***Algorithms and Data structures***

***Topic: Getting Started***

***Week: 1***

***Takeaways***

* *Takeaway 1*
  + *A computation transforms of a sequence of symbols into another.*
* *Takeaway 2*
  + *A computational problem maps input data to output data. Mathematical functions clearly capture how these inputs map to the appropriate output*
* *Takeaway 3*
  + *An algorithm is a finite sequence of non-ambiguous instructions, which processes its inputs to produce the solution of a computational problem. To work efficiently, algorithms store their data into dedicated data structures.*
* *Takeaway 4*
  + *There is an direct relationship between the actions we stipulate in an algorithm and the capabilities of the computer we use to execute it.*

***1 Computation***

* ***CS is all about computation***
  + *A computation is transformation of data*
    - *It consumes some data and produces some data*
* *What is Data?*
  + *Data is an overload term*
    - *Example:*
      * *Texts*
      * *Numbers*
      * *Pictures*
      * *Audio recording*
  1. ***Computational Problems***
* *Computational problems*
  + *Are problems which can be solved by a computation*

***1.2 Algorithms***

* *An algorithm has inputs and outputs. It consumes some data and produces some results.*
* *An algorithm is finite: it must terminate at some point and cannot have an infinite number of steps.*
* *An algorithm is well-defined, and each step is non-ambiguous.*
* *An algorithm is effective and can be carried out by either a machine or human with pen and paper in a finite amount of time. Each step must be feasible.*

*Do not confuse algorithm and computation. Algorithm is the list of steps to follow whereas the computation is what happen when a computer goes through a particular addition.*

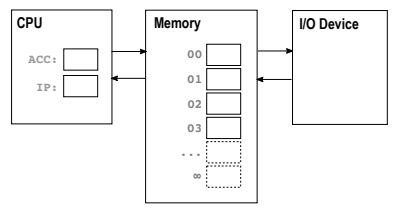
1. *How to Describe an Algorithm*

* *Using natural language*
* *Using Flowcharts*
* *Using Pseudocode*
* *Using a program*

1. *RAM*

*Takeaways*

* *Takeaway 1*
  + *The RAM model defines the actions that the machine understands.*
* *Takeaway 2*
  + *There are many ways to encode a given piece of pseudocode into machine code. It is important to understand-at a high-level-how a compiler does that.*
* *Takeaway 3*
  + *A program is an algorithm encoded using a programming language. This program can be converted into machine code for execution by a computing device.*
* *Random Access Machines*
  + *RAM is an abstract machine, a “model” of computation:*
    - *It is a blueprint that carries out computation.*
* *Architecture*
  + *A random-access machine is a machine that mimics the behavior of a real computer. It manipulates data encoded as symbols*
  + *Three following components:*
    - *An I/O device that the machine uses to exchange sequence of symbols with the user. We can think of this as a screen and a keyboard for example.*
    - *A memory which contains infinitely many cells. Each memory cell can contain an arbitrary long sequence of symbols and has a unique identifier which will allow to read and write anywhere in this memory.*
    - *A central processing unit (CPU) the carries out arithmetic and logical operations (addition, subtraction, comparisons, etc.). This CPU has two registers, namely ACC and IP which can both hold any arbitrary long sequence of digits.*
      * *ACC is the accumulator and holds intermediate results*
      * *IP is the instruction pointer and contains the address where the next instruction is located.*

**

*Algorithm Correctness*

*Takeaways*

* *Takeaway 1*
  + *Provided with valid inputs, an algorithm…*
    - *is partially correct when it never terminates with an incorrect result but may not terminate.*
    - *Is totally correct when it always terminates with a correct result,*
* *Takeaway 2*
  + *A pre-condition is what we assume to hold before we execute instructions A post-condition is what we assume to hold after we executed instructions.*
* *Takeaway 3*
  + *The syntax and the semantic of the language are the basis of the deduction system we use to reason about the correctness of algorithm.*
* *Takeaway 4*
  + *Reasoning about correctness at the RAM instruction level is possible but tedious. When possible, we will reason at the pseudo-code level.*
* *Takeaway 5*
  + *When reasoning about correctness, loops are the main obstacle.*
    - *We tackle partial correctness by identifying a loop invariant, which is true, before, after and during the loop.*
    - *We tackle termination by identifying a loop variant, which is a quantity that decreases with each iteration and can only be negative after the loop.*
* *Takeaway 6*
  + *Testing is very useful in practice, but it comes with important theoretical implications:*
    - *It cannot show the absence of “bug”, only their presence*
    - *It cannot show non-termination.*
* *Takeaway 7*
  + *Proving, testing, and debugging all require a detailed understanding of the algorithm. The concepts we use to build proof directly support debugging programs:*
    - *Pre-conditions are checked explicitly at the beginning of procedures.*
    - *Invariants are checked within the procedures using assertion*
    - *Post conditions are checked in the automated tests.*
* *Functional Correctness*
  + *It must terminate at some point. In other words, the RAM must reach a HALT instruction and stops. If does not always terminate, the algorithm is, at best, partially correct.*
  + *When it terminates, it must produce a correct output for all possible valid inputs. A correct output satisfies the constraints set by the problem. A correct output satisfies the constraints set by the problem. An algorithm is thus incorrect if one can found at least one set of inputs for which the algorithm output is wrong.*

***Topic: Efficiency***

***Week: 2***

Takeaway

* Takeaway 1
  + Benchmarking describes the programs but not the underling algorithms.
* Takeaway 2
  + How we measure the time and space needed for a computation ultimately depends on the underlying computation model.
* Takeaway 3
  + The memory used in a computation boils down to the number of memory cells used to store actual data. By convention, we will only account for intermediate results, discard inputs and outputs.
* Takeaway 4
  + The time spent in a computation boils down to the time spent executing all the instructions.
* Takeaway 5
  + In practice, we do not know precisely the RAM instructions that would be generated by a compiler so we will only account arithmetic and logic operations.
* Computational Complexity
  + The performance of a single computations.
  + Estimate time and memory required by these computations using RAM and generalize higher-level code.

Algorithm analysis

Takeaways

* Takeaway 1
  + We model algorithms efficiency as a function from the input size to the measure of interest, be it time, space, or something else. This function allows us to make prediction.
* Takeaway 2
  + Best, worst, and average cases all assume a given input size. They capture additional variations due to the actual data given.
* Takeaway 3
  + The average case always requires additional assumptions that describes which inputs are the most likely. The analysis thus often relies on probabilities.
* Best, Worst and Average Cases
  + The best case, where the least number of resources is needed. That is the fastest scenario if we talk about time or the scenario that use the least memory.
  + The worst case, which requires the most resources. If we consider runtime, that is the lowest execution paths.

Orders of Growth

Takeaways

* Takeaway 1
  + We compare algorithms efficiency by comparing their efficiency models. The comparison is seldom straightforward as best, worst, and average case comparisons may not agree.
* Takeaway 2
  + Comparing the efficiency of algorithms is only meaningful when the efficiency models assume the same model of computation.
* Takeaway 3
  + We use asymptotic analysis to simplify the models obtain from algorithm analysis. Any kind of bound can possible describe any kind of scenario.
* Asymptotic Analysis
  + Upper bounds (Big-O) are families of functions thar are always greater than f given a constant factor.
  + Lower bounds (Big-Ω) are families of functions that are always lesser f given a constant factor.
  + Approximations are families of functions that resemble f given constant factors.
* Upper bounds
  + Upper bounds are functions that are always greater for large inputs. If a function f admits an upper bound g, we can think of it as f g.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

* Lower bounds
  + A lower bound is the counter part of an upper bound: This bound is a function that is “lesser” than the function of interest.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

* Approximations
  + Finally, we can also search a single function that approximates our model. This the big-Theta notation, which finds both an upper bound and a lower bound at the same time.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

* Other bounds
  + Little o – Little-o also represents a family of functions that accept an upper bound, but the definition is stricter.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

* + Little-w – Just as bi-Omega is the counter part of Big-O, little-w is the counterpart of little-o.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

***Topic: Arrays***

***Week: 3***

Takeaway

* Takeaway 1
  + A data type is a set of sequences of machine symbols that adhere to a specific representation and which we manipulate using specific programming interface.
* Takeaway 2
  + A data structure is the representation chosen for a particular data type.
* Takeaway 3
  + An abstract data type (ADT) captures the inherit relationships between the procedures that form the programming interface. It includes a domain, a set of operations, and a set of axioms that constrain the behavior of the procedures.

Abstract Data Type

* Data Type
  + Symbolic representation
  + Programming interface
* Symbolic Data Representation
  + ASCII
* Abstract Data Types
  + An abstract data type (ADT) defines the programmer’s expectations over a programming interface, irrespective of its representation.
  + Defining an ADT
    - An abstract data type defines three elements: a set of domains, a set of operations and a set of axioms. Formally, an ADT defines an algebra over the possible values of the data types.
      * The domains – describe what the ADT is about, including all the data types manipulated by the programming interface.
      * The operations – The operations are the procedures that we can use to manipulate our ADT. They are often categorized into constructor, queries and commands.
      * The Axioms – are the relationships between the procedures that the ADT supports.

Amortized Analysis

Amortized analysis is a method of analyzing the costs associated with a data structure that average the worst operations out over time. Often, a data structure has one particularly costly operation, but I don’t get performed very often. That data structure shouldn’t be labeled a costly structure just because that one operation that is seldom performed, is costly.

***Topic: Recursion***

***Week: 4***

**What is Recursion?**

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve. Recursion may be difficult to understand, but the best way to figure out how it works is to experiment with it.

**Et bilde som inneholder tekst

Automatisk generert beskrivelse**

**Recursion Example**

In this example recursion is used to add a range of numbers together by breaking it down into the simple task of adding two number:

**Example explained:**

Et bilde som inneholder tekst

Automatisk generert beskrivelseWhen the function is called, it adds parameter and returns the result. When becomes 0, the function just returns 0. When running, the program follows these steps to the figure on the right. Since the function does not call itself when is 0, the program stops and returns the result.

**Halting Condition**

Et bilde som inneholder tekst

Automatisk generert beskrivelseJust as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter becomes 0.

In this example, the function adds a range of numbers between a start and an end. The halting condition for this recursive function is when end is not greater than start.

Here is four illustrations examples of the use of recursion, providing java implementation:

* The factorial function (commonly denoted as n!) is a classic mathematical function that as a natural recursive definition.
* An English ruler has a recursive pattern that is a simple example of a fractal structure.
* Binary search is among the most important computer algorithms. It allows us to efficiently locate a desired value in data set with upwards of billions of entries.
* The file system for a computer has a recursive structure in which directories can be nested arbitrarily deeply within other directories. Recursive algorithms are widely used to explore and manage these file systems.

**The Factorial Function**

To demonstrate the mechanics of recursion, we begin with a simple mathematical example of computing the value of the factorial function. The factorial of a positive integer n, denoted n! is defined as the product of the integers from 1 to n. if n = 0, then n! is defined as 1 by convention. More formally, for any integer n 0,

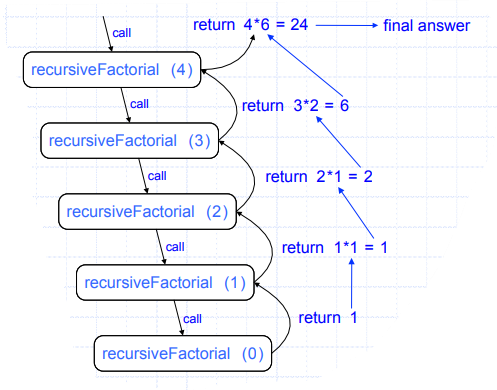
This recursive function can be formalized as:

**A Recursive Implementation of the Factorial Function**

Recursion is not just a mathematical notation; we can use recursion to design a Java implementation of the factorial function, as shown in the code under:

Et bilde som inneholder tekst

Automatisk generert beskrivelse

This method does not use any explicit loops. Repetition is achieved through repeated recursive invocations of the method. The process is finite because each time the method is invoked, its argument is smaller by one, and when a base case is reached, no further recursive call is made. We can illustrate the execution of a recursive method using a **recursion trace**. Each entry of the trace corresponds to a recursive call. Each new recursive method call is indicated by a downward arrow to a new invocation. When the method returns, an arrow showing this return is drawn and the return value may be indicated alongside this arrow. Here is an example of such a trace for factorial function:

**Drawing an English Ruler**

The English ruler pattern is a simple example of a fractal, that is, a shape that has a self-recursive structure at various levels of magnification. Ignoring the lines containing 0 and 1, let us consider how to draw the sequence of ticks lying between these lines. The central tick (at ½ inch) has length 4. Observe that the two patterns of ticks above and below this central tick are identical, and each has a central tick of length 3.

In general, an interval with a central tick length L 1 is composed of:

* An interval with a central tick length L – 1
* A single tick of length L
* An interval with a central tick length L – 1

Although it is possible to draw such a ruler using an iterative process, the task is considerably easier to accomplish with recursion. Our implementation consists of three methods, as shown in the Figure. The main method, drawRuler, manages the construction of the entire ruler. The utility method, drawLine, draws a single tick with a specified number of dashes.

**Binary Search**

Binary search is used to efficiently locate a target value within a sorted sequence of n elements stored in an array. When the sequence is unsorted, the standard approach to search for a target or exhausting the data set. This algorithm is known as linear search, or sequential search, and it runs in O(n) time (linear time) since every element is inspected in the worst case.

When the sequence is sorted and indexable, there is a more efficient algorithm. If we consider an arbitrary element of the sequence with value v, we can be sure that all elements are prior to that in the sequence with value v, and that all elements after that element in the sequence have values less than or equal to v. This observation allows us quickly “home in” on a search target using a variant of the children’s game “high-low”. We call an element of the sequence a candidate if, at the current stage of the search, we cannot rule out that this item matches the target. The algorithm maintains two parameters, low and high, such that all the candidate elements have index at least low and at most high. Initially, low = 0 and high = n – 1. We then compute the target value to the median candidate, that is, the element with index:

We consider three cases:

* If the target equals the median candidate, then we have found the item we are looking for, and the search terminates successfully.
* If the target is less than median candidate, then we recur on the first half of the sequence, that is, on the interval of indices from low to mid – 1.
* If the target is greater than the median candidate, then we recur on the second half of the sequence, that is, on the interval of indices from mid + 1 to high.

An unsuccessful search occurs if low > high, as the interval [low, high] is empty. This algorithm is known as binary search. We give Java implementation in the figure under, and an illustration of the execution of the algorithm in the figure. Whereas sequential search runs in O(n) time, the more efficient binary search runs in O(log n) time. This is a significant improvement, given that if n is 1 billion, log n is only 30.

Et bilde som inneholder tekst

Automatisk generert beskrivelse

In the code earlier we consider three cases as explained before:

* If the target equals data[mid], then we have found the target.
* N if target < data[mid], then we recur on the first half of the sequence (n – 1).
* N if target > data[mid], then we recur on the second half of the sequence (n +1).

Et bilde som inneholder bord

Automatisk generert beskrivelse

**File Systems**

Modern operating systems define file-systems directories (also called “folders”) in a recursive way. Namely, a file system consists of a top-level directory, and the content of this directory consist of files and other directories, which in turn can contain files and other directories, and so on. The operating system allows directories to be nested arbitrarily deeply, although by necessity there must be some base directories that contain only files, not further subdirectories.

***Topic: Hashing***

***Week: 5***