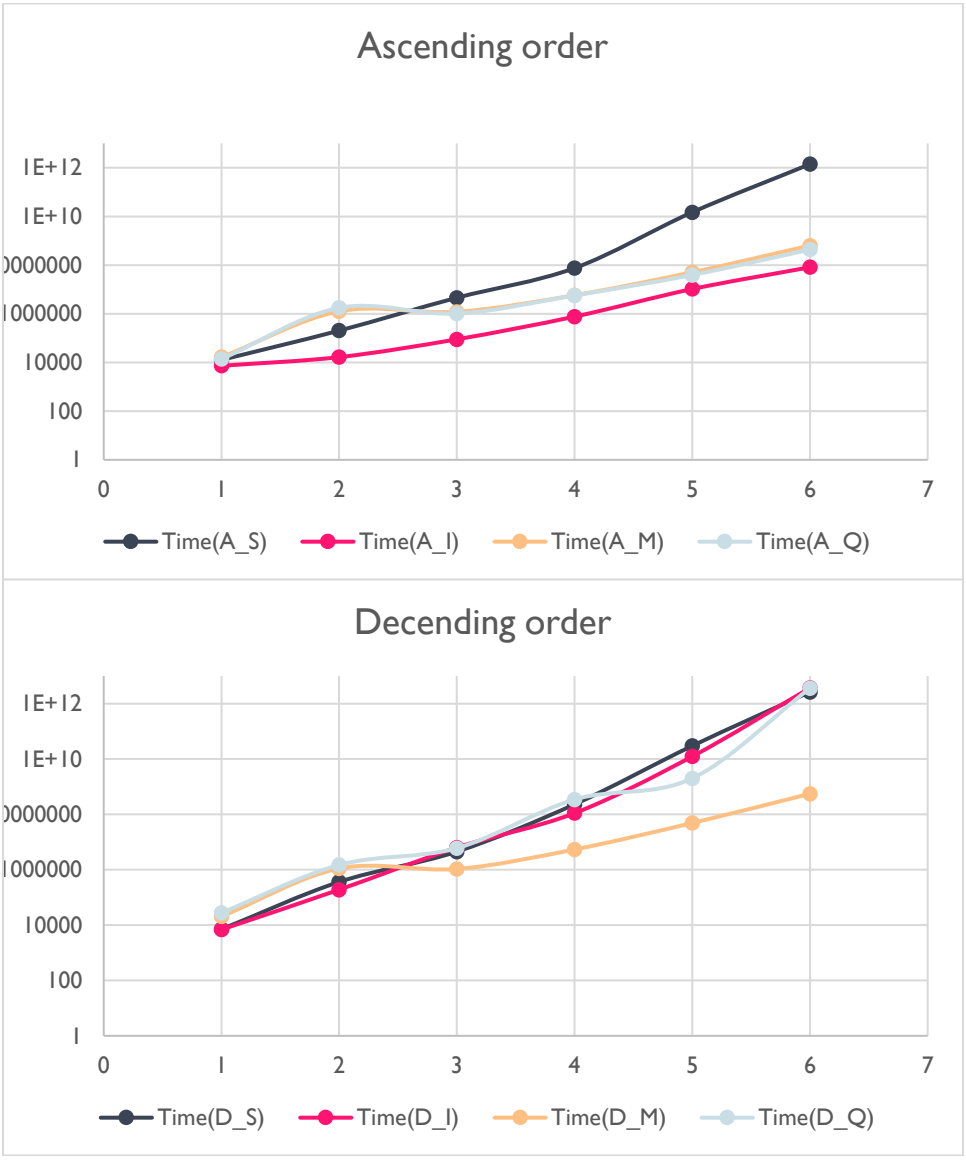
-> the second quick sort is called $n/_{2^{nd}}$ -> log(n)

1. The experimental method used.

- Java
- import java.util.*; (println , Random- nextInt(), nanoTime( )
- import java.util.Arrays; (used sort)
- import java.util.Collections; (reverseOrder())
- import java.io.*;

1. The data collected (use tables and graphs to help illustrate this).





S = selection
I = insert
M = Merge
Q = quick

Interpretation:

Time Complexities
Quick sort:
Worst Case Complexity [Big-O]: O(n2)
It occurs when the pivot element picked is either the greatest or the smallest element.

This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.
However, the quicksort algorithm has better performance for scattered pivots.

Best Case Complexity [Big-omega]: O(n*log n)
It occurs when the pivot element is always the middle element or near to the middle element.

Average Case Complexity [Big-theta]: O(n*log n)
It occurs when the above conditions do not occur.

-This algorithm is efficient for solving small size arrays but not large size arrays.

Time Complexities:
Select sort :(Data sort type doesn't matter)

Worst Case Complexity: O(n2)
If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

Best Case Complexity: O(n2)
It occurs when the array is already sorted

Average Case Complexity: O(n2)
It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

This algorithm is useful for sorting arrays with a small number of elements but does not preform as well with large arrays
Time Complexity:

Insert Sort:
Worst Case Complexity: O(n2)
Suppose an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

Each element must be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = n^2.

Best Case Complexity: O(n)
When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.
Average Case Complexity: O(n2)
It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Time Complexity
Merge sort : The data sort type doesn't matter
Best Case Complexity: O(n*log n)

Worst Case Complexity: O(n*log n)

Average Case Complexity: O(n*log n)
This algorithm is very useful in sorting arrays of large as well as small size.

Conclusion

Advantage and disadvantage of merge sort

Adv:
It is quicker for larger lists because unlike insertion and bubble sort it doesnt go through the whole list seveal times.
Dis:
Uses more memory space to store the sub elements of the initial split list.

Advantage and disadvantage of selection sort

Adv:
Doesn't depend data sort type
Dis:
Appropriate only for small N since N2 grows rapidly

Advantage and disadvantage of insert sort

Adv:
Best sorted data as it was a linear complexity of space and time .
Dis:
Not best for random sort data.

Advantage and disadvantage of Quicksort

Adv: Sorting n objects takes only n (log n) time.
Dis:
It is a recursive process.
In the worst-case scenario, it takes quadratic (i.e., n2) time.

Finally, the best sorting algorithms depend on the case of data that developer deal with and the limitation like memory or time or complexity or all.

The end.