FUNDAMENTAL ALGORITHMS

Laboratory Assignments Guide

Tudor Mureşan

Rodica Potolea

Camelia Lemnaru

U.T. PRESS
Cluj-Napoca, 2018
ISBN 978-606-737-293-9

Table of Contents

Introduction

General Requirements

For this laboratory you are asked to implement and analyze (empirically) the correctness and efficiency of a number of algorithms. The programming language(s) are C/C++, the style is procedural. You are not allowed to use any additional libraries, containing existing implementations of necessary data structures or algorithms. The programming environment installed in the lab is Visual Studio, but you may use any environment you feel comfortable with on your own notebook. You are encouraged to use the pseudo-code provided at the course/seminar to implement the algorithms, and employ any information from your course/seminar notes. Also, keep a copy of the *Introduction to Algorithms* book by Thomas Cormen et al. close, since you will probably need to consult it often while trying to solve the assignments. For language-related information, use the *MSDN* library or *Google* search. We will be using moodle for this lab, so all the materials you need will be posted there. You will have to enroll in the current year course in the first lab session (the teaching assistant will provide the key and any other necessary information).

Assignments and Grading

Each assignment will be graded individually. At the end of the laboratory sessions (Week 14 probably), you will be given a quiz test consisting of a small number of questions from the assignments. The final mark at the laboratory is based on the average of the individual assignment grades. Starting from this average, a penalty can be applied for constant late delivery (check the delivery extensions sub-section). How well you do on the final quiz may influence the final laboratory grade by 1 point (+/-).

Assignment deliverables

- **Pseudo-code** on paper (your implementation should start from the pseudo-code)
- C/C++ procedural **implementation**, code should be commented
- The source file should have a header (block comments) containing:
    - Personal identification information (name, group)
    - Problem specification
    - Start and end date
    - Special evaluation requirements if necessary
    - **Conclusions and personal interpretations, running time, best/worst cases, memory, etc**
- **Running example(s)**: You must give a running example of your algorithm/procedures, on reasonably small-sized input(s)

- **Analysis**: generation and interpretation of output charts/tables, according to the individual assignment requirements

Assignment delivery

The majority of assignments are 1-week assignments. At the end of the session, you have to present your *code+analysis+interpretation* to the teaching assistant. If your assignment is not complete, you present what you have so far. You are allowed to continue working at your assignment and deliver it later (check the section on **delivery extensions**). When you consider that your assignment is complete, upload the C/C++ source file using the form provided on the moodle.

If you have more than one source file for your algorithm (excluding the additional .h files you will be provided as supporting material during the lab sessions), upload a .zip archive containing all your source files. Do not include specific environment project files! Do not use any other archiving method (only .zip is allowed).

Assignment grading

Each assignment is graded incrementally – the requirements are to be completed incrementally. Each assignment has four grading thresholds: grades 5, 7, 9 and 10. The requirements for each grade are assignment specific and are provided in the assignment description.

Delivery extensions

The majority of assignments are 1-week assignments. If, however, you do not finish by the end of the laboratory session, assignments may have the following extensions:

- *Extension_1*: the assignment can be delivered at the beginning of the following laboratory session, at the beginning of the session
- *Extension_2*: For maximum grade 8, some assignments can be delivered at the beginning of the second following the original deadline (not all assignments have Extension_2)

Note:

! Constant delivery of assignments (>50%) at Extension_1 may be penalized with 1 point from the final mark.

!! Constant delivery of assignments (>50%) at Extension_2 may be penalized with 1 point at the final mark.

Missing laboratory sessions

All assignments have to be solved (to a certain extent). If, for some reason, you miss **one** laboratory session, you may solve the corresponding assignment at home and present it at the beginning of the following session. You must attend the laboratory with your group! (i.e. you are not allowed to attend a laboratory session with another teaching assistant). If you want to switch to another lab session, you must do it by the second week of school, with the approval of the course instructor, the lab instructor of the session you want to switch from, and the lab instructor of the session you want to switch to.
If you miss more than 1 laboratory session (but not more than 3!), you will be asked to solve other assignments (and not the ones you have missed).

Coding Guidelines (the basics)

http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html
http://www.cs.swarthmore.edu/~newhall/unixhelp/c_codestyle.html
http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Evaluation Tricks – how to evaluate the time complexity of your algorithms

For most assignments, you are asked to evaluate (perhaps comparatively), the running time of 1/several algorithms in the average case (and, for some assignments, in the best and worst case). You have to identify additional memory requirements also, and write such observations in the header of your source code!
However, before you pass on to the evaluation, make sure your algorithms work correctly!!!

Therefore, for each analysis case, you have to generate the according input data, of varying size (the superior limit in usually 10.000, the increment 100 – but this may differ for some assignments). For the average case, you have to repeat the measurement, at least 5 times, and report the average of your measurements. When you are asked to compare algorithms, make sure you test the algorithms in the same conditions! (same random input data for the average case, or the correct corresponding data for the best/worst cases).

What do you measure? When evaluating the running time, for each run of your algorithm, you have to count the number of operations performed by your algorithms, i.e. the number of assignments and the number of comparisons performed on the input structure, or on corresponding auxiliary variables (i.e. for sorting, no operations on indices or flags should be counted!!!). These measurements are saved, either in a file (usually .csv), or by using a library which we provide, the Profiler (which is intended to support the algorithm input data and chart generation process)

How do you analyze? Use the recorded measurement data to generate required charts/tables, which will help you interpret the results (use either a tool such as MS Excel, or the Profiler, to so this). You may need, in some cases, to limit the *Ox* or *Oy* values on your charts, for better visualization of certain characteristics (overlapping, behavior on small sized inputs, asymptotic behavior, etc.)

Hint: feel free to be aggressively inquisitive with your charts, try to get as much out of them as you can (of course, do not try to see something that is not there). You will be rewarded accordingly for your efforts.

Write all your observations regarding the analysis in the analysis part of the Header section in your main source code file (provided you have more than 1 source files for your assignment).

Example: *Compare insertion sort and selection sort in the average case, for input sequences of sizes 100->10.000 (using an increment of 100).*

Thus, you have to generate a random sequence for each intermediate size (100, 200, …, 10000). Then, apply both sorting methods (on the same sequence, independently) and count, for each, the number of assignments and the number of comparisons. Since we are dealing with the average case, repeat this 5 times, and write the average of the measurements in a .csv file, as you are shown in the *Introductory Laboratory session*. Then, using the results in the .csv file, generate three Excel scatter plots (you are shown how to do that in the *Introductory Laboratory session* as well): 1 comparing the number of comparisons used by the two methods, 1 for the number of assignments, and 1 for the overall number of operations performed by the two methods.

Don't forget you also have to interpret the three charts (order of complexity, which method is better for small/large sized inputs, etc.) – and add these observations to the header of your source code file.

Assignment No. 0: Introductory Session

First off, make sure you have read the **guide to the laboratory sessions** (available on the moodle course page). In this introductory session, you will get used to working with *Visual Studio* by writing a more complex Hello World C/C++ application. Also, you will see how to generate the data to evaluate your algorithms and how to generate the required charts (either with *MS Excel* or by using a framework written in C++). This assignment is not graded.

Introduction to *Visual C++*

To create a new C/C++ project, using the wizard:

- *File – New – Project... – Win32 –* ***Win32 Console Application*** *– Name: HelloWorld – Location:* ….choose… *– OK – Next –* ***Empty project*** *– Finish*;
- *Solution Explorer – Source Files – Add – New Item – C++ File (.cpp) – Name: HelloWorld – Add*;
- Include stdio.h and conio.h, write your main function in which you print „Hello, world!", on the screen (use getch()) to keep the console from closing until you hit a key.
- Compile and run your application

Working with files

Now, to extend your application, do the following (use Google or MSDN library for help, or ask the teaching assistant):

- Declare an array of integers of size MAX_SIZE – constant defined by you
- Read a sequence of *n* numbers from the keyboard, and store them in the array
- Print the *n* numbers in the array
- Create and open a file, write the numbers from the array in the file, and close the file (check the file to see it worked)
- Now open the previous file, read the contents and print them on the screen (don't forget to close the file at the end)

Generating test cases for the algorithms (best, worst, average)

In order to test your algorithms, you have to generate a series of input sequences, such as: a sorted array of integers (of given size), a random array of integers (of given size), etc.

Since generating an ordered sequence is straightforward, let us focus on generating a random sequence of integers. We suggest two alternatives:

1. Using the Profiler Framework (available on the moodle course page)
2. Using the random number generator available in C/C++

1. How to use the **Profiler Framework**: check the profiler guide on the moodle course page
2. How to use the random number generator available in C/C++:
   - Read about rand(,)srand()functions and RAND_MAX constant:
     - http://www.cplusplus.com/reference/cstdlib/rand/
     - http://www.cplusplus.com/reference/cstdlib/srand/

- http://www.cplusplus.com/reference/cstdlib/RAND_MAX/

- Write a sequence of code/function which:
  - Generates *n* random numbers, using the rand() function alone, stores them in an array, then prints them on the screen; what happens when you run your program the second time?
  - Change your sequence of code such that the sequence of *n* random numbers differs between runs

Exercises:

1. Write a function which generates an array of *n* random integers between *Low* and *High*, and returns the array; print the contents of the array in a file
2. Write a function which generates a sorted array of random integers; print the contents of the array in a file

Generating charts for the analysis of algorithms

Again, you have two options for generating the evaluation charts:

1. Using the **Profiler Framework**, same as before: check the profiler guide on the moodle course page
2. Use *MS Excel*
3.

1. How to use the **Profiler Framework**: check the profiler guide on the moodle course page
2. How to use *MS Excel*:
   - First, from your program, you have to save your analysis data in a *.csv* (**c**omma-**s**eparated **v**alues) file. You are free to use your own format for the file. However, it is a good idea to use the following format:

     Size_of_problem, No_assignments, No_comparisons, No_assignments+No_comparisons

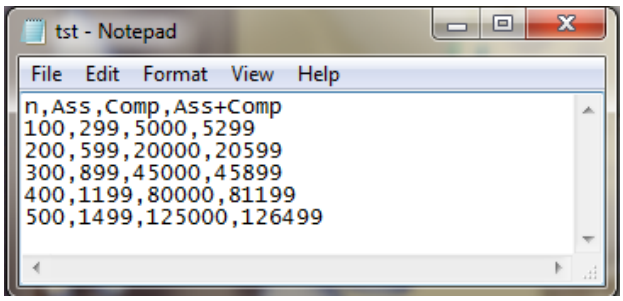


Figure 0.1 - .csv data

The figure above represents an example of how a *.csv* file might look like for one analysis case – input size 100 to 500. You can choose to use the same file for all cases

of an algorithm (best, average, and worst). How many columns will your *.csv* require then?

- Importing data to *MS Excel* (version 2010): if your data is properly formatted and the extension is .csv, *Excel* should recognize it and open it correctly:

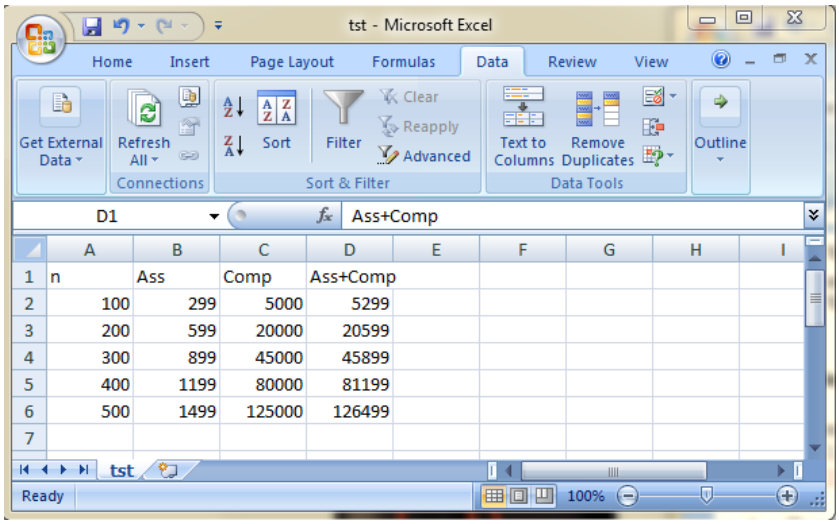


Figure 0.2 - .csv data imported in MS Excel

However, if *Excel* places your values in the same column (probably you used a different column separator than the one set in *Excel*), use the *Data->Text to Columns* wizard to correct this (ask the teaching assistant for help). Also, you may import your data in Excel by using the *Data->Get External Data* wizard (again, ask the teaching assistant)

- Building the chart: select the data rows and columns; then go to *Insert->Charts->Scatter* and select the second type (connected points). For the above data, what you get should look like Figure 3.
  Note that the number of assignments, although linear, looks constant when placed on the same chart with the number of comparisons or with the sum (both quadratic). As a rule, whenever one curve cannot be visualized correctly because of the difference in growth rate with the other curves, it is best to place it also on a separate chart, by itself (try to do this by yourself).

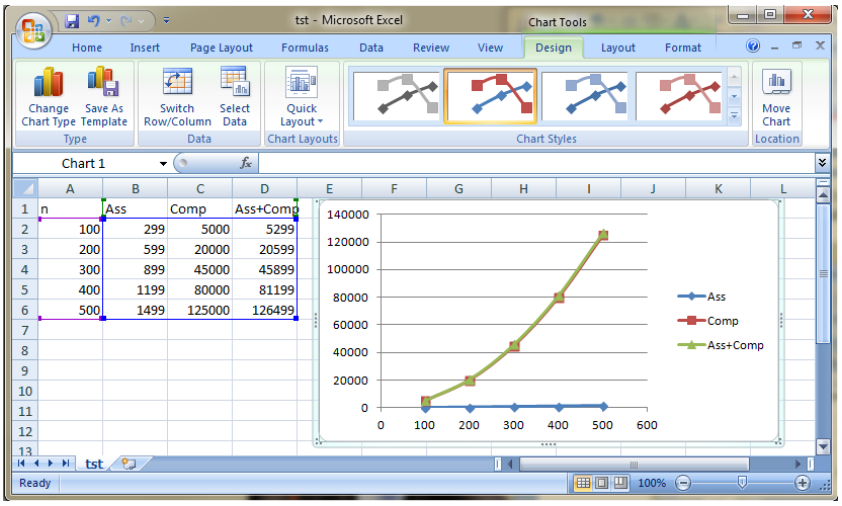


Figure 0.3 – Scatter plots in MS Excel

- Additionally, you can name your chart, label the axes, scale the axes – you may need to perform scaling when comparing algorithms – on small inputs, for example. Try to identify how these operations are performed in *Excel* (ask the teaching assistant for help whenever you need guidance).

- ! Don't forget you also have to interpret the charts, and place your comments in comments at the beginning of your source code file

Exercise: Write a C/C++ program which writes in a file, for *n* starting from 100 to 10.000 (with a 100 increment), the following values (for each value of *n* use a separate line):

*n, log(n), n*log(n), n^2, n^3, 2^n*

Use the values in the file to build scatter plots for these functions, either by using *MS Excel* or the *Profiler Framework*.

Below, you have several charts which exemplify the results you should observe for this exercise. All charts have been generated using the Profiler. The initial group of charts have been generated for small values of n (1 to 30):

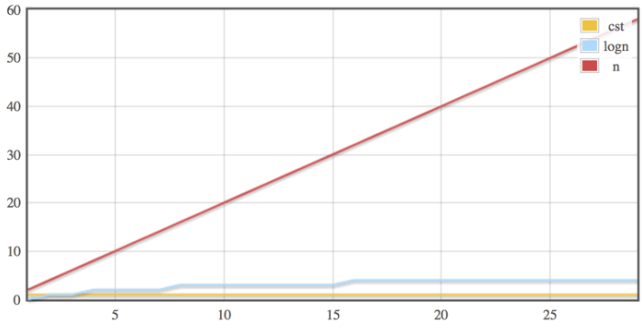


Figure 0.4: Comparison of constant, logarithmic and linear growth, very small n

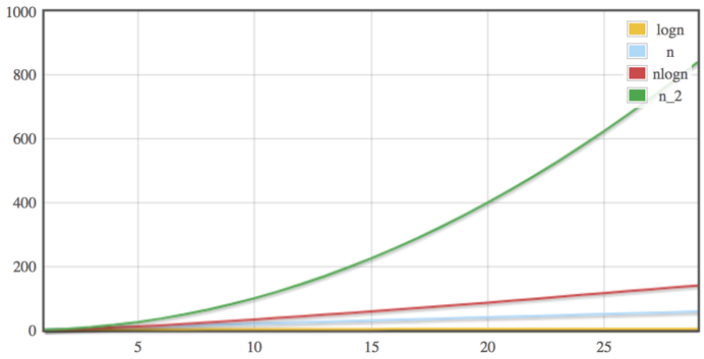


Figure 0.5: Comparison of logarithmic, linear, linearithmic (nlogn) and quadratic ($O(n^2)$) growth, very small n

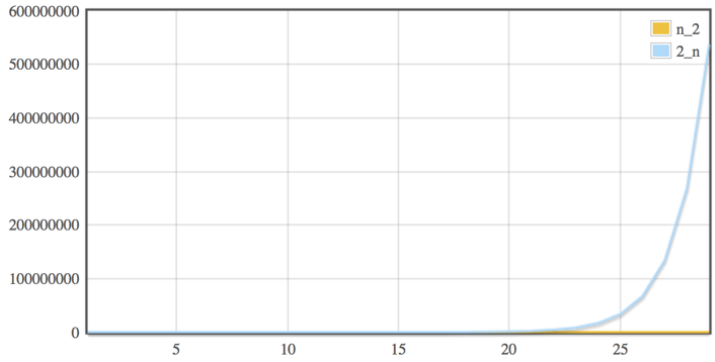


Figure 0.6: Comparison of quadratic ($O(n^2)$) and exponential growth, very small n

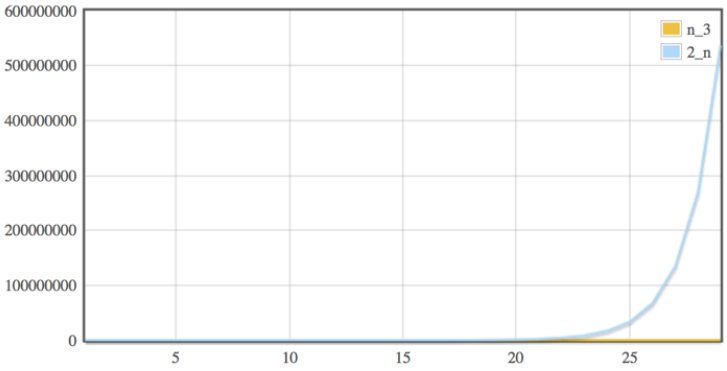


Figure 0.7: Comparison of polynomial ($O(n^3)$) and exponential growth, very small n

What can you observe for the charts in figures 0.4-0.7? Perhaps the most striking observation is the fact that, even for small sized inputs, a polynomial growth function (be it n^2, n^3) seems constant in comparison to an exponential function. Also, the logarithmic function grows much slower than the linear function, while *O(nlogn)* is closer to the linear curve than it is to a quadratic one. You will see, in the following assignments, that an *O(nlogn)* growth can be easily

mistaken as a linear growth. You can check, however, the numbers (divide the entire range of values to *n*, or to *nlogn*, respectively, to see which "fits" better).

The following group of charts consider a slightly larger value of n (1 to 100):

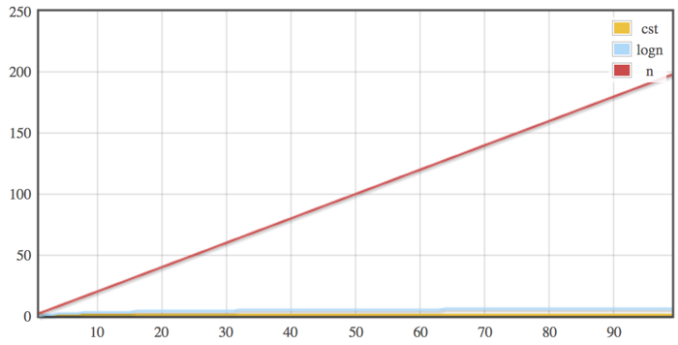


Figure 0.8: Comparison of constant, logarithmic and linear growth, n<100

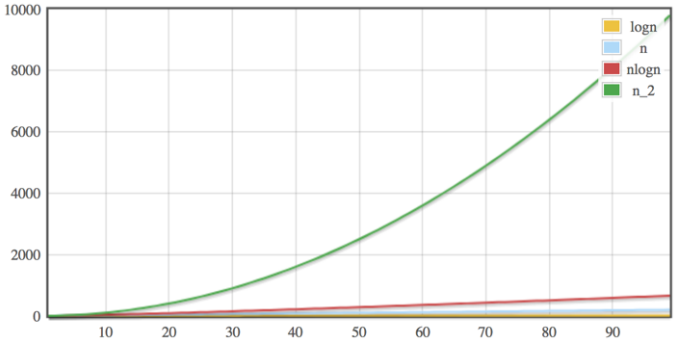


Figure 0.9: Comparison of logarithmic, linear, linearithmic (nlogn) and quadratic ($O(n^2)$) growth, n<100

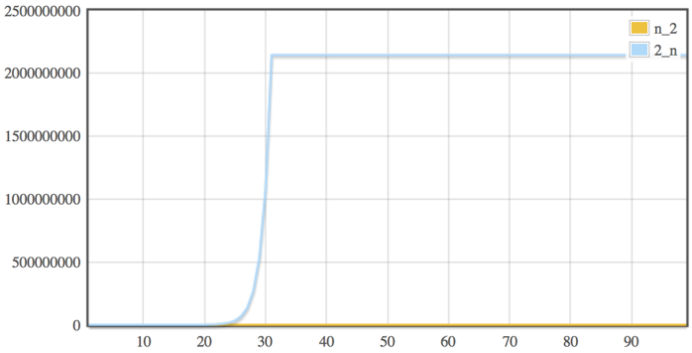


Figure 0.10: Comparison of quadratic ($O(n^2)$) and exponential growth, n<100

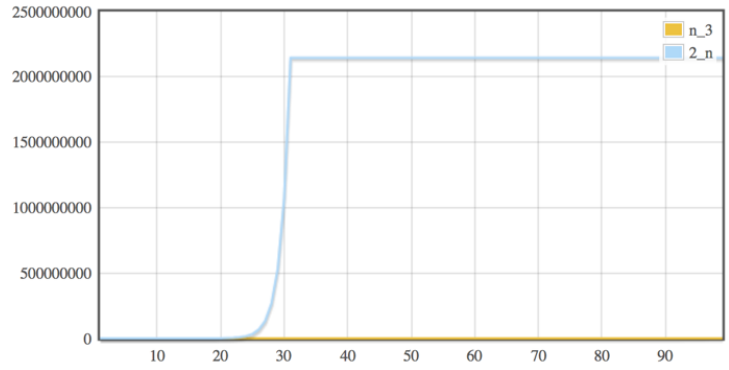


Figure 0.11: Comparison of polynomial ($O(n^3)$) and exponential growth, n<100

For the charts in figures 0.8-0.11, we can observe, first of all, that the values for the exponential function produce an overflow at values 2^{31}. As expected, the growth of any polynomial function is undetectable at this size either.
Moreover, the difference between logarithmic and linear functions, and linearithmic and quadratic functions becomes more significant at this size also. Keep in mind that we are still at relatively small values of n.

In the next group of charts, we further increase n to 1000:

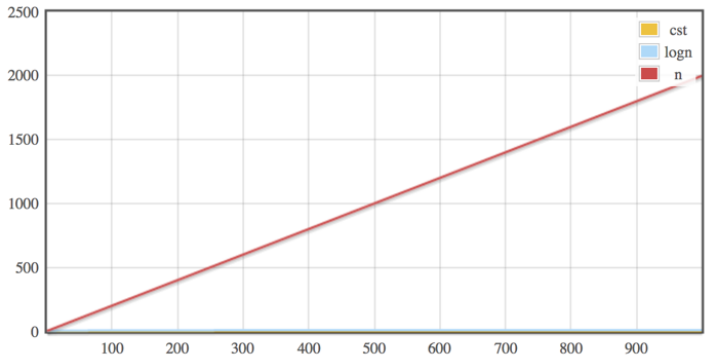


Figure 0.12: Comparison of constant, logarithmic and linear growth, n<1000

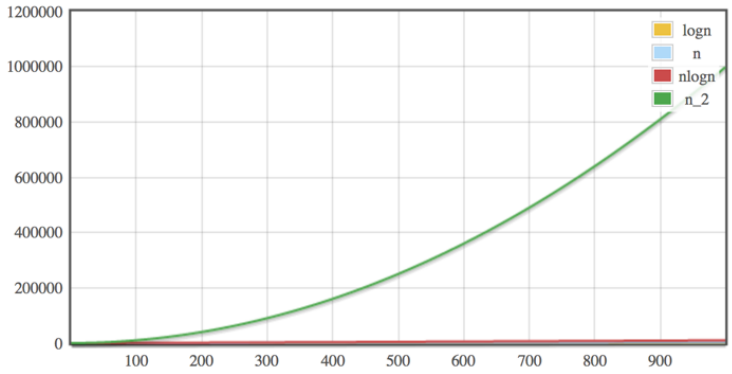


Figure 0.13: Comparison of logarithmic, linear, linearithmic (nlogn) and quadratic ($O(n^2)$) growth, n<1000

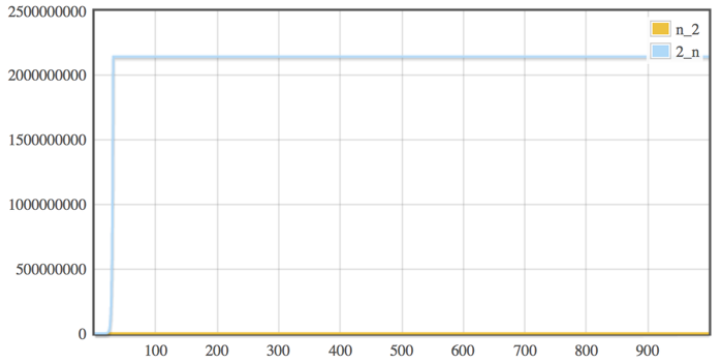


Figure 0.14: Comparison of quadratic ($O(n^2)$) and exponential growth, n<1000

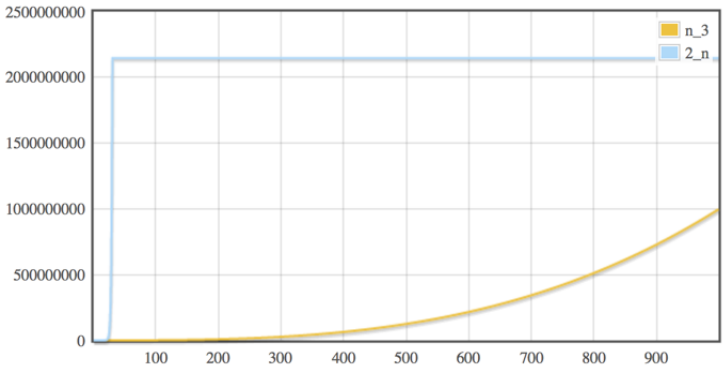


Figure 0.15: Comparison of polynomial ($O(n^3)$) and exponential growth, n<1000

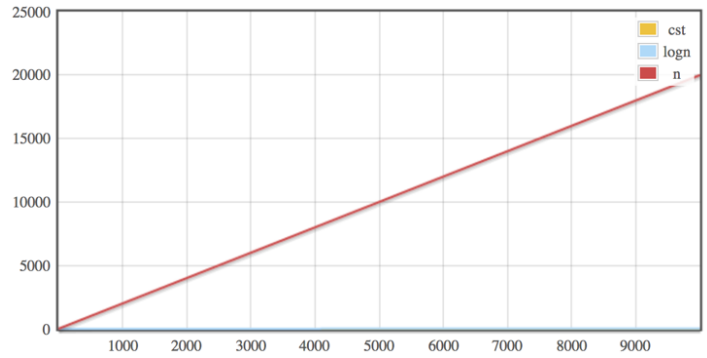


Figure 0.16: Comparison of constant, logarithmic and linear growth, n<10000

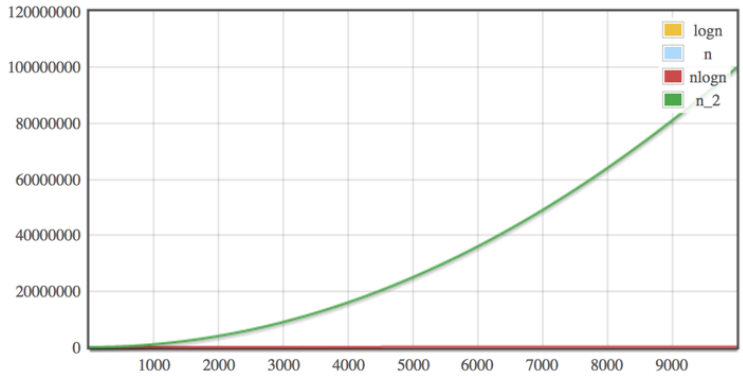


Figure 0.17: Comparison of logarithmic, linear, linearithmic (nlogn) and quadratic ($O(n^2)$) growth, n<10000

As expected, when *n* increases further, the difference between the faster growing functions and the smaller growing functions is accentuated. For *n* going up to 1000, however, the only difference which is still still small enough to consider at this point is that between a linear and a linearithmic growth, which can be observed below, in figure 0.18:

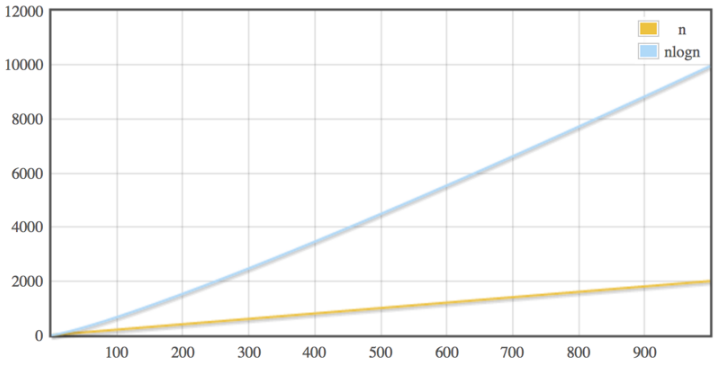


Figure 0.18: Comparison of linear and linearithmic (nlogn) growth, n<1000

We can observe that both curves seem to have the same shape – visually, they both look linear, but with different constants. However, by checking the effectively how the values grow, one can see the difference between the two functions.

The last group of figures show the same charts as before, but for *n* going up to 10000. We have eliminated the exponential growth from this group of charts, since it is evident that they did not differ that much from their versions for n<1000.

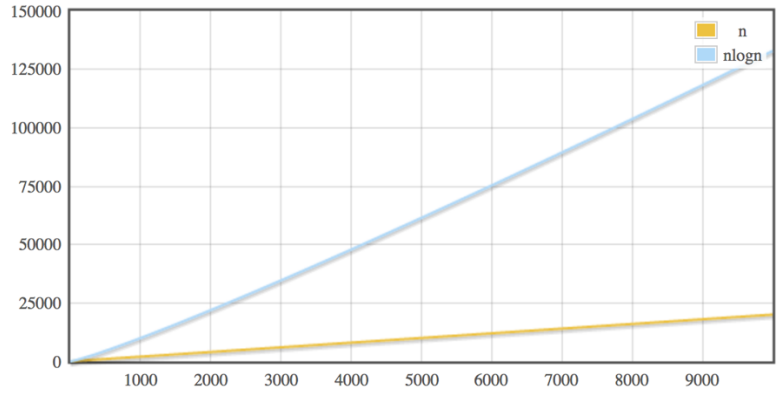


Figure 0.19: Comparison of linear and linearithmic (nlogn) growth, n<10000

Figure 0.19 depicts the comparison between a linear and a linearithmic growth, for n<10000. We can see that the gap between the two seems to increase, however there is still no apparent visual difference. Which means that for these two growths one must always also check the numbers, to make sure which one applies.

The charts presented so far will help you accurately identify the shape of the curves you get for your algorithms, but they were also meant to make you observe the immense difference between different complexity classes.

Perhaps a more concrete example is necessary for this also. Table 0.1 on the next page presents the expected running times for different complexity levels, for an input of varying sizes.

Table 0.1: time taken by algorithms having different complexity classes, for different sizes of the input problem (assume time to execute a constant running time algorithm to be 10^{-7} seconds)[1]

Complexity Class	N = 10	N = 100	N = 1,000	N = 10,000	...	N = 1,000,000
O(1)	$1x10^{-7}$ seconds	$1x10^{-7}$ seconds	$1x10^{-7}$ seconds	$1x10^{-7}$ seconds	...	$1x10^{-7}$ seconds
$O(\log_2 N)$	$3.3x10^{-7}$ seconds	$6.6x10^{-7}$ seconds	$10x10^{-7}$ seconds	$13.3x10^{-7}$ seconds	...	$20x10^{-7}$ seconds
O(N)	$1x10^{-7}$ seconds	$1x10^{-6}$ seconds	$1x10^{-5}$ seconds	$1x10^{-4}$ seconds	...	$1x10^{-2}$ seconds
$O(N\log_2 N)$	$3.3x10^{-7}$ seconds	$6.6x10^{-6}$ seconds	$10x10^{-5}$ seconds	$13.3x10^{-4}$ seconds	...	$20x10^{-3}$ seconds
$O(N^2)$	$1x10^{-6}$ seconds	$1x10^{-4}$ seconds	$1x10^{-2}$ seconds	1 second	...	2.7 hours
$O(N^3)$	$1x10^{-5}$ seconds	$1x10^{-2}$ seconds	10 seconds	2.7 hours	...	$3x10^3$ years
$O(2^N)$	$1x10^{-5}$ seconds	$4x10^{21}$ centuries	Forget about it	Forget about it	...	Forget about it

[1] Inspired from: http://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/aa/

Assignment No. 1: Analysis & Comparison of Direct Sorting Methods

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** 3 direct sorting methods (Bubble Sort, Insertion Sort – using either linear or binary insertion and Selection Sort)

Input: sequence of numbers $< x_1, x_2, \dots, x_n >$
Output: an ordered permutation of the input sequence $< x'_1 \leq x'_2 \leq \cdots \leq x'_n >$

You may find any necessary information and pseudo-code in your Seminar no. 1 notes (Insertion Sort is also presented in the book[2] – *Section 2.1*). Make sure that, for each of the required sorting methods, you select its efficient version (whenever more than one version has been provided to you).

Thresholds

Threshold	Requirements
5	Implement 1 direct sorting method, exemplify correctness and evaluate it (at least in the average case) – at least 1 chart
7	Compare 2 direct sorting methods (best, average and worst case), i.e. implementation, exemplify correctness and analysis (charts)
9	Compare 3 direct sorting methods (best, average and worst case), i.e. implementation, exemplify correctness and analysis (charts)
10	Discussion, interpretations, efficiency, compare, stability

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to prove your algorithm(s) work, so you should also prepare a demo on a small-sized input (which may be hard-coded in your main function).

1. You are required to compare the three sorting algorithms, in the **best**, **average** and **worst** cases. Remember that for the **average** case you have to repeat the measurements m times (m=5 should suffice) and report their average; also for the **average** case, make sure you always use the **same** input sequence for all three sorting methods – to make the comparison fair; make sure you know how to generate the **best/worst** case input sequences for all three methods.

2. This is how the analysis should be performed for a sorting method, in any of the three cases (**best**, **average** and **worst**):
   - vary the dimension of the input array (*n*) between [100…10000], with an increment of maximum 500 (we suggest 100);
   - for each dimension, generate the appropriate input sequence for the sorting method; run the sorting method counting the operations (i.e. number of assignments, number of comparisons and their sum).

   ! Only the assignments („=”) and comparisons („<”,”==”,”>”,”!=”) which are performed on the input structure and its corresponding auxiliary variables matter.

3. For each analysis case (**best**, **average** and **worst**), generate charts which compare the three methods; use different charts for the number of comparisons, number of assignments and total number of operations. If one of the curves cannot be visualized correctly because the others have a larger growth rate (e.g. a linear function might seem constant when placed on the same chart with a quadratic function), place that curve on a separate chart as well. Name your charts and the curves on each chart appropriately.

4. Interpret the charts and write your observations in the header (block comments) section at the beginning of your main .cpp file.

The charts below exemplify the form of the growth curves you should get.
! The absolute values might differ, the shape counts.

Average Case Charts

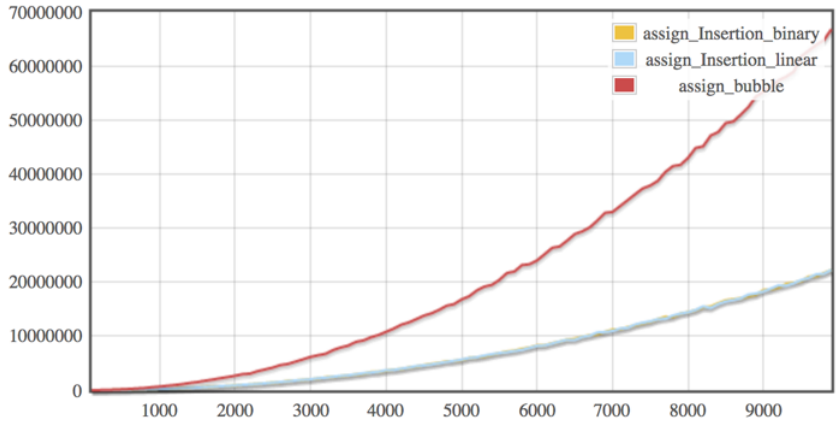


Figure 1.1: Average case assignments, binary insertion, linear insertion and bubble sort

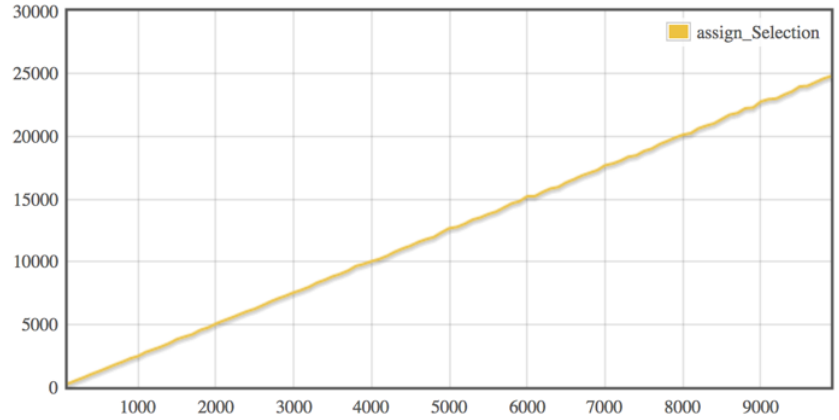


Figure 1.2: Average case assignments, selection sort

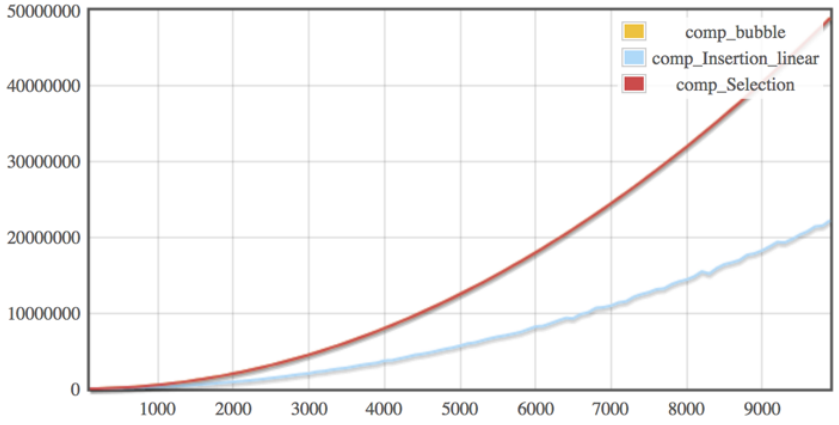


Figure 1.3: Average case comparisons, selection, linear insertion and bubble sort

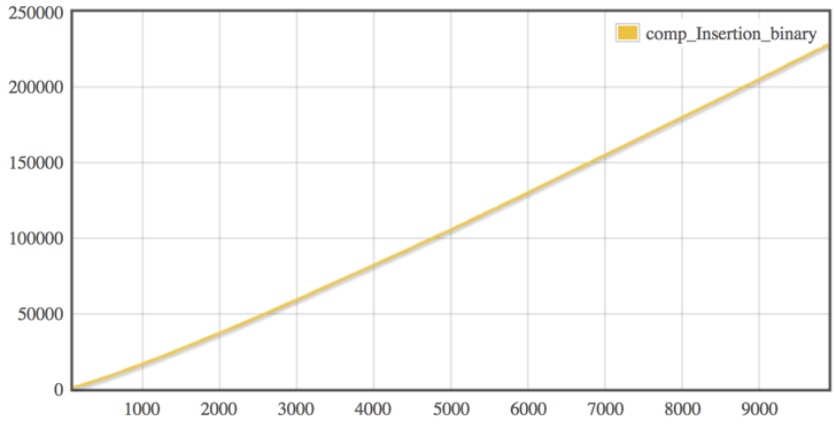


Figure 1.4: Average case comparisons binary insertion sort

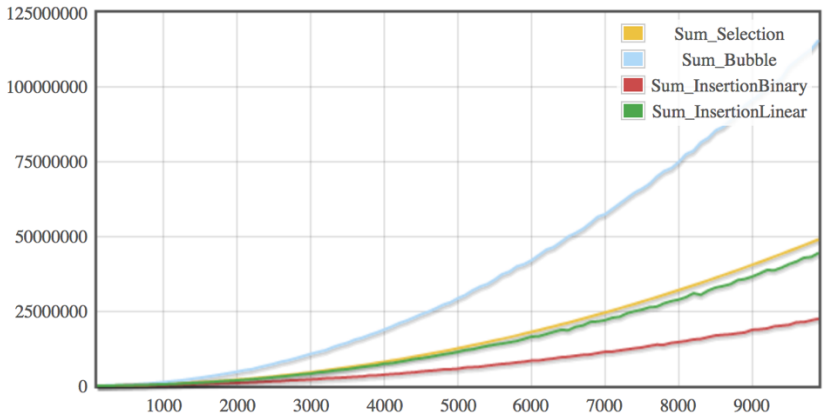


Figure 1.5: Average total number of operations, all sorts

The charts in figures 1.1 – 1.5 present the expected shapes for the complexity measurements obtained in the *average analysis case*, by several direct sorting methods. You should try to compare the shapes of the curves obtained by your implementations to the shapes depicted here.

Best Case Charts

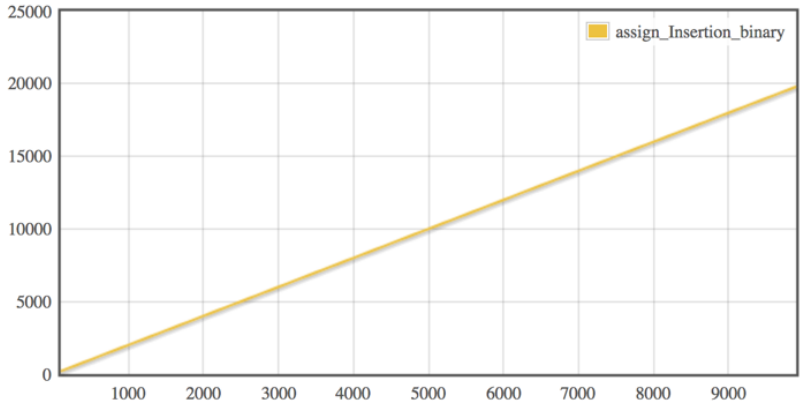


Figure 1.6: Best case assignments, binary insertion sort

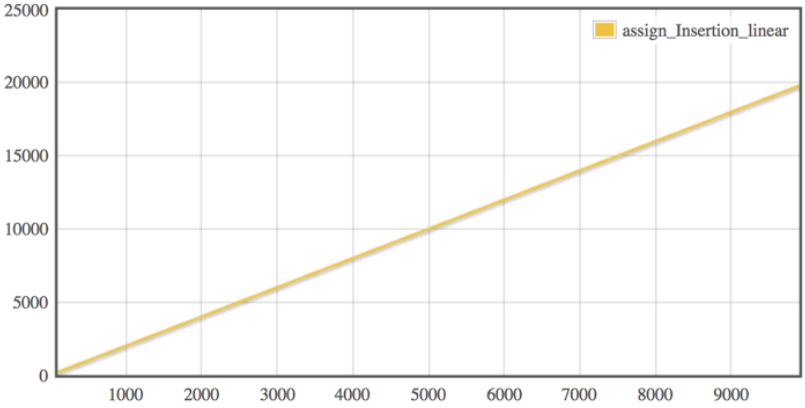


Figure 1.7: Best case assignments, linear insertion sort

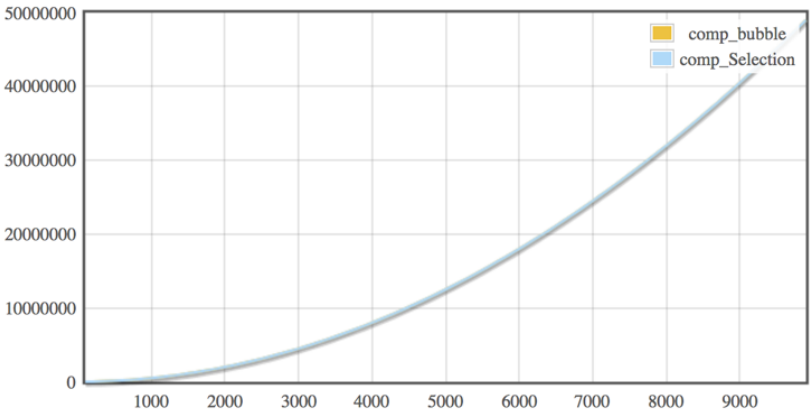


Figure 1.8: Best case comparisons, selection and bubble sort

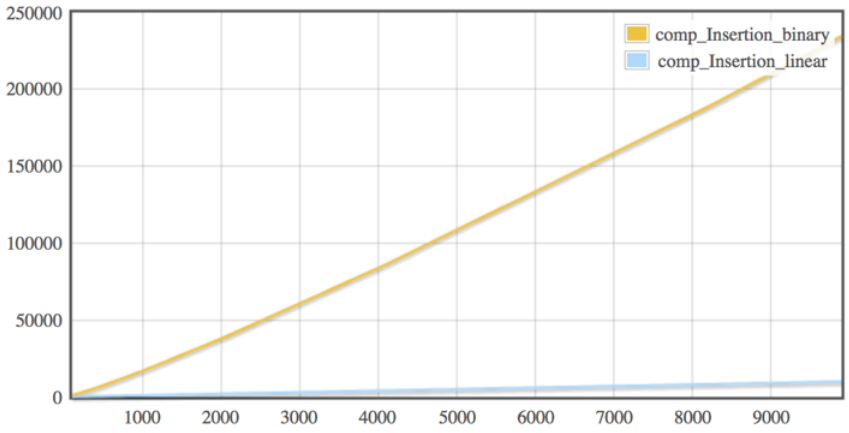


Figure 1.9: Best case comparisons binary and linear insertion sort

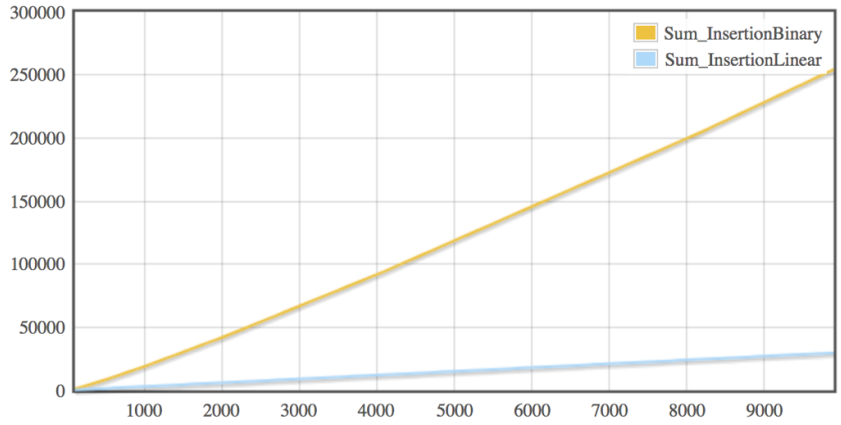


Figure 1.10: Best case total operations, binary and linear insertion sort

For the *best case analysis*, (figures 1.6-1.10) you should get 0 assignments for the bubble sort and selection sort algorithms. Therefore, the growth curves for the total number of operations for these algorithms coincide with the ones in figure 1.8. Also, it is possible to implement a slightly improved version of bubble sort, which makes a linear number of comparisons in the best case. What is the shape for the number of comparisons of binary insertion in this analysis case?

Worst Case Charts

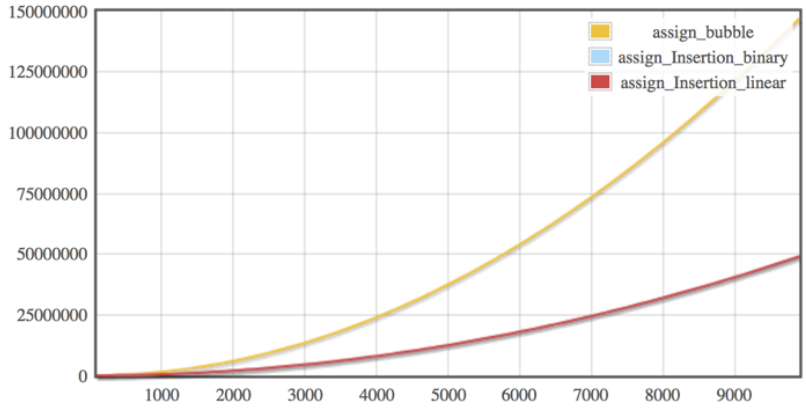


Figure 1.11: Worst case assignments, binary insertion, linear insertion and bubble sort

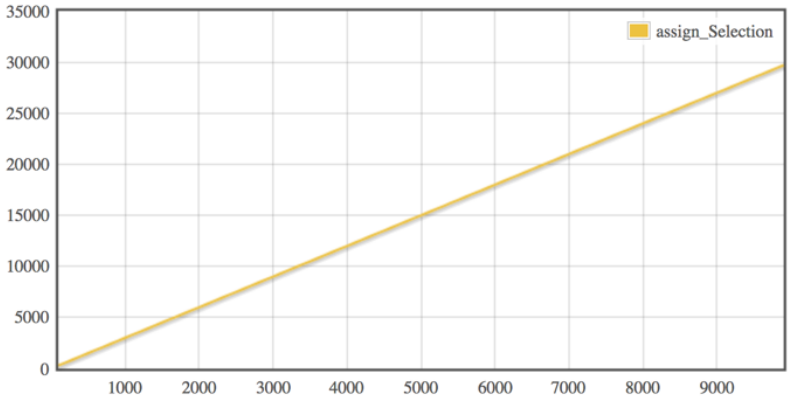


Figure 1.12: Worst case assignments, selection sort

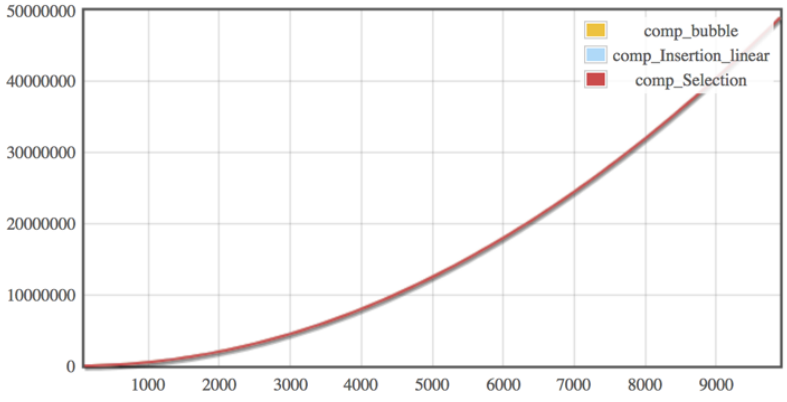


Figure 1.13: Worst case comparisons, selection, linear insertion and bubble sort

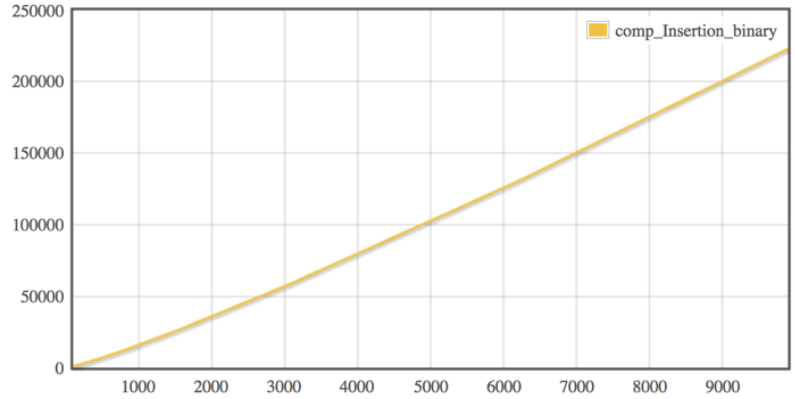


Figure 1.14: Worst case comparisons, binary insertion sort

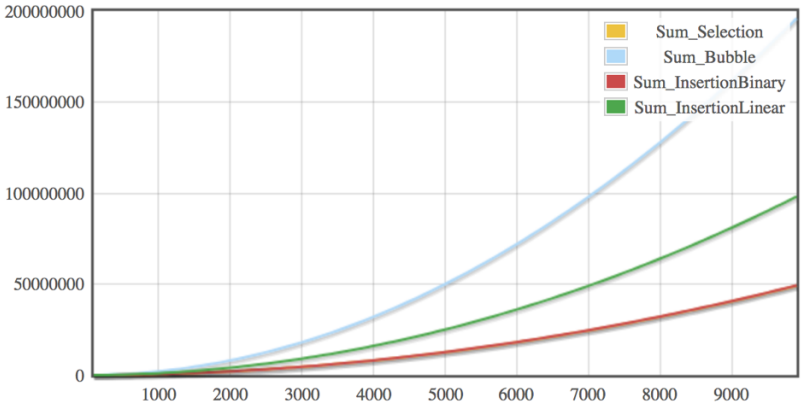


Figure 1.15: Worst case total number of operations, all sorts

Assignment No. 2: Analysis & Comparison of Bottom-up and Top-down Build Heap Approaches

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** two methods for building a heap, namely the *bottom-up* and the *top-down* strategies. Additionally, you have to implement heapsort.

You may find any necessary information and pseudo-code in your course notes, or in the book:

- Bottom-up: *section 6.3* (*Building a heap*)
- Heapsort: *section 6.4* (*The Heapsort algorithm*)
- Top-down: *section 6.5* (*Priority queues*) and *problem 6-1* (*Building a heap using insertion*)

Thresholds

Threshold	Requirements
5	Implement and exemplify correctness of bottom-up build heap procedure
6	Implement and exemplify correctness of heapsort
7	Implement and exemplify correctness of top-down build heap procedure
9	Comparative analysis of the two build heap methods, in the average case
10	Interpretations, advantages/disadvantages of each approach

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to prove your algorithm(s) work on a small-sized input.

1. You are required to compare the two build heap procedures in the **average** case. Remember that for the **average** case you have to repeat the measurements m times (m=5)

and report their average; also for the **average** case, make sure you always use the **same** input sequence for the two methods – to make the comparison fair.

2. This is how the analysis should be performed:
   - vary the dimension of the input array (n) between [100…10000], with an increment of maximum 500 (we suggest 100).
   - for each dimension, generate the appropriate input sequence for the method; run the method, counting the operations (assignments and comparisons, may be counted together for this assignment).

   ! Only the assignments and comparisons performed on the input structure and its corresponding auxiliary variables matter.

3. Generate a chart which compares the two methods under the total number of operations, in the average case. If one of the curves cannot be visualized correctly because the other has a larger growth rate, place that curve on a separate chart as well. Name your chart and the curves on it appropriately.

4. Interpret the chart and write your observations in the header (block comments) section at the beginning of your main .cpp file.

5. Only the correctness of heapsort is demonstrated, the analysis is not necessary.

6. (*extra – for extra credit*) Try to compare the two build heap procedures in the **worst** case. What do you observe?

Example Charts

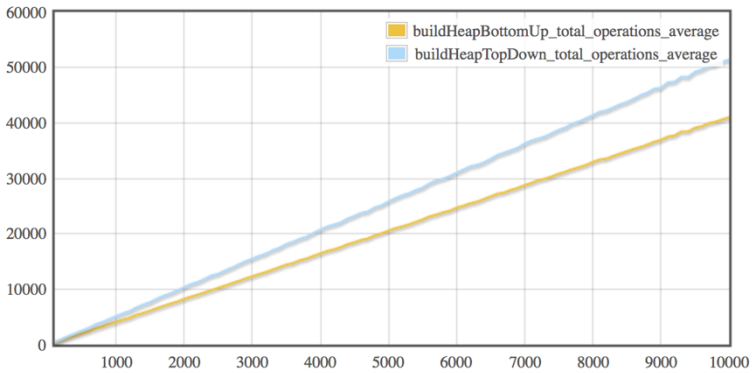


Figure 2.1: Total number of operations for the two build heap strategies, average case analysis

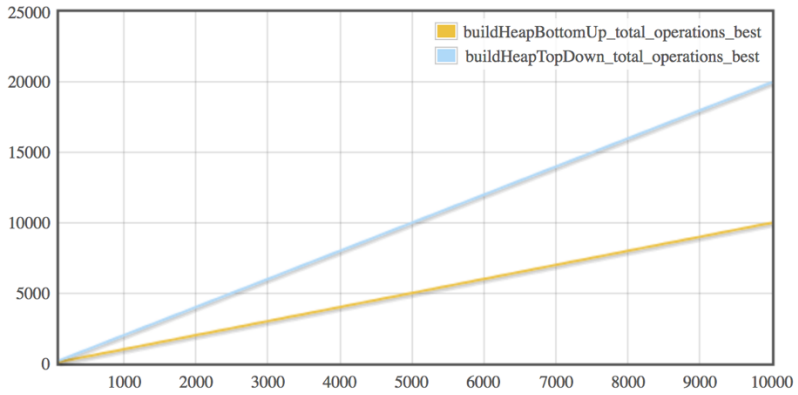


Figure 2.2: Total number of operations for the two build heap strategies, best case analysis

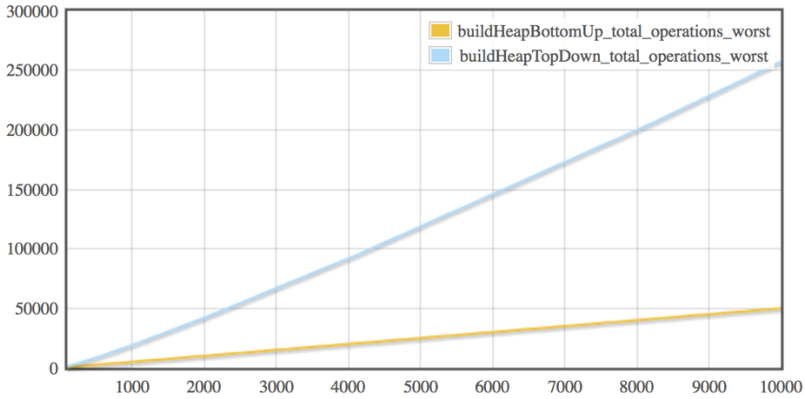


Figure 2.3: Total number of operations for the two build heap strategies, worst case analysis

The charts in figures 2.1-2.3 exemplify the shapes you should obtain. We have also provided the best case behavior. The actual values you get might differ, but the growth should be the same, and so is the relative ordering of the two algorithms' complexities.

Assignment No. 3: Analysis & Comparison of Advanced Sorting Methods – Heapsort and Quicksort. QuickSelect

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** *Quicksort and Quick-Select (Randomized-Select)*. You are also required to analyze the complexity of *Quicksort* and *Heapsort* (Implemented in Assignment No. 2) comparatively.

You may find any necessary information and pseudo-code in your course notes, or in the book:

- Heapsort: *Chapter 6* (*Heapsort*)
- Quicksort: *Chapter 7* (*Quicksort*)
- Randomized-Select: *Chapter 9*

Thresholds

Threshold	Requirements
5	QuickSort: implementation, exemplify correctness and average case analysis
7	QuickSelect (Randomized-Select): implementation and exemplify correctness
9	Comparative analysis of the Quicksort and Heapsort
10	Generate and evaluate best and worst case for QuickSort; interpretations, efficiency

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to prove your algorithm(s) work on a small-sized input.

1. You are required to compare the two sorting procedures in the **average** case. Remember that for the **average** case you have to repeat the measurements m times (m=5) and report their average; also for the **average** case, make sure you always use the **same** input sequence for the two methods – to make the comparison fair.

2. This is how the analysis should be performed:
   - vary the dimension of the input array (n) between [100…10000], with an increment of maximum 500 (we suggest 100).
   - for each dimension, generate the appropriate input sequence for the method; run the method, counting the operations (assignments and comparisons, may be counted together).

   ! Only the assignments and comparisons performed on the input structure and its corresponding auxiliary variables matter.

3. Generate a chart which compares the two methods under the total number of operations, in the average case. If one of the curves cannot be visualized correctly because the other has a larger growth rate, place that curve on a separate chart as well. Name your chart and the curves on it appropriately.

4. Interpret the charts and write your observations in the header (block comments) section at the beginning of your main .cpp file.

5. Evaluate Quicksort in the **best** and **worst** cases also – total number of operations. Compare the performance of Quicksort in the three analysis cases. Interpret the results.

6. For QuickSelect (Randomized-Select) no explicit complexity analysis needs to be performed, only the correctness needs to be demonstrated on sample inputs.

Example Charts

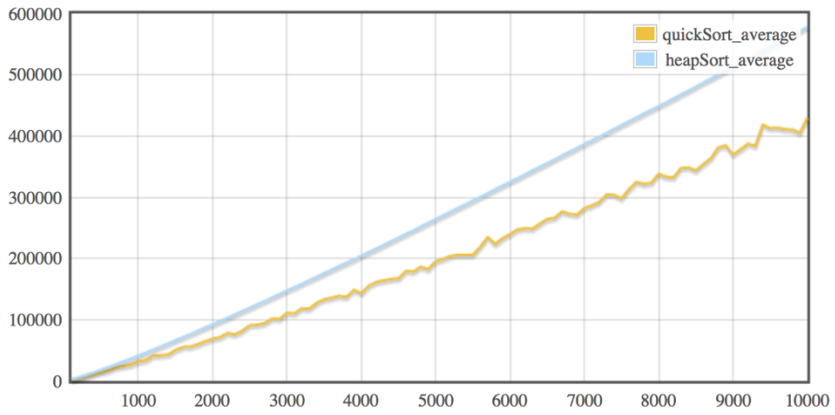


Figure 3.1: Total number of operations for Heapsort and Quicksort, average case analysis

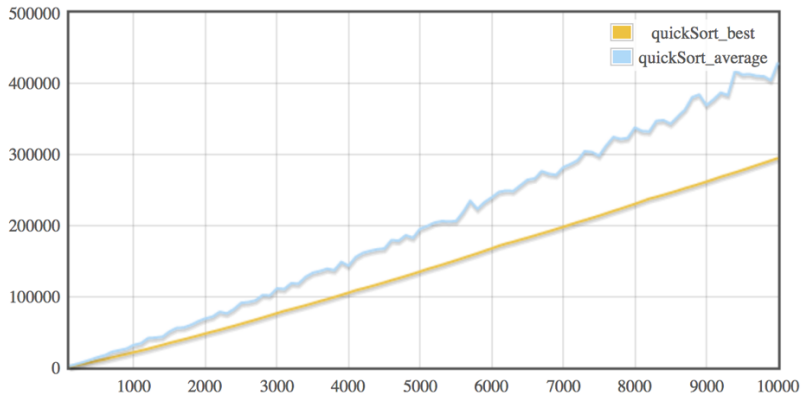


Figure 3.2: Comparison of Quicksort behavior, average and best case

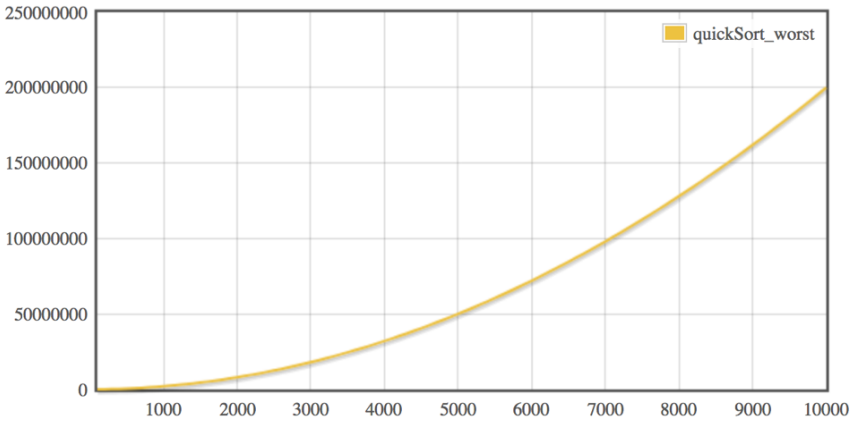


Figure 3.3: Quicksort behavior, worst case

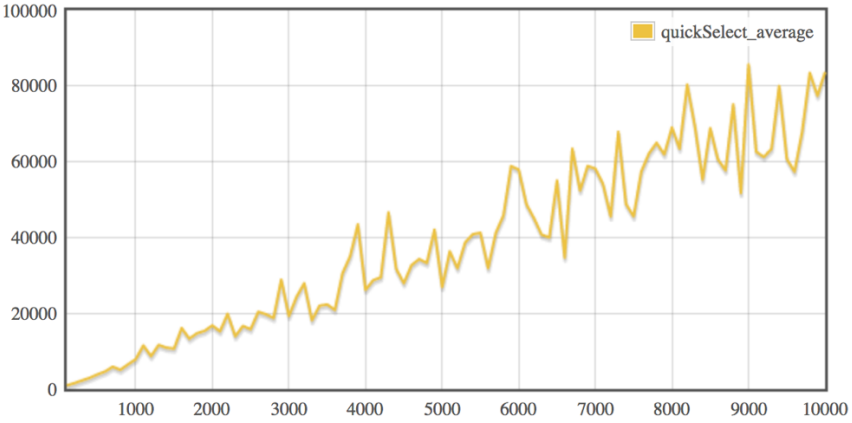


Figure 3.3: Quick-select behavior, average case

These are the shapes you should obtain for the curves. We have also included Quick-select average case analysis. The absolute values you get may differ. What are the shapes of the growth curves. Verify that the curves you obtain have the same growth.

Assignment No. 4: Merge k Ordered Lists Efficiently

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** an O(n*log*k) method for **merging k sorted sequences**, where n is the total number of elements. (Hint: use a heap, see seminar no. 2 notes).

Implementation requirements:

- Use linked lists to represent the k sorted sequences and the output sequence

Input: k lists of numbers $< a_1^i, a_2^i, \dots, a_{m_i}^i < a_1^i, a_2^i, \dots, a_{m_i}^i >$, $\sum_{i=1}^{k} m_i = n \sum_{i=1}^{k} m_i = n$

Output: a permutation of the union of the input sequences: $< a_1^{'} \leq a_2^{'} \leq \cdots \leq a_n^{'}$ $a_1^{'} \leq a_2^{'} \leq \cdots \leq a_n^{'} >$

Thresholds

Threshold	Requirements
5	Generate k random sorted lists, having n elements in total (n and k given as parameters); merge 2 lists
7	Adapt heap operations to work on new structure (list_index, key); use min-HEAP
9	Correct and complete algorithm implementation, with demo on a small-sized input
10	Evaluation, interpretations, discussion

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to show your algorithm works on a small-sized input (e.g. k=4, n=20).

We will make the average case analysis of the algorithm. Remember that, in the average case, you have to repeat the measurements several times. Since both **k** and **n** may vary, we will make each analysis in turn:

1. Choose, in turn, 3 constant values for k (k1=5, k2=10, k3=100); generate k random sorted lists for each value of k so that the combined number of elements in all the lists (n) varies between 100 and 10000, with a maximum increment of 400 (we suggest 100); run the algorithm for all values of n (for each value of k); generate a chart that represents the sum of assignments and comparisons done by the merging algorithm for each value of k as a curve (total 3 curves).

2. Set n = 10.000; the value of k must vary between 10 and 500 with an increment of 10; generate k **random** sorted lists for each value of k so that the combined number of elements in all the lists is 10000; test the merging algorithm for each value of k and generate a chart that represents the sum of assignments and comparisons as a curve.

3. Interpret your charts.

Example Charts

The charts in figures 4.1 and 4.2 illustrate the shapes you should obtain for this analysis. Absolute values may differ slightly.

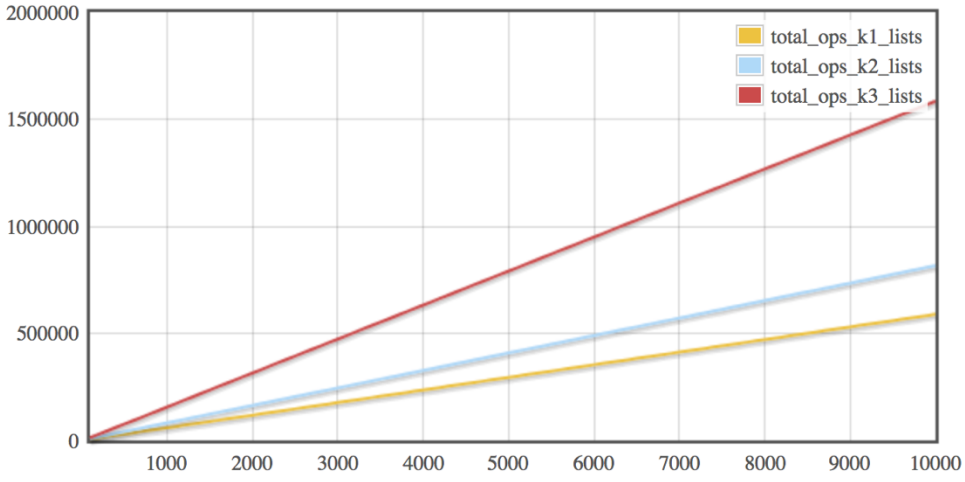


Figure 4.1: Total number of operations for the k-way merge heap-based algorithm, average case analysis, using 3 different values for k (5,10,100) and varying n between 100 and 10000

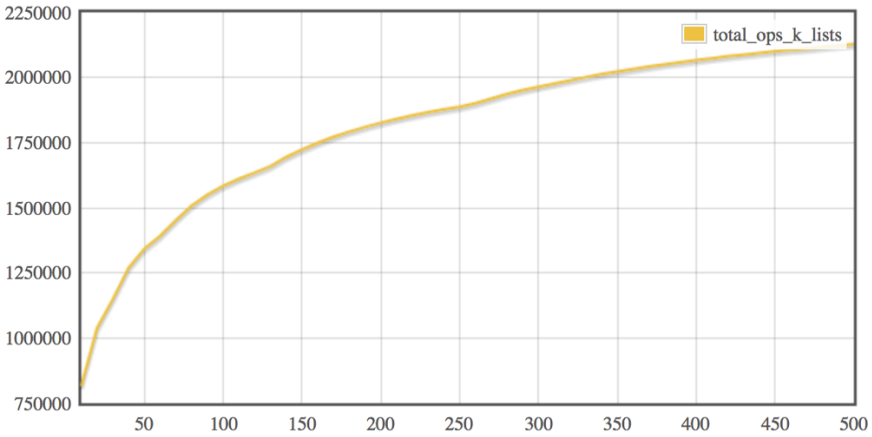


Figure 4.2: Total number of operations for the k-way merge heap-based algorithm, average case analysis, for n=10000, and varying k between 10 and 500

Assignment No. 5: Search Operation in Hash Tables

Open Addressing with Quadratic Probing

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** the *insert* and *search* operations in a hash table using open addressing and quadratic probing.

You may find any necessary information and pseudo-code in your course notes, or in the book, in section 11.4 Open addressing.

The use of closed and open specifies if it is mandatory to use a certain position or data structure.

Hashing (refers to the hash table)

- Open Hashing
  - Free to leave the hash table to hold more elements at a certain index e.g. chaining
- Closed Hashing
  - Not more than one element can be stored at a certain index e.g. linear/quadratic probing

Addressing (refers to the final position of the element with respect to its initial position)

- Open Addressing
  - The final address is not completely determined by the hash code, it also depends on the elements which are already in the hash table e.g linear/quadratic probing
- Closed Addressing
  - The final address is always the one initially calculated (there is no probing) e.g. chaining

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to prove your algorithm(s) work on a small-sized input.

You are required to evaluate the *search* operation for hash tables using open addressing and quadratic probing, in the average case (remember to perform 5 runs for this). You will do this in the following manner:

1. Select N, the size of your hash table, as a prime number around 10000 (e.g. 9973, or 10007);

2. For each of several values for the filling factor $\alpha \in \{0.8, 0.85, 0.9, 0.95, 0.99\}$, do:

    a. Insert n random elements, such that you reach the required value for α ($\alpha = n/N$)

    b. Search, in each case, m random elements ($m \sim 3000$), such that approximately half of the searched elements will be *found* in the table, and the rest will *not* be *found* (in the table). Make sure that you sample uniformly the elements in the *found* category, i.e. you should search elements which have been inserted at different moments with equal probability (there are several ways in which you could ensure this – it is up to you to figure this out)

    c. Count the operations performed by the search procedure (i.e. the number of cells accessed during the search)

3. Output a table of the form:

Filling factor	Avg. Effort *found*	Max. Effort *found*	Avg. Effort *not-found*	Max. Effort *not-found*
0.8				
0.85				
...	...		...	
0.99				

Avg. Effort = Total effort / Number of elements
Max. Effort = Maximum number of accesses performed by a single search operation

4. Interpret your results.

Example Results

The chart below exemplifies the values you should obtain when evaluating the average effort of the search operation. The results are averaged over 5 independent runs.

Alpha	Avg Found	Avg Not Found	Avg Max Found	Avg Max Not Found
0.80	1.89	5.31	18.4	28
0.85	2.15	7.26	28.8	44
0.9	2.4	11.18	41.4	88
0.95	3.01	22.15	70.6	129
0.99	4.41	102.82	237.4	841

Assignment No. 6: Dynamic Order Statistics

Allocated time: 4 hours

Implementation

You are required to implement **correctly** and **efficiently** the management operations of an **order statistics tree** (chapter 14.1 from the book). Dynamic order statistics algorithms perform searches according the order of elements. Elements are searched by their position in the sorted sequence, which is called the rank of the element.

In this assignment you see a structure which can implement such operations efficiently and analyze empirically the complexity of dynamic order statistics operations. You have to use a balanced, augmented Binary Search Tree. Each node in the tree holds, besides the necessary information, also the *size* field (i.e. the size of the sub-tree rooted at the node).

The management operations of an **order statistics tree** are:

- BUILD_TREE(n)
  - *builds* a balanced BST containing the keys 1,2,...n (*hint:* use a divide and conquer approach)
  - make sure you initialize the *size* field in each tree node

- OS-SELECT(tree, i)
  - selects the element with the i^{th} smallest key (having rank i)
  - the pseudo-code is available in chapter 14.1 from the book

- OS-DELETE(tree, i)
  - you may use the deletion from a BST, without increasing the height of the tree (why don't you need to rebalance the tree?)
  - keep the *size* information consistent after subsequent deletes
  - there are several alternatives to update the *size* field without increasing the complexity of the algorithm (it is up to you to figure this out).

Does OS-SELECT resemble anything you studied this semester?

Thresholds

Threshold	Requirements
5	BUILD_TREE - correct and efficient implementation; demo for n=11,
7	OS_SELECT & OS_DELETE - correct and efficient implementation, demo
9	Management operations evaluation
10	Interpretations, discussion

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to prove your algorithm(s) work on a small-sized input (11) i.e. pretty-print the initially built tree and, for a few elements (3), OS-SELECT by a randomly selected index and pretty-print the tree after its OS-DELETE).

Once you are sure your program works correctly:

- vary n from 100 to 10000 with step 100;
- for each n (don't forget to repeat 5 times),
    - build a tree with elements from 1 to n
    - perform n sequences of OS-SELECT and OS-DELETE operations using a randomly selected index based on the remaining number of elements in the BST

Evaluate the computational effort as the sum of the comparisons and assignments performed by each individual operation.

Example Chart

The chart in figure 6.1 shows the shape you should obtain by performing the analysis of the ensemble of algorithms presented in this assignment, in the average case. Absolute values may differ slightly.

What order of magnitude doe the curve have?

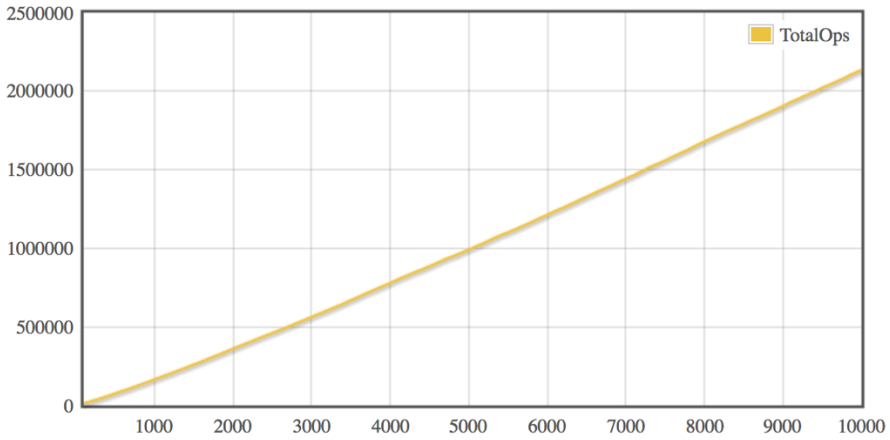


Figure 6.1: Total number of operations for the dynamic order statistics algorithms (build tree, insert, OS-SELECT), average case analysis, for n between 100 and 10000

Assignment No. 7: Multi-way Trees

Transformations between different representations

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** linear time transformations between three different representations for a multi-way tree:

- **R1**: *parent representation*: for each key you are given the parent key, in a vector.
- **R2**: *multi-way tree representation*: for each node you have the key and a list (e.g. vector, linked list) of children nodes
- **R3**: *binary tree representation*: for each node, you have the key, and two pointers: one to the first child node, and one to the brother on the right (i.e. the next brother node)

Also, you are required to write a *pretty print* procedure on *R3*, which performs a preorder traversal on the binary representation and outputs the tree in a friendly manner (see the image on the next page for an example).

Therefore, you are given as input a multi-way tree in the *parent* representation (*R1*). You are required to implement T1, which transforms the tree to a *multi-way* representation (*R2*), then T2, which transforms from the *multi-way* representation to the *binary* representation (*R3*). Then, on the binary representation, you are asked to write a pretty print procedure (using a pre-order traversal).

You should be able to design the necessary data structures by yourselves. You may use intermediate structures (i.e. additional memory).

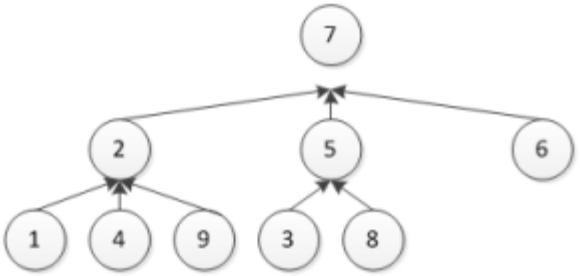
Evaluation

You should run your algorithms on a sample input tree (you may use the one in the example provided on the next page). Output (in a readable manner) the tree in each of the three representations (for *R1* simply print the parent vector; for *R3* it is enough to call the pretty print procedure).

Explain what data structures you employed for the *R2* and *R3* representations. You should assess the efficiency of your methods: i.e. do your transformations run in *O(n)*? Also, explain the necessity for any additional memory employed by your algorithms.

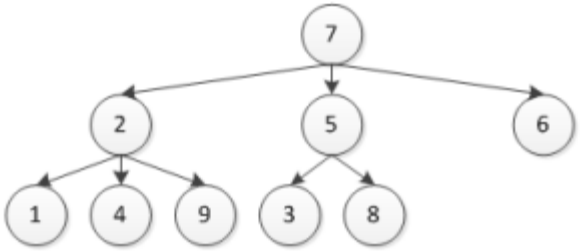
Input (R1): Π = {2, 7, 5, 2, 7, 7, -1, 5, 2}

1 2 3 4 5 6 7 8 9



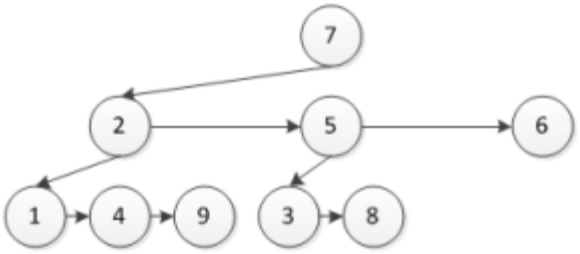
R1: parent representation

T1:parent -> multi-way



R2: multi-way representation

T2: multi-way -> binary



R3: binary representation

PP: prrety_print (binary)

```
7
    2
        1
        4
        9
    5
        3
        8
    6
```

Pretty print

Assignment No. 8: Disjoint Sets

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** the base operations for **disjoint set** (Section 21.1 from the book) and the **Kruskal's algorithm** (searching for the minimum spanning tree) using disjoint sets.

You have to use a tree as the representation of a disjoint set. Each tree holds, besides the necessary information, also the *rank* field (i.e. the height of the tree).

The base operations on **disjoints sets** are:

- MAKE_SET (x)
  - creates a set with the element *x*

- UNION (x, y)
  - makes the union between the set that contains the element *x* and the set that contains the element *y*
  - the heuristic *union by rank* takes into account the height of the two trees so as to make the union
  - the pseudo-code can be found in the chapter 21.3 from the book

- FIND_SET (x)
  - searches for the set that contains the element *x*
  - the heuristic *path compression* links all nodes that were found on the path to *x* to the root node

Thresholds

Grade	Requirements
5	Correct implementation for: MAKE_SET, UNION and FIND_SET + demo
7	Correct and **efficient** implementation for: Kruskal's algorithm
9	Evaluate the disjoint sets operations using Kruskal's algorithm
10	Interpretations, discussion

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! The correctness of the algorithm must be proved on a small-sized input (i.e. create 10 initial sets and execute the sequence: UNION and FIND_SET for 5 pairs of objects; print the contents of the resulting sets).

Once you are sure your program works correctly:

- vary n from 100 to 10000 with step 100;
- for each n
    - build a random graph with random weights on edges (**n** nodes, **n*4** edges)
    - find the minimum spanning tree using Kruskal's algorithm

Evaluate the computational effort as the sum of the comparisons and assignments performed by each individual base operation on disjoint sets.

Example Chart

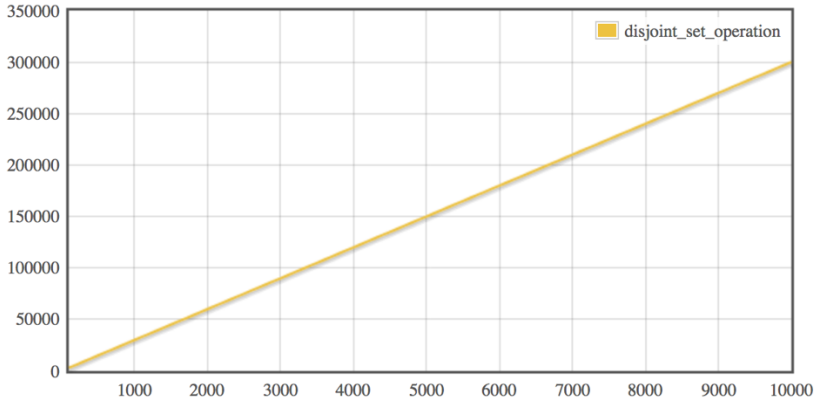


Figure 8.1: Total number of calls made to disjoint set operations (MAKE-SET, FIND-SET, UNION) in the Kruskal algorithm, as a function of n (the total number of MAKE-SET operations), average case analysis, for n between 100 and 10000

Assignment No. 9: Breadth-First Search

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** the *Breadth-First Search (BFS)* graph algorithm (Section 22.2 from the book). For graph representation, you should use adjacency lists. You are also required to pretty-print the resulting tree/forest of trees (use *Assignment 8* to achieve this) – only for the demo.

Thresholds

Grade	Requirements
5	Correct and **efficient** implementation of BFS
7	Correct and **efficient** implementation for the pretty print strategy
9	Evaluation
10	Interpretations, discussion

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to prove your algorithm works on a small-sized graph (which you may hardcode in your main function), i.e. for a small-sized graph, print the BFS tree/forest of trees.

Since, for a graph, both |V| and |E| may vary, and the running time of BFS depends on both (how?), we will make each analysis in turn:

1. Set |V| = 100 and vary |E| between 1000 and 5000, using a 100 increment. Generate the input graphs randomly – make sure you don't generate the same edge twice for the same graph. Run the BFS algorithm for each pair value and count the number of operations performed; generate the corresponding chart (i.e. the variation of the number of operations with |E|).

2. Set |E| = 9000 and vary |V| between 100 and 200, using an increment equal to 10. Repeat the procedure above to generate the chart which gives the variation of the number of operations with |V|.

3. Interpret your charts.

Example Charts

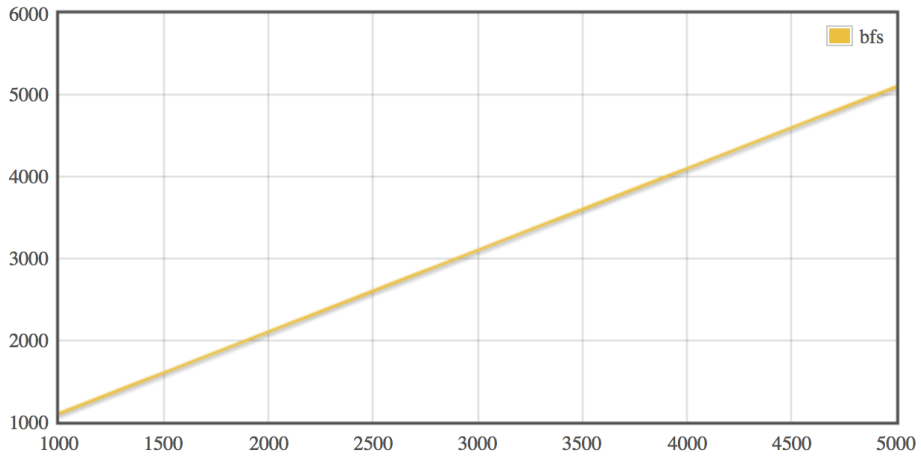


Figure 9.1: Total number of performed by the BFS algorithm, average case analysis, for E between 100 and 5000, and V=100

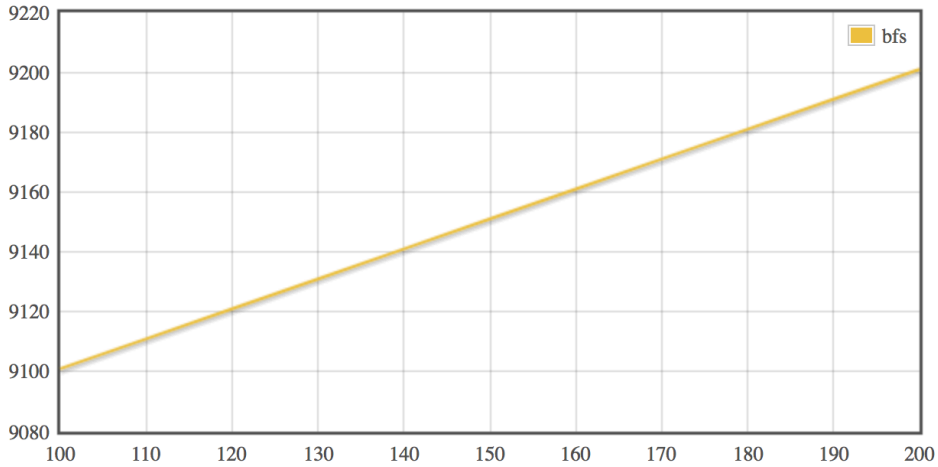


Figure 9.2: Total number of performed by the BFS algorithm, average case analysis, for E =9000 and V between 100 and 200

Assignment No. 10: Depth-First Search

Allocated time: 2 hours

Implementation

You are required to implement **correctly** and **efficiently** the *Depth-First Search (DFS)* graph algorithm (Section 22.3 from the book). For graph representation, you should use adjacency lists. You are also required to:

- Implement Tarjan's algorithm for detecting strongly connected components, on a directed graph (https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm)
- Implement topological sorting (described in *Section 22.4*)

Thresholds

Grade	Requirements
5	Correct and **efficient** implementation of DFS
8	Correct and **efficient** implementation of Tarjan and topological sort on a directed graph
9	Evaluation
10	Interpretations, discussion

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! Exemplify the correctness of your algorithm/implementation by running it on a smaller graph:

- Print the initial graph (the adjacency lists)
- Print all strongly connected components of the graph
- A list of nodes sorted topologically (should this list be nonempty? if empty, why so?)

Since, for a graph, both |V| and |E| may vary, and the running time of DFS depends on both (how?), we will make each analysis in turn:

1. Set |V| = 100 and vary |E| between 1000 and 5000, using a 100 increment. Generate the input graphs randomly – make sure you don't generate the same edge twice for the same graph. Run the DFS algorithm for each <|V|, |E|> pair value and count the number of operations performed; generate the corresponding chart (i.e. the variation of the number of operations with |E|).

2. Set |E| = 9000 and vary |V| between 100 and 200, using an increment equal to 10. Repeat the procedure above to generate the chart which gives the variation of the number of operations with |V|.

3. Interpret your charts.

Example Charts

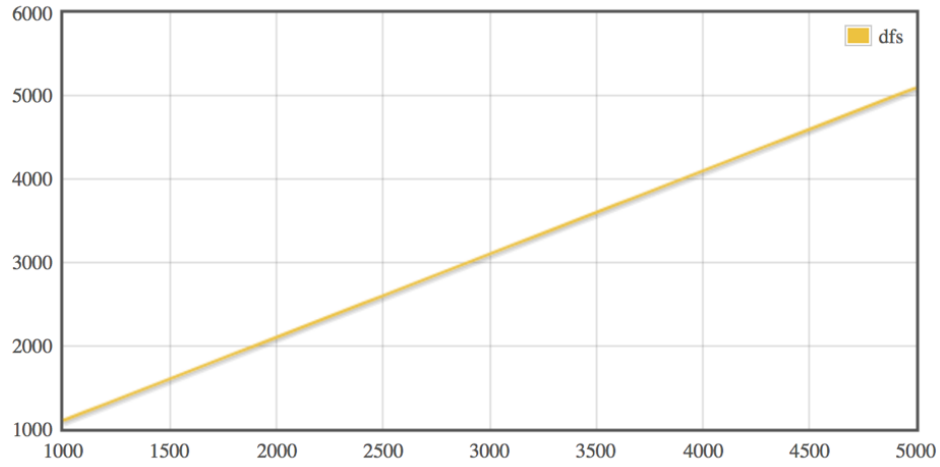


Figure 10.1: Total number of performed by the DFS algorithm, average case analysis, for E between 100 and 5000, and V=100

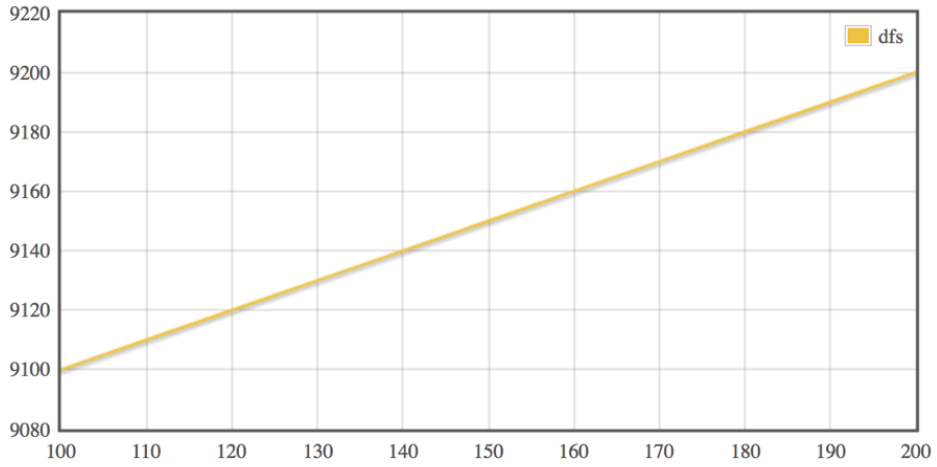


Figure 10.2: Total number of performed by the DFS algorithm, average case analysis, for E =9000 and V between 100 and 200

The charts in figures 10.1 and 10.2 indicate the expected growth pattern you should obtain for your curves.

Bibliography

[1] Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest. Clifford Stein. *Introduction to Algorithms*. Third Edition. The MIT Press. Cambridge, Massachusetts London, England, 2009