Week 1 – Data Structures and Algorithms

**Elegant program** has an algorithm that is:

1. Easy to understand, code, and debug
2. Efficiently using computer’s resources

**Asymptotic analysis**

* Evaluates the efficiency of an algorithm
* Gives a way to define the inherent difficulty of a problem

**Data structure** = (in the most general sense) any data representation and its associated operations. Even an integer or floating-point number stored on the computer can be viewed as a simple data structure.

A solution is said to be **efficient** if it solves the problem within the required resource constraints.

**Data-centered** view of program design process:

1. Analyse the problem to determine the basic operations that must be supported, such as inserting a data item into the data structure, deleting a data item for the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation
3. Select the data structure that best meets these requirements

Resource constraints on certain key operations normally drive the data structure selection process. **Issues relating to the relative importance** of these operations are addressed by the following questions:

1. Are all data items inserted into the data structure at the beginning, or are the insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically get by with simpler data structures to get an efficient implementation, while dynamic applications often require something more complicated.
2. Can data items be deleted? If so, this will probably make the implementation more complicated.
3. Are all data items processed in some well-defined order, or is search for specific data items allowed? “Random access” search generally requires more complex data structures.

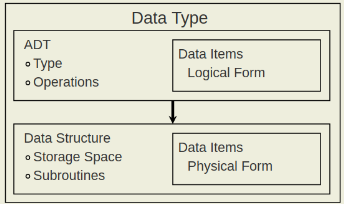
**Problem** = task to be performed it is best thought as a function in the mathematical sense (matching between inputs (the domain) and outputs (the range)). The values making up and input are function ***parameters***. A specific value for the parameters is called an ***instance*** of the problem.

**Algorithm** = method or a process followed to solve a problem. A given algorithm solves only one problem. Something can only be called an algorithm if it has the following properties:

1. It must be correct. It must compute the desired function, converting each input to the correct output.
2. It is composed of a series of concrete steps, meaning, the action described by that step is completely understood and doable by the person or machine that must perform the algorithm. Each stop must also be doable in a finite amount of time.
3. There can be no ambiguity as to which step will be performed next.
4. It must be composed of a finite number of steps.
5. It must terminate.

**Program** = instance or concrete representation of an algorithm. Terms algorithm and program are often used interchangeably but by definition an algorithm must provide sufficient detail that it can be converted into a program when needed.

**Type** = collection of values. Have both logical form and physical form. In terms of an ADT, the definition of the data type is its logical form, and the implementation is its physical form.



* **Simple type** = values contain no subparts, eg. an integer
* **Aggregate/composite type** = containing several pieces of information
* **Abstract data type (ADT)** = specification of a data type within some language, independent of an implementation. The interface for the ADT is defined in terms of a type and a set of operations on that type. The behaviour of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access (=**encapsulation**)

**Data item** = piece of information or a record whose value is drawn from a type. A member of a type.

**Data type** = type together with a collection of operations to manipulate the type. Integer is a member of the integer data type. Addition is an example of an operation on the integer data type.

**Array** = Contiguous block of memory locations, where each memory location stores one fixed-length data item. By this meaning it is a physical data structure. However, an array can also mean a logical data type composed of a (typically homogenous) collection of data items, with each data item identified by an index number.

**Data structure** = implementation for an ADT. In an object-oriented language, an ADT and its implementation together make up a **class**. Each operation associated with the ADT is implemented by a **member function** or a **method**. The variables that define the space required by a data item are referred to as **data members**. An **object** is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

Data structure often refers to **data stored in a computer’s main memory.**

**File structure** = refers to the organization of data on peripheral storage, such as a disk drive or CD.

Week 2 – Comparing Algorithms

When estimating an algorithm’s performance, the **number of basic operations** required by the algorithm to process and input of a certain size is of primary consideration. Size is often the number of inputs processed. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations but summing the contents of an array containing n integers is not, because the cost depends on the value on n (i.e., the size of the input).

Example

Text, letter

Description automatically generated

Size of the problem is A.length

The basic operation is to compare an integer’s value to that of the largest value seen so far.

It is reasonable to assume that it takes a fixed time to do one such comparison regardless of the value of the two integers or their position in the array.

Because most important factor affecting running time is normally size of the input, for a given input size **n** we often express **T** to run the algorithm as a function of n, written as **T(n)** always assuming the value is a non-negative.

**c** being the amount of time required to compare two integers in function largest, it does not matter to us what the precise value of **c** might be. The total time to run largest is therefore approximately **cn**, because we must make **n** comparisons, with each comparison costing **c** time. Expressed by the equation, the largest-value sequential search algorithm has a running time of **T(n)=cn.**

**Growth rate** = rate at which the cost of the algorithm grows as the size of its input grows.

Diagram

Description automatically generated with medium confidenceA picture containing chart

Description automatically generated

When analysing an algorithm normally we are not interested in the best case, because it might happen only rarely and generally be too optimistic for a fair characterization of the algorithm’s running time. If the best-case has a high chance of occurring, the **best-case analysis** is useful. The Quicksort algorithm can take advantage of the best-case running time of Insertion Sort to become more efficient.

The advantage of analysing **worst-case** is that you know for certain that the algorithm must perform at least that well. This is important for real-time applications.

**Average-case analysis** is not always possible to perform. It requires that we understand how the actual inputs to the program, and their costs, are distributed with respect to the set of all possible inputs to the program. For example, the sequential search algorithm on average examines half of the array values. This is only true if the element with value **K** is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does not necessarily examine half of the array values in the average case.

* Incorrect assumptions about data distribution can have disastrous consequences on a program’s space or time performance. Unusual data distributions can also be used to advantage, such as is done by self-organizing lists.

Regardless of the algorithm’s growth rate: Constant factors never affect the relative improvement gained by a faster computer.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation.

For these reasons, **we usually ignore the constants** when we want an estimate of the growth rate for the running time or other resource requirements of an algorithm. This simplifies the analysis and keeps us thinking about **the most important aspect: the growth rate**. This is called **asymptotic** **algorithm** **analysis**. To be precise, asymptotic analysis refers to the study of an algorithm as the input size “gets big” or reaches a limit (in the calculus sense).

In some rare cases, where comparing algorithms meant to run on small values of **n**, the constant can have a large effect.

**Upper bound** = highest growth rate that the algorithm can have, **big-Oh notation**. If the upper bound for an algorithm’s growth rate (for, say, the worst case) is (f(n)), then we would write that this algorithm is “in the set O(f(n)) in the worst case”.

Table

Description automatically generated

Simplifying rules

Text

Description automatically generated

The first rule: if some function g(n) is an upper bound for cost function, then any upper bound for the function g(n) is also an upper bound for the cost function.

The second rule: any multiplicative constants can be ignored in equations when using big-Oh notation.

The third rule: given two parts of a program run in sequence, only the more expensive part needs to be considered.

The fourth rule: If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place.

**Lower bound** = used to describe the least amount of a resource that an algorithm needs for some class of input. Denoted by Ω, pronounced “big-Omega”.

**Theta-notion** = When the upper and lower bounds are the same within a constant factor. Θ (big-Theta).

**Space/time tradeoff** **principle** = one can often achieve a reduction in time if one is willing to sacrifice space or vice versa. Many programs can be modified to reduce storage requirements by “packing” or encoding information. “Unpacking” or decoding the information requires additional time.

**Disk-based space/time tradeoff principle** = the smaller you can make your disk storage requirements, the faster your program will run. This is because the time to read information from disk is enormous compared to computation time, so almost any amount of additional computation needed to unpack the data is going to be less than the disk-reading time saved by reducing the storage requirements.

Instead of growth rate and running time, the term **computational complexity** is commonly used when describing how efficient the algorithm is.

Data Structures and their qualities

**Hash Table**

* Allows for extremely fast exact-match search
* Records can be modified quickly when the modification does not affect its space requirements
* Supports efficient insertion of new records
* Deletion can be supported efficiently but too many deletions lead to some degradation in performance for the remaining operations
* Can be reorganized periodically to restore peak efficiency
* Can not perform efficient range queries

**Binary search tree**

* Supports large databases
* Supports efficient insertion and deletion of data records
* Supports range queries