**Table of Contents**

1. Merge Sort……………………………………………………………………………………..

* 1. Problem Description and Explanation………………………………………………...
  2. Tabulation, Output and Analysis of array sizes, actual cost, expected cost…………..

1.2.1 Input arrays, sorted array, and analysis……………………………………..

1.2.2 With more values……………………………………………………………

1.2.3 Output and Graphical analysis of algorithm……….....................................

1.3 Source Program……………………………………………………………………….

2. Heap Sort……………………………………………………………………………………..

2.1 Problem Description and Explanation……………….................................................

2.2 Tabulation, Output and Analysis of array sizes, actual cost, expected cost………….

2.2.1 Input arrays, sorted array, and analysis……………………………………..

2.2.2 With more values……………………………………………………………

2.2.3 Output and Graphical analysis of algorithm……….....................................

2.3 Source Program……………………………………………………………………….

3. Heap sort for Priority Queue Problem………………………………………………………..

3.1 Problem Description and Explanation………………………....................................

3.2 Tabulation, Output and Analysis of array sizes, actual cost, expected cost…………

3.2.1 Input arrays, sorted array, and analysis…………………………………….

3.2.2 With more values……………………………………………………………

3.2.3 Output and Graphical analysis of algorithm……….....................................

3.3 Source Program………………………………………………………………………

4. Quick sort And Graphical Comparisons of Merge, Heap and Quick Sorts…………………..

4.1 Problem Description and Explanation………………………....................................

4.2 Tabulation, Output and Analysis of array sizes, actual cost, expected cost…………

4.2.1 Input arrays, sorted array, and analysis…………………………………….

4.2.2 With more values……………………………………………………………

4.2.3 Output and Graphical analysis of algorithm……….....................................

4.3 Source Program……………………………………………………………………….

5. Maximum Sub-Array problem……………….………………………………………………..

5.1 Problem Description and Explanation……………………………………..................

5.2 Input arrays, sorted array, and analysis……………………………………………….

5.2.1 With more values……………………………………………………………

5.3 Source Program……………………………………………………………………….

6. Hashing and Hash Tables…………………….………………………………………………..

6.1 Problem Description and Explanation……………………………………..................

6.2 Input arrays, sorted array, and analysis……………………………………………….

6.2.1 With more values……………………………………………………………

6.3 Source Program……………………………………………………………………….

7. Matrices Chain Multiplication Problem….….………………………………………………..

7.1 Problem Description and Explanation……………………………………..................

7.2 Print out m [i, j] and s [i,j] tabulations with same matrices as in textbook…………..

7.3 Print out m [i, j] and s [i,j] tabulations with matrices generated by myself….……..

7.4 Source Program……………………………………………………………………….

8. Huffman Codes Problem………………….….………………………………………………..

8.1 Variable length encoding, Input Texts for compression and decompression...............

8.2 Fixed length encoding…………………………………………………… …………..

8.3 Comparisons…………………………………………………………………….…….

8.4 Source Program……………………………………………………………………….

**Chapter1 – Merge Sort**

***The merge sort algorithm*** follows the divide-and-conquer approach: they

break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem. The divide-and-conquer paradigm for merge sort involves three steps at each level of the recursion.

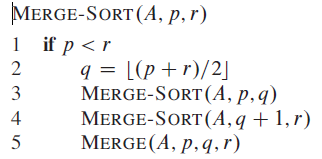
**Divide:** Divide the n-element sequence to be sorted into two subsequences of n=2

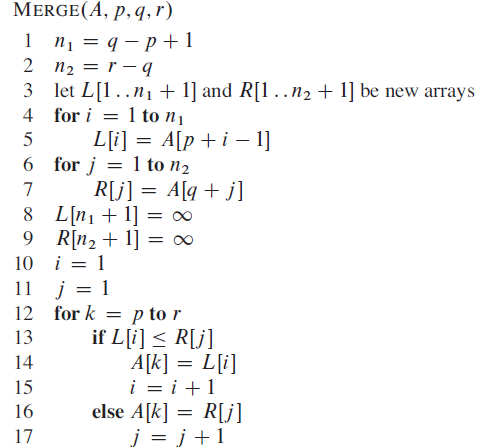
elements each.

**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

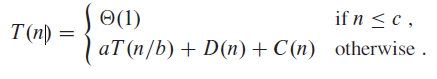
The algorithm for merge sort is:





The MERGE-SORT routine divides the array into two arrays recursively. The purpose of MERGE-SORT is only dividing the array. The MERGE routine conquers and combines the arrays. It performs sorting and combining the sorted arrays together. The heart of merge sort algorithm lies in MERGE routine.

The MERGE-SORT routine runs recursively. The actual question is how many times the recursion occurs? It occurs Θ (lgn) times, which is actual computing timing of MERGE-SORT routine. On the other hand the MERGE routine takes total computation time of Θ (n) .Therefore the total computation time for merge sort algorithm is Θ (nlgn).The total computation equation is:



**Divide:** The divide step just computes the middle of the subarray, which takes

constant time. Thus, D (n) = **Θ (1).**

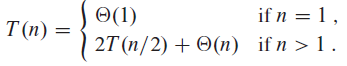
**Conquer:** Recursively solve two sub problems, each of size n=2, which contribute

2T (n/2) to the running time.

**Combine:** We have already noted that the MERGE procedure on an n-element

subarray takes time Θ **(n)**, and so C(n)= **Θ(n)**.

Worst-case running time T (n) of merge sort:



* 1. **Problem Description and Explanation**
* The problem is to implement merge sort algorithm using any programming language and also to calculate the total running time of the merge sort program i.e., actual counting, which calculates number of times each statement executes. The actual counting is compared with NlgN value for different array sizes (N).
* Tabular representation of the comparison between NlgN and actual counting for the given N is asked to be drawn. Five different N values are asked to be used.
* The Graphical representation of actual counting and NlgN is also asked to be drawn. The path followed by each one should be almost parallel. The actual counting and NlgN values will not be same value. They differ by some constant value.
* I implemented the program in java programming language using notepad++ editor and jdk7 java software.
* Almost closely followed the algorithm described in the class.
* Used random number generator for input numbers.
* Compared the values for five different array sizes.
* Drawn the tabular form and graphical form of the representation.
* In merge function where it uses two arrays to compare and combine into one final sorted array. In my program I used one array temp for comparison, which compares left part of the array with right part.
* Used counter variable for actual counting to count number of execution it goes through for each given array size.
  1. **Tabulation, Output and Analysis of array sizes, actual cost, expected cost of generated data**

|  |  |  |
| --- | --- | --- |
| **N** | **Actual Counting** | **NlogN** |
| 10 | 43 | 33.2 |
| 20 | 107 | 86.4 |
| 30 | 177 | 147.2 |
| 40 | 255 | 212.8 |
| 50 | 335 | 282.2 |

**1.2.1Input arrays, sorted array, and analysis**

Enter the array size**: 10**

Unsorted array**: 6 0 5 8 4 1 9 5 6 4**

Sorted array**: 0 1 4 4 5 5 6 6 8 9**

Counting number of steps**: 43**

Enter the array size**: 20**

Unsorted array**: 3 5 1 2 1 1 0 1 8 3 3 0 8 2 4 4 1 8 1 9**

Sorted array**: 0 0 1 1 1 1 1 1 2 2 3 3 3 4 4 5 8 8 8 9**

Counting number of steps**: 107**

Enter the array size**: 30**

Unsorted array**: 0 3 4 9 6 7 5 3 3 5 5 2 9 9 8 0 5 7 0 1 1 8 6 5 3 3 7 0 8 4**

Sorted array**: 0 0 0 0 1 1 2 3 3 3 3 3 4 4 5 5 5 5 5 6 6 7 7 7 8 8 8 9 9 9**

Counting number of steps**: 177**

Enter the array size**: 40**

Unsorted array**: 5 9 4 3 9 1 4 3 8 8 4 5 6 5 4 3 1 4 4 4 4 8 3 3 7 9 9 0 8 9 1 4 4**

**0 2 6 7 6 3 3**

Sorted array**: 0 0 1 1 1 2 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 6 6 6 7 7 8 8 8**

**8 9 9 9 9 9**

Counting number of steps**: 255**

Enter the array size**: 50**

Unsorted array**: 1 9 9 8 8 9 4 5 7 4 7 8 9 9 1 5 6 5 5 2 3 2 4 8 3 0 4 9 4 9 2 4 0**

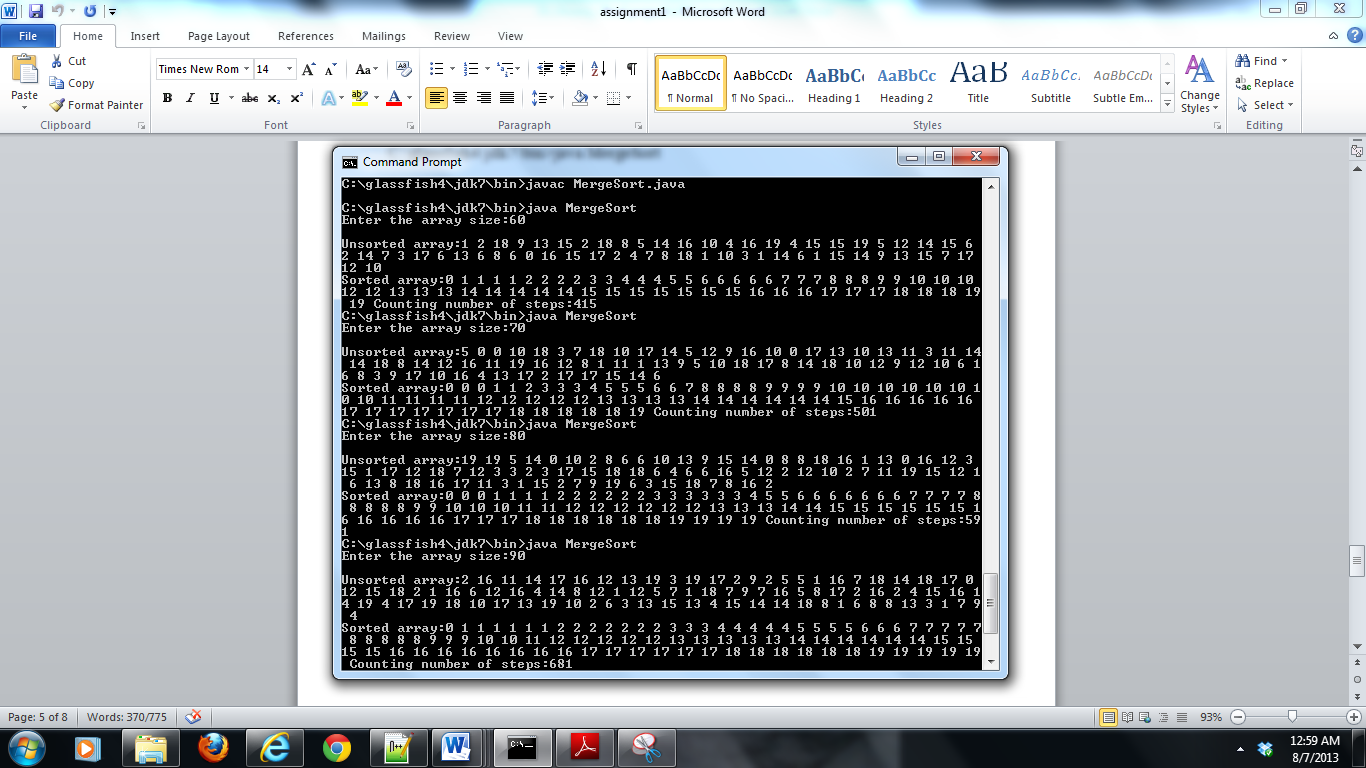
**0 0 9 8 2 9 9 0 7 7 5 6 6 9 1 2 7**

Sorted array**: 0 0 0 0 0 1 1 1 2 2 2 2 2 3 3 4 4 4 4 4 4 5 5 5 5 5 6 6 6 7 7 7 7 7**

**8 8 8 8 8 9 9 9 9 9 9 9 9 9 9 9**

Counting number of steps**: 335**

**1.2.2 with more values**



**1.2.3 Output and Graphical analysis of Algorithm**

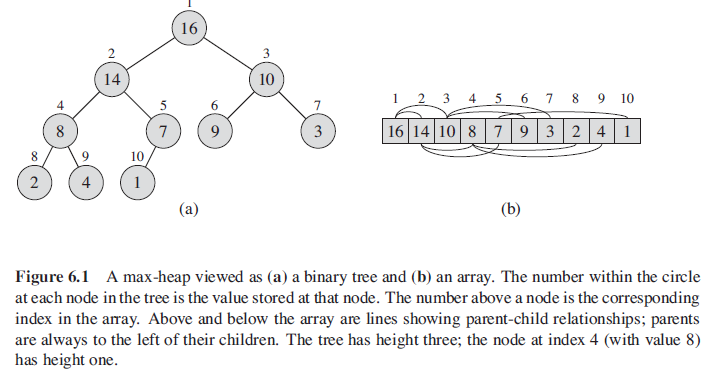
**Chapter 2 – Heap sort**

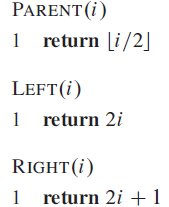
***Heap*** is a binary tree using a data structure HEAP to manage information during execution of algorithm. Theheapdata structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes:

A: *length*, gives the number of elements in the array

A: *heap*-*size*, which represents how many elements in the heap are stored within

array A.





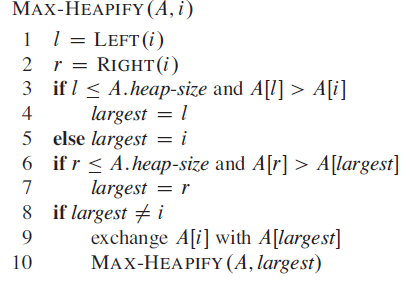
There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds,

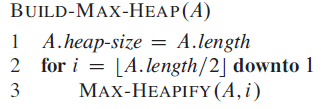
the values in the nodes satisfy a ***heap property***, the specifics of which depend on

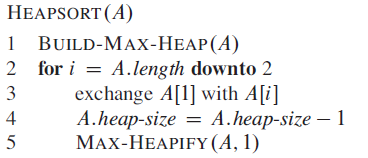
the kind of heap. In a ***max-heap***, the max-heap propertyis that for every node i

other than the root, **A [PARENT (i)]>=A[i].**That is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way; the **min-heap property**is that for every node i other than the root, **A [PARENT (i)] <=A[i].**The smallest element in a min-heap is at the root.

**Algorithms for Heapsort:**







The heart of heap sort is the MAX-HEAPIFY routine. The largest node is at root node. This routine is called recursively, so the height of the tree is Θ (lgn).Therefore total computation tine for MAX-HEAPIFY is **Θ (lgn).**

The BUILD-MAX-HEAP method takes **O (n)** times. The HEAPSORT procedure takes time **O(nlgn),** since the call to BUILD-MAXHEAP takes time **O(n)** and each of the n -1 calls to MAX-HEAPIFY takes time O(lgn).**Therefore the total computation time for heap sort is O(nlgn).**

Heapify is used by operating system. When many jobs are needed to be executed, the maximum (highest) priority job is executed. It is also called priority queue.

**2.1 Problem Description and Explanation**

* The problem is to implement heap sort algorithm using any programming language and also to calculate the total running time of the heap sort program i.e., actual counting, which calculates number of times each statement executes. The actual counting is compared with NlgN value for different array sizes (N).
* Tabular representation of the comparison between NlgN and actual counting for the given N is asked to be drawn. Five different N values are asked to be used.
* The Graphical representation of actual counting and NlgN is also asked to be drawn. The path followed by each one should be almost parallel. The actual counting and NlgN values will not be same value. They differ by some constant value.
* I implemented the program in java programming language using notepad++ editor and jdk7 java software.
* Almost closely followed the algorithm described in the class.
* Used random number generator for input numbers.
* Compared the values for five different array sizes.
* Drawn the tabular form and graphical form of the representation.
* With my program the user enters array size and using random generator the array is filled with numbers which are unsorted. Using heap sort all the elements are sorted and displayed. Actual counting steps are also displayed.
* Used counter variable for actual counting to count number of execution it goes through for each given array size.

**2.2 Tablulation,Output and Analysis of array sizes,actual cost,expected cost of generated data**

|  |  |  |
| --- | --- | --- |
| **N** | **Actual Counting** | **NlogN** |
| 10 | 57 | 33.2 |
| 20 | 133 | 86.4 |
| 30 | 232 | 147.2 |
| 40 | 316 | 212.8 |
| 50 | 421 | 282.2 |

**2.2.1 Input arrays,sorted array, and analysis**

Enter the array size: **10**

Unsorted array:

**2 5 7 17 19 1 15 9 0 15**

Sorted Array using Heapsort:

**0 1 2 5 7 9 15 15 17 19**

TOTAL STEPS: **57**

Enter the array size: **20**

Unsorted array:

**2 10 0 0 9 6 2 7 16 2 17 8 15 14 9 4 14 6 15 3**

Sorted Array using Heapsort:

**0 0 2 2 2 3 4 6 6 7 8 9 9 10 14 14 15 15 16 17**

TOTAL STEPS: **133**

Enter the array size: **30**

Unsorted array:

**11 11 12 14 2 11 14 12 12 10 4 16 14 7 16 16 12 12 13 17 17 8 3 12 12 3 2 12 12 16**

Sorted Array using Heapsort:

**2 2 3 3 4 7 8 10 11 11 11 12 12 12 12 12 12 12 12 12 13 14 14 14 16 16 16 16 17 17**

TOTAL STEPS**: 232**

Enter the array size:**40**

Unsorted array:

**16 3 1 15 9 4 10 18 13 18 14 1 12 19 17 15 10 17 5 8 11 16 1 8 6 0 12 16 14 11 6 4 6 15 6 15 8 5 5 15**

Sorted Array using Heapsort:

**0 1 1 1 3 4 4 5 5 5 6 6 6 6 8 8 8 9 10 10 11 11 12 12 13 14 14 15 15 15 15 15 16 16 16 17 17 18 18 19**

TOTAL STEPS: **316**

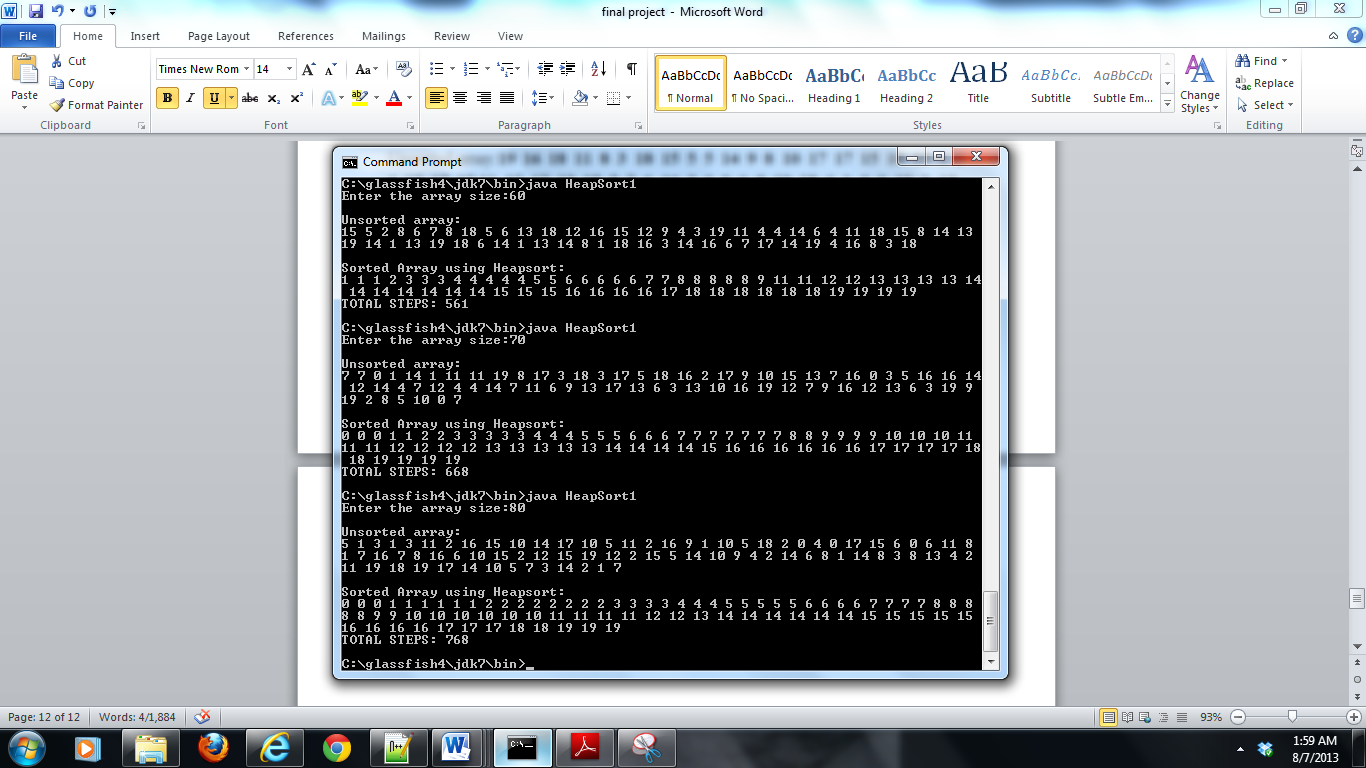
Enter the array size:**50**

Unsorted array:**19 16 18 11 8 3 18 15 5 5 14 9 8 10 17 17 15 10 13 1 5 6 17 17 17 11 17 17 13 15 5 7 6 16 2 6 0 6 9 10 18 6 1 0 0 19 2 13 9 18**

Sorted Array using Heapsort: **0 0 0 1 1 2 2 3 5 5 5 5 6 6 6 6 6 7 8 8 9 9 9 10 10 10 11 11 13 13 13 14 15 15 15 16 16 17 17 17 17 17 17 17 18 18 18 18 19 19**

TOTAL STEPS: **421**

**2.2.2 with more values**



**2.2.3 Output and Graphical analysis of algorithm**

**Chapter 3-Heap sort for priority Queue problem**

***Priority Queue*** is a popular application of heap. It is used by operating system. When many jobs are needed to be executed, the maximum (highest) priority job is executed.

A priority queueis a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A max-priority queuesupports the following operations:

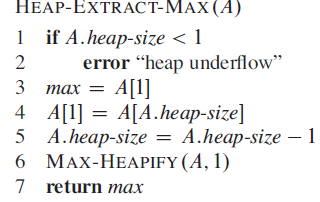
INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation S = S U {x}.

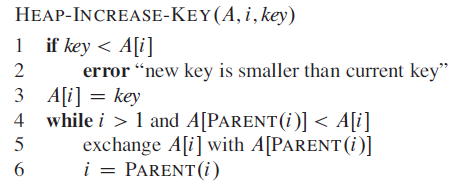
MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

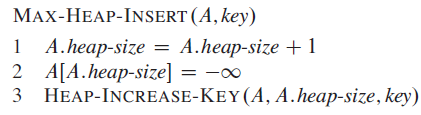
INCREASE-KEY(S, x, k) increases the value of element x’s key to the new value k,which is assumed to be at least as large as x’s current key value.

Algorithms for priority queue using heap:









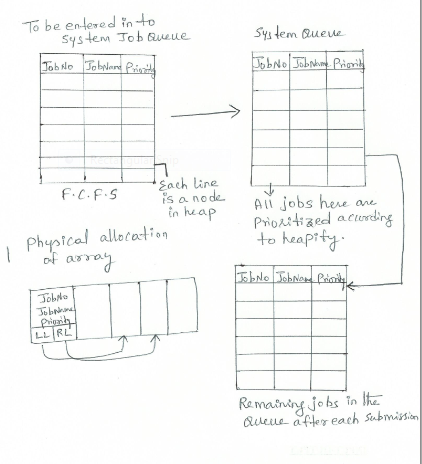
The procedure HEAP-MAXIMUM implements the MAXIMUM operation in **Θ (**1) time. The running time of HEAP-EXTRACT-MAX is O (lg n), since it performs only a constant amount of work on top of the O (lg n) time for MAX-HEAPIFY. The running time of HEAP-INCREASE-KEY on an n-element heap is O(lg n), since the path traced from the node updated in line 3 to the root has length O(lg n).The running time of MAX-HEAP-INSERT on an n-element heap is O(lg n).Therefore a heap can support any priority-queue operation on a set of size n

in O (lg n) time.

**3.1 Problem Description and Explanation**

* Program to implement priority queue using heap with nodes containing job name, jobno and priority.
* Perform Insertion of new node into the heap. Based on the priority of the new node it is adjusted into the heap according to MAX-HEAPIFY method.
* Perform display of all the nodes in the priority queue.
* Perform Extraction of the maximum node based on priority from the queue for execution. After extracting that node is removed from the priority queue.
* Once the extraction of maximum priority job is removed it needs to be sorted, so that highest priority job will be the root node..
* Used random generator for jobname,jobno and priority
* Based on the priority the program is manipulated.
* Implemented the program in c++ programming language.
* The insertion, extraction, displaying and sorting operation is performed in the program. The user is given the choice of selection on the type of action to be performed.
* If the user selects insertion, new nodes are inserted. If the user selects sort, nodes are sorted. If user selected extraction, the node with highest priority is extracted from the queue. If the user selected display, the remaining nodes in the priority queue are displayed.

**3.2 Tabulation, Output and Analysis of array sizes, actual cost, expected cost**



The above figure is the actual problem to be solved. The F.C.F.S table is the table with nodes containing jobNo, job Name, priority. These values need to be entered by the user. These jobs are sent to system queue; where the jobs are prioritized according to their priority using heapify method.

The highest priority job is removed from the queue for execution. The remaining job in the queue after each submission is present in the queue.

**3.2.1 Input arrays, sorted array and analysis**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION: **1**

ENTER NUMBER OF JOBS IN PRIORITY QUEUE: **5**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 4**

**DISPLAYING REST OF THE JOBS REMAINING IN THE QUEUE READY FOR EXECUTION**

**JOBNAME JOBNO PRIORITY**

**rAQBc 41 11478**

**sa1xV 29358 16827**

**vJcrg 9961 32391**

**wTiiz 14604 18716**

**trQF6 19718 186**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 2**

**Number of Nodes: 4**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 4**

**DISPLAYING REST OF THE JOBS REMAINING IN THE QUEUE READY FOR EXECUTION**

**JOBNAME JOBNO PRIORITY**

**trQF6 19718 1869**

**rAQBc 41 11478**

**sa1xV 29358 16827**

**wTiiz 14604 18716**

**vJcrg 9961 32391**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 3**

**HIGHEST PRIORITY JOB READY FOR EXECUTION IS:**

**vJcrg 9961 32391**

**3.2.2 with more values**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 4**

**DISPLAYING REST OF THE JOBS REMAINING IN THE QUEUE READY FOR EXECUTION**

**JOBNAME JOBNO PRIORITY**

**trQF6 19718 1869**

**rAQBc 41 11478**

**sa1xV 29358 16827**

**wTiiz 14604 18716**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 1**

ENTER NUMBER OF JOBS IN PRIORITY QUEUE**: 6**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 2**

**Number of Nodes: 9**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 4**

**DISPLAYING REST OF THE JOBS REMAINING IN THE QUEUE READY FOR EXECUTION**

**JOBNAME JOBNO PRIORITY**

**trQF6 19718 1869**

**exuie 778 9040**

**rAQBc 41 11478**

**sa1xV 29358 16827**

**wTiiz 14604 18716**

**2dREbb 15350 24084**

**iIgxIX 8942 24370**

**3EDNhm 30333 25547**

**oLBPh7 27644 27529**

**zBlax3 19912 31322**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 3**

**HIGHEST PRIORITY JOB READY FOR EXECUTION IS:**

**zBlax3 19912 31322**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 4**

**DISPLAYING REST OF THE JOBS REMAINING IN THE QUEUE READY FOR EXECUTION**

**JOBNAME JOBNO PRIORITY**

**trQF6 19718 1869**

**exuiea 778 9040**

**rAQBc 41 11478**

**sa1xV 29358 16827**

**wTiiz 14604 18716**

**2dREbb 15350 24084**

**iIgxIX 8942 24370**

**3EDNhm 30333 25547**

**oLBPh7 27644 27529**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 3**

**HIGHEST PRIORITY JOB READY FOR EXECUTION IS:**

**OLBPh7 27644 27529**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 4**

**DISPLAYING REST OF THE JOBS REMAINING IN THE QUEUE READY FOR EXECUTION**

**JOBNAME JOBNO PRIORITY**

**trQF6 19718 1869**

**exuiea 778 9040**

**rAQBc 41 11478**

**sa1xV 29358 16827**

**wTiiz 14604 18716**

**2dREbb 15350 24084**

**iIgxIX 8942 24370**

**3EDNhm 30333 25547**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 1**

ENTER NUMBER OF JOBS IN PRIORITY QUEUE**: 3**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 2**

**Number of Nodes: 10**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 4**

**DISPLAYING REST OF THE JOBS REMAINING IN THE QUEUE READY FOR EXECUTION**

**JOBNAME JOBNO PRIORITY**

**trQF6 19718 1869**

**JOyEbb 11323 2082**

**Wy6Ebb 19954 7376**

**exuiea 778 9040**

**rAQBcα 41 11478**

**sa1xVα 29358 16827**

**wTiizα 14604 18716**

**2dREbb 15350 24084**

**iIgxIX 8942 24370**

**KIDEbb 13931 24626**

**3EDNhm 30333 25547**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 3**

**HIGHEST PRIORITY JOB READY FOR EXECUTION IS:**

**3EDNhm 30333 25547**

OPTION1: INSERT NODE

OPTION2: BUILD MAX HEAP

OPTION3: EXTRACT MAXIMUM NODE

OPTION4: DISPLAY REMAINING NODES

OPTION5: EXIT

CHOOSE YOUR OPTION**: 5**

**Chapter 4- Quicksort And Graphical Comparisons of Merge,Heap and Quick Sorts**

Quicksort, like merge sort, applies the divide-and-conquer paradigm.

**Divide:** Partition (rearrange) the array A[p.. r] into two (possibly empty) subarrays

A[p.. q] and A[q+1.. r] such that each element of A[p.. q] is less than or equal to A[q+1..r], which is, in turn, less than or equal to each element

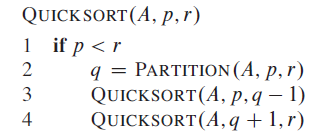
**Conquer:** Sort the two subarrays A[p.. q] and A[q+1.. r] by recursive calls

to quicksort.

**Combine:** Because the subarrays are already sorted, no work is needed to combine

them: the entire array A [p.. r] is now sorted.

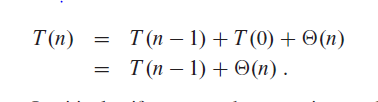
Algorithms for Quicksort:





The worst case total running time for quicksort is Θ(n2).The average running case total running time is Θ(nlgn).

Worst case partitioning is divided between one element in one zone and n-1 elements in another.



Best case partitioning is dividing the array into two equal parts.



Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still O(n lg n) but with a slightly larger constant hidden by the O-notation.

**4.1 Problem Description and Explanation**

* The problem is to implement quicksort algorithm using any programming language and also to calculate the total running time of the quicksort program i.e., actual counting, which calculates number of times each statement executes. The actual counting is compared with NlgN value for different array sizes (N).
* Tabular representation of the comparison between NlgN and actual counting for the given N is asked to be drawn. Five different N values are asked to be used.
* The Graphical representation of actual counting and NlgN is also asked to be drawn.Here other two sorting algorithms mergesort and heapsort is also drawn to compare the results.
* I implemented the program in java programming language using notepad++ editor and jdk7 java software.
* Almost closely followed the algorithm described in the class.
* Used random number generator for input numbers. Used same input values for all three sorting programs to compare actual counting.
* With my program the user enters array size and using random generator the array is filled with numbers which are unsorted. Using quicksort all the elements are sorted and displayed. Actual counting steps for mergesort, heapsort and quicksort are also displayed.
* Used counter variable for actual counting to count number of execution it goes through for each given array size.

**4.2 Tabulation, Output and Analysis of array sizes, actual cost, expected cost of generated data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NlogN** | **ARRAYSIZE N** | **QUICKSORT** | **MERGESORT** | **HEAPSORT** |
| **33.2** | **10** | **30** | **43** | **57** |
| **86.4** | **20** | **79** | **107** | **137** |
| **147.2** | **30** | **165** | **177** | **223** |
| **212.8** | **40** | **227** | **255** | **318** |
| **282.2** | **50** | **299** | **335** | **412** |

**4.2.1 Input arrays, sorted array, and analysis**

Enter the array size: **10**

Unsorted array: **196 100 115 157 194 0 82 199 46 58**

Sorted array: **0 46 58 82 100 115 157 194 196 199**

QUICKSORT ACTUAL COUNTING: **30**

MERGESORT ACTUAL COUNTING: **43**

HEAPSORT ACTUAL COUNTING: **57**

Enter the array size: **20**

Unsorted array: **43 25 120 134 10 100 49 90 185 122 76 189 20 93 66 107 5 54 52 17**

Sorted array: **5 10 20 25 43 49 52 54 66 76 90 93 100 107 120 122 134 172 185 189**

QUICKSORT ACTUAL COUNTING:**79**

MERGESORT ACTUAL COUNTING:**107**

HEAPSORT ACTUAL COUNTING:**137**

Enter the array size:30

Unsorted array**: 40 133 142 133 179 120 194 9 132 94 83 107 188 120 79 25 160 63 149 62 18 18 31 178 160 113 47 186 187 161**

Sorted array:**9 18 18 25 31 40 47 62 63 79 83 94 107 113 120 120 132 133 133 142 149 160 160 161 178 179 186 187 188 194**

QUICKSORT ACTUAL COUNTING:**165**

MERGESORT ACTUAL COUNTING**:177**

HEAPSORT ACTUAL COUNTING:**223**

Enter the array size**:40**

Unsorted array:**153 133 42 94 178 62 75 91 130 106 107 67 109 184 132 132 127 130 54 21 160 82 5 53 32 119 128 115 77 176 54 75 10 189 109 131 17 156 158 22**

Sorted array:**5 10 17 21 22 32 42 53 54 54 62 67 75 75 77 82 91 94 106 107 109 109 115 119 127 128 130 130 131 132 132 133 153 156 158 160 176 178 184 189**

QUICKSORT ACTUAL COUNTING:**227**

MERGESORT ACTUAL COUNTING**:255**

HEAPSORT ACTUAL COUNTING:**318**

Enter the array size**:50**

Unsorted array**:92 15 45 121 42 180 141 46 104 175 144 175 45 165 165 125 108 107 17 61 13 152 188 100 55 58 156 106 56 120 162 31 37 18 119 133 83 22 82 195 53 63 112 4 41 189 175 134 142 145**

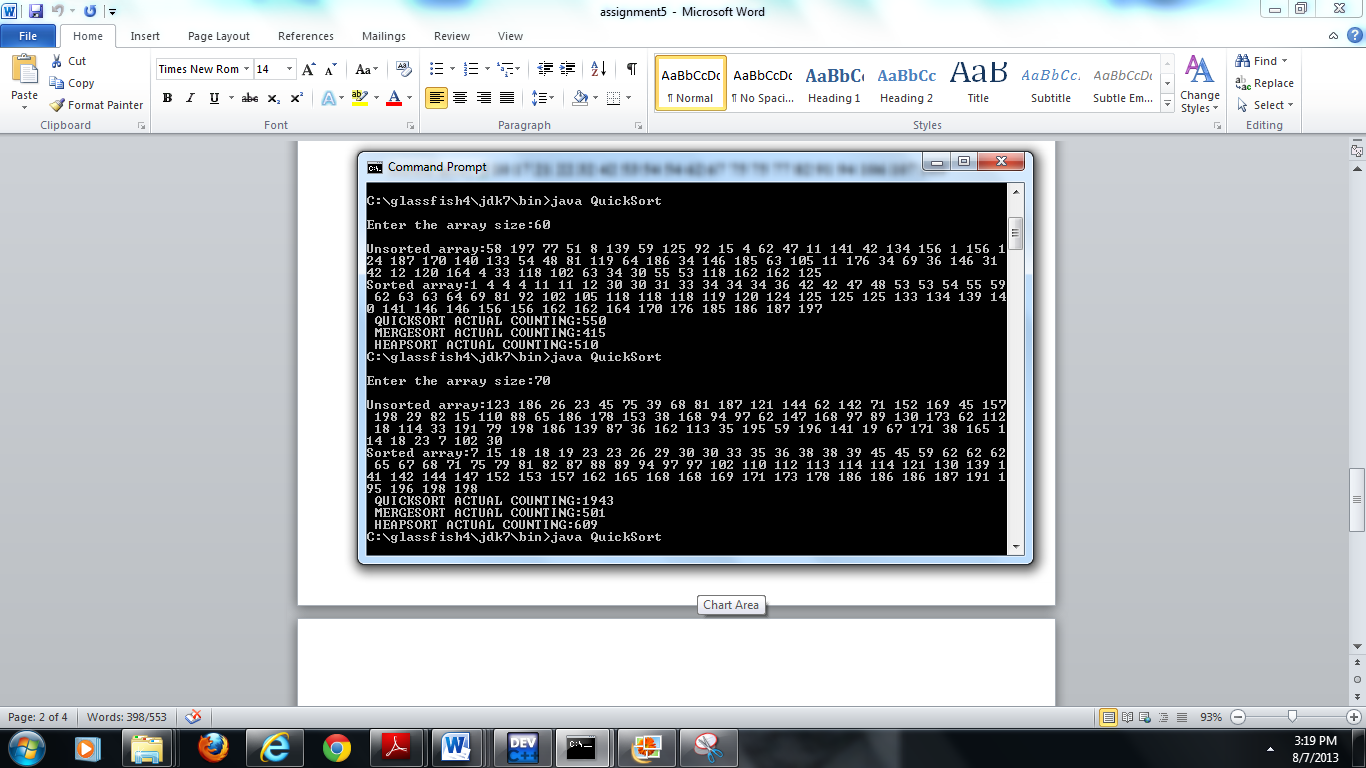
Sorted array:**4 13 15 17 18 22 31 37 41 42 45 45 46 53 55 56 58 61 63 82 83 92 100 104 106 107 108 112 119 120 121 125 133 134 141 142 144 145 152 156 162 165 165 175 175 175 180 188 189 195**

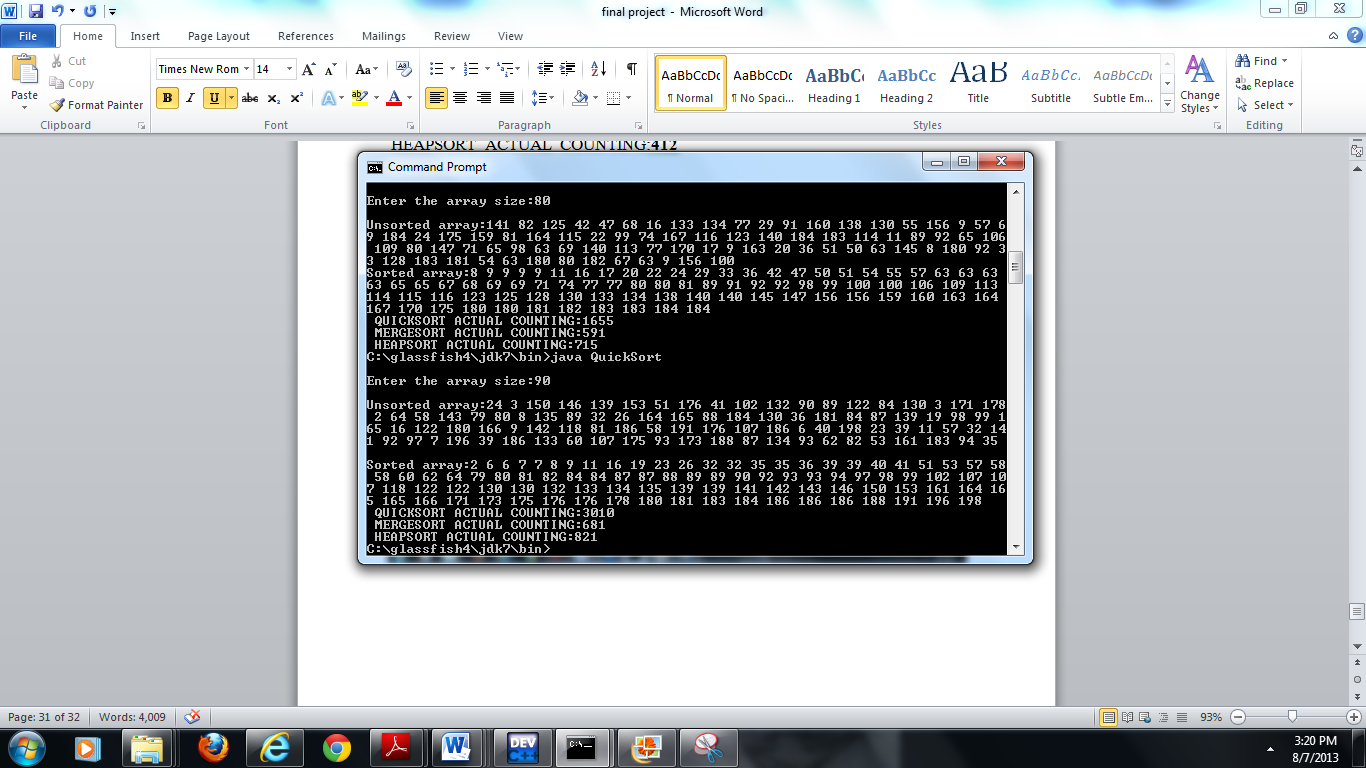
QUICKSORT ACTUAL COUNTING:**299**

MERGESORT ACTUAL COUNTING**:335**

HEAPSORT ACTUAL COUNTING:**412**

**4.2.2 with more values**





**4.2.3 Output and Graphical analysis of algorithm**

**Chapter-5 Maximum Sub-Array problem**

Maximum subarray problem is finding the subarray with maximum value in the given array. This can be used in shares problem. Every day the share values can go up or down. You might think that you can always maximize profit by either buying at the lowest price or selling at the highest price. You maximize profit by buying at the lowest price and selling at the highest price., then it would be easy to determine how to maximize profit: find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference.

A maximum subarray is to find the nonempty contiguous subarray of an array A whose values have the largest sum.

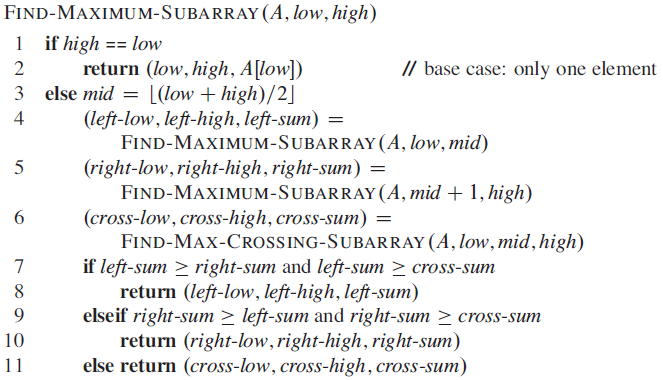
There are many methods that this problem can be performed. Here I used divide and conquer method. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say *mid*, of the subarray, and consider the subarrays A[*low..* *mid*] and A[*mid+* 1.. *high].A*ny contiguous subarray A[i.. j] of A[*ow..* *high]* must lie in exactly one of the following places:

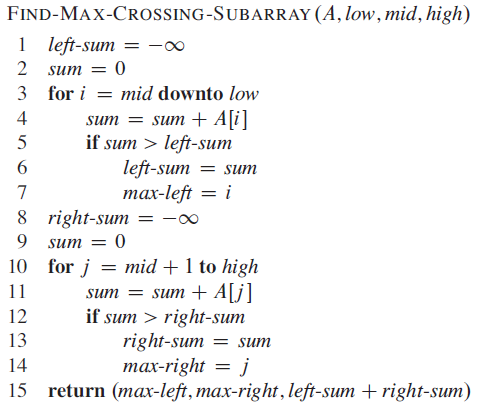
entirely in the subarray A[*low..* *mid]*, so that *low<=* i<= j<= *mid*,

entirely in the subarray A[*mid+* 1.. *high]*, so that *mid* < i<= j<= *high*, or

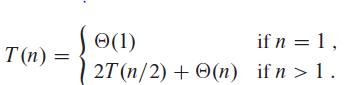
crossing the midpoint, so that *low<=* i<= *mid* < j <= *high*.

Algorithms for Maximum subarray problem:





A recurrence for the running time T(n) of FIND-MAXIMUM-SUBARRAY:



This recurrence is the same as recurrence for merge sort. This recurrence has the solution **Θ(**n lg n).

**5.1 Problem Description and Explanation**

* Program to calculate Maximum Subarray using divide and conquer method. For 5 companies share values for 30days are compared individual and calculated maximum subarray, which is customers who can get maximum profit from their shares. Finally all 5 companies’ maximum profit is compared and the highest maximum subarray value is calculated.
* Used Divide and conquer method and the algorithm. Three methods are used. One to calculate maximum subarray within the range low and mid indexes. A second method is within the range mid+1 and high. These two methods are recursive methods.Third method calculates maximum subarray within the range from low to high.
* Performed this problem in java programming language. Where the user enters number of companies share values to be compared. The user also enters number of days of share value to be compared.
* Each company values are randomly generated within the range the user specified. From the price value of each company, the change in the share value from the previous day is calculated.
* For the change for each company the maximum subarray value is calculated.
* For each maximum subarray value for each company is compared and the highest share value for a particular company is calculated.

**5.2 Input arrays, sorted array and analysis**

ENTER NUMBER OF COMPANIES, WHOSE SHARES ARE TO BE COMPARED: **5**

ENTER NUMBER OF DAYS OF THE SHARES DATA TO BE COMPARED:

**30**

ENTER THE RANGE MIN AND MAX: BASED ON WHICH SHARE VALUES ARE RANDOMLY GENERATED FOR 1 COMPANY:

**10 20**

ENTER THE RANGE MIN AND MAX: BASED ON WHICH SHARE VALUES ARE RANDOMLY GENERATED FOR 2 COMPANY:

**40 50**

ENTER THE RANGE MIN AND MAX: BASED ON WHICH SHARE VALUES ARE RANDOMLY GENERATED FOR 3 COMPANY:

**70 90**

ENTER THE RANGE MIN AND MAX: BASED ON WHICH SHARE VALUES ARE RANDOMLY GENERATED FOR 4 COMPANY:

**10 90**

ENTER THE RANGE MIN AND MAX: BASED ON WHICH SHARE VALUES ARE RANDOMLY GENERATED FOR 5 COMPANY:

**30 80**

DISPLAYING RANDOM SHARE VALUES GENERATED FOR 1 COMPANY:

**14 17 15 19 13 13 14 15 19 18 13 14 14 16 15 19 18 18 17 18 20 13 14 13 15 18 19 14 18 14**

DISPLAYING RANDOM SHARE VALUES GENERATED FOR 2 COMPANY:

**50 43 43 47 46 47 47 42 49 46 48 45 43 44 46 49 46 49 47 43 45 50 42 49 48 46 46 48 45 44**

DISPLAYING RANDOM SHARE VALUES GENERATED FOR 3 COMPANY:

**81 89 89 74 74 85 88 76 82 72 84 78 83 72 89 72 73 87 87 83 88 82 72 74 76 87 76 74 87 90**

DISPLAYING RANDOM SHARE VALUES GENERATED FOR 4 COMPANY:

**59 27 41 22 81 79 62 67 49 90 26 86 26 26 64 74 53 51 54 70 66 19 58 31 35 19 31 60 75 78**

DISPLAYING RANDOM SHARE VALUES GENERATED FOR 5 COMPANY:

**77 45 40 62 62 64 32 35 33 43 58 43 43 49 47 44 70 77 43 59 68 80 72 69 43 32 65 67 69 48**

CHANGE IN THE SHARE VALUE FOR 1 COMPANY IS:

**3 -2 4 -6 0 1 1 4 -1 -5 1 0 2 -1 4 -1 0 -1 1 2 -7 1 -1 2 3 1 -5 4 -4**

CHANGE IN THE SHARE VALUE FOR 2 COMPANY IS:

**-7 0 4 -1 1 0 -5 7 -3 2 -3 -2 1 2 3 -3 3 -2 -4 2 5 -8 7 -1 -2 0 2 -3 -1**

CHANGE IN THE SHARE VALUE FOR 3 COMPANY IS:

**8 0 -15 0 11 3 -12 6 -10 12 -6 5 -11 17 -17 1 14 0 -4 5 -6 -10 2 2 11 -11 -2 13 3**

CHANGE IN THE SHARE VALUE FOR 4 COMPANY IS:

**-32 14 -19 59 -2 -17 5 -18 41 -64 60 -60 0 38 10 -21 -2 3 16 -4 -47 39 -27 4 -16**

**12 29 15 3**

CHANGE IN THE SHARE VALUE FOR 5 COMPANY IS:

**-32 -5 22 0 2 -32 3 -2 10 15 -15 0 6 -2 -3 26 7 -34 16 9 12 -8 -3 -26 -11 33 2 2**

**-21**

MAXIMUM SUB\_ARRAY FOR **1** COMPANY

LOWER INDEX FOR THE MAXIMUM SUB\_ARRAY IS: **5**

HIGHER INDEX FOR MAXIMUM SUB\_ARRAY IS: **8**

TOTAL SUM FOR THE MAXIMUM SUB\_ARRAY IS: **6**

MAXIMUM SUB\_ARRAY FOR **2** COMPANY

LOWER INDEX FOR THE MAXIMUM SUB\_ARRAY IS: **8**

HIGHER INDEX FOR MAXIMUM SUB\_ARRAY IS: **21**

TOTAL SUM FOR THE MAXIMUM SUB\_ARRAY IS: **8**

MAXIMUM SUB\_ARRAY FOR **3** COMPANY

LOWER INDEX FOR THE MAXIMUM SUB\_ARRAY IS: **23**

HIGHER INDEX FOR MAXIMUM SUB\_ARRAY IS: **29**

TOTAL SUM FOR THE MAXIMUM SUB\_ARRAY IS: **18**

MAXIMUM SUB\_ARRAY FOR **4** COMPANY

LOWER INDEX FOR THE MAXIMUM SUB\_ARRAY IS: **4**

HIGHER INDEX FOR MAXIMUM SUB\_ARRAY IS: **9**

TOTAL SUM FOR THE MAXIMUM SUB\_ARRAY IS: **68**

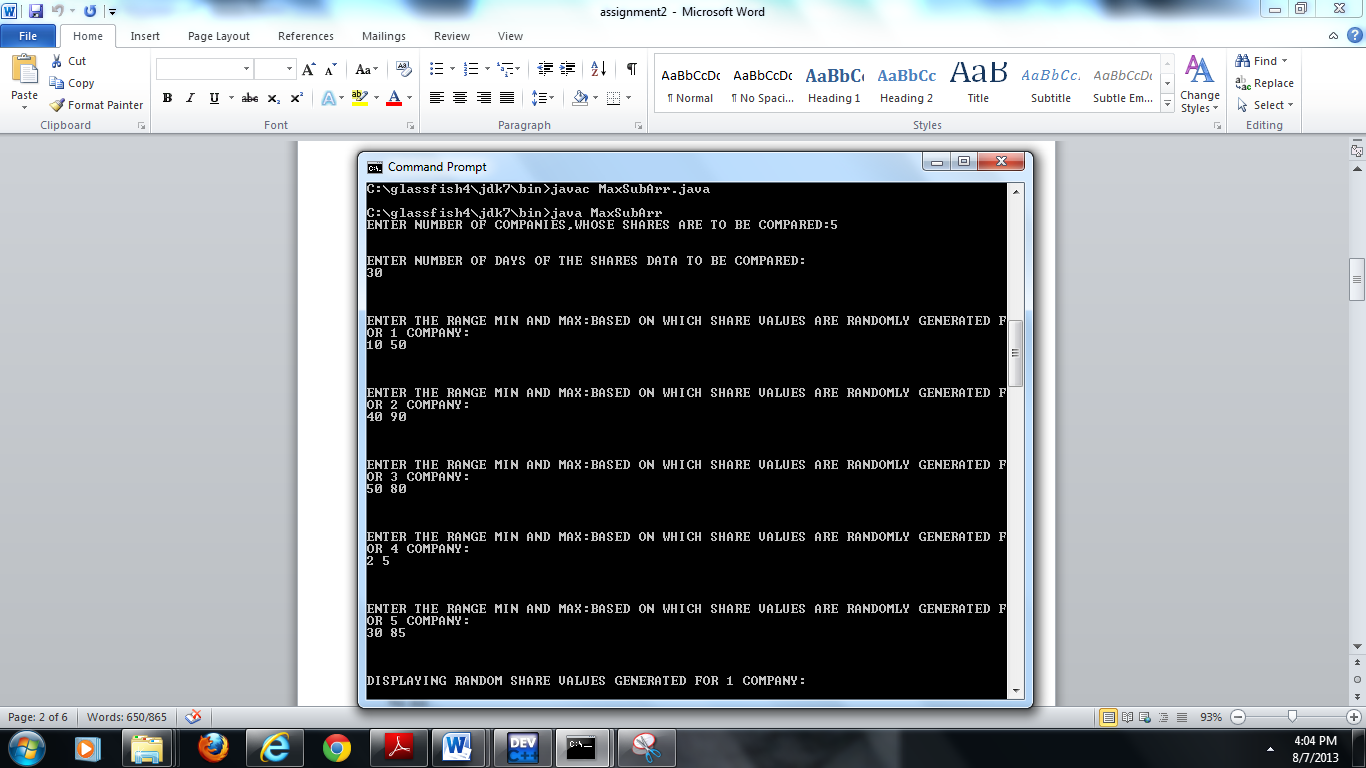
MAXIMUM SUB\_ARRAY FOR **5** COMPANY

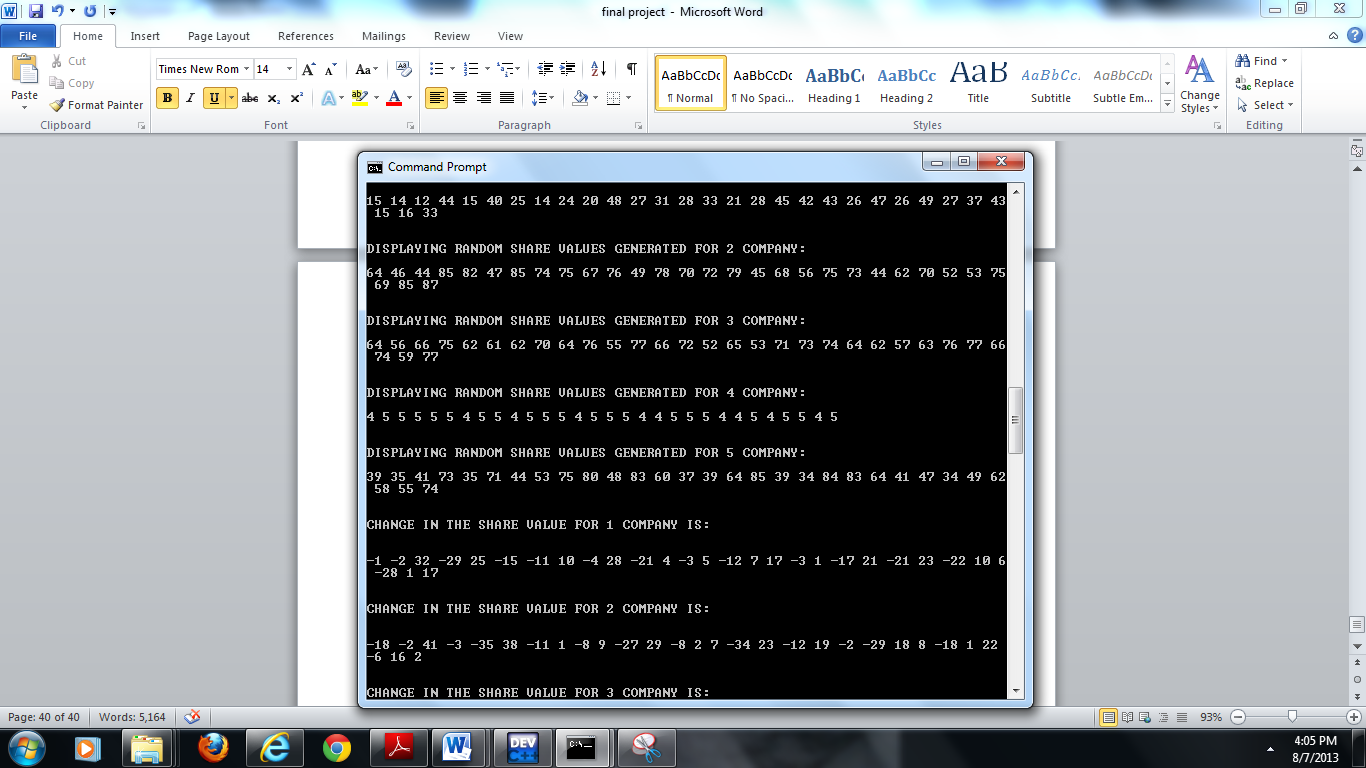
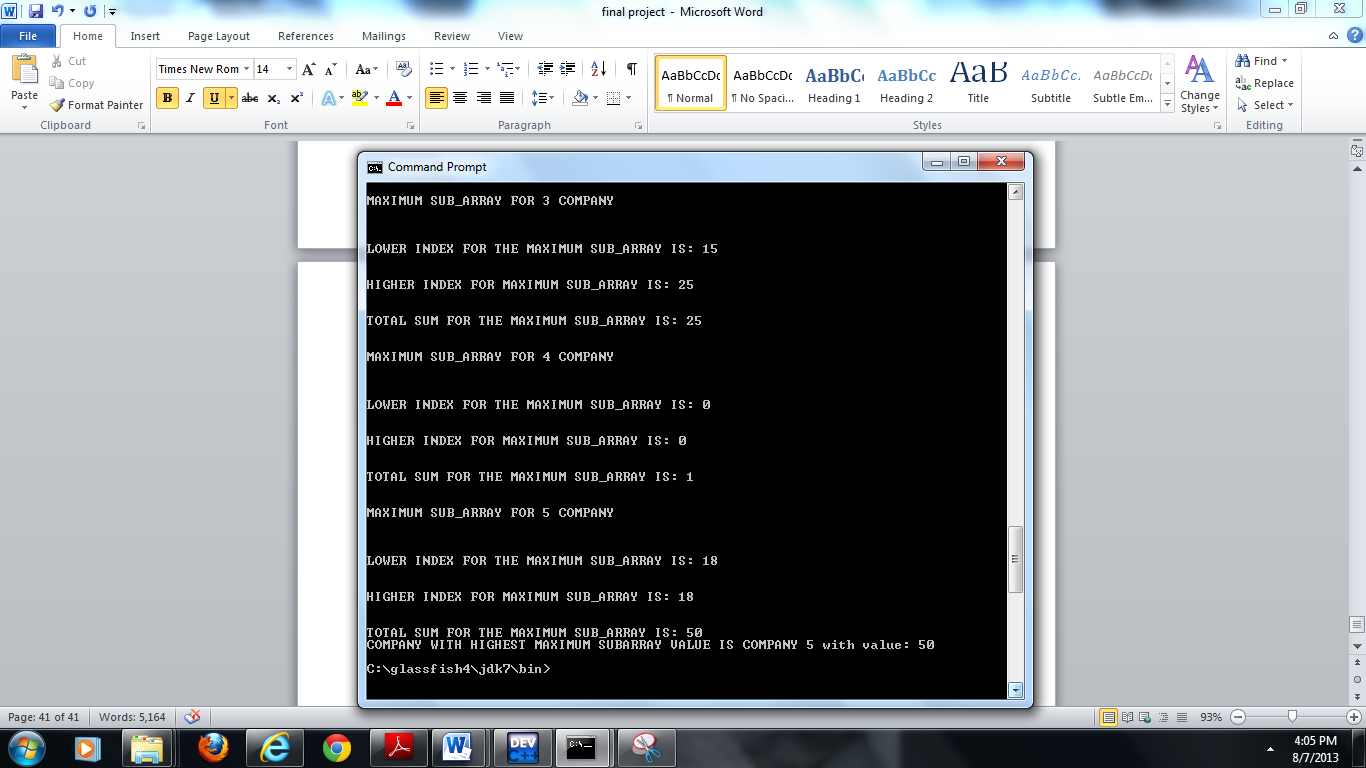
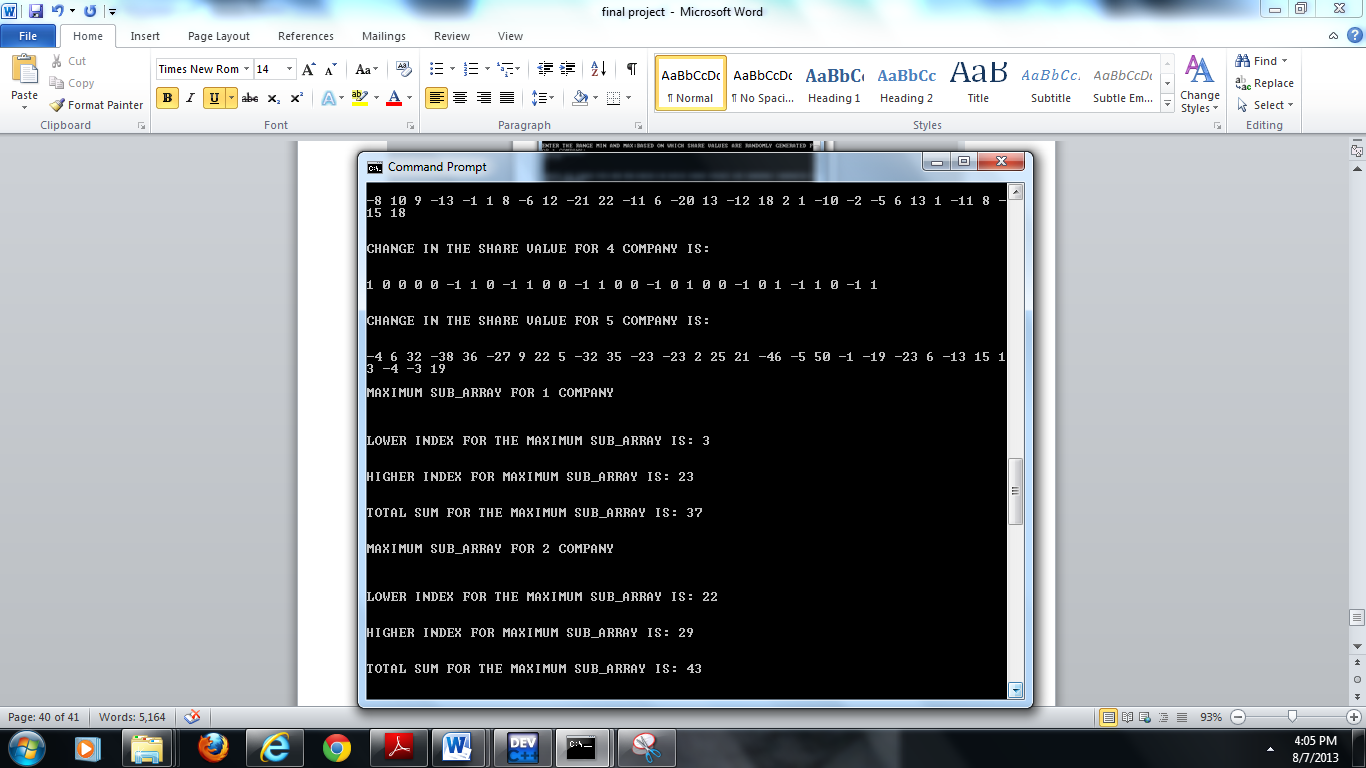
LOWER INDEX FOR THE MAXIMUM SUB\_ARRAY IS: **7**

HIGHER INDEX FOR MAXIMUM SUB\_ARRAY IS: **21**

TOTAL SUM FOR THE MAXIMUM SUB\_ARRAY IS: **48**

COMPANY WITH HIGHEST MAXIMUM SUBARRAY VALUE IS COMPANY **4** with value: **68**

**5.2.1 with more values**



**Chapter 6- Hashing and Hash Tables**

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an

arbitrary position in an array in O(1) time. Hashing is an extremely effective and practical technique: the basic dictionary operations require only O(1) time on the average.

With hashing, this element is stored in slot h(k); that is, we use a ***hash function*** h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a ***hash table*** T[0.. m- 1]

Where the size m of the hash table is typically much less than |U|. We say that an

element with key k ***hashes*** to slot h (k); we also say that h(k) is the ***hash value*** of

key k.

A good hash function satisfies (approximately) the assumption of simple uniform

hashing: each key is equally likely to hash to any of the m slots, independently of

where any other key has hashed to.

There are different methods to calculated hash functions: 1.Division method 2.Multiplication method 3.Polynomial method 4.Partition of key 5.Use part of characters6.Open addressing 7.Linear probing 8.Quadratic probing 9.Double hashing.

I want to discuss division method, multiplication method and quadratic probing.

In division method, the hash function is **kmodm** where k is key and m is hash table size.m should be prime number and far from power of 2.If two keys gets same slot then they use linked list, so that all the numbers that share the slot are in linked list. Which avoids collision but worst case if the entire keys map to same slot then it is similar to search in linked list.

In multiplication method, the hash function is integral part of m\*(kAmod1)).Where A is a constant value= 0:6180339887 … where 0<A<1.This method is similar to division method except here it uses constant A and multiplies with m.

In quadratic Probing, there is no chaining and the keys are actual stored in the hash table. The hash table is slightly bigger than number of keys. To avoid collision, when two keys share same slot the hash function is calculated again. Based on that value the slot is assigned. This is repeated till finding empty slot or reached the table size.

Hi(x) = (hash(x) +i2) mod m where f (i) =i2.

**6.1 Problem Description and Explanation**

* Program to implement hash table using hash function in three different methods. Division method, multiplication method and quadratic probing method. Implemented in java programming language. Implemented three methods
* 150 student ID’s need to be stored using hash function and hash table. The hash table size is prime number, which is calculated within the program for the given range of numbers. Hashing performed for student ID’s are randomly generated.
* Hash table is built based on the array size which depends on prime number for division method and multiplication method.
* All student ID’s are inserted into hash table using key, which is calculated using hash function.
* If the key already holds student ID, it will be inserted using chaining i.e. using linked list. This is performed in division method and multiplication method.
* Quadratic probing doesn’t use linked list, there is no chaining. It directly inserts into the table based on the key. If the slot is already occupied then the hash function is calculated again till it finds empty slot for the data.
* Performs search on student ID’s on all three methods.
* Perform insertion in to hash tables using three methods.

**6.2 Input arrays, sorted array, and analysis**

**PROGRAM TO FIND PRIME NUMBERS BETWEEN THE RANGE:**

ENTER FIRST +VE NUMBER: **32**

ENTER SECOND +VE NUMBER:**64**

**prime number 37**

**prime number 41**

**prime number 43**

**prime number 47**

**prime number 53**

**prime number 59**

**prime number 61**

**CHOOSE ARRAY SIZE(m) FROM THE PRIME NUMBERS WHICH IS NOT CLOSE TO POWER OF 2:59**

OPTION1:**DIVISION METHOD**

OPTION2:**MULTIPLICATION METHOD**

OPTION3:**QUADRATIC PROBING**

OPTION4:**EXIT**

ENTER YOUR CHOICE:**1**

**Index 0 has 545042 --> 771189 --> 139771**

**Index 1 has 169508 --> 427102 --> 967896 --> 560855**

**Index 2 has 899162**

**Index 3 has 484629 --> 473832**

**Index 4 has**

**Index 5 has 402149**

**Index 6 has 869076 --> 554075**

**Index 7 has 163201**

**Index 8 has 949082 --> 844534 --> 539504**

**Index 9 has 200432 --> 297015 --> 710133 --> 506052**

**Index 10 has 708718 --> 715503**

**Index 11 has 519801 --> 513429**

**Index 12 has 558565 --> 207869 --> 144916**

**Index 13 has 308878**

**Index 14 has 599395 --> 192531 --> 429534 --> 436260 --> 762648**

**Index 15 has 478092 --> 926610 --> 965373**

**Index 16 has 581697 --> 198374 --> 463225 --> 602052**

**Index 17 has 705598 --> 237492**

**Index 18 has 768906 --> 469599 --> 760764**

**Index 19 has 529190**

**Index 20 has 495797 --> 836758 --> 693624 --> 748907 --> 813630**

**Index 21 has 876938 --> 572026**

**Index 22 has**

**Index 23 has 252071 --> 825610**

**Index 24 has 816171 --> 543591**

**Index 25 has 612799 --> 347417**

**Index 26 has 964263 --> 519226 --> 635515**

**Index 27 has 436037 --> 664485 --> 795052**

**Index 28 has 795584 --> 564717 --> 277564 --> 419931 --> 146997**

**Index 29 has 702070 --> 307537 --> 210659 --> 779537 --> 350725 --> 174669**

**Index 30 has 988988 --> 520233 --> 145996 --> 946095 --> 119151 --> 585959 --> 468785**

**Index 31 has 746204 --> 260575 --> 480173 --> 930638 --> 435805**

**Index 32 has 175203 --> 458462**

**Index 33 has 560061 --> 720187**

**Index 34 has 643606 --> 124996 --> 973711 --> 683608**

**Index 35 has**

**Index 36 has 681427**

**Index 37 has 299521 --> 212732**

**Index 38 has 787157 --> 581719**

**Index 39 has 388141 --> 401652**

**Index 40 has 435932 --> 752939 --> 833769 --> 895011**

**Index 41 has 229079 --> 319172 --> 295041 --> 451922 --> 385783 --> 782322 --> 810465**

**Index 42 has 805923 --> 602668 --> 211203 --> 321238 --> 917610**

**Index 43 has 903333**

**Index 44 has**

**Index 45 has 374872 --> 372276 --> 146601**

**Index 46 has**

**Index 47 has 475587 --> 464200**

**Index 48 has 205427**

**Index 49 has**

**Index 50 has 601378 --> 643386 --> 842924**

**Index 51 has 791005**

**Index 52 has 343196 --> 879506 --> 482967**

**Index 53 has 141830**

**Index 54 has 696136 --> 492527 --> 102183**

**Index 55 has 506452 --> 396004 --> 587872 --> 137348 --> 879686 --> 659498**

**Index 56 has 701861**

**Index 57 has 992614**

**Index 58 has 304970 --> 434711 --> 352878 --> 330222**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **842924**

**StudentID found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **330221**

**StudentID not found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**0**

OPTION1:**DIVISION METHOD**

OPTION2:**MULTIPLICATION METHOD**

OPTION3:**QUADRATIC PROBING**

OPTION4:**EXIT**

ENTER YOUR CHOICE:**2**

**Index 0 has**

**Index 1 has 887736 --> 718090**

**Index 2 has 212778 --> 213854 --> 917071**

**Index 3 has 781627 --> 330058**

**Index 4 has 301579 --> 345685 --> 873300**

**Index 5 has**

**Index 6 has 656333 --> 328275**

**Index 7 has 638745 --> 795416 --> 273990**

**Index 8 has 770021**

**Index 9 has 850015 --> 139046 --> 240521 --> 377884**

**Index 10 has 218409 --> 354285 --> 374059 --> 422257**

**Index 11 has 922935 --> 692548 --> 954087 --> 208717**

**Index 12 has 557570 --> 622348 --> 401742**

**Index 13 has 298416 --> 220574**

**Index 14 has 927862 --> 979830 --> 428493**

**Index 15 has 686563 --> 324591 --> 357628**

**Index 16 has**

**Index 17 has 804309**

**Index 18 has 403056 --> 304919**

**Index 19 has 269476**

**Index 20 has 901595 --> 568890 --> 286611 --> 132181**

**Index 21 has 146300 --> 675567**

**Index 22 has 353376 --> 670954 --> 655306 --> 582399 --> 218220 --> 948175 --> 138548**

**Index 23 has 835034**

**Index 24 has 140480 --> 416248**

**Index 25 has 213798 --> 468051 --> 741324 --> 766732**

**Index 26 has**

**Index 27 has 189945 --> 307390 --> 325334 --> 950387**

**Index 28 has 977858 --> 560025 --> 377658 --> 284922 --> 533397 --> 575762**

**Index 29 has 649169 --> 802413 --> 524582 --> 653117**

**Index 30 has 789760 --> 977028**

**Index 31 has 280555 --> 313804**

**Index 32 has 836806 --> 956691 --> 417431**

**Index 33 has 824009 --> 276010 --> 658078**

**Index 34 has 547055 --> 357923 --> 749772**

**Index 35 has**

**Index 36 has 280937 --> 690497 --> 261252**

**Index 37 has 786461 --> 119865**

**Index 38 has**

**Index 39 has 423926 --> 807235 --> 428916**

**Index 40 has 138521 --> 772690 --> 318126 --> 888250**

**Index 41 has 516353 --> 174299**

**Index 42 has 753018 --> 660604 --> 641762 --> 155169 --> 197534 --> 723662 --> 626080 --> 546175**

**Index 43 has 406152 --> 701550**

**Index 44 has 109568**

**Index 45 has**

**Index 46 has 942231 --> 281091**

**Index 47 has 103316 --> 498537 --> 654941**

**Index 48 has 768938 --> 152624 --> 759966**

**Index 49 has 192062 --> 626763 --> 616427**

**Index 50 has 652959 --> 676681 --> 629415 --> 676770 --> 799815**

**Index 51 has 709265 --> 581340**

**Index 52 has 181773 --> 285311**

**Index 53 has 193452 --> 341062 --> 908136**

**Index 54 has 906806 --> 380034 --> 340575 --> 562456**

**Index 55 has 271218 --> 943859 --> 160593**

**Index 56 has 765652 --> 680147 --> 778051**

**Index 57 has**

**Index 58 has 301985 --> 139070**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **758051**

**StudentID not found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **908136**

**StudentID found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **778052**

**StudentID not found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**0**

OPTION1:**DIVISION METHOD**

OPTION2:**MULTIPLICATION METHOD**

OPTION3:**QUADRATIC PROBING**

OPTION4:**EXIT**

ENTER YOUR CHOICE:**3**

ENTER TOTAL NUMBER OF STUDENTS:**60**

ENTER HASH TABLE SIZE:**67**

**Index 0 has 352554**

**Index 1 has 291183**

**Index 2 has 761926**

**Index 3 has**

**Index 4 has 984904**

**Index 5 has 801391**

**Index 6 has 843936**

**Index 7 has 674027**

**Index 8 has 566623**

**Index 9 has**

**Index 10 has 370118**

**Index 11 has 904980**

**Index 12 has 275516**

**Index 13 has 265332**

**Index 14 has 755368**

**Index 15 has**

**Index 16 has**

**Index 17 has 455349**

**Index 18 has 394848**

**Index 19 has 988871**

**Index 20 has 569185**

**Index 21 has 953431**

**Index 22 has 204439**

**Index 23 has 912826**

**Index 24 has 784728**

**Index 25 has**

**Index 26 has 370871**

**Index 27 has 572274**

**Index 28 has 991962**

**Index 29 has 284042**

**Index 30 has 330272**

**Index 31 has 738706**

**Index 32 has 888115**

**Index 33 has 865133**

**Index 34 has 706007**

**Index 35 has 219924**

**Index 36 has 797800**

**Index 37 has 507423**

**Index 38 has 963431**

**Index 39 has 331421**

**Index 40 has 730873**

**Index 41 has 572422**

**Index 42 has 463213**

**Index 43 has 544686**

**Index 44 has 631974**

**Index 45 has 705287**

**Index 46 has 118687**

**Index 47 has 478896**

**Index 48 has 517756**

**Index 49 has 948635**

**Index 50 has 568880**

**Index 51 has 320108**

**Index 52 has 528276**

**Index 53 has 345098**

**Index 54 has 262338**

**Index 55 has 496324**

**Index 56 has 837354**

**Index 57 has 560778**

**Index 58 has 650494**

**Index 59 has 962178**

**Index 60 has**

**Index 61 has**

**Index 62 has 578406**

**Index 63 has 199187**

**Index 64 has 955819**

**Index 65 has 836827**

**Index 66 has 825908**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **560778**

**StudentID not found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **578406**

**StudentID found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**1**

Enter a StudentID to search **825907**

**StudentID not found**

DO YOU WANT TO SEARCH FOR STUDENTID:1/0

**0**

OPTION1:**DIVISION METHOD**

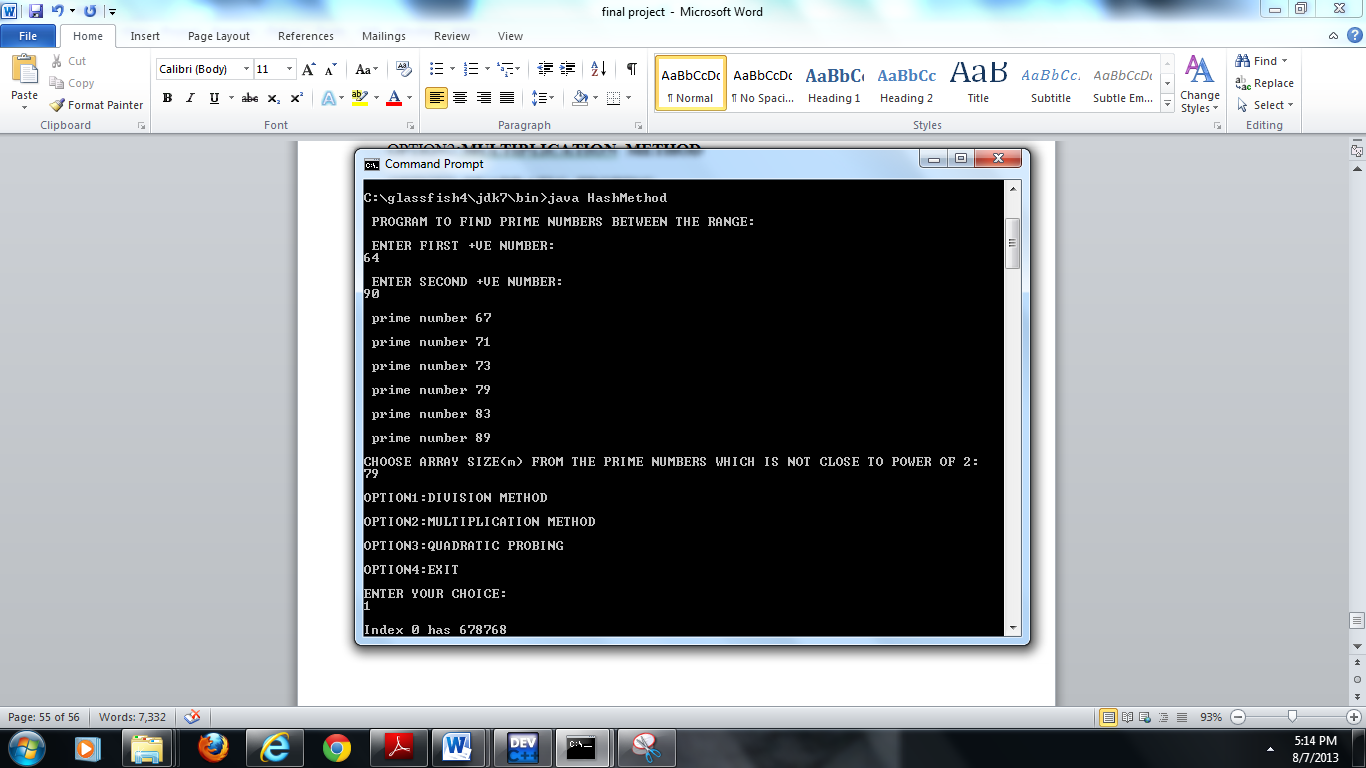
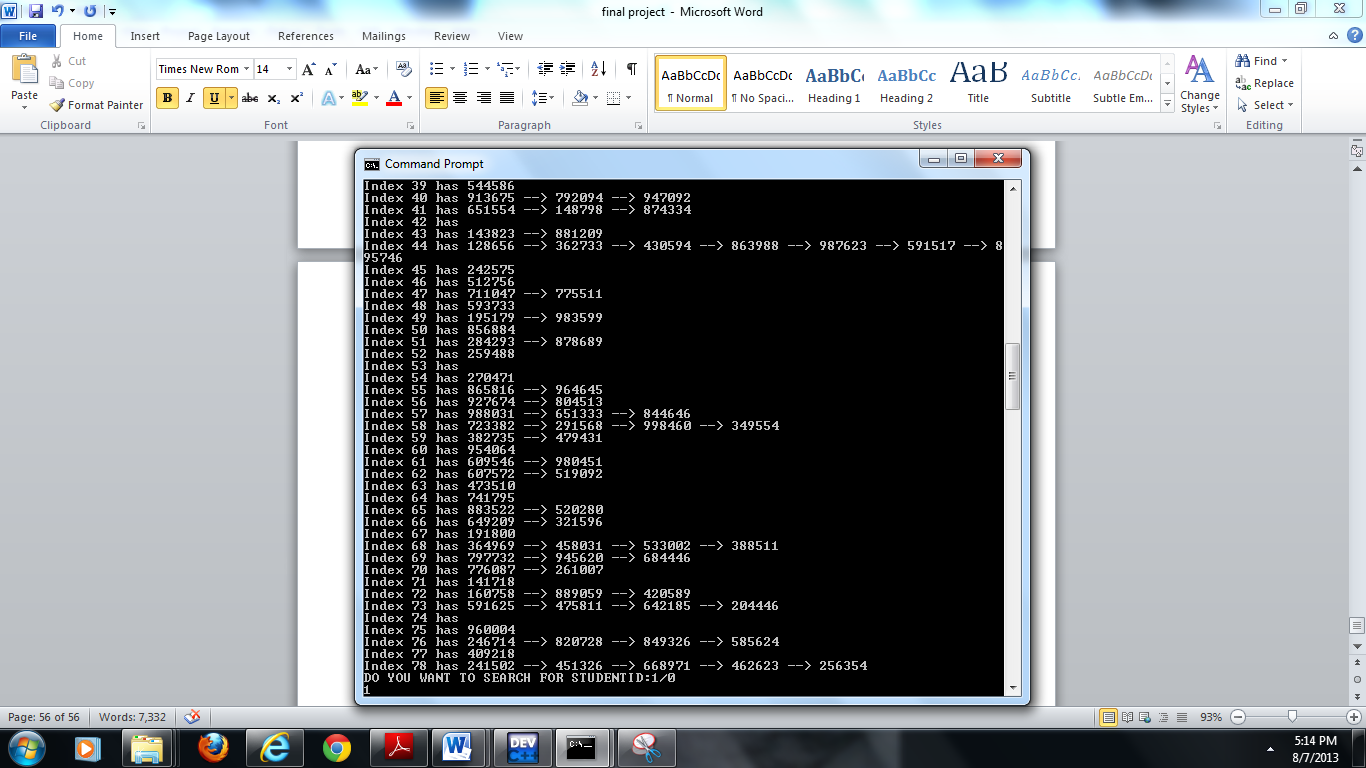
OPTION2:**MULTIPLICATION METHOD**

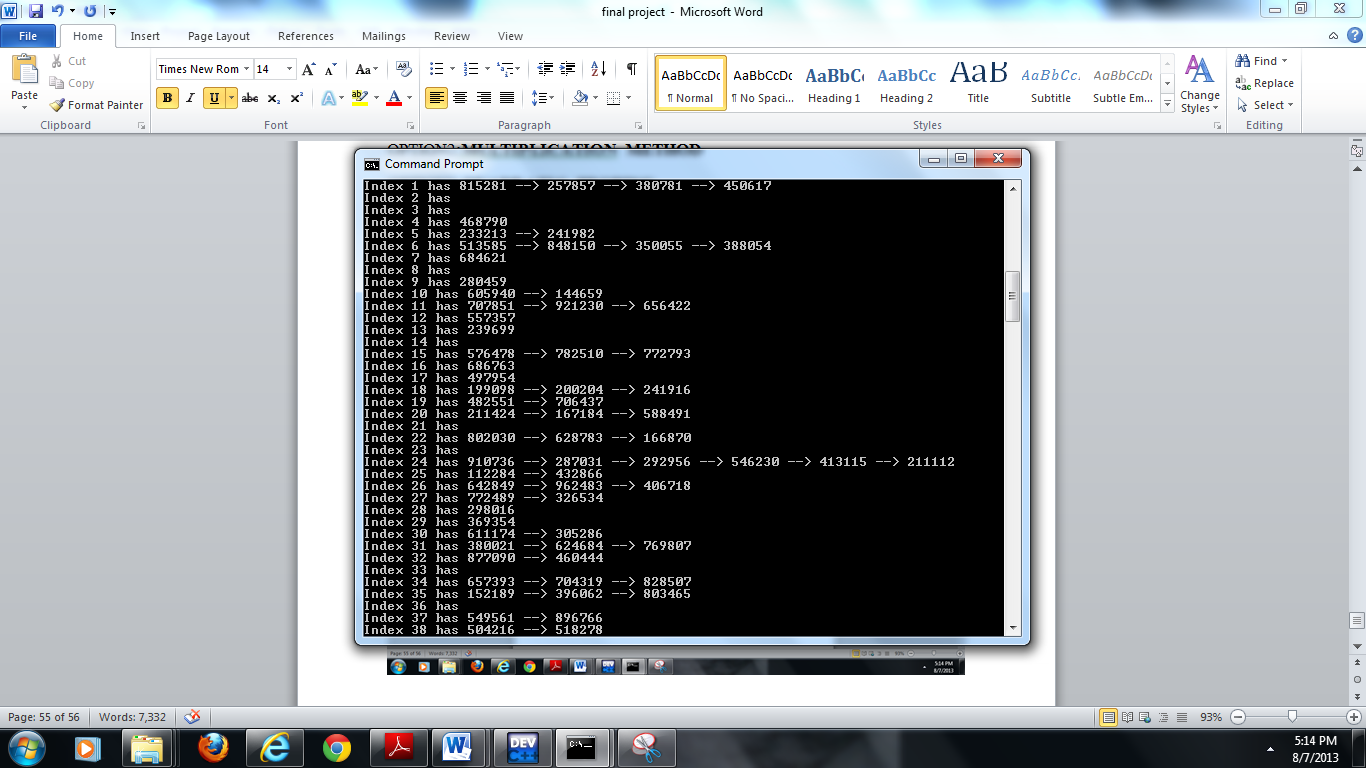
OPTION3:**QUADRATIC PROBING**

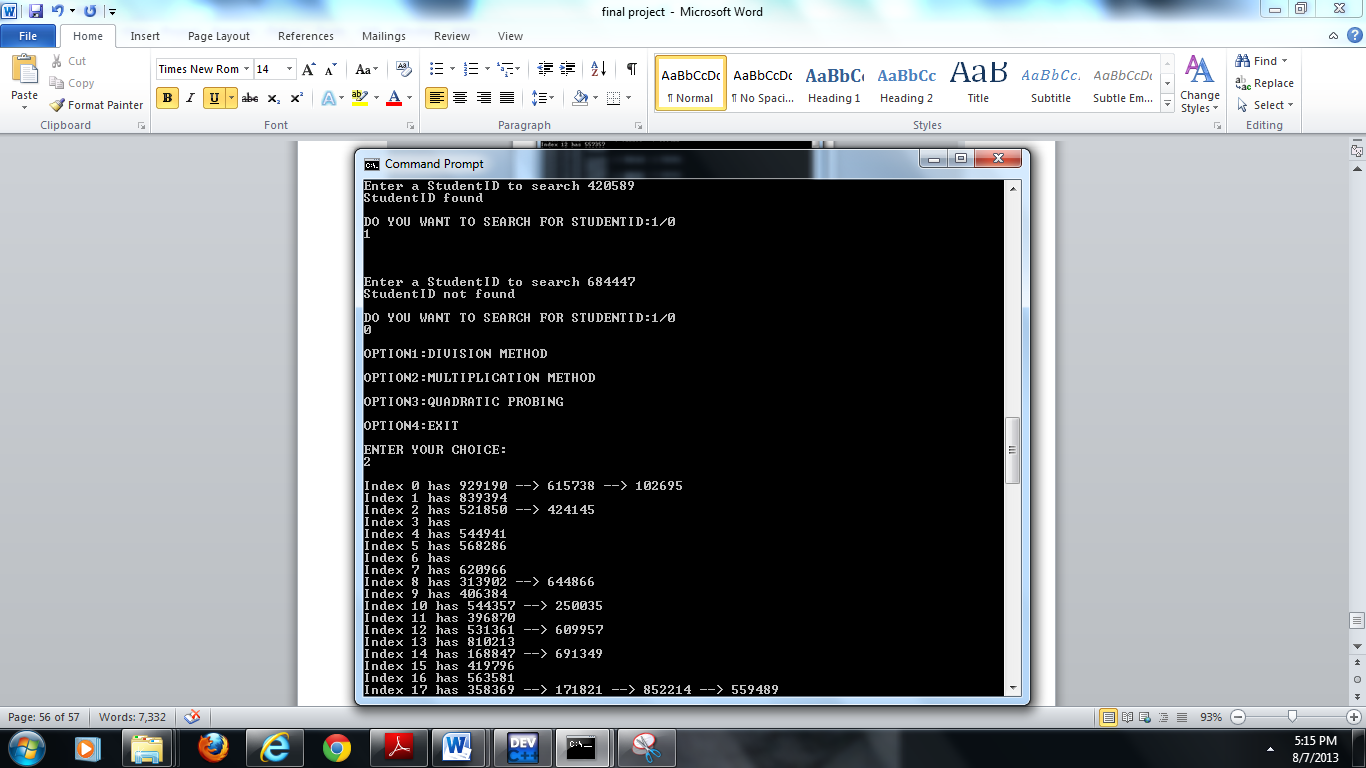
OPTION4:**EXIT**

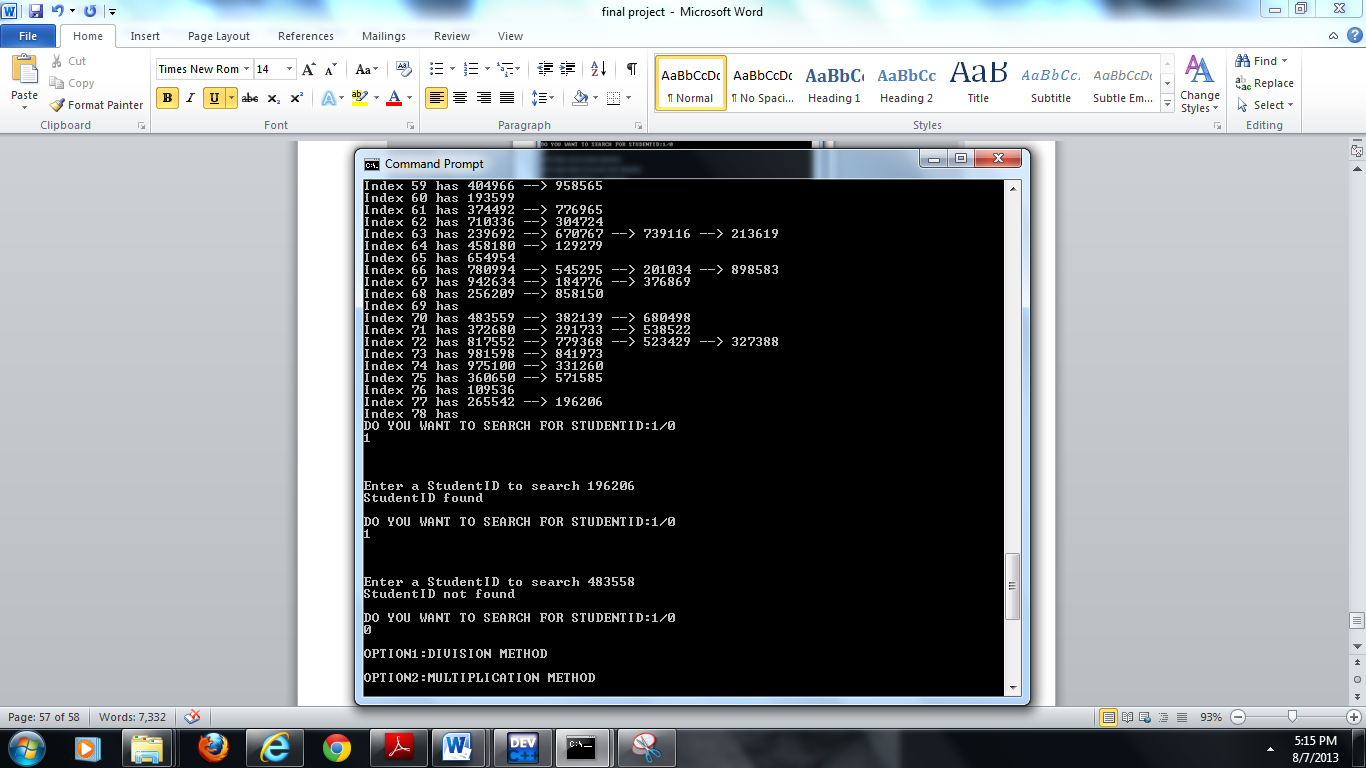
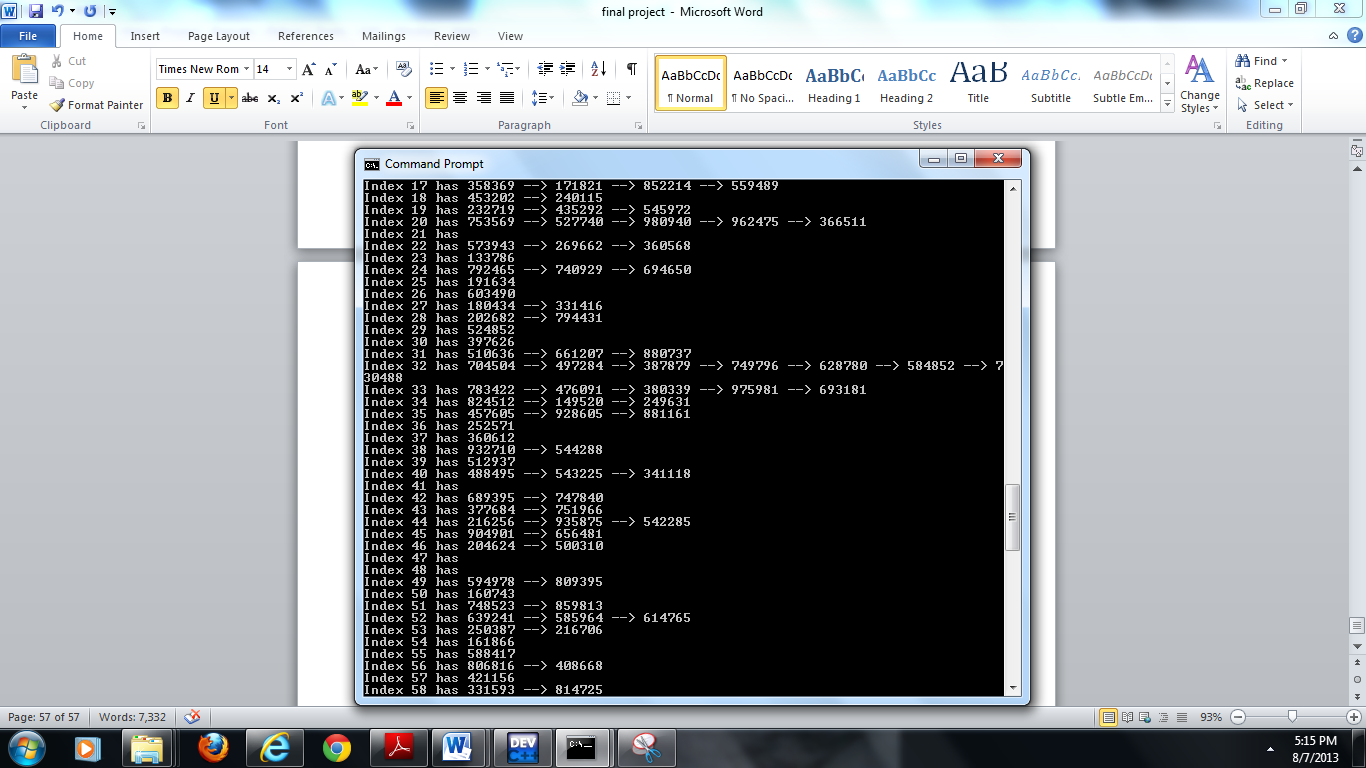
ENTER YOUR CHOICE:

**4**

**6.2.1 with more values**







**Chapter 7- Matrix Chain Multiplication**

Matrix chain multiplication is an example of dynamic programming. Dynamic programming applied for optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value. When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.

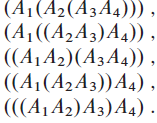
2. Recursively defines the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

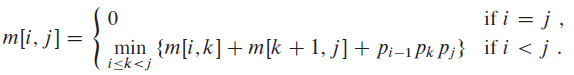
4. Construct an optimal solution from computed information.

Multiplying Pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is A1, A2, A3, A4, then we can fully parenthesize the product

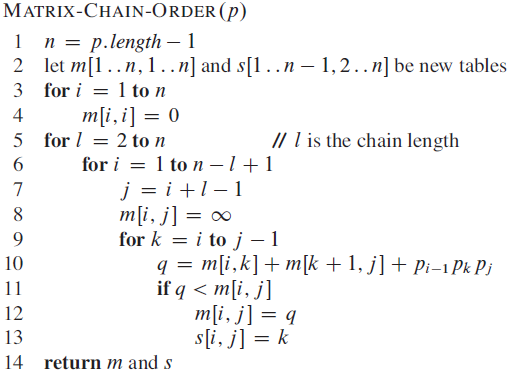
A1A2A3A4 in five distinct ways:



The minimum cost of parenthesizing the product can be calculated recursively using below formula:

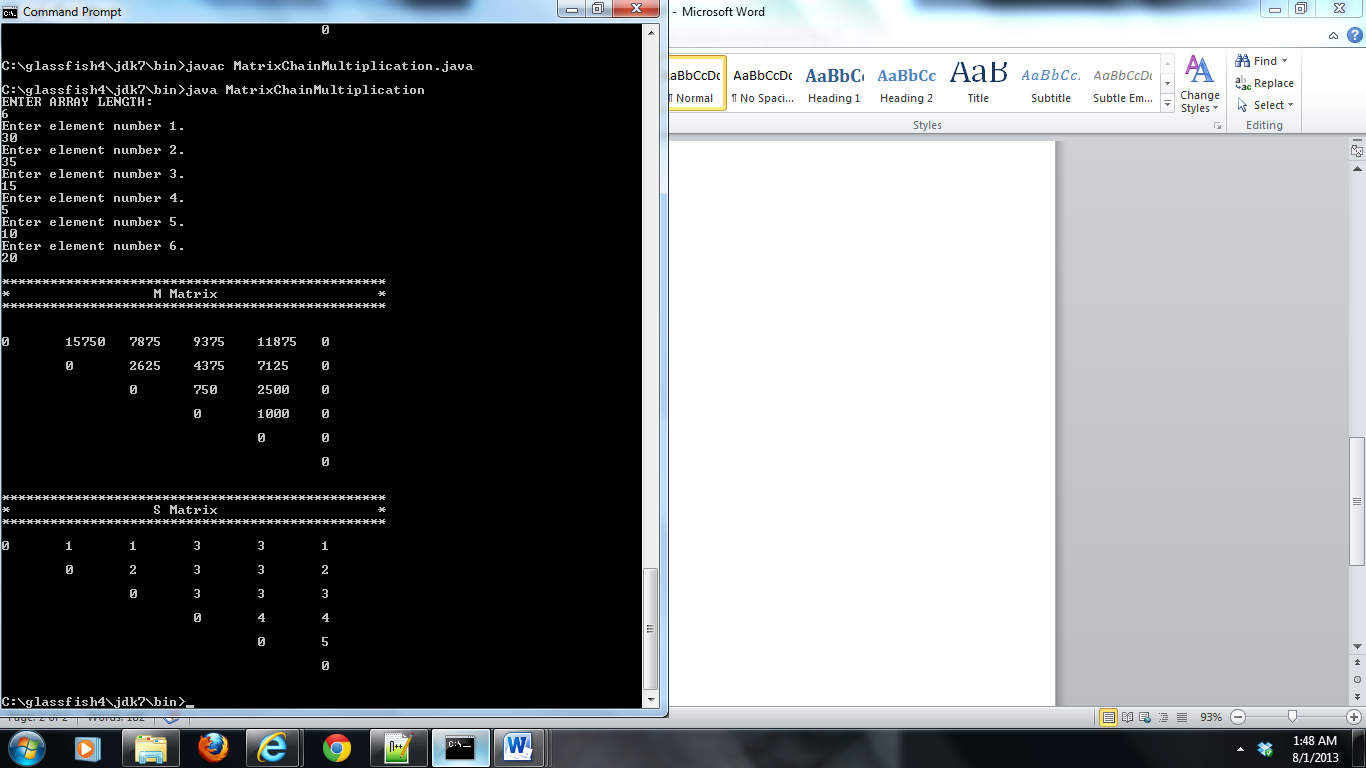


Algorithm followed by matrix chain multiplication is:

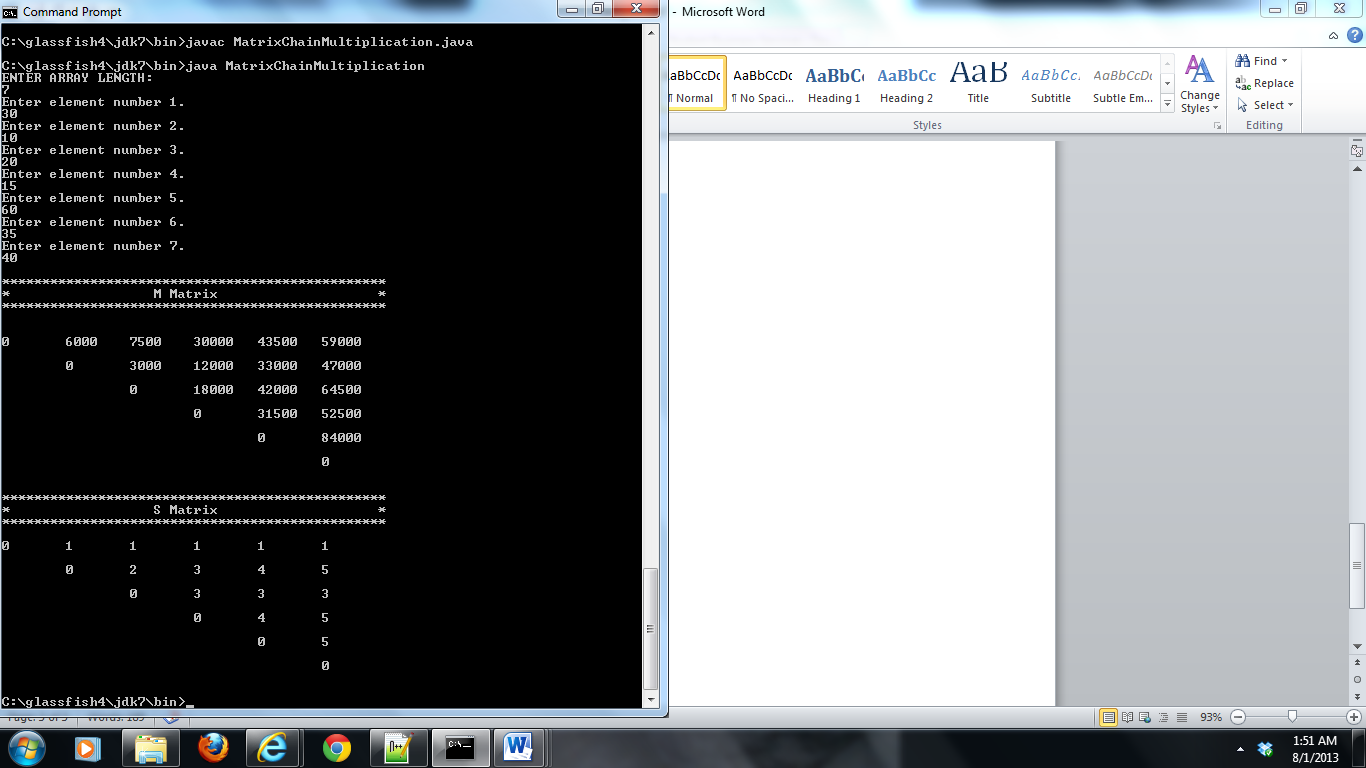


**7.1 Matrices Chain Multiplication Problem**

* Implemented the problem in java programming language
* Executed for textbook example and personal example
* Followed the algorithm from the class.

**7.2 Print Out m [i, j] and s[i,j] tabulations with same matrices as in the textbook**

**7.3 printout m[i][j] and s[I,j] tabulations with matrices generated by yourself**



**Chapter 8 Huffman Codes Problem**

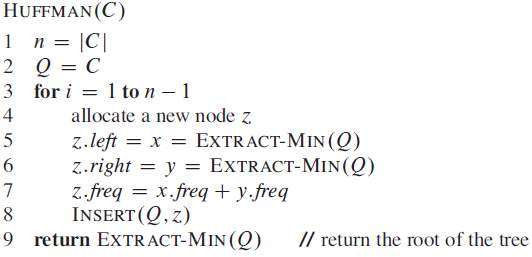
**8.1 Variable length encoding**

Huffman’s greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string. There are many options for how to represent a file of information. Here, we consider the problem of designing a ***binary character code*** (or ***code*** for short) in which each character is represented by a unique binary string, which we call a codeword.

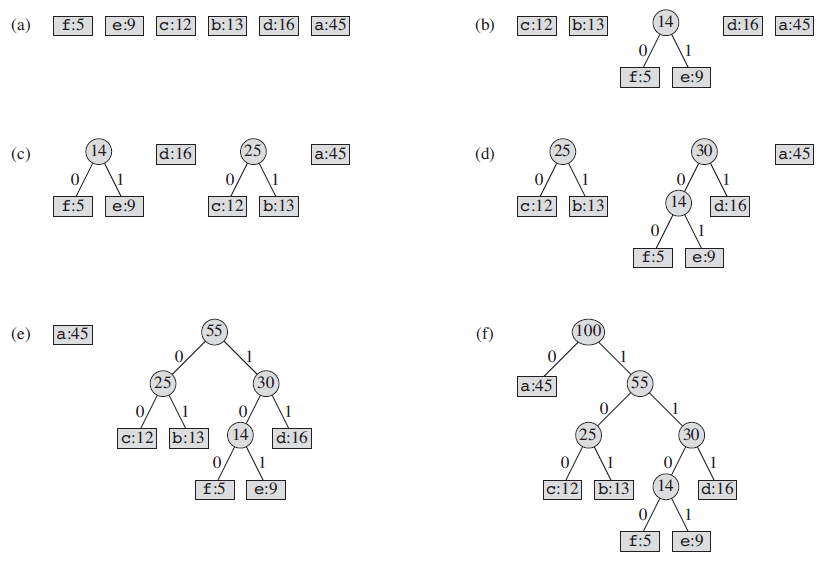
A ***variable-length code*** can do considerably better than a fixed-length code, by

giving frequent characters short codewords and infrequent characters long codewords.

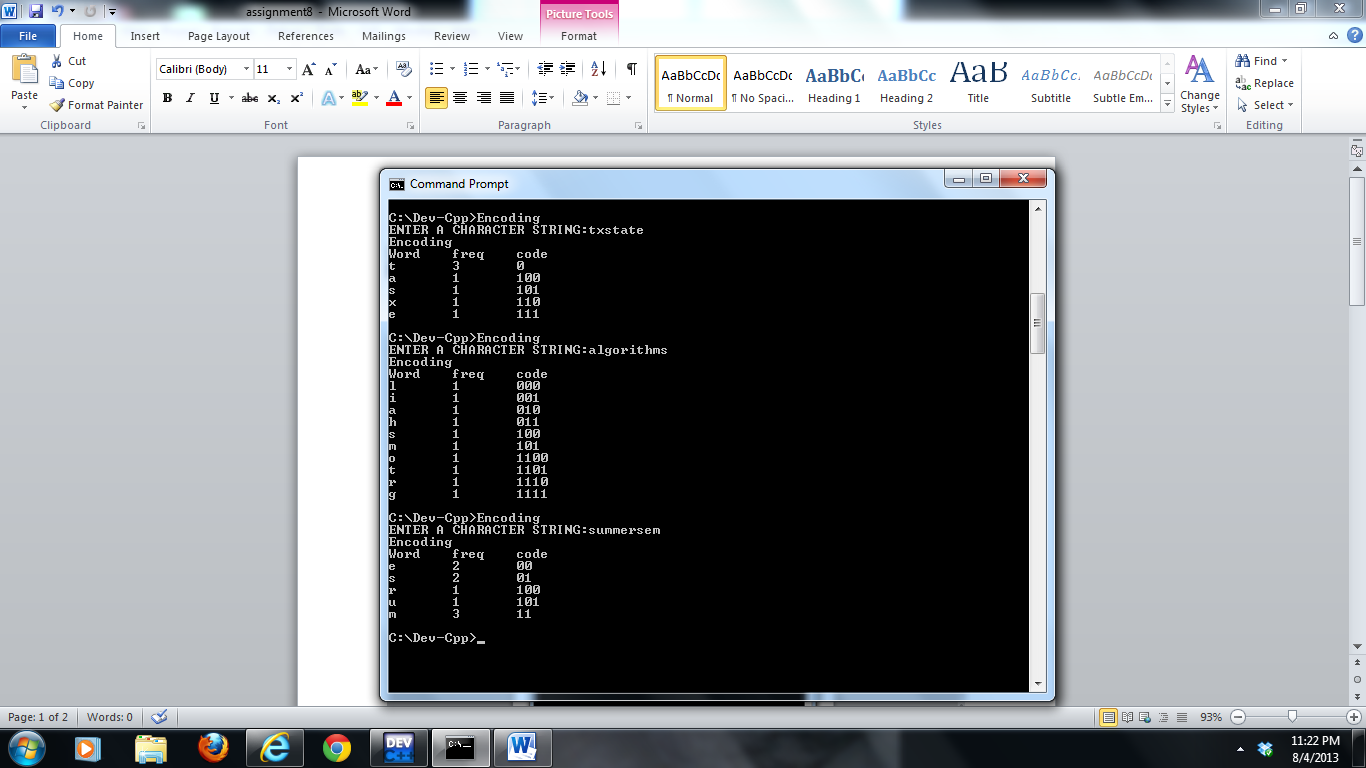
Algorithm for Huffman Codes:



Consider the below figure, where 6 characters along with their frequency is given. Building Huffman algorithm is first combine the infrequency characters and proceed in this manner till the complete characters are implemented.

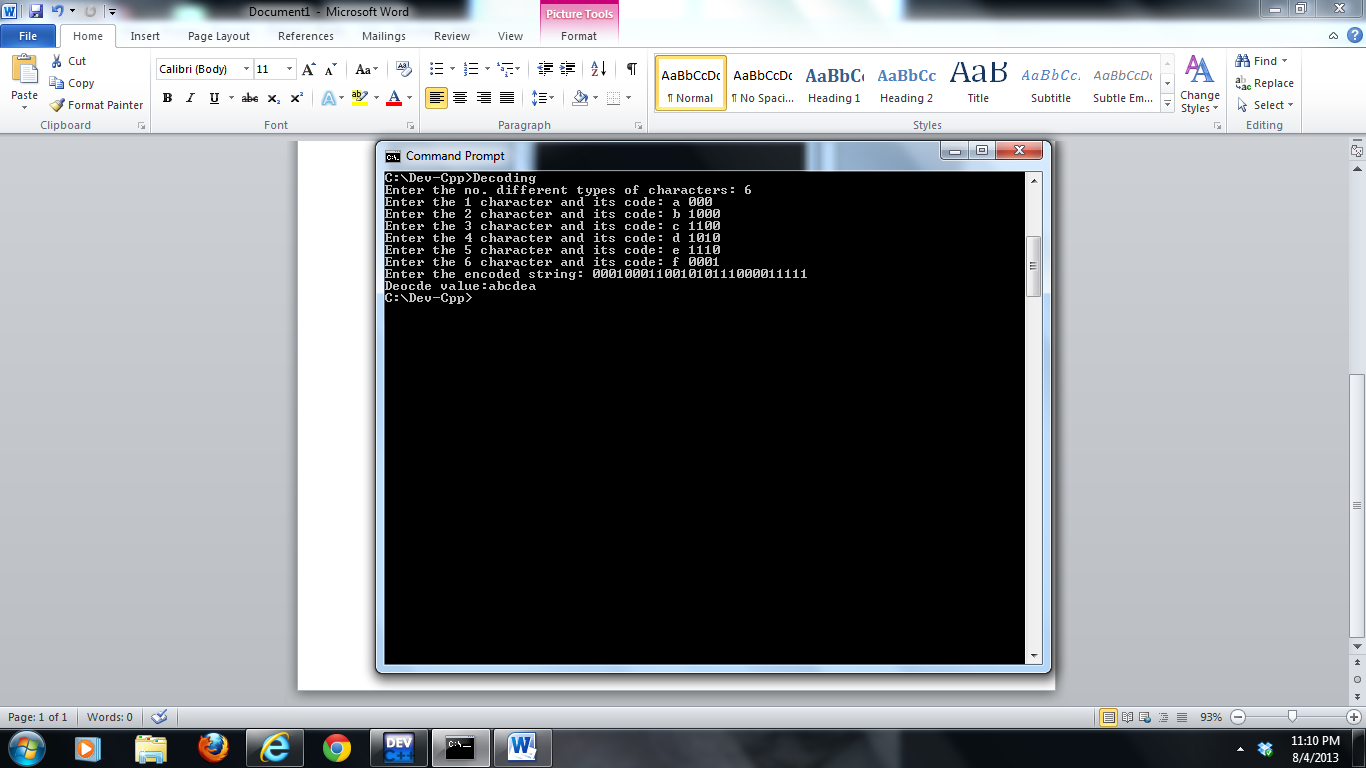


**Input Texts for compression**



**Decompressed texts**





**8.2 Fixed length encoding**

In a fixed-length code, we need 3 bits to represent 6 characters:

a = 000, b = 001. . . f = 101. This method requires 300,000 bits to code the

entire file with 100000 bits data.

Fixed length representation:



**8.3 Comparisons**

* Fixed length representation uses 25% more space than variable length codeword.
* Variable length codeword is more efficient then fixed length code word.