Algorithms and Data Structures 2020

Hans Georg Schaathun

1. september 2020

Welcome to *Algorithms and Data Structures* 2020. The module resources comprise

Lecturer Professor Hans Georg Schaathun hasc@ntnu.no. Please ask if anything is unclear.

Language English is the primary language for the module, because it also serves as an elective for the (international) MSc programme. Norwegian may be used in groups where everybody is fluent.

Main Session (four hours). Online. The main session will start and end with a plenary lecture. The middle part will be group work.

Auxiliary Session (two hours) Normally on campus, although one online session will probably be provided for those unable to attend on campus. The classes is divided into a couple of groups, according to room capacity. The session is used for group or individual work supervised by teaching assistants. Sometimes, video lectures will be made available to be watched before and/or after the auxiliary sessions.

Compulsory assignments must be handed in, but they are not graded. However, roughly half of the exam paper will be drawn from the compulsory projects, and thus it will pay off to do the assignments properly. See Section 1.2.

Blackboard BlackBoard is used for announcement, and if we are not allowed to have on-campus sessions, BB Collaborate will be used for joint sessions as well.

Learning material is this site, which is dynamic and is updated as we go along.

Textbook Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser: *Data Structures and Algorithms in Java*, 6th Edition International Student Version.

In principle, the syllabus is the entire book as well as the the lecture notes on this site. However, from the textbook, Chapters 1 and 7 are cursory material, as they concern Java specifics.

NP-Completeness is an important topic not covered by the main textbook. You have to rely on the lecture notes for that final topic.

Additional Reading It is worth reading *Introduction to Algorithms*, Third Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. For better or for worse, it is heavier than Goodrich, Tamassia, and Goldwasser in more than one sense of the word.

Previous Exam Papers Sorry, this is a new module, so no such papers exist. The best guideline you have are the the exercises (compulsory and otherwise) which you are assigned week by week.

1 Practical Information

1.1 Learning Outcomes

A module on Algorithms and Data Structures is supposed to make you better programmers. We do that by stepping away from actual programming, and try to view computer programs from the outside. We shall learn to solve problems, describe solutions, analyse properties of the program, and prove correctness without being distracted by the details of implementation.

For most students it may be a good idea to do some programming exercises in order to link what you learn to what you already knew. That will often help you learn the new material better. This module, however, will be assessed based on abstract discussion of programmable solutions, rather than on actual programmed solutions.

The key skills and competencies to learn are

1. being able to communicate, discuss, and evaluate potential solutions and partial designs for computer programs.
2. being able to prove that an algorithm is correct.
3. being able to assess the runtime complexity of an an algorithm.
4. being able to choose suitable algorithms and data structures for a given information system. (This will also require you to be familiar with a range of standard algorithms and data structures.)
5. being able to construct variations of algorithms.

1.2 Compulsory Assignments

There are weekly compulsory assignments. They must be handed in on time even if you have not completed the task as set.

Each submission needs to contain the following items:

1. Draft solution to the problem(s) assigned. This may be incomplete, but it has to be handed in.
2. A comment on your solution. Why did you approach the problem in this way.
3. A reflection note. What did you learn from the exercise? What do you have to work on? How is it relevant (or not) for your career?

4. When you have received feedback on a previous submission, you have to include a reflection on that feedback. What did you learn from the feedback, and what are you going to do about it?

Please, **submit on time even if you cannot complete the technical task.** If you are not satisfied with the technical work, reflect upon this in the reflection note. What is difficult? What can you do? What do you want to understand? Finally, alert the module convener or the TA that you would like feedback on this particular submission.

Exact deadlines can be found in BlackBoard. The recommended submission format is in Micro-Soft OneNote, via BlackBoard.

The technical solution (Item 1) can be done collaboratively. The comment and reflection (Items 2-4) are personal and must be done individually.

In order to sit the exam, you have to hand in 10 out of the 14 weekly assignments on time. You should hand in all of them, and when you are allowed four lapses, it is only to allow for illness and other mitigating circumstances. There is therefore **no excuse** for not handing in ten of the assignments on time.

In order to provide feedback on your work, we will ask you to go through your submission with us, one-to-one, on at least two occasions during the semester. This will require you to meet up, either physically or online, to discuss your most recent assignment. This is not supposed to be a polished and formal presentation, but an casual chat about what you have already written in the submission. We cannot give feedback to all students every week. If you want to discuss a particular piece at a particular time, please let us know. Otherwise, we will just tell you what to present when.

The feedback will never lead to a fail, but you have to deal with the feedback and include reflection upon it in your next submission.

The compulsory assignments are not graded, but approximately half of the exam will be drawn directly from work you have done in the compulsory assigment.

2 Foundations of Algorithms

Reading 1. Goodrich, Tamassia, & Goldwasser: Chapter (1), 2, 4.1, and 4.4.

Chapter 1 and most of Chapter 2 is cursory background material, but it is useful reading because it will help you connect the theoretical and high level contents of the module to your concrete experience with Java programming.

Algorithm Theory works with **Language-Independent Models**. We do *not* want to be tied up in the details of particular programming languages. This is important. We choose a simpler language to be understandable by a wider audience, across different language traditions (C, Java, Ada, Python).

The first thing to do in this module is to learn the basics of this language. We shall try to map the concepts to Java jargon as we go along.

2.1 Key Concepts

2.1.1 Description of Algorithms

Consider the sorting of a bridge hand. Thirteen cards to be sorted in increasing order.

We sort first by suit, so that

$$\spadesuit > \heartsuit > \diamondsuit > \clubsuit$$

and then within each suit so that

$$A > K > Q > Kn > 10 > 9 > 8 > 7 > 6 > 5 > 4 > 3 > 2.$$

Step 1. Specification of the Concrete Problem

Input. A hand of 13 cards.

Output. A hand of 13 cards sorted in increasing order.

Step 2. Specification of the Abstract Problem

Input. An array of n objects, $A_1, A_2, \ldots, A_n$.

Output. An array of n objects sorted in increasing order.

The objects can be of any type (class), as long as we have a binary relation $\leq$, so that for any two objects x, y we can determine whether $x \leq y$ is true or false.

In algorithm theory, we tend to think of the objects as numbers, so that we have an intuitive grasp on what $\leq$ means, but there is no loss of generality. In Object Oriented Programming, the notion of less than or equal is encapsulated in the class, and we may have to rewrite it as x.`isSmallerThanOrEqual`(y) instead of $x \leq Y$. In Functional Programming, the function implementing $\leq$ may be passed as an argument alongside the array A using a lambda expression which also the later versions of Java supports, and in this case it may be rewritten as `isSmallerThanOrEqual`(x, y) instead of

Step 3. Pseudo-Code

One way to solve the sorting problem is the following.

Input: Array A_i, $i = 1, \ldots, n$
Output: The same array A_i, $i = 1, \ldots, n$ sorted in place so that $A_1 \leq A_2 \leq \ldots \leq A_n$

```
1        for i := 2, 3, ..., n
2            j := i
3            while (j ≥ 2) and (A_j < A_{j-1})
4                swap A_j and A_{j-1}
5                j := j − 1
```

Observe how we combine well-known programming constructs (for, while), mathematical notation ($\geq, A_i, A_j$), and natural language (swap). This hybrid language is called pseudo-code. There is no standard for how to write it. As long as it is legible and unambiguous to human readers, it is ok.

I have in this case used := as the assignment operator, just to illustrate the variation you will encounter. Java and C uses the equality sign = for assignment, and for equality they use a double equality sign ==.

Some authors would prefer pseudo-code closer to their favourite programming language, for instance:

```
1        for i = 2 to n
2            j = i
3            while (j >= 2) and (A[j] < A[j−1]
4                swap A[j] and A[j−1]
5                j = j−1
```

This is a matter of taste, more community taste than personal taste though, but you may have to deal with more than one community..

The ‘swap’ line deserves some comment, as we use this in both the example styles, in spite of its being far from known programming languages, where it would have to be rewritten as

```
1        t   = A_j
2        A_j = A_i
3        A_i = t
```

The swap statement is a simple and easily understandable instruction in natural language, and most human reader would find the programming construct much harder to read. This is why we resort to natural language here. In the choice of style, you should always strive to maximise the reader’s comprehension.

2.1.2 Analysis

Empirical versus Theoretical Analysis

In the first year, you have had to test your programs, to see if they work as intended. Testing is, of course, important both in its informal forms and in more structured and analytical forms, where we can talk about empirical analysis.

Empirical analysis is limited, however, by the number of examples you have time to test. Each test you make will only validate one single case, and there may be an infinite number of cases with different properties.

Algorithm theory, and therefore this module, focuses on theoretical analysis, searching for proofs that are valid for any data set. Note that theoretical analysis will also help in the design of good and complete test sets for empirical analysis.

Correctness of Insertion Sort

Let’s see what it does.

The index i is the card we are looking at. Cards to the left have already been looked at, and cards to the right not yet. In the first iteration, $i = 2$, so there is only one card to the left. Obviously an array of a single card is always sorted. The inner loop (while) tries to insert the ith card in the correct position among the $i - 1$ previous cards.

Some programming languages allow assertions, establishing claims relevant to the analysis. Let’s write them into the pseudo-code as follows.

```
1        for  i := 2, 3, ..., n
2            assert  A_1 ≤ A_2 ≤ ... ≤ A_{i-1}
3            j := i
4            while  (j ≥ 2)  and  (A_j < A_{j-1})
5                swap  A_j  and  A_{j-1}
6                j := j - 1
```

```
7                 assert  A_1 ≤ A_2 ≤ ... ≤ A_i
8             assert  A_1 ≤ A_2 ≤ ... ≤ A_n
```

When the programming languages support assertions, they are tested in debug mode, to identify places where critical assumptions are broken. In theoretical analysis, they are claims used to structure the proof. The two first assertions are examples of *loop invariants*, i.e. properties which invariable holds in every iteration of the loop.

The purpose of the while loop, is to reestablish the loop invariant for when the index i increases.

The first iteration of while compares A_i to A_{i-1} (because $j = i$). If A_i is larger, it belongs where it is and the while loopis not entered. If it is smaller, the two cards are swapped, and the loop continues to compare it with the next card.

When the while loop terminates, the i first cards are in sorted order, and in the next iteration of the for loop, we can again say that the cards to the left of i are sorted.

Complexity of Insertion Sort

- Number of instructions. What is an instruction?
- Formal computing models
- Worst case, best case, and average case
- Complexity

2.2 Projects and Exercises

2.2.1 Compulsory Projects

These projects should be solved as part of the weekly assignment. When you phrase your answers, you should *always* write with fellow students in mind. Write so that they would understand, and be convinced by your argument. If you are unsure what is a comprehensible argument, discuss it with fellow students. Always use drawings and sketches when they are more comprehensible than prose.

Øvingsoppgåve 2.2.1 (Sorting Cards)**.** Consider the problem of sorting a hand of cards. How would you do it naturally?

If you have a deck of cards, shuffle and draw at least ten cards at random. If you do not, you can write down ten (or more) numbers at random.

Look at the cards/numbers. How would you intuitively start to sort them?

Answer the following,

1. Is your approach systematic or not?
2. Can you write down pseudo-code describing how you do it?

3. If you cannot, why is that? Can you make changes so that it is possible?
4. Is your approach similar to insertion sort? What are the main differences?

(You should do this exercise together with other students, and discuss and compare approaches. Do not spend more than about 30 minutes on the problem. There is no ideal answer.)

Øvingsoppgåve 2.2.2 (Letter Frequencies)**.** Simple substitution ciphers work by replacing each letter in the alphabet with another. To encrypt a text, the same substitution is applied throughout the text. Such ciphers are easily broken by using frequency analysis.

Consider a program which takes a text as input and outputs frequency tables for each letter in the text. I.e. for each letter in the alphabet, output the number of occurrences of this letter in the text. (See also P-2.22 in the textbook.)

You are going to describe (not implement) such a program. Think throught the following questions first:

- How do you model the text (input)?
- How do you model the frequency tables (output)?
- How do you parse the text?

Describe your program in the form of an algorithm, with pseudo-code and precise definitions of input and output.

- How do you know that the algorithm produces the correct answer?
- How many operations does the algorithm require when the text is n characters long and there are k letters in the alphabet?

2.2.2 Exercises

Øvingsoppgåve 2.2.3 (Change Making)**.** Design and describe a program (algorithm) which takes two amounts (numbers) as input. One is the amount charged and the other the amount given. The program should return the number of each kind of bill and coin to give back as change for the difference between the amount given and the amount charged. The values assigned to the bills and coins avalaible can be based on the monetary system of any current or former government. Try to design your program so that it returns the fewest number of bills and coins as possible.

You can for instance use the denominations available for Norwegian *kroner:*

$$1, 5, 10, 20, 50, 100, 200, 500, 1000$$

Does your algorithm always produce the fewest number of bills and coints? Would it always give the fewest coins and bills if you change the denominations?

Consider the similar problem of selecting stamps to make up a given amount of postage. Suppose the stamps have values 1, 24, 33, and 44 pence. Test the algorithm to find the required stamps for 67p postage.

(Cf. Goodrich, Tamassia, & Goldwasser P-2.26)

Øvingsoppgåve 2.2.4 (Selection Sort)**.** Selection Sort is a sorting algorithm, sinilar to insertion sort. It can be described as follows:

Input: Array `A[]` of size `n`.
Output: The same array `A[]` in sorted order.

```
1          for i := 1 to n-1
2              for j := i+1 to n
3                  if A[i] > A[j]
4                      swap A[i] with A[j]
```

1. Rewrite the pseudo-code in Java. Note in the indices in particular. Is the first element `A[0]` or `A[1]`.
2. How many swap operations must be made in the worst case? What about the best case?
3. Can you prove that the algorithm is correct, i.e. produces a sorted array? Compare it to the proof for insertion sort.

2.3 Home study

2.3.1 Proof Techniques

Video: Proof by Example and Counter-Example

Video: Induction and Loop Invariants

- **TODO** video on Induction and loop invariants

2.3.2 Review Material

These videos where made for the old module in Discrete Mathematics 2013-2015, but they are also relevant here.

Sorting Algorithms

Video: Insertion Sort

Video: Selection Sort

Proofs of Correctness

Video: Proof of Insertion Sort

3 Data Structures

Reading 2. Goodrich, Tamassia, & Goldwasser: Chapter (2), 3

3.1 Key Concepts

3.1.1 LinkedList and ArrayList

In Java, you are probably familiar with the following Data Structures or Data Types:

- Array
- ArrayList
- LinkedList

What is the difference between them?

Array is a primitive data type. Let's leave that aside for now.

Both ArrayList and LinkedList are classes inheriting the abstract class List. Therefore they share a number of methods.

- add(index,element)
- get(index)
- remove(index)

There are others, but these three are particularly interesting from an algorithmic point of view. We shall discuss their complexity.

3.1.2 ADT (Abstract Data Type)

Definisjon 3.1.1 (Abstract Data Type)**.** An Abstract Data Type (ADT) is a *set of operations*.

In Object Oriented Programming the set of operations will normally be a set of methods. Java can codify an ADT as an interface, and to implement the ADT a class must implement the interface. However, the ADT define the semantics (behaviour) of the methods, while the interface only defines the syntax (call signature). Thus an implementation of the interface is not necessarily an implementation of the ADT.

The ADT itself, is the mathematical model describing the interface and its semantics. An API, in contrast, is the interface of the implementation.

1. Static and Dynamic Definitions
    - Java Generics
    - Type parameters

3.1.3 The Array ADT

The Array ADT would normally provides methods to

- *get* the element at a given position
- *set* (change) the element at a given position

The Java ArrayList class in Java provides many more methods. Most importantly, you can change the size of the ArrayList. Adding or removing an element at the end of the ArrayList is an important functional extension. When Arrays are used in the theoretical literature, the size is usually assumed to be fixed.

THe other methods of ArrayList are convenience methods which can be implemented using the methods above.

3.1.4 The List ADT

The List ADT is stateful. It always has a cursor pointing to the current position.

- *add* an element at the current position
- *remove* an element at the current position
- *next* get the next element in the list

The ListIterator

Notably, the list classes in Java does not implement an interface representing the List ADT. Instead, the List interface has the `listIterator()` which returns an object implementing the ListIterator interface, which represents the List ADT.

This makes it possible to access the same List concurrently in different subprograms, because each iterator maintains its own cursor.

In Java:

- AbstractList (Abstract Class)
- LinkedList and ArrayList (Concrete Classes)
- List (interface) $\sim$ Array (ADT)
- ListIterator (Java) $\sim$ List (ADT)
- LinkedList also implements the Deque ADT

3.1.5 Algorithm Theory

The interesting questions in algorithm theory is how we implement the ADTs so that they are correct and efficient, and exactly how efficient each operation is in terms of complexity.

3.1.6 Copies

1. Equality
2. Cloning - shallow and deep copies

3.2 Projects and Exercises

3.2.1 Compulsory Assignment

Øvingsoppgåve 3.2.1 (Experimental Running Time)**.** Compare the running time of LinkedList and ArrayList in Java.

Use the file ArrayListDemo.java as a basis. Compile and run the program, and look at the output as well as the source code. What does this test tell us about the performance of ArrayList?

It is possible that the program runs out of memory. In that case you may have to increase the heap size with an option to JVM.

1. Modify the program to use LinkedList instead of ArrayList. Compile and run the new version. What does it tell us about the performance of LinkedList? What are the advantages and disadvantages of LinkedList compared to ArrayList?
2. Modify the programs to initialise a smaller array, maybe about 100 elements, and test both ArrayList and LinkedList. How do the two implementations compare when the list is small?
3. Try with some different list sizes to see how the run time developes. Which operations run in constant time? Linear time? Slower than linear?
4. Review the theoretical properties of LinkedList and ArrayList, and compare your empirical output to the theory. Are the test results reasonable?
5. When would you prefer to use ArrayList and when would you prefer LinkedList? Try to find example applications where you would recommend one or the other.

Øvingsoppgåve 3.2.2 (Optional)**.** In the above tests, we measured in milliseconds, and mostly only a single operation at a time. This does not always allow us an exact comparison.

There are two things you can do:

1. Use the java.lang.System.nanoTime() function instead.
2. Make several, similar operations in sequence, e.g. for instance 100 get() calls on adjacent indices.

Try to modify the test programs, and see if you can learn a bit more about the cases where the first test showed zero milliseconds.

(Remember that we rarely care much about the time of a single operations, but when the same operation is made thousands or millions of times, even a difference of less than a millisecond may matter.)

Øvingsoppgåve 3.2.3 (Based on Goorich *et al* C-3.20)**.** Suppose you are making a multiplayer game with $n \geq 1000$ players, numbered from 1 to n, interacting in an enchanted forest. The winner is the player who first meets every other player. The game is constructed so that a function, meet(i, j) is called every time player i meets player j,

Design the algorithm to be implemented in meet, so that it records who has met whom, and reliably detects when someone has won (i.e. has met every other player).

You should also make an init() algorithm to initialise the data structure used by meet(i, j).

1. Can you demonstrate that the algorithm is correct?
2. What is the time complexity?
3. Is it possible to make it faster?

Hint. You may have to record both whether or not i and J have met, and how many players i has met.

3.2.2 Exercises

Øvingsoppgåve 3.2.4 (Hall of Fame)**.** Consider a Computer Game where you are going to implement a *Hall of Fame*, i.e. a list of, say, the 100 best scores ever achieved. What data structure would you choose? What benefits and disadvantages do you consider to make that choice?

Describe the data structure (do not implement it) with all the operations required in the application. For each operation, consider the following:

1. How fast/slow is the operation?
2. When and how often do you expect the operation to be used?

Would your choices change if the number of scores to be stored changes?

Øvingsoppgåve 3.2.5 (Based on Goorich *et al* C-3.19)**.** Design an algorithm, shuffle, which takes a list A with n elements and reorders it such that every permutation is equally likely. Assume that you have a function random(n) to return a random integer in the range $0, 1, \ldots, n-1$.

1. What is the running time (complexity) of your algorithm?
2. Is it possible to solve the problem in linear time?
3. Is it best to use ArrayList or LinkedList as input and output? Why?

Øvingsoppgåve 3.2.6. Goodrich *et al* C-3.22–23. Comment on the run time complexity for your solutions.

Øvingsoppgåve 3.2.7. Goodrich *et al* C-3.24. Comment on the run time complexity for your solutions.

Øvingsoppgåve 3.2.8. Goodrich *et al* C-3.25.

Comment on the run time complexity for your solution.

4 Complexity Analysis

Reading 3. Goodrich, Tamassia, & Goldwasser: Chapter 4

4.1 Key Concepts

4.1.1 Complexity

1. Induction and Deduction
2. Common functions with examples
3. Algorithm Theory focuses on infinitely large datasets: asymptotics
4. Big-O
5. Recursion and loop
6. Mathematical Induction and Loop Invariants
7. Computing Model

4.1.2 Recursion

1. **TODO** video on Induction and recursion

4.2 Projects and Exercises

This week, I ask you not to start with the compulsory projects. The Wednesday Group Work section contains two warm-up problems, and then a discussion exercise which I want to discuss in the plenary debrief, after your group discussion.

You should be able to complete those three problems in the group session, and you may have time to proceed with the compulsory projects.

4.2.1 Wednesday Group Work

Øvingsoppgåve 4.2.1 (Goodrich *et al* R-4.1 rephrased)**.** The number of operations executed by algortihms A and B is $8n \log n$ and $2n^3$, respectively. Determine n_0 such that A| is better than B for $n \geq n_0$

Øvingsoppgåve 4.2.2 (Goodrich *et al* R-4.7 rephrased)**.** Order the following functions by asymptotic growth rate:

$$4n \log n + 2n, 2^{65}, 2^{\log n}, 4n + 100 \log n, 4n, 1.1^n, n^2 + 10n, n^3, n \log n, n^{32}.$$

Øvingsoppgåve 4.2.3 (Goodrich *et al* C-4.43 rephrased)**.** Claim: *In any flock of sheep, all sheep have the same colour*

An alleged proof goes as follows.

Base Case: $n = 1$ A single sheep clearly has the same colour as itself.

Induction Step: Consider a flock of $n > 1$ sheep. Take one sheep a out. The remaining $n - 1$ sheep have the same colour by induction. Now, replace a and take a different sheep b out. Again, the remaining $n - 1$ sheep have the same colour by induction. Hence, all the sheep in the flock have the same colour.

In reality, we know that a flock with black and white sheep has been observed, so what is wrong with the argument?

4.2.2 Compulsory Assignment

Øvingsoppgåve 4.2.4 (Goodrich *et al* C-4.49 rephrased)**.** Let $p(x)$ be a polynomial, i.e.

$$p(x) = \sum_{i=0}^{n} a_i x^i.$$

1. Describe an $O(n^2)$-time algorithm to compute $p(x)$.
2. Improve the algorithm to $(n \log n)$ time by improving the calculation of x^i.
3. Now consider rewriting as

   $$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n) \cdots))).$$

   How many arithmetic operations do you need to calculate this (in Big-O notation).

Øvingsoppgåve 4.2.5 (Goodrich *et al* R-4.31 rephrased)**.** Al and Bob are arguing about their algorithms. Al claims his $O(n \log n)$-time algorithm is *always* faster than Bob'S $O(n^2)$-time algorithm. To settle the issue they run a set of experiments. To Al's dismay, they find that if $n < 100$, the $O(n^2)$-algorithm runs faster, and only when $n \geq 100$ is the $O(n \log n)$-time one better. Explain how this is possible.

4.2.3 Practice Problems

Øvingsoppgåve 4.2.6 (Goodrich *et al* C-4.35 rephrased)**.** Show that

$$\sum_{i=1}^{n} i^2 = O(n^3).$$

Øvingsoppgåve 4.2.7 (Goodrich *et al* R-4.28 rephrased)**.** Given an n-element array X. Algorithm A chooses $\log n$ elements from X at random and executes an $O(n)$-time calculation for each. What is the worst-cae running time of B.

Øvingsoppgåve 4.2.8 (Goodrich *et al* R-4.30 rephrased)**.** Given an n-element array X. Algorithm D calls Algorithm E on each element X_i. Algorithm E runs in $O(i)$ time when called on X_i. What is the worst-case run time of D?

Øvingsoppgåve 4.2.9 (Goodrich *et al* P-4.55 rephrased)**.** Make an experimental analysis to test the hypothesis that Java's Array.sort() method runs in $O(n \cdot logn)$ time on average.

1. Proof techniques C-4.x

5 Stacks and Queues

Reading 4. Goodrich, Tamassia, & Goldwasser: Chapter 6

1. Stack
2. Queue

5.1 Projects ??

- P-5.5 Capital Gain (FIFO)
- P-6.3 Text Editor
- C-6.20 Sorting Data Packets
- Skyline.
    1. Data structure
    2. Linear or Split and Conquer?

6 Trees

Reading 5. Goodrich & Tamassia: Chapter 7

6.1 Key Concepts

6.2 Projects and Exercises

6.2.1 Compulsory Assignment

6.2.2 Exercises

7 The Heap

Reading 6. Goodrich & Tamassia: Chapter 8

7.1 Key Concepts

7.2 Projects and Exercises

7.2.1 Compulsory Assignment

- P-8.8 Job Scheduler

7.2.2 Exercises

8 Maps and Hash tables

Reading 7. Goodrich & Tamassia: Chapter 9

8.1 Key Concepts

- The Map interface
- Ordered Maps

8.2 Projects and Exercises

8.2.1 Compulsory Assignment

8.2.2 Exercises

9 Search Trees

Reading 8. Goodrich & Tamassia: Chapter 10

9.1 Key Concepts

9.2 Projects and Exercises

9.2.1 Compulsory Assignment

9.2.2 Exercises

10 Text Processing

Reading 9. Goodrich & Tamassia: Chapter 12. (Chapter 12.4, see next section.)

10.1 Key Concepts

10.1.1 Searching and Pattern Matching

10.1.2 Dynamic Programming

10.2 Projects and Exercises

10.2.1 Compulsory Assignment

10.2.2 Exercises

11 Greedy Algorithms and Huffmann

Reading 10. Goodrich & Tamassia: Chapter 12.4

11.1 Key Concepts

11.2 Projects and Exercises

11.2.1 Compulsory Assignment

11.2.2 Exercises

12 Sorting

Reading 11. Goodrich & Tamassia: Chapter 11

12.1 Key Concepts

12.2 Projects and Exercises

- Sorting - Technical Variations
- Big-O

12.2.1 Compulsory Assignment

12.2.2 Exercises

13 Shortest Path

Reading 12. Goodrich & Tamassia: Chapter 13

13.1 Algorithms

Learn (1) proofs and (2) complexity for Dijkstra's Algorithm.

Graph Algorithm Problems

1. DFT and BFT
2. Topological Sort
3. Transitive Closure
4. Shortest Path
5. Spanning Tree

13.2 Projects and Exercises

13.2.1 Public Transport

13.2.2 Network Broadcast

13.2.3 Google Maps

Practical judgement rather than algorithmic

1. Calculation
2. Caching

13.2.4 Compulsory Assignment

13.2.5 Exercises

14 Memory

Reading 13. Goodrich & Tamassia: Chapter 14

14.1 Key Concepts

14.2 Projects and Exercises

1. C-15.12/13. External memory data structures. (also other C-15.*)

14.2.1 Compulsory Assignment

14.2.2 Exercises

15 NP-Completeness

15.1 Projects and Exercises

15.1.1 Compulsory Assignment

15.1.2 Exercises