Hacettepe University
Department of Computer Engineering
BBM204 Programming Lab.
Spring 2021
Programming Assignment 1

Subject: Analysis of Algorithms
Submission Date: 10.03.2021
Deadline:24.03.2021 23:59
Programming Language: Java

1. Introduction

In this experiment, you will analyze different algorithms and compare their running times. You are expected to measure running times of the algorithms which are listed in the Problem section, and comment about the results. This will allow you to observe under what conditions the theoretical complexities are valid.

2. Background

Analysis of algorithms is the area of computer science that provides tools to analyze the efficiency of different methods of solutions. Efficiency of an algorithm is depending on different parameters such as execution time, memory space, disk space and etc.. Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of the analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis;

- **Worst-case** − The maximum number of steps taken on any instance of size a.
- **Best-case** − The minimum number of steps taken on any instance of size a.
- **Average case** − An average number of steps taken on any instance of size a.

Algorithm analysis is important in practice because the accidental or unintentional use of an inefficient algorithm can significantly impact system performance. In time-sensitive applications, an algorithm taking too long to run can render its results outdated or useless. An inefficient algorithm can also end up requiring an uneconomical amount of computing power or storage in order to run, again rendering it practically useless.

A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modelled input
- Calculate the total running time by multiplying the time by the frequency for each operation, then

adding all the products.

2.1 Example of Algorithm analysis

On these experiments, you will measure time requirements of the algoritms and compare their time complexities. A time complexity analysis should focus on gross differences in the efficiency of algorithms that are likely to dominate the overall cost of a solution. An example of algorithm analysis is given below;

Table 1. Running times and Unit cost of each lines of code

Code Fragment	Unit Cost	Times
i=1;	c1	1
sum=0;	c2	1
while (i≤n) {	c3	n+1
j=1;	c4	n
while (j≤n) {	c5	n*(n+1)
sum =sum+i;	c6	n*n
j=j+1;	c7	n*n
}	-	-
i=i+1;	c8	n
}	-	-

Total Cost = c1 + c2 + (n+1) *c3 + n*c4 + n*(n+1) *c5+n*n*c6 + n*n*c7 + n*c8.

The time required for this algorithm is proportional to n^2 which is determined as growth rate and it is usually denoted as $O(n^2)$. The details of the notations are not given in this assignment paper, you should research and use it for your comments.

Also, you can see list of some classes of functions with example expression when analyzing the running time of an algorithm in Table 2. In each case, c is a positive constant and n increases without bound. As you can see in table, for large inputs, efficiency matters more than CPU speed. For example, an O (log n) algorithm on a slow machine will outperform an O(n) algorithm on a fast machine

Table 2. Some classes of function and their big-O notation

name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
nlogn time	$4n\log_2 n$, $n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$

slower

3. Problem

The main objective of this assignment is to show the relationship between the running time of implementations with the theoretical asymptotic complexities. You are expected to design an experiment showing that the empirical data follows the corresponding asymptotic growth function. To do so, you will have to consider how to reduce the noise in your measurements and plot the running times to reveal the asymptotic complexities.

You are given 5 different algorithms for different purposes and their pseudocodes that are listed below.

3.1 Comb Sort Algorithm

```
procedure combsort( input : list of sortable items )
 gap = input.size // Initialize gap size
 shrink = 1.3 // Set the gap shrink factor
 sorted = false

while sorted = false
 # Update the gap value for a next comb
 gap = floor(gap / shrink)
 if gap ≤ 1
       gap = 1
       sorted = true // If there are no swaps this pass, we are done
 end if
 # A single "comb" over the input list
 i = 0
 while i + gap < input.size // See Shell sort for a similar idea
       if input[i] > input[i+gap]
       swap(input[i], input[i+gap])
       sorted = false
       // If this assignment never happens within the loop,
       // then there have been no swaps and the list is sorted.
       end if
       i = i + 1
    end while
  end while
end procedure
```

3.2 Gnome Sort Algorithm

```
procedure gnomeSort(a[]):
    pos := 0
    while pos < length(a):
        if (pos == 0 or a[pos] >= a[pos-1]):
            pos := pos + 1
        else:
            swap a[pos] and a[pos-1]
            pos := pos - 1
```

3.3 Shaker Sort Algorithm

```
procedure ShakerSort(A : list of sortable items) is
    do
        swapped := false
        for each i in 0 to length(A) - 2 do:
            if A[i] > A[i + 1] then // test whether the two elements are
in the wrong order
                swap(A[i], A[i + 1]) // let the two elements change
places
                swapped := true
            end if
        end for
        if not swapped then
            // we can exit the outer loop here if no swaps occurred.
            break do-while loop
        end if
        swapped := false
        for each i in length(A) - 2 to 0 do:
            if A[i] > A[i + 1] then
                swap(A[i], A[i + 1])
                swapped := true
            end if
        end for
    while swapped // if no elements have been swapped, then the list is
sorted
end procedure
```

3.4 Stooge Sort Algorithm

```
function stoogesort(array L, i = 0, j = length(L)-1){
        // If the leftmost element is larger than the rightmost element
   if L[i]>L[j] then
        // Swap the leftmost element and the rightmost element
    L[i] ↔ L[j]
        // If there are at least 3 elements in the array
    if (j - i + 1) > 2 then
        t = floor((j - i + 1) / 3)
        stoogesort(L, i  , j-t)  // Sort the first 2/3 of the array
        stoogesort(L, i+t, j)    // Sort the last 2/3 of the array
        stoogesort(L, i  , j-t)  // Sort the first 2/3 of the array again
     return L
 }
```

3.5 Bitonic Sort Algorithm

```
# The sequence to be sorted starts at index position low,
# the parameter cnt is the number of elements to be sorted.
  procedure compAndSwap(a, i, j, dire):
      if (dire==1 and a[i] > a[j]) or (dire==0 and a[i] < a[j]):
            a[i],a[j] = a[j],a[i]
      end if
  end procedure
  procedure bitonicMerge(a, low, cnt, dire):
```

```
      if cnt > 1:
        k = cnt/2
        for i in range(low , low+k):
            compAndSwap(a, i, i+k, dire)
        bitonicMerge(a, low, k, dire)
        bitonicMerge(a, low+k, k, dire)
      end if
  end procedure
procedure bitonicSort(a, low, cnt, dire):
      if cnt > 1:
            k = cnt/2
            bitonicSort(a, low, k, 1)
            bitonicSort(a, low+k, k, 0)
            bitonicMerge(a, low, cnt, dire)
      end if
  end procedure
# Caller of bitonicSort for sorting an array of length N in ASCENDING order
  procedure sort(a,N, up):
      bitonicSort(a,0, N, up)
  end procedure
```

You must execute this algorithm on randomly generated integer numbers. For this purpose, you must carry out the following main steps;

1. Generate n object which include at least two attributes (type of one attribute must be integer to use for sorting operations and you are free to create types of other attributes. Assign value for each attributes randomly)
2. Run the algorithms on randomly generated integer attribute to sort objects (run each of the algorithms with the same input)
3. Record the execution times.
4. Carry out steps 1-3 for different input size (n) as shown in Table 3.
5. You should plot a graph that will show the relation between n and the execution time.

Construct a table as shown in Table 3 for step 3 which record execution time for each algorithms.

For the algorithms given below you may have different **average-case** and **worst-case** complexities. Design an experiment to show both the **average-case** and the **worst-case** complexities of this algorithm, by generating the inputs accordingly. In addition, you are expected to determine related inputs to observe **stability** of sorting algorithm.

Table 3. Example Table

Algorithms/n	100	300	500	700	1100	1300	1500	1700	1900	2100	2300	2500
Comb Sort												
Gnome Sort												
Shaker Sort												
Stooge Sort												
Bitonic Sort												

4. Report

- You should clearly express the idea behind this assignment.
- In your report, you should plot a graph showing the relation between the input size and execution times. You are expected to reveal the Asymptotic complexity of the algorithm with your experiments. For example, for an algorithm with an average case complexity of O(n), you should be able to get the execution times in roughly a line. Some common growth functions in algorithm analysis is shown Figure 1. You can plot the execution times for each algorithm comparatively in a single figure or separately in different figure.
- You should demonstrate the results with plots, discuss your random number generation strategies for each. Report how you designed your experiments clearly.
- In addition, you are expected to determine related inputs to observe stability of sorting algorithm and interpret them as which sorting algorithms are stable, what is advantages and disadvantages of stability and discuss how you obtained observations on this subject as a result of the experiments.
- You must argue the algorithms in terms of time complexity and memory requirements with your own sentence.
- You must show your analysis in detail for each algorithm in your report.
- Your report has to include table as shown in Table 3, figures as shown in Figure 1 and your analysis for the experiment. You can discuss all sorting algorithms on one figure, or you can create different figures for each sorting algorithm.

Hint: Executing your algorithm once for an input size might be very noisy leading to plots with lots of spikes. Consider repeating the experiments with same size input and averaging the results.

Hint: You will require enough number of executions to have a clear plot, consider different number of input sizes to experiment with. If your input sizes are too small, the difference might be too small.

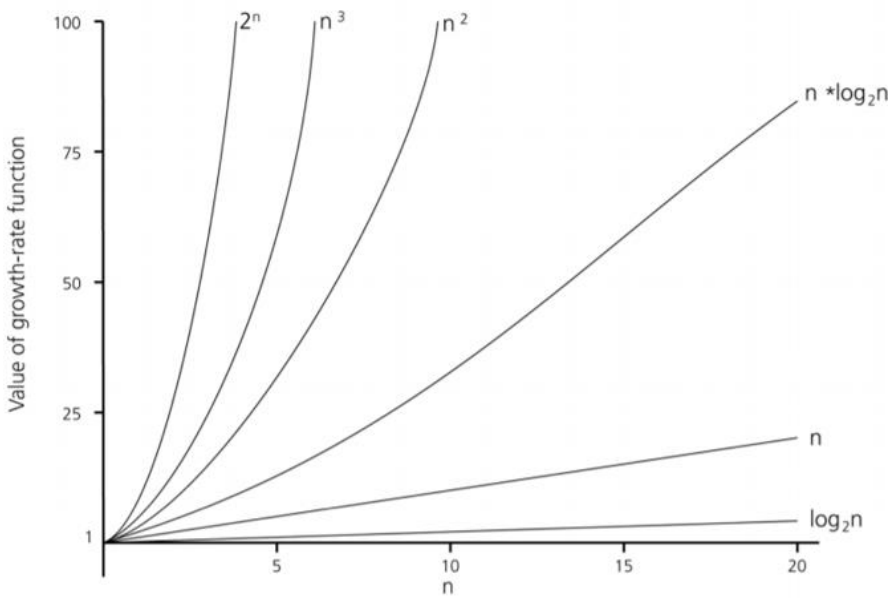

Figure 1. Some common growth functions in algorithm analysis

5. Grading and Important Notes

Table 4. Grading Policy

Task	Point
Submitted	1
Coding standards	9
Clean code (Implementing each sorting algorithm is 6 point)	30 (5x6=30)
Report (Discussing each sorting algorithm is 12 point)	60 (5x12=60)
Total	**100**

- Your work will be graded over a maximum of 100 points according to the grading policy stated in Table 4.
- Do not miss the submission deadline.
- Save all your work until the assignment is graded.
- Regardless of the length, use UNDERSTANDABLE names to your variables, classes and Functions. The names of classes, attributes and methods should obey Java naming convention. This expectation will be graded as Coding standards.
- Source code readability is a great of importance for us. Thus, write READABLE SOURCE CODE, COMMENTS and clear MAIN function. This expectation will be graded as clean code.
- You will submit your as following file hierarchy:

<student id.zip>
→ report <DIR>
→ report.pdf <FILE>
→ src <DIR>
→ Main.java(Required)
→ *.java(optional)