

**WIA2005 Algorithm Design And Analysis**  
Faculty of Computer Science and Information Technology  
University of Malaya

**REPORT 1**

(Lab 1 – Lab 6)

**Prepared by**

|  |  |
| --- | --- |
| Name | Matrix No. |
| MUHAMMAD AMIRUL BIN HARUN | WES150016 |
| MUHAMMAD FARIS BIN JAAFAR | WEM150013 |
| MUHAMMAD AZRI BIN ZAINUDDIN | WER150018 |
| IRSYAD BIN SHAMSUDIN | WES150009 |

Lecturer’s Name

Dr. Asmiza binti Abdul Sani

Contents

[Lab 1 3](#_Toc478892728)

[Part A 3](#_Toc478892729)

[Question 1 3](#_Toc478892730)

[Question 2 3](#_Toc478892731)

[Question 3 3](#_Toc478892732)

[Question 4 3](#_Toc478892733)

[Question 5 3](#_Toc478892734)

[Question 6 3](#_Toc478892735)

[Question 7 3](#_Toc478892736)

[Part B 4](#_Toc478892737)

[Question 1 4](#_Toc478892738)

[Question 2 4](#_Toc478892739)

# Lab 1

## Part A

### Question 1

### Question 2

### Question 3

### Question 4

### Question 5

### Question 6

### Question 7

## Part B

### Question 1

### Question 2

# Lab 2

## Part 1- Implementation

### Question 1

Bubble sort is a popular sorting algorithm, but inefficient. It works by swapping adjacent elements that are out of order.

The running time for bubble sort are:

* Best-case running time: Ω(n)
* Average-case running time: Θ(n^2)
* Worst-case running time: O(n^2)

**Pseudocode**

BubbleSort(A)

for i = 1 to A.length-1

for j = A.length downto i+1

if A[j] < A[j-1] //comparing between two adjacent elements

swap position A[j] with A[j-1]

### Question 2

Counting sort elements based on numeric keys between a specific range. No comparison is done using sorting. Usually used as subroutine in other sorting algorithms.

In other words, it counts the number of occurrence in a value (e.g. if ‘2’ appears three times in an array). Then doing some arithmetic to calculate the position of each object in the output sequence.

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **Count** | 0 | 2 | 2 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

2) Modify the count array such that each element at each index stores the sum of previous counts.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **Count** | 0 | 2 | 2 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| **Sum** | 0 | 2 | 4 | 4 | 5 | 6 | 6 | 7 | 7 | 7 |

The modified count array indicates the position of each object in the output sequence.

3) Output each object from the input sequence followed by decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index.

The running time for bubble sort are:

* Best-case running time: Ω(n+k)
* Average-case running time: Θ(n+k)
* Worst-case running time: O(n+k)

**Pseudocode**

Counting-Sort(A, B, k)

Let C[0 … l] be a new array

For i = 0 to k

C[i] = 0

For j = 1 to A.length

C[A[j]] = C[A[j]] + 1

For I = 1 to k

C[i] = C[i] + C[i-1]

For j = A.length downto 1

B[C[A[j]]] = A[j]

C[A[j]] = C[A[j]] - 1

### Question 3

Bucket sort is a sorting algorithm in which, buckets are created, and elements are put into the buckets. Then sort the algorithm inside each bucket using some sorting algorithm (e.g. Insertion sort).

The elements are taken out and joined to get the sorted result.

**Pseudocode**

Bucket-Sort(A)

Let B[0 … n-1] be a new array

n. = A.length

for i = 0 to n-1

make B[i] an empty list

for i = 1 to n

insert A[i] into list B[n\*A[i]]

for i = 0 to n-1

sort list B[i] with insertion sort

concatenate the list B[0], B[1], …, B[n-1] together in order

### Question 4

Shell sort is basically a variation of the insertion sort. It works by comparing the elements between gap (h, usually starts from h=n/2), and reducing the h-value dividing by 2 (h/2) until 1.

**Pseudocode**

Shell-Sort(A, n)

For gap = n/2 down to 0, gap/=2

For i = gap until n

For j=i until gap

A[j] = A[j-gap]

A[j] = temp

## Part 2 - Discussion

### Advantage and Disadvantage of Algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| **Sorting  Algorithm** | **Time Complexity** | | |
| **Best-Case** | **Average-Case** | **Worst-Case** |
| Bubble Sort | Ω(n) | Θ(n^2) | O(n^2) |
| Counting Sort | Ω(n + k) | Θ(n + k) | O(n + k) |
| Bucket Sort | Ω(n + k) | Θ(n + k) | O(n^2) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) |

* Green = Excellent
* Yellow = Fair
* Brown = Bad
* Purple = Horrible

In worst-case, counting sort is the best sorting method, while other sorting algorithms perform horribly.

In average-case, both counting sort and bucket sort works excellently, while the other two perform horribly.

In best-case, both counting sort and bucket sort works excellently, followed by bubble sort and shell sort.

Overall, counting sort has the best performance in all cases, followed by bucket sort, bubble sort and shell sort.

**Bubble-Sort**

* Best-Case: Ω (n) - This is the case of the already-sorted or almost sorted sequence
* Average: Θ (n^2)
* Worst-Case: O(n^2) - At maximum, there will be n passes through the data, and each

**Counting Sort**

* All cases: O(n+k) - Where n is the number of elements in input array and k is the range of input.

Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.

**Bucket Sort**

* Best-Case and Average-Case: Ω (n + k) - When the data being sorted can be distributed between the buckets perfectly or when buckets are as small as possible.
* Worst-Case: O(n^2) - When all elements are allocated to the same bucket. Since individual buckets are sorted using another algorithm, if only a single bucket needs to be sorted, bucket sort will take on the complexity of the inner sorting algorithm.

**Shell Sort**

* Best-Case: Ω(n log(n)) - Depends on the increment sequence
* Average-Case & Worst-Case: Θ(n(log(n))^2 - Depends on the increment sequence

# Lab 3

## Part 1

### Binary Search

### Powering a number

Powering a number (e.g. 2 to the power of 3) involves into breaking it into smaller parts.  
  
 If an, where n is the subset of N,

* If n is even, an = an/2 \* an/2
* If n is odd, an = a(n-1)/2 \* a(n-1)/2 \* a

In all cases, the time complexity is Θ (log n).

### Fibonacci numbers

### Matrix multiplication

### Merge sort

### Quick sort

Quicksort is similar to insertion sort, as it sorts “in place”.

It involves three steps:

1. **Divide**

Partition the array into two subarrays around a pivot (x) such that elements in lower subarray ≤ x ≤ elements in upper subarray.

1. **Conquer**

Recursively sort the two subarrays

1. **Combine**

Trivial

* Best-case: Ω (n log n) – when the array is partitioned evenly
* Worst-case: O (n2) – when
  + the input is already sorted or reverse sorted
  + partitioning occurs around min or max element
  + one side of partition always has no elements

**Pseudocode**

Quicksort(A, p, r)

If p < r

Then q <- partition (A, p, r)

Quicksort (A, p, q-1)

Quicksort (A, q+1, r)

## Part 2

### Finding the Matric Number

# Lab 4

## Part 1 - Implementation

### Question 1

### Question 2

### Question 3

# Lab 5

## Part A

Randomized Selection is a mixed of randomization and divide-and-conquer method. It is modelled after quicksort, in which the input array is partitioned recursively. However, unlike quicksort, only side of the partition is recursively processed.

It is used to find the ith smallest element (e.g. i=5, then 5th smallest element in an array). In other words, ith order statistic of a statistical sample is equal to its ith-smallest value.

There are two cases for this method; lucky and unlucky. Assuming that the elements are distinct:

* Lucky case: T(n) = T (9/10) + Θ (n)

= Θ (n)

* Unlucky case: T(n) = T (n-1) + Θ (n)

= Θ (n2)

**Pseudo-Code**

Randomized-Select(A, p, r, i)

If p == r

Return A[p]

q = Randomized-Partition (A, p, r)

k = q -p + 1

return A[q]

else if i < k

return Randomized-Select (A, p, q-1, i)

else

return Randomized-Select (A, q+1, r, i-k)

### Question 1

In this question, pseudo-code of Randomized Selection is given as a method. There are four parameters required, an array of integers (A), leftmost value (p), rightmost value (r) and order statistic of i (i). There are three steps:

1. **Divide**

If p is equal to r, then return the value at p position. If not, pick a pivot (x) from the sequence, and use it to the partition the sequence so that it becomes:

A[1], . . . , A[k − 1], A[k] = x, A[k + 1], . . . , A[n]

for some k. All elements on the left of A[k] are smaller than x and all elements on the right are larger than x.

1. **Conquer**

Then, we compare i with k. There are three cases:

* If i is equal to k, return x, because x is the kth smallest.
* Else if I is less than k, make a recursive call to find the ith smallest in the left-hand side partition (A[1], . . ., A[k − 1])
* Else, if i is more than k, make a recursive call to find the ith smallest in the right-hand side partition (A[k+1], . . ., A[n])

1. **Combine**

### Question 2

There are nine wells located at different location, marked with an x-axis and y-axis coordinate (x, y). To get the minimized total length of the spurs from all nine wells connected to a main pipeline, a median of x-axis and y-axis need to be calculated, and in linear time.

Apply randomized selection method to find the median. Considering that there are nine wells (n = 9), and it is an odd number, a median can be found by median = n/2.

Hence, the median order statistics from x-axis and y-axis coordinates can be calculated.

Another method is:

* (x1 + x2 + x3 + … + xn)/n, xi is coordinate of x-axis
* (y1 + y2 + y3 + … + yn)/n, yi is coordinate of y-axis

Considering that the main pipeline is running from east to west, we can ignore the x-axis coordinate, as the total length of spurs depends on the y-axis coordinates from all wells to the main pipeline at y-axis coordinate.

# Lab 6

## Part A

### Question 1

### Question 2

### Question 3