Computer Science Book

Written by Bryce Summers (BryceSummers.com)

Last Updated: July 20, 2015

Contents

I Fundamentals **9**

1 Mathematical Objects **11**

2 Fundamental Distinctions **13**

- 2.1 Chapter Overview . . . 13
- 2.2 Descriptions . . . 13
- 2.3 Interfaces . . . 13
  - 2.3.1 Types . . . 14
  - 2.3.2 Abstract Data Types . . . 14
- 2.4 Implementations . . . 14
  - 2.4.1 Data Structures . . . 14
- 2.5 Analysis . . . 15
- 2.6 ”Cooking” Example . . . 15
  - 2.6.1 Scenario . . . 15
  - 2.6.2 Description . . . 15
  - 2.6.3 Interface . . . 15
  - 2.6.4 Implementation . . . 15
  - 2.6.5 Analysis . . . 16

3 Memory, Data, and Interpretation **17**

- 3.1 Overview . . . 17
- 3.2 Bits . . . 17
- 3.3 Memory . . . 17
- 3.4 Byte . . . 18
- 3.5 2’s Complement . . . 18
- 3.6 Booleans . . . 19
- 3.7 Pointers . . . 19
- 3.8 Structures . . . 19
- 3.9 Pairs . . . 19
- 3.10 Static Memory Allocation . . . 19
- 3.11 Dynamic Memory Allocation . . . 20
- 3.12 Memory Deallocation . . . 20
- 3.13 Chars . . . 20
- 3.14 Strings . . . 20

4 General Definitions 21
- 4.1 Imperative Programming . . . 21
- 4.2 Purely Functional Programming . . . 21
- 4.3 Declarative Programming . . . 22
- 4.4 Object Oriented Programming . . . 22
- 4.5 Algorithm . . . 22
- 4.6 Computation and Procedures . . . 22
- 4.7 Deterministic vs. Randomized . . . 22
- 4.8 Sequential versus Parallel . . . 22
- 4.9 Recursion . . . 22

5 Data Structure Definitions 23
- 5.1 Overview . . . 23
- 5.2 Ephemeral vs. Persistant . . . 23
- 5.3 Packaged vs. Unpackaged. . . . 23
- 5.4 Indexed vs. Non-Indexed . . . 24
- 5.5 Finite vs. Infinite . . . 24
- 5.6 Fixed Size vs. Resizable . . . 24
- 5.7 Ordered vs. Unordered . . . 25
- 5.8 Enumeratable and Iterable . . . 25
- 5.9 Transversals . . . 25
- 5.10 Eager vs. Lazy . . . 26
- 5.11 Simple vs. Performant vs. Efficient . . . 26
- 5.12 Amortized versus Non Amortized . . . 26
- 5.13 Endogeneous vs. Exogeneous . . . 27
- 5.14 Null vs. Node terminals . . . 27
- 5.15 Deterministic vs. Randomized . . . 27
- 5.16 Sparse vs. Dense . . . 28
- 5.17 Concurrency safe. . . . 28
  - 5.17.1 Motivation . . . 28
  - 5.17.2 Mutual Exclusion . . . 28
  - 5.17.3 Lock-Free . . . 28

6 Analysis 29
- 6.1 Overview . . . 29
- 6.2 Correctness Analysis . . . 29
  - 6.2.1 Mathematical Proof . . . 29
  - 6.2.2 Empirical Testing . . . 30
  - 6.2.3 Formal Methods . . . 30
- 6.3 Resource Analysis . . . 31
  - 6.3.1 Models . . . 31
- 6.4 Exact Analysis . . . 32
- 6.5 bigO . . . 33
  - 6.5.1 Overview . . . 33
  - 6.5.2 Formal Definitions . . . 33
- 6.6 Asymptotic Analysis . . . 33

- 6.6.1 Overview . . . 33
- 6.6.2 The Adversary. . . . 34
- 6.6.3 Worst Case . . . 34
- 6.6.4 Average Case . . . 35
- 6.6.5 Best Case . . . 35
- 6.6.6 Expected Case . . . 35
- 6.7 Amortized Analysis . . . 35
  - 6.7.1 Motivation . . . 35
  - 6.7.2 Banker's Method . . . 36
  - 6.7.3 Potential Functions . . . 37
  - 6.7.4 Potential Method . . . 38
- 6.8 Analysis of Parallel Algorithms . . . 39
  - 6.8.1 work and span . . . 39
  - 6.8.2 Efficiency vs. Speedup . . . 39
- 6.9 Analysis of Dynamic Programming . . . 39
- 6.10 Analysis of Recursive Procedures . . . 39
  - 6.10.1 Brick Method . . . 39
- 6.11 Arithmetic Intensity . . . 40

II Abstract Data Types and Data Structures 41

7 ADTs 43
- 7.1 Overview . . . 43
- 7.2 Function . . . 43
  - 7.2.1 Interface . . . 43
  - 7.2.2 Finite Functions . . . 43
  - 7.2.3 Infinite Functions . . . 43
  - 7.2.4 Real World Examples. . . . 44
- 7.3 Stack . . . 44
  - 7.3.1 Interface . . . 44
  - 7.3.2 Real World Examples. . . . 44
- 7.4 Queue . . . 44
  - 7.4.1 Interface . . . 44
  - 7.4.2 Real World Examples. . . . 44
- 7.5 Deque . . . 44
  - 7.5.1 Interface . . . 45
  - 7.5.2 Real World Examples. . . . 45
- 7.6 Priority Queue . . . 45
  - 7.6.1 Interface . . . 46
  - 7.6.2 Real World Examples. . . . 46
- 7.7 Set . . . 46
  - 7.7.1 Interface . . . 46
  - 7.7.2 Real World Examples. . . . 46
- 7.8 Multi-Set . . . 46
  - 7.8.1 Interface . . . 46

7.8.2 Real World Examples. 46
7.9 Relations 46
7.9.1 Interface 46
7.9.2 Real World Examples. 46
7.10 Trees 46
7.10.1 Interface 46
7.10.2 Full Trees 46
7.10.3 Complete Trees 46
7.10.4 Binary Trees 46
7.10.5 Binary Search Trees 46
7.10.6 Balanced Binary Search Trees. 46
7.10.7 Heaps 46
7.10.8 Tree Transversals 46

8 Data Structures 47
8.1 Overview 47
8.2 Arrays 49
8.2.1 Fixed Size Arrays 49
8.2.2 Strings 49
8.3 Lists 49
8.3.1 Endogenous vs. exogenous 49
8.3.2 Single vs. Double 49
8.3.3 Linear vs. Circular 49
8.4 Unbounded Arrays (UBA) 49
8.4.1 Overview 49
8.4.2 Amortized Analysis. 49
8.5 Stacks 49
8.5.1 Interface 49
8.5.2 Array Implementations 49
8.5.3 List Implementations 49
8.5.4 Persistant Stacks 49
8.5.5 Max/Min Stacks 49
8.5.6 Function Call Stack 49
8.6 Queues 49
8.6.1 Interface 49
8.6.2 List Implementation 49
8.6.3 Array Implementation 49
8.6.4 Fairness 49
8.6.5 Pipelining 49
8.6.6 Queuing Theory 49
8.6.7 Two Stacks 49
8.7 Dequeues 49
8.8 Heaps 49
8.9 Union Find Structures 49
8.10 Hash Tables 49
8.11 Maps 49

8.12 Bloom Filters 49
8.13 Link-Cut trees 49
8.14 Binary Search Trees 49
8.15 D-ary Trees 49
8.16 Graphs 49
8.17 Cycle 49
8.18 Streams 49
8.19 Tries 49
8.20 Suffix Tries 49
8.21 Binary Descision Diagrams 49
8.22 Bit Sets 49

III Fixed Size Data Structures. 51

IV Algorithms 53

9 Sorting Algorithms 55
9.1 Selection Sort 55
9.2 Insertion Sort 55
9.3 Merge Sort 55
9.4 Quick Sort 55
9.5 Radix Sort 55
9.6 Bucket Sort 55
9.7 Binary Search Tree Sort 55
9.8 Heap Sort 55

10 Search Algorithms 57
10.1 Collection Searches 57
  10.1.1 Linear Search 57
  10.1.2 Binary Search 57
10.2 Graph Searches 57
  10.2.1 BFS 57
  10.2.2 DFS 57
  10.2.3 A* Search 57
  10.2.4 Weighted A* Search 57
  10.2.5 A* recomputations 57
10.3 Game Tree Searches 57
  10.3.1 MiniMax 57
  10.3.2 AlphaBeta Pruning 57
  10.3.3 Iterative Deepening 57

11 Common Algorithms for Linear Collections of Numbers 59
  11.0.4 Partitioning 59
  11.0.5 Linear Reduce Operations. $\mathcal{O}(n)$ work, $\mathcal{O}(log(n))$ span. 60
  11.0.6 Median (Middle) / kth largest 62

12 Parallel Algorithms **65**
12.1 Inclusive Scan 65
12.2 Addition 65

V Inefficient and Obscure Algorithms and Data Structures **67**

Part I

Fundamentals

Chapter 1

Mathematical Objects

To fully appreciate the material covered in this book, it is important that you are familiar with many mathematical objects, including sets, functions, orderings, numbers, relations, graphs, trees. At the present time, most purely mathematical material will not be explicity developed withing this book. The Author may write material regarding these subjects that will eventually be found at BryceSummers.com

Chapter 2

Description, Interface, Implementation, and Analysis

2.1 Chapter Overview

When thinking like a computer scientist, we often want to mimic the workings of reality, especially with regards to procedures or processes that modify the state of reality. Before we can effectivly mimic reality it is important to explicitly list several distinctions or perspectives on how a facet of reality may be perceived. In this Chapter, we will use the proccesss of a woman asking her husband to cook dinner as an example to help illustrate the distinctions that we will make. By understanding these distinctions now, you will be able to ease your mental workload for the rest of your computer science related life.

2.2 Descriptions

A description is a somewhat informal way of communicating. For example, a ceo at a large corporation use descriptions to describe their goals to their colleagues, where the colleagues would be responsible for acting on the intent and getting the desried results. Often Computer Scientists will interact with non computer scientists, such as the media, clients, users, and potential sponsers at the description level to convey the motivation behind their work. The description level could also be though of as the idea level.

2.3 Interfaces

An interface is a formal mathematical **specification** for how a given system should act. If the user of the system behaves in a certain way, the specification should provide a detailed description of what behavior the system should exhibit. Often interfaces are used to get multiple independantly produced systems to work together such as with the APIs that are becoming better known by the general public. Interfaces are also used to rigourously outline the workings of a describable idea in a form that can be understood at a later point in time. Many humans think of great ideas, but forget them because they did not write them down in sufficient detail that they could be recreated. Once an idea is written in such a form that it can be understood in the same way by multiple independant humans, it may be called a proper specification.

Interfaces are the primary means by which information and details may be abstracted from the user of a system. (This is why the development of knowledge is of linear complexity and not worse!)

2.3.1 Types

In Computer Science, a type is a way of labeling groups of objects. For example, if I added two entities of type number, I would expect the result to also be a number, as apposed to a letter, chicken, or asparagus. We will be using types to describe specific objects and how objects of given types interact with each other. A Type can be thought of as a type of interface that describes what information and procedures the object may be used for.

2.3.2 Abstract Data Types

An Abstract data type (ADT) is an interface that describes the behavior of a given type. ADTs are commonly associated with mathimatical objects, such as functions, trees, graphs, sets, etc. Abstract Data Types are implemented by Data Structures, which we will discuss in the Implementation section. The main idea is that Abstract Data Types describe **What** a system should do and Data Structures specify **how** a machine acomplishes the behavior specified in th ADT.

2.4 Implementations

An implementation is a specification of the particular details of **how** a system will peform a given undertaking. An implementation is said to **implement** a given interface if it corectly provides detailed procedures that satisfy the expected behavior as described in the interface.

For example, imagine a lamp. The **interface** for the lamp is that it can be turned on and off, where it produces light in the on state and does not produce light in the off state. The **implementation** on the other hand is all of the details abstracted from the user, such as the chemical reactions used to produce light, the routing of electricity, the methods of heat dissipation, the efforts to properly manage the bightness of the light, the materials chosen in the light's construction, etc. All of these details are not readily apparent, nor neccessary for the average user. These implementation details consist of all of the design descisions that went into how the implementation ultimatly performs as specified by an interface.

2.4.1 Data Structures

A Data structure is a very important concept in the realm of programming, wherebye an abstract data type is implemented correctly by a given program that can be executed on an actual computational machine. In this book, we will discuss quite a few data structures. They are mostly used to abstract common data management tasks that are used as subroutines in complex programs and help reduce the cognitive load on programmers and Computer Scientists interested in the analysis of algorithms.

2.5 Analysis

Analysis is the proccess by which we verify that a given implementation correctly implements a given interface. For example, banks rely on analysis to ensure that their monetary systems are secure from unlawful activity such as malicious programs that steal money.

In addition to purely correctness focused analysis, is the analysis of the amount of resources used for a particular task. Important resources in the realm of computer programming include the time a computation takes, the amount of memory a computation requires, and the amount of energy consumed. Analysis of this form is very important, because many correct algorithms are not useful if they cannot be run in a reasonable amount of time with the memory capabilities of an actual machine.

We will have a much more detailed discussion of different types of analysis in chapter 6.

2.6 "Cooking" Example

To conclude this chapter, we will properly investigate how these distinctions can be used to view a non technical scenario. For a thorough technical investigation of Arrays using these distinctions, please see Section 8.2.

2.6.1 Scenario

Please consider the common situation in which a woman asks her husband to cook dinner.

2.6.2 Description

The task is that of cooking dinner. The husband and wife each have likely eaten countless dinners in their life and both understand that making dinner is the proccess of producing food in an edible and pleasing form that may be consumed in the evening. The Wife merely needs to declare her intent in a conscise fashion using the word dinner for the husband to understand the intent. This is possible through the power of a shared definition.

2.6.3 Interface

In this example, the wife is said to be using her husband's interface. The husband's interface lists all of the actions that a husband is capable of performing such as that of preparing dinner. The details of how the husband decides on dinner, prepares it, cooks it, and the timings of his actions are abstracted from the wife so she does not need to worry about them. If the wife wanted more information, then we could extend the husband's interface to allow for him to answer questions and provide more details.

2.6.4 Implementation

The husband's implementation of the making dinner action would be the details and decsions that he makes, including a procedure for deciding whether a hamburger or a salad should be served, when to set the table, which beverages to serve, whether to serve dinner inside or outside, and all other relevant details pertaining to the action.

2.6.5 Analysis

To Analyze whether the husband makes an appropiate dinner in an appropiate manner would require us to look at the expectations of the wife and see if they are met by the husband's performance of the action, and in general by proving that the husband will always prepare the meal appropiatly given his internal descision procedures. In the real world, humans are not very likely to err at points in their lives, so it may be appropriate to only determine whether a paricular instance of the husband performing the action is correct or we could make statements about the probability of the husband succeeding at the action.

Chapter 3

Memory, Data, and Interpretation

3.1 Overview

In this section we will be developing the basics of how information is represented and interpreted on a machine. These ideas are of paramount importance to the Computer Scientists' and programmers concptual model or the abstract practical state of a computer.

3.2 Bits

How does one differentiate between two things? In the real world, we are constantly differentiating between ideas, objects, people, etc every minute of our lives. A computer is essentially a machine that is able to differentiate and act upon a huge, but finite number of states. The basic building block of state information is the **bit**, which is defined to take on two distinct states, which we will call

$$1 \texttt{ and } 0$$

Depending on the context the states of a bit can be called alternative names such as *true* and *false*, or *on* and *off*, but what is important is that the states are *different.* Bits become more and more powerful as they work *together* as you will see in the following sections.

3.3 Memory

Actual Computers are capable of representing quite a few more than 2 states and they acomplish this representational feat by combining bits into larger structures. If 1 bit can represent 2 states, then if two of them are combined, then together they can represent $2 \cdot 2 = 4$ states. More explicitly, 2 bits can represent the following states:

$$00, 01, 10, \texttt{and } 11$$

It is important to note the these bits must be interpretted as acting together, because with an alternative interprettation the bits could be seen as independant unrelated entities. We will discover that in order to effectivly represent, modify, and interpret the states of a machine, it is very important to both think of combinations of bits and to think of a separation between different groups of bits. **Memory** is an array of bits stored on a computer that represents the entirety of the current state of the computer. One can think of it as a very long line of boxes, each of which contains a bit in one of its two states, as in Figure 3.1 the boxes are called **memory locations**.

0	1	0	1	0	1	0	0	0	0	1	0	1	0	···

Table 3.1: Memory Array

3.4 Byte

A **Byte** is a group of 8 bits, which is capable of representing 2^8 different states. If we interpret each state as a binary number composed of bits representing binary digits, then bytes range from the state 0 to $2^8 - 1 = 255$. Please see Figure 3.2 for an representation of the number 100. Other primitive number sizes such as integers, shorts, and longs are represented by combinations of multiple bytes. Primitive number are the basic finitie sized numbers that are used as fundamental states in the execution of programs.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	0	1	0	0

Table 3.2: A Byte Representing the number 100. $(2^6 + 2^5 + 2^2)$

3.5 2's Complement

Negative numbers are represented by mapping half of the possible states of a primitive number of size N to negative numbers between -1 and 2^{N-2} inclusive. The particulars of this mapping are up to the implementation, but a popular mapping is what is known as 2's complement arithemetic, where the numbers are represented such that when any two numbers are interpreted as positive numbers, added together, and then remapped to appropiate positive or negative numbers, then the result is correct modulo 2^N. One way to convert from a positive number to its negation is by flipping, i.e. changing the state, of every bit and then incrementing the number by 1 using base 2 arithemetic. Please see Figure 3.3 for an example of the conversion of 100 to -100. Of particular note, please see that $100 + (-100)$ overflows to 0!

100	0	1	1	0	0	1	0	0
~ 100	1	0	0	1	1	0	1	1
-100	1	0	0	1	1	1	0	0

Table 3.3: Bit representations of 100, bit flipped 100, and -100 ($=\sim 100 + 1$).

3.6 Booleans

Booleans are a primitive type supported by many programming languages that can take on 2 states, *true* and *false*. They are like bits, except for memory alignment reosons are usually implemented as byte sized or greater sections of memory that are restricted to only take on 2 states. It may be more memory and time efficient to use the boolean operations on larger types such as integers when trying to represent arrays of booleans in software, such as when using bit sets. Booleans are used to represent the results of any question or function that returns yes or no answers.

3.7 Pointers

Every memory location has an address that the computer can use to locate the memory location, much like mailmen use our home addresses to located our houses. A pointer is a primitive type that represents an adress. On a 32 bit machine, pointers take up 32 bits and can therefore address up to 2^{32} distinct memory locations, whereas 64 bit machines have pointers that are 64 bits wide. Pointers are used to form connections between disparate groups of data. One can say that a region of memory representing a pointer p **points to** the region of memory that is addressed by the value of p.

FIXME : Put in a simple diagram with an 4 bit computer.

3.8 Structures

Structures are types formed by placing several different types in a predetermined sequence. For instance I might make a structure containing 3 chars to represent a person's three main initials or a structure containing a pointer to a structure that represents a university and an integer representing the current school year. Structures will be very important to packaging data types into useful forms and allow for the creation of metaphors that ease the cognitive load on the programmer. Structures are called objects in the paradigm of object oriented programming.

3.9 Pairs

Pairs are simple structures that associate two typed objects. They are often used to represent simple utility objects, to represent connections, or to represent simple not quite primitive objects.

3.10 Static Memory Allocation

Also known as the Stack.

3.11 Dynamic Memory Allocation

Also known as the Heap.

3.12 Memory Deallocation

Garbage Collected vs. Manual Memory Deallocation

3.13 Chars

Also known as characters, chars are regions of memory the size of a byte that can represent states for various letters and other symbols used in the representation of text. As long as the computer interprets the charters state in the same way each time when it encodes, decodes, and displays the characters to the user on a visual display, it does not matter which bit states coorespond to which character. This has lead to many different encodings for sets of characters including the Ascii encoding and the Unicode encoding. Programmers type using english characters, instead of sequences of 1s and zeros, but the machine views their programs as sequences of characters.

3.14 Strings

Strings are implemented by a sequence of characters followed by a terminal character. The reason is that strings vary from the trivial string of size 0 to strings containing the entire works of William Shakespeare. If the languages were to package size information for a strings, then they would need to add a field to the beginning of every string that is of sufficient bit size to store the number cooresponding to the largest possible string. This would increase the amount of size required for all strings in existance and would not be very useful as most operations, such as printing and concatenation on strings will iterate through all characters anyways. If the size of the string is desired by the programmer, then it can be found in linear time by iterating through all of the characters until the terminal character is found and counting how many characters are seen. Strings do have an overhead of one character, because they need to allocate space for and store the terminal character. Because strings are implemented in this fashion, they require the presense of a dynamic memory allocator.

Chapter 4

General Definitions

4.1 Imperative Programming

Imperative programming is a programming paradigm wherebye programmer directly command (hence "imperative") the computer to modify regions of data to perform a computation. They do this by writing lines of code that act as instructions for how the computer should proceed at any point in time, including which instructions to execute next, which sections of memory to modify, and how to group lines of code into modular tasks called functions.

4.2 Purely Functional Programming

Purely Functional Programming is a programming paradigm whereby programmers specify Types and operations to map between types. The programmer does not give the computer instructions, but rather informs it as to how the decomposition between types work. A computation is thought of as the evaluation of a root function that may have many sub functions used in determining its output type. A hallmark of Purely Functional Programming is that their are no side effects, all input values to functions are unmodified, and all output values to functions are determined only as a function of the input values.

4.3 Declarative Programming

4.4 Object Oriented Programming

4.5 Algorithm

4.6 Computation and Procedures

4.7 Deterministic vs. Randomized

4.8 Sequential versus Parallel

A **sequential** computation is one that is run on one processor.

A **parallel** computation is one whose work is distributed to more than one processor. Paralllel computations can only be performed if the work of the computation can be decomposed into independant pieces or pieces that may be processed concurrently. Please see section 6.8 for a discussion of the analysis of parallel algorithms. Please see 12 for a discussion of parallel algorithm design and examples.

4.9 Recursion

Chapter 5

Data Structure Definitions

5.1 Overview

When describing Data Structures, we will use terminology to help us to explicity defferentiate between the properties of various structures. You may wish to refer to Chapter 7 for descriptions of Abstract data types that may be used in this section to provide examples of the properties. You may also with to refer to Chapter 8.2 for a discussion of arrays, Chapter 8.3 for a discussion of lists, and Section 3.14 for a discussion of strings.

5.2 Ephemeral vs. Persistant

These terms are also know as mutable vs. immutable or Dynamic vs. Frozen repectively.

Let S and S' be the states of a data structure before and after an operation respectivly.

A data structure is said to be **ephemeral** if succeeding operations on S are not guranteed to be correct. Ephemeralness is a hallmark of data structures that permit operations that mutate the internal state of a structure, such as an oepration that replaces a the value of a byte with another byte. Traditionally, ephemeral data structures are associated with imperative programming, but they may be programmed in functional languages that are not pure in the sense that they allow mutations or side effects.

A data structure is said to be **Persistant** if both S and S' may be correctly used in future operations. Another way of saying this is that the data structure stores all of its previous states. Most well designed persistant data structures are capable of maintaining their history of states very memory and time efficiently and may be used in applications that require the same data to be used in different branches of execution.

5.3 Packaged vs. Unpackaged.

A data structure is said to be **Packaged** if it exists with some metadata or external interpretational knowledge that specifies information about the structure such as the locations in memory of all of its particular parts or how large the structure is. For instance, bytes can be viewed as packaged, because externally we know that all bytes are of size 8 bits and take up the contiguous region of memory starting at their address and extending for 8 bytes in order of increasing memory address. Arrays are also packaged, because they exist with a given element size and a total number of bytes. They only work because we have these pieces of information.

A data structure is said to be **Unpackaged** if it exists without rigourous structural data. When viewing an unpackaged structure, it is impossible to determine all of the properties of the structure merely upon inspection. Most Programming languages implement strings as unpackaged structures, because they are defined as a sequence of char data fields followed by a terminal character instead of explicitly store the size of the string at the beginning. Please see the section 3.14 for more information regarding strings.

5.4 Indexed vs. Non-Indexed

A data structure of size S is said to be **Indexed** if every natural number between 0 and $S - 1$ cooresponds to a distinct element that is contained within the data structure and an efficient lookup operation exists returns the proper element given one of these natural numbers. In this case, we generally consider efficient lookup functions to be those that run in constant time. The canonical example of an indexed collection would be arrays. I suppose a $\mathcal{O}(log(S))$ function would be efficient for an Augmented tree structure with stored tree size fields.

A data structure is said to be **Non-Indexed** if it is not said to be Indexed.

5.5 Finite vs. Infinite

A data structure is said to be of **Finite** size their is a finite bound to the number of elements stored in the structure.

A data structure is said to be of **Infinite** size if the structure theoretically represents a collection of an infinite amount of elements. One example of an infinite data structure would be a stream that represents the natural numbers and gives the user a incrementally larger number every time. Although it never explicitly stores all of its elements in memory, it is able to represent all of its elements through its mathematical procedure. Another property of infinite structures is that they are often iterable, but not entirly enumeratable. Infinite data structures are usually associated with explicit transformation functions and mathematical definitions.

Please see section 8.18 for more information on streams and 10.3 for more information about essentially infinite game trees.

5.6 Fixed Size vs. Resizable

A data structure is said to be of **Fixed Size** if it requires a constant amount of memory throughout its lifespan. Arrays and most primitive data types are of fixed size as well as some specilized versions of many of the more complex data structures. Electrical and Comptuer Engineers often use fixed size data structures for applications that lack a dynamic memory allocator and have constrained memory resources.

A data structure is said to be **Resizable** if it may enlarge of decrease its total memory requirements throughout its lifespan. Resizable structures require the presense of a dynamic memory allocator

Please see sections 3.10 and 3.11 for discussions of static and dynamic memory allocation.

5.7 Ordered vs. Unordered

A data structure with size n is said to be **Total ordered** if there is a total order associated with the elements contained withing the structure and all of the elements are arranged in the structure in a way that respects this ordering. The presense of an ordering enables data structures to provide $\mathcal{O}(log(n))$ time complexity searchs through the use of binary search, whereas the fastest search time for an ordered collection is a linear search. Please see sections 10.1.1 and 10.1.2 for more information about basic collection searching.

A data structure is said to be **Ordered** if there is a well defined transversal for the data structure. In other words all iterations through the structure are guranteed to yield the same partial enumerations. Every element is also guranteed to be found at well defined locations throughout the structure. Most data structures that are ordered such as arrays and lists will have the oldest added elements nearer to the start of the structure than the youngest added elements.

A data structure is said to be **Unordered** if there there is no well defined transversal through the data structure and elements are not guranteed to be found at consistant locations. The primary example of such as structure would be a set, which cares only about inclusion rather than the relative locations of the elements.

5.8 Enumeratable and Iterable

These terms can mostly be used interchangably, but I have decided to define them in the context of this book in subtly different ways.

A data structure D is said to be **Enumeratable** if it can be converted into an indexable data structure such as an array. Naturally this translation would ideally be done in linear time. The output structure may be called the **enumeration** of D

A data structure D is said to be **Iterable** if it can provide an iterator that cooresponds to the current collection of elements inside of the structure. The order in which the elements are iterated through by the iterator coorespond to their positions in a **partial enumeration** of D.

Both the enumeration and partial enumeration of D will be ordered if and only if D is an ordered data structure. The enumeration of infinite sized data structures is infinite, so it may not be explicity produced on an actual finite memory machine. This is why infinite data structures may be used practically with lazy evaluation or streaming based algorithms or those that are of a mathematical nature that require properties of the sturcture, but not the structure itself.

5.9 Transversals

A **Transversal** of a data structure is the procedure for determining the order of its enumeration.

The most common transversal for an indexed data structure such as an array is to iterater from the lowest indice to the highest indice. There are countless other possible transversals possible, such as the reverse transversal or a transversal that visits every even element first and then visits the odd elements. You will find that transversals will be implicitly mentioned many times throughout this book, so please keep your eyes peeled.

Please see section **??** for a discussion of more complicated tree transversals.

5.10 Eager vs. Lazy

A data structure is said to be **Eager** if all computations are explicitly carried out in full at the moment its operations are called.

A data structure is said to be **Lazy** if it allows for the data structure to store operational requests and delay executing them until the user makes a request that requires an element to be revealed from the structure. For example, consider an implementation of a stream that represents all of the natural numbers. If the stream was implemented eagerly, then it would spend an infinite amount of time multiplying all of the natural numbers by 2, which would not be practical. If instead we implemented the stream lazily, then we could store the multiplication operation inside the stream and just multiply all numbers by 2 as they come out of the stream. Lazy evaluation is of paramount importance to computations on infinite data structures as well as aplications where the user does not need to query every element's state after an operation. Some functional programming languages use lazy evaluation to increase the efficiency of purely functional computations.

5.11 Simple vs. Performant vs. Efficient

An implementation of a data structure is said to be **Simple** if it can easily be read and understood by an programmer that did not write the code.

An implementation of a data structure is said to be **Performant** if it runs to completion relatively quickly compared to the standards of its real world time period.

An implementation of a data structure is said to be **Efficient** if its asymptotic complexity is competitive with the asymptotic complexity of the leading simple implementations of its time period.

In this book we will focus on simple and efficient data structures, because the author hopes to convey broad concepts to the reader. We will try to eschew data structures that are too complicated, because they are not as fun to program and to not shed as much light on the beautiful general themes of computer science.

5.12 Amortized versus Non Amortized

A data structure is said to be **non Amortized** if every one of its operations are guranteed to run efficiently every time they are called. These data structure are safe to use in real time applications such as visual systems and Aircrafts, where an operation that takes too much time might jepardize the proper performance of the system.

A data structure D is said to be **Amortized** if it runs efficiently relative to the number of operations that it has performed. In other words an amortized data structure has efficient time complexities for each its operations averaged over all of the times the operations have been called. Amortization is a powerful way of viewing data structures that allows us to use many efficient data structures that would not be possible without amortization.

Please see section 6.7 for a discussion of amortized analysis.

5.13 Endogeneous vs. Exogeneous

A data structure is said to be **Endogeneous** if the values of the elements are stored withing the structure. For instance Arrays are endogenous because they directly contain the memory used to stor their elements. Endogeneous structures allow us to conserve space and ensure the locality of data by directly storing the element data within the structure. A data structure is said to be **Exogeneous** if pointers to the elements are stored within the structure, instead of the values of the elements themselves. This allows us to add the exact same memory addressed element to multiple structures and modifications to the element affect all of the exogeneous structures that the element has been added to.

5.14 Null-terminated vs. Null-node Terminated.

In many node based data structures, most notably binary search trees, the structure can either terminate through the use of null pointers, or structure nodes simply containing null data. Depending on the implementation choice, the implementation may be able to perform fewer null pointer checks and simplify the implementation of operations such as rearrangement or tree balancing.

5.15 Deterministic vs. Randomized

A data structure is said to be **Deterministic** if given the same sequence of operations and inputs, the data structure will behave the same way every time. Deterministic data structures have strict gurantees on their time analysis bounds, which is useful for applications that cannot afford any expensive operations. On the other hands, because the analysis gurantees are strict, the time complexity of deterministic operations may be higher due to degenerate sequences of operations that an adversary can use to make the data structure run inefficiently.

A data structure is said to be **Randomized** if it contains a random component that causes it to not behave in exactly the same way each time. The time bounds for randomized data structures are often based on the average or expected performance for the operations, which means that sometimes the data structure may by chance perform slowly. On the other hand randomization can be a powerful tool for improving the ideal performance of a data structure and combatting an adversary.

Please see Section 6.6 for a discussion of asymptotic analysis that has been mentioned in this section.

5.16 Sparse vs. Dense

A data structure is said to be **Dense** if it allocates memory for all elements up to the data structure size. These structures are time efficient, but require a lot of memory. These data structures waste a lot of space when most of their data is non existant, uniform, or just a bunch of null pointers.

A data structure is said to be **Sparse** if is allocates only enough memory for the actual useful data currently stored at a given time in addition to the overhead of storing structures to locate these data pieces. Sparse data structures are good in situations where indexes are not contiguous or in situations where the majority of the data fields are null.

5.17 Concurrency safe.

FIXME : Expand this section or move it somewhere else.

5.17.1 Motivation

Sometimes we would like separate processors to modify the same state of a structure at the same time. These proccessors are said to be running concurrently. FIXME : Make sure to define concurrency and mention concurrent uniproccessors.

5.17.2 Mutual Exclusion

Mutual exclusion is a technique for ensuring the safety of concurrent modification to data by only affording one processor permission to make reads and writes at any given time. Some standardized techniques for implementing mutual exclusion are semaphors, mutexes, and locks. FIXME : Expand this a bit.

5.17.3 Lock-Free

If it is desirable for multiple processors to access a structure without waiting for permission, then lock free data structures may be implemented that ensure the correctness of the operations no matter which order the proccessors perform their concurrent modifications. These data structures are generally difficult to implement and require the use of atomic system provided operations such as compare and swaps.

Please see the following project repository for an example project using Lock-Free Union Find structures Project Repository: `https://github.com/Bryce-Summers/Randomized_Acyclic_Connected_Sub_Graphs`

Chapter 6

Analysis

6.1 Overview

In this chapter, we will discuss how we can use analysis to reason about the efficieny and corectness of our procedures. Analysis is not explicitly used when we write code, but it is used when a Computer Scientist wants to decide which procedures are more fit for particular purposes than others and it is also used to increase our confidence in the correctness and safety of a given procedure. We will mostly analyze the performance of independant modular functions and conclude that an entire program is correct if every component function is correct. So it suffices to discuss the analysis of programmatic functions.

6.2 Correctness Analysis

6.2.1 Mathematical Proof

To prove that a function is theoretically correct, it is necessary to rigoursously formalize what the desired behavior is in the language of mathematics and then prove that any inputs to the function imply the correct output and side effects. For instance if I had a function

$$\texttt{ADD : int*int} \to \texttt{int}$$

that takes two numbers and returns their sum with no side effects, then to prove its correctness we would look at the implementation and reason that the return result must be the sum of the numbers for all possible inputs to the function. In the specification we would also need to specify whether integer arithmetic should overflow or throw an error when the sum exceeds the size cooresponding to the representational capabilities of the machine's primitive integer type. It is also possible to prove that every line in the program is safe and correct as implied by the previous lines in the program. Programmers naturally develop this reasoning gradually and subconsiously as they gain more and more experience writing programs. Skilled programmers use this sort of reosoning informally on a day to day basis when writing correct code.

6.2.2 Empirical Testing

Empirical testing is the process by which a programmer runs their function with different inputs and then checks to see if the return results are correct. As moe and more inputs produce valid outputs, their is a higher and higher probability that the program is correct. Often Empirical testing is a good way to gain confidence one's program, but for a greater sense of surety it is important to use reaosoning such as in the mathematical proof section. It is perfectly possible to write a hard coded program that returns the correct results for a finite set of test cases, but is does not contain any mathematical logic. Programs like these are good for lookup tables, but will fail for applications that demand prescise answers for all inputs.

Unit Testing

One way of automating the empirical testing of an application is through unit testing, where we write auxiliary functions that test inputs and return true iff the input is validated. Often, we can even just inform the user of incorrect functions and terminated the execution of the program immediatly to allow the user to correct the erroneous function. Unit Test as very useful when making changes to well tested libraries, because the correctness of a library can be automatically checked after the modification has been made without a human manually going through an retesting the library. Whenever a programmer finds a set of inputs that produces erroneous results but is not caught by the existing tests, they can add a new unit test to the testing suite that checks for the new execution case. An example of unit test that checks for the correctness of an add function is the following:

```
// Returns true iff the ADD function correctly adds integers.
boolean test_add(int a, int b, int*int -> int ADD)
{
   return (a + b) == ADD(a, b);
}
```

When unit testing it is often very helpful to write an assertion helper function:

```
void ASSERT(boolean b)
{
   if(b == false) throw new Error("Assertion Failed");
}
```

Then, we could test the correctness of the `ADD` function using the `ASSERT` function without writing a specific function like the `test_add` function as follows:

```
// a = some integer, b = some integer.
ASSERT(a + b == ADD(a,b));
```

6.2.3 Formal Methods

When the utmost correctness of a system is required, such as in Nuclear powerplants and Aircraft, formal methods may be used to rigrously test and prove the correctness of the system.

Please see section 8.21 for a discussion of Binary Descision Diagrams that are data structures sometimes used in formal correctness checking.

6.3 Resource Analysis

Analysis in this section deals with the amount of certain resources such as *memory, time*, and *energy* that are used during a computation. Analysis of this sort is very important for the following reasons.

- If a computation takes too much *time*, then it is not useful.
- If a computation takes up too much *memory*, then a machine cannot correctly perform the computation.
- If a computation takes up too much *energy*, then the running of the machine will be too expensive, the world will have less energy for others, and the machine may create real world dangers due to excessive heat.

Please note that the memory requirements of a computation are sometimes refered to as the *space* requirements. It is also of note that the implementations of many proedures make many time and space tradeoffs.

6.3.1 Models

In order to correctly talk about resources in a consistant fashion admidst the variations and complexity of real world machines, we need to create abstract conceptual models that we will use to reoson about the performance of reality.

Time

From an analysis point of view we will think of time in terms of the number of operations that need to be performed. We will generally treat simple operations such as arithmetic between two integers, or loading and storing from a memory address as unitary operations.

Memory

We will think of memory in terms of the number of fixed sized chunks of memory used, for instance the size of an integer or a word on a machine. It does not matter which size chunk we use as long as we are consistant.

Energy

We will not discuss energy consumption very much in this book, but elsewhere it is necessary to develop a model for which operations a machine uses energy. Often minimizing data movement and improving the locality of memory accesses can improve the energy consumption of a computation.

6.4 Exact Analysis

An analysis is said to be *exact* if it derives an explicit formula for the amount of system resources an algorithm will use. For instance if we were to do an exact analysis of an addition function, then we would see that it requires space for 3 integers, including the two input integers and the 1 output integer. We would also see that it takes 1 operation to perform the computation, although with a more detailed knowledge of the machine it would also probably require an overhead of several operations to push the function and the inputs onto the stack and pop them off when the function returns.

Exact analysis is very important in parallel applications that wish to achieve a constant time speedup and those wishing to make their programs as fast as possible.

6.5 bigO

6.5.1 Overview

It is rather rare for different computations to take up the exact same number of resources. We often find that several algorithms that utilize the same resources behave relativly similar and it it is difficult to perceive a difference. Sometimes when we run algorithms on larger and larger problem sizes the differences become more apparent. BigO notation is a useful mathematical tool that can be used to group algorithms based on how similar their performance is as their problem sizes become very large. BigO notation groups all procedures into **complexity classes** that effectively elliminate constants and illuminate those procedures that should have comparable performance.

Warning

It is important to distinguish BigO notation from the exact resources that procedures use, because it is possible for two procedures to have the same complexity class and to have one of them perform 1 million times or worse slower than the other.

6.5.2 Formal Definitions

The Formal definition of a function f that grows no faster than g (big O):

$$f(n) \in \mathcal{O}(g(n)) \leftrightarrow \exists(k > 0).\exists(n_0).\forall(n > n_0).f(n) \leq k \cdot g(n)$$

The Formal definition of a function f that grows no slower than g (big Omega):

$$f(n) \in \Omega(g(n)) \leftrightarrow \exists(k > 0).\exists(n_0).\forall(n > n_0).f(n) \geq k \cdot g(n)$$

The Formal definition of a function f being in the compexity class defined by function g (big Theta):

$$f(n) \in \Theta(g(n)) \leftrightarrow f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$$

BigO notation is used to define upperbounds on the resources used by a function, whereas BigOmega is used to define lower bounds and BigTheta is used to define exact complexity classes for functions.

Please see Figure 6.1 for a table illustrating the relative performances of different complexiety classes for different sized problems.

After looking at the table, it should become apparent to you that all of the complexety classes from 2^n and beyond are extremely slow and complexity classes beyond $n \cdot lg(n)$ are relatively slow. We will be using BigO notation alot in our discussion of Asymptotic Analysis in section 6.6.

6.6 Asymptotic Analysis

6.6.1 Overview

As seen in section 6.5, we can use big $\mathcal{O}$ notation to make easily differentiate between the complexities of different algorithms. We will now put bigO notation to use to analyze the

	Problem Size	2	4	8	16	32
Name	**Complexity**					
Constant Time	$\mathcal{O}(1)$	1	1	1	1	1
Logarithmic	$\mathcal{O}(log_b(n))$	1	2	3	4	5
Linear	$\mathcal{O}(n)$	2	4	8	16	32
	$\mathcal{O}(n \cdot log_b(n))$	2	8	24	64	160
Quadratic	$\mathcal{O}(n^2)$	4	16	64	256	1024
Cubic	$\mathcal{O}(n^3)$	8	64	512	4096	32768
Exponential	$\mathcal{O}(2^n)$	4	16	256	65536	4294967296
Factorial	$\mathcal{O}(n!)$	2	24	40320	$\sim 2.09 \times 10^{13}$	$\sim 2.63 \times 10^{35}$
Non-Computable	$\mathcal{O}(BusyBeaver(n))$	6	107	HUGE	HUGE	HUGE

Table 6.1: Time Approximations relative to constant time operations in order of increaseing complexity and problem size. b is 2 for the approximations, but logarithms of all bases are in the same somplexity class.

performance of algorithms form several general viewpoints that may help us make descisions on which algorithms and procedures are worth implementing. This is one of the most important features of a Computer Science viewpoint in that it is of paramount importance that we can make well informed descisions upon how information should be processed.

It is important to note that Asymptotic analysis can be applied to all resources include both time and memory.

6.6.2 The Adversary.

Usually in our analysis of procedures, we will imagine a theoretical user called the *aversary* who always knows how to feed our program with input data that makes our procedure perform in the worst possible way relative to the performance goal we are trying to optimize.

For a real life example consider a bus system. Someone has the job of deciding on how many busses to send along each routes, but a sports stadium may act as an adversary by providing huge numbers of bus riders at specific stops that serve as outliers to the general usage of the bus system. If the designer wanted a general solution without special casing the sporting events, much life many of the vanilla algorithms we will discuss, then he would need to send huge numbers of busses to the sporting stadium at all times, even when there is no event, in order that the system is guranteed to never crash. Of course, once he knows more about the problem and can communicate with the sporting stadium, he can develop a more efficient algorithm by only routing extra busses to the stadium when games begin and end. In the same way, we can often develop much more efficient algorithms when we have more informationa and are alright making our algorithm less generally applicable.

6.6.3 Worst Case

A worst case analysis of an algorithm seeks to find out what is the maximum amount of a given resource the algorithm will use over all possible executions.

Consider the problem of finding a shirt in a set of drawers, assuming the shirt is indeed in the set of drawers. In the worst case you will have to look through every single drawer, therefore the worst case analysis of this algorithm would label it as of linear time complexity in terms of the number of drawers.

6.6.4 Average Case

An average case analysis of an algorithm describes the average amount of a given resource that is used over all possible executions.

Going back to our shirt and drawers example, and further assuming that all shirts are distinct and the same number of shirts are in each drawer, the average amount of drawers to find a shirt averaged over all possible initial shirt searches would be the number of drawers divided by 2, because half of the shirts are in each half of the drawers and the timings perfectly average out to searching half of the drawers. One can also think of the middle drawer as the point where it is equally likely that a shirt has been or not been found.

6.6.5 Best Case

A Best case analysis describes the smallest possible number of resources an algorithm will use out of all possible executions.

In our drawer example, the best case would be 1 drawer if we found the shirt we were looking for in the very first drawer.

6.6.6 Expected Case

The expected case is quite like the average case, but is mostly used with regards to randomized algorithms that are likely to use varying resources even when called with the same inputs. Expected case analysis determines how much of a given resource an algorithm is expected to use for any particular inputs. Usually the complexity is the greatest expected value out of any set of inputs, in other words what is the highest probable value that an adversary can force your algorithm to perform at.

6.7 Amortized Analysis

6.7.1 Motivation

Often when doing worst case analysis, it is hard to prevent all cases from being attacked by an adversary. Often their will be some operations that when in a particular state will be quite slow compared to the rest of the work. This peculiarity may convince some algorithm designers to give up on certain algorithms and never implement them.

Thankfully, many of these algorithms when viewed in light of **Amortized Analysis** may be understood to actually be quite efficient. We will use Amortized analysis to show that a sequence of operations uses an efficient amount of resources relative to the amount of operations that have been performed. In this sense we are averaging the performance of sequences of operations rather than viewing each operation independantly.

Often amortization schemes only apply to sequences of operations that start from a trivial state, if we were to start the analysis at a complicated state, then we would likely need to add a start up cost to the time complexity of the Analysis.

6.7.2 Banker's Method

In the banker's method, we add theoretical coins to a theoretical bank every time we perform an operation. Whenever we perform an expensive operation that takes up more time than our desired time complexity, we pay for the extra cost with the coins we have saved in the bank. If we can prove that for any sequence of operations, we can always pay for expensive operations and then we can say that the algorithm runs in the amortized time complexity of the non expensive operations.

Example

Consider the problem of incrementing a natural number represented by bits by 1. We would like to prove that we can do this in amortized constant time in terms of the nuber of bits flipped. Please see Figure 6.2 for an ilustration of the first 4 operations in this procedure.

Initial State	0	0	0
1 bit flipped	0	0	1
2 bits flipped	0	1	0
1 bit flipped	0	1	1
3 bits flipped	1	0	0

Table 6.2: First 4 operations in **Natural Number Incrementation Procedure** along with the costs in number of flipped bits.

For our analysis we will allocate 2 coins per operation, therebye seeking to derive a constant time complexity bound. The procedure procedes from the initial state as follows:

- 1 Bit is flipped, 1 out of the 2 coins allocated for this operation is used to pay for the flipped bit. The 1 remaining coins are stored. There is now 1 coin in the bank.
- 2 bits are flipped. 2 coins from the allocated 2 coins are paid, 0 coins are stored. There is still 1 coin in the bank.
- 1 bit is flipped. 1 coin pays for the operation, 1 coin is stored in the band. There are now 2 coins in the bank.
- 3 bits are flipped. The two allocated coins are paid in addition to 1 coin from the bank. The bank now has 1 coin.
- $\cdots$

By examining this pattern, one can see that we will always have enough coins in the bank to pay the costs of all operations when done in sequence. The key idea is that every time a bit is flipped from 0 to one, it stores one coin into the bank in order to pay for its eventual flipping back to 0. In this way, we can be certain that our bank account will never be overdrawn.

Turnstile vs. Cash Register Model

In many problems such as those viewed in amortized analysis, there are sequences of operations that increase some property, such as a counter as seen in the natural number incrementation procedure. a natural extension of these procedures is the ability to reverse the process and decrease the property while maintaining efficient time bounds.

A good way of thinking of this is the difference between *turnstiles* that only allow people in one direction and *cash registers* that allow money to be both increased and decreased.

What would happen if we naively implemented decrementation as the reverse of incrementation and allowed arbitrary sequences of incrementations and decrementations. Well, then we could have a situation where a number represented by an arbitraryly large number of 1s is incremented, then decremented repeatedly. Lets call the size in bits of this number n. Each of these operations would require $N + 1$ flips, which would eventually deplete and then overdraw any finite bank acount stored using any remotely frugal constant coin allocation policy. Alas, our algorithm now runs in linear time in the number of bits without the possibility of amotization as in the incrementation only example.

One fabulous way to write an amortized constant time algorithm for the incrementation plus decrementation problem is to use *trits*, which are objects that can take on 3 states: 1, 0, and -1. At this time, I will leave it as an **excerscise** to the reader to design an efficient algorithm using trits.

6.7.3 Potential Functions

A *potential function* Φ is any function that maps the states of a system, such as an algorithm or data structure, to non-negative numbers. Potential functions also map the initial state of a data structure to 0.

$$\Phi(S) \geq 0, \Phi(S_{\texttt{initial}}) = 0$$

Potential functions are useful in many types of analysis including mathematical proofs and amortized analysis.

6.7.4 Potential Method

The *Potential Method* is the name of the technique where potential functions are used to prove amortized bounds on the total cost of a sequence of operations.

Let C be a cost, o be an operation, and K be a non-negative constant of proportionality. We can define the amortized cost in resources as follows:

$$C_{\texttt{amortized}}(o) = C_{\texttt{actual}}(o) + K \cdot (\Phi(S_{\texttt{after}} - \Phi(S_{\texttt{before}})) \tag{6.1}$$

A proof of an upper bound on the amortized cost of a sequence of operations is also a valid proof for an upperbound on the total actual cost of the sequence of operations. Let $O = o_0, o_1, o_2, \cdots$ and also define the following:

$$C_{\texttt{amortized}}(O) = \sum_i C_{\texttt{amortized}}(o_i) \tag{6.2}$$

$$C_{\texttt{actual}}(O) = \sum_i C_{\texttt{actual}}(o_i) \tag{6.3}$$

We can then substitute in equation 6.1 to equation 6.2:

$$C_{\texttt{amortized}}(O) = \sum_i (C_{\texttt{actual}}(o_i) + K \cdot (\Phi(S_{i+1}) - \Phi(S_i)))$$

We can then Substitute the expression as follows:

$$C_{\texttt{amortized}}(O) = C_{\texttt{actual}}(O) + \sum_i (K \cdot (\Phi(S_{i+1}) - \Phi(S_i)))$$

The second term is a telescoping seriese so all except the initial and final terms cancel out:

$$C_{\texttt{amortized}}(O) = C_{\texttt{actual}}(O) + K \cdot (\Phi(S_{\texttt{Final}}) - \Phi(S_{\texttt{Initial}}))$$

Because we defined $\Phi(S_{\texttt{Initial}})$ to be 0 and $\Phi(S_i) \geq 0$ for all i, we can conclude that:

$$\Phi(S_{\texttt{Final}}) - \Phi(S_{\texttt{Initial}}) \geq 0$$

Therefore the amortized cost serves as an upper bound on the total cost.

$$C_{\texttt{amortized}}(O) \geq C_{\texttt{actual}}(O)$$

Also the amortized cost bound for a single operation is an upperbound for the average actual cost of single operations.

Key Idea

1. Determine a useful *potential function* Φ.
2. Prove $C_{\texttt{amortized}}(o)$ is in $\mathcal{O}(g(n))$ using equation 6.1, which suffices to prove an upper bound on the average cost of any operation.
3. Prove $\Phi(S_{\texttt{Final}}) - \Phi(S_{\texttt{Initial}}) \geq 0$, which is trivial if you started from the trivial initialization state.

6.8 Analysis of Parallel Algorithms

6.8.1 work and span

- The **work** of an algorithm is the amount of resources it must use in total.
- The **span** of an algorithm is the largest amount of resources used by any particular processor.

6.8.2 Efficiency vs. Speedup

- The speedup of a parallel algorithm is $\frac{\texttt{execution time (using 1 processor)}}{\texttt{execution time (using } P \texttt{ processors)}}$
- The Efficiency of a parallel algorithm is the the speedup divided by the total amount of work the algorithm does.

When designing parallel algorithms, we want to minimize span as much as possible, while trying to keep the work invariant. If we have to increase the amount of work to decrease the span, then our algorithm becomes less efficient and will burn more energy than required doing theoretically unneeded work.

6.9 Analysis of Dynamic Programming

1. Count the number of problem instances, P
2. Determine a bound on the resources used to solve one problem instance, R.
3. Bound the total resources of the problem by the resources it takes to solve all of the problem instances, $P \cdot R$

6.10 Analysis of Recursive Procedures

The simplest way to prove bounds regarding recursive procedures is to sum up the work done at every recursive instance, although this may be be hard to do at times and lead to nasty summations.

6.10.1 Brick Method

If the total amount of work done at each recursive depth decreases by at least a constant factor, then a bound for the work done for the initial function call will also bound the total amount of work done.

If the total amount of work done at each recursive depth increases by at least a constant factior, then a bound for the amount of work done at the very last recursive step will also bound the total amount of work done.

FIXME: Add diagrams and extra expanded informations.

6.11 Arithmetic Intensity

The arithmetic intensity of a procedure is the ratio of useful operations vs. The total number of operations performed including overhead.

Consider the problem of setting every element X of an array to $2X + 1$ and the following code:

```
int[] A; // Array of integers.
for(int i = 0; i < N; i++)
{
  A[i] = A[i]*2;
}

for(int i = 0; i < N; i++)
{
  A[i] = A[i] + 1;
}
```

This code spends $2*N$ operations to increment the loop counter, 2 operations to initialize the loop variable, $2*N$ operations checking the $i < N$, $2*N$ memory lookups, $2*N$ useful multiplication and addition operations, and $2*N$ memory stores. Disregarding the memory loads and stores, this code does $2*N$ useful operations, but $4*N+2$ overhead operations to manage the looping. Therefore, the **Arithmetic Intensity** would be $\frac{1}{3}$

This means that only around one third of our execution time is being used to do work that we consider to be useful.

We can improve this by rewriting the code as follows.

```
int[] A; // Array of Integers.
for(int i = 0; i < N; i++)
{
int val = A[i];
A[i] = val*2 + 1;
}
```

The arithmetic intensity is now around $\frac{1}{2}$. If we desire a greater arithmetic intensity, we would need to find more useful work that can be done within the body of the loop iteration.

Part II

Abstract Data Types and Data Structures

Chapter 7

ADTs

7.1 Overview

In this Section we will discuss the mathematical definitions of various Abstract Data Types. For particular implementations of these Types, please see section 8, which discusses Data Structures.

7.2 Function

7.2.1 Interface

A function f associates elements of type K with elements of type V and allows for the associated value to be looked up. Elements K are said to be in the domain of f and elements V are said to be in the range of f. If a key is looked up that has no associated value, then an undefined state is returned.

- `void associate(K key, V value)`
- `V lookup(K key)`

7.2.2 Finite Functions

Finite functions provide a finite number of associations from the domain to the range. They require a finite number of associate operations to populate the association.

7.2.3 Infinite Functions

Infinite functions ar those that provide an infinite number of associations from the domain to the range, such as functions that are defined on the entire range of infinite sets, such as the set of natural numbers. In order to implement an infinite function, it is necessary to specify a mathematical relationship between the inputs and the outputs. Many functions that programmers write are of this form.

7.2.4 Real World Examples.

Many indexing and lookup systems are examples of functions, including the dewey decimal system used in libraries to help the patrons convert between the name and description of a book and the desired book itself.

7.3 Stack

Stacks allow people to store Elements and remove them such that the last element added to the stack will be the first element removed from the stack. (LIFO).

7.3.1 Interface

- `bool isEmpty()`
- `void push(E elem)`
- `E pop()`

7.3.2 Real World Examples.

Most one sided shopping cart storage areas work like a stack. Stacks of logs usually work like a stack in that the last log added to the stack will be on top and therefore the first one to be removed later.

7.4 Queue

Queues allow people to store Elements and remove them such that the fisrt element added to the queue will be the first element removed from the queue. (FIFO).

7.4.1 Interface

- `bool isEmpty()`
- `void enqueue(E elem)`
- `E dequeue()`

7.4.2 Real World Examples.

The lines at a super market usually function like a queue. Toaster ovens with conveyor belts function like a queue in that the first piece of bread that is toasted is the first one that will be done.

7.5 Deque

.

7.5.1 Interface

.

7.5.2 Real World Examples.

.

7.6 Priority Queue

.

7.6.1 Interface

7.6.2 Real World Examples.

7.7 Set

7.7.1 Interface

7.7.2 Real World Examples.

7.8 Multi-Set

7.8.1 Interface

7.8.2 Real World Examples.

7.9 Relations

7.9.1 Interface

7.9.2 Real World Examples.

7.10 Trees

7.10.1 Interface

7.10.2 Full Trees

7.10.3 Complete Trees

7.10.4 Binary Trees

7.10.5 Binary Search Trees

7.10.6 Balanced Binary Search Trees.

7.10.7 Heaps

7.10.8 Tree Transversals

Chapter 8

Data Structures

8.1 Overview

Throughout this part of the book, we will be developing many of the standard data structures use in imperative computation. There is quite a lot of beauty in the interelations between each of these structures. You will find that many of these structures provide that same capabilities albeit with different resource complexities and that many of the earlier data structures may be seen as special cases of the later structures. Of particular note are graphs, which are ADTs cooresponding to mathematical relations, which are sufficiently general that they can represent most of the other data structures presented in this book.

8.2 Arrays

8.2.1 Fixed Size Arrays

Description

Interface

Implementation

Analysis

8.2.2 Strings

8.3 Lists

8.3.1 Endogenous vs. exogenous

8.3.2 Single vs. Double

8.3.3 Linear vs. Circular

8.4 Unbounded Arrays (UBA)

8.4.1 Overview

8.4.2 Amortized Analysis.

8.5 Stacks

8.5.1 Interface

8.5.2 Array Implementations

Amortized Analysis

8.5.3 List Implementations

8.5.4 Persistant Stacks

8.5.5 Max/Min Stacks

8.5.6 Function Call Stack

8.6 Queues

8.6.1 Interface

8.6.2 List Implementation

8.6.3 Array Implementation

8.6.4 Fairness

8.6.5 Pipelining

8.6.6 Queuing Theory

Part III

Fixed Size Data Structures.

Part IV

Algorithms

Chapter 9

Sorting Algorithms

9.1 Selection Sort

9.2 Insertion Sort

9.3 Merge Sort

9.4 Quick Sort

9.5 Radix Sort

9.6 Bucket Sort

9.7 Binary Search Tree Sort

9.8 Heap Sort

Chapter 10

Search Algorithms

10.1 Collection Searches

10.1.1 Linear Search

10.1.2 Binary Search

10.2 Graph Searches

10.2.1 BFS

10.2.2 DFS

10.2.3 A* Search

10.2.4 Weighted A* Search

10.2.5 A* recomputations

10.3 Game Tree Searches

10.3.1 MiniMax

10.3.2 AlphaBeta Pruning

10.3.3 Iterative Deepening

Chapter 11

Common Algorithms for Linear Collections of Numbers

This section will discuss algorithms for many of the standard statistics that are often computed from linear collections of data.

11.0.4 Partitioning

Non destructive Partitioning. $\mathcal{O}(n)$ **work,** $\mathcal{O}(n)$ **memory,** $\mathcal{O}(log(n))$ **span.**

```
// Partition into elements less, equal, and greater than the given value.
void partition(int[] collection, int[] lesser, int[] equal, int[] greater, int value)
{
  for(int i = 0; i < len; i++)
  {
    int c_val = collection[i];
    if (c_val <  value) then (lesser .add(c_val));
    if (c_val == value) then (equal  .add(c_val));
    if (c_val >  value) then (greater.add(c_val));
  }
}
```

Destructive Partitioning. $\mathcal{O}(n)$ **work,** $\mathcal{O}(1)$ **memory**

11.0.5 Linear Reduce Operations. $\mathcal{O}(n)$ work, $\mathcal{O}(log(n))$ span.

Location Based Element Search

The following algorithm can be used to find the index of a particular element in a collection when a binary function is used to determine the relative fitness of candidates.

```
// Returns index leftmost index i in indices, such that collection[i] is a minimal.
int reduce(int[] collection, int ... indices, op : (int[], int, int) --> int))
{
  int best_index = indices[0];

  for(int  i = 0; i < indices.size; i++)
  {
    best_index = op(collection, best_index, indices[i]);
  }
  return best_index;
}



// Returns index i, such that collection[i] is the leftmost minnimum element.
int reduce(int[] collection, op : op : (int[], int, int) --> int))
{
  reduce(collection, [0, ..., collection.size - 1], op);
}
```

Minimization and Maximization

```
// -- Fitness Functions.
int min(int[] collection, int index_a, int index_b)
{
  return collection[index_a] <= collection[index_b] ? index_a : index_b;
}

int max(int[] collection, int index_a, int index_b)
{
  return collection[index_a] >= collection[index_b] ? index_a : index_b;
}

// -- Functions that find location of maximum / minnimum element in the collection.

int max(int[] collection) = reduce(collection, max);
int min(int[] collection) = reduce(collection, min);
```

Valued based Associative Binary Reduction Operations

The following algorithm can used to apply a binary operator iterativly over a collection of data. Example associative binary operators include *addition*, *multiplication*, *min*, *max*, *and*, *xor*, *etc*

```
int reduce(int[] collection, op : (int, int) --> int)
{
  for(int i = 0; i < collection.size; i++)
  {
    result = op(result, i);
  }
  return result;
}
```

Reductions for several common algorithms

```
int min (int[] collection) = reduce(collection,  min);
int max (int[] collection) = reduce(collection,  max);
int sum (int[] collection) = reduce(collection, add, 0);
int mean(int[] collection) = sum(collection) / collection.size;
```

11.0.6 Median (Middle) / kth largest

We will define the median of a linear collection of data to be the value that would be at the middle indice if the collection were sorted. An equivilant definition is the value v such that half of the values not equal to v are less than v and half are greater.

Median of 3 Numbers. $\mathcal{O}(1)$ work.

Please see Section 11.0.5 for the implementation of $min(int[], int...)$

Determines the index i such that $collection[i]$ is the median of $collection[index\_a]$, $collection[index\_b]$, and $collection[index\_c]$.

```
int med3_index(int[] collection, int index_a, int index_b, int index_c)
{
  int min_index = min(collection, index_a, index_b, index_c);
  int not_min_index_1 = (min_index == index_a) ? index_b : index_a;
  int not_min_index_2 = (min_index == index_c) ? index_b : index_c;

  return min(colection, not_min_index_1, not_min_index_2);
}

// Returns value of the median of 3 numbers.
int med3(int a, int b, int c)
{
  int min = min(a, b);
  int not_min_1 = max(a, b);// Implied by last line.
  int not_min_2 = max(min, c);

  return min(not_min_1, not_min_2);
}
```

Median of 5 Numbers. $\mathcal{O}(1)$ work. 7 comparisons.

Note : 6 comparisons is optimal.

```
// Returns value of the median of 3 numbers.
int med5(int a, int b, int c, ind d, int e)
{
  sort(ref a, ref b);// 1
  sort(ref c, ref d);// 2

  sort(ref a, ref c);// 3
  sort(ref b, ref d);// 4

  // a and d are too high and low to be the median.
  return med3(b, c, e);// 5, 6, 7.
}
```

Verification Procedure. $\mathcal{O}(n)$ **work,** $\mathcal{O}(log(n)$ **span.**

A procedure for determining whether or not a particular element is the median.

```
bool isMedian(int median_value, Iterable<int> collection)
{
  int lesser = 0, greater = 0;

  for e in collection
  {
    if (e < median_value) then (lesser++)
    if (e > median_value) then (greater++)
  }
  return (|lesser - greater| <= 1);
}
```

To determine the kth largest element, replace the return statement with:

```
return lesser <= k && greater < collection.size - k;
```

Brute Force Median Find Algorithm. $\mathcal{O}(n^2)$ **work,** $\mathcal{O}(log(n))$ **span.**

```
int Median(Iterable<int> collection)
{
  for e in collection
  {
    if (isMedian(e, collection)) then (return e)
  }
  throw new Error("Cannot Get Here.");
}
```

Pseudo Median. $\mathcal{O}(1)$ **work.**

```
int pseudo_median(int[] collection)
{
  int index_1, index_2, index_3 = random_integer in [0, collection.size);
  return med3_index(collection, index_1, index_2, index_3);
}
```

Quick Select Algorithm. $\mathbb{E}(\mathcal{O}(n))$ **work.**

Please see 11.0.4 for partitioning algorithms.

```
int Median(int[] collection, int k)
{
  int val = collection[pseudo_median(collection)];
  int[] lesser, equal, greater;
  partition(collection, lesser, equal, greater, val);

  if(k < lesser.size) then return Median(lesser, k);

  int size_non_greater = lesser.size + equal.size;
  if(k > size_non_greater) then return Median(greater, k - size_non_greater);

  return val;
}
```

Linear Time Selection Algorithm. $\mathcal{O}(n)$ **work.**

To achieve a guranteed linear time Median find algorithm, replace the constant time pseudo_median implementation with the following procedure, where the Median function referenced therin is a co-recursive call to the quick select based median.

```
int linear_pseudo_median(int[] collection)
{
  int[] medians;

  for(int i = 0; i < collection.size/5; i++)
  {
    int j = i*5;
    medians[j] = med5(collection[j + 0], collection[j + 1], collection[j + 2],
                      collection[j + 3], collection[j + 4]);
  }
   return Median(medians);
}
```

The resulting recurrence solves to $\mathcal{O}(n)$:

$$T(n) = T\left(\frac{2}{10}\right) + T\left(\frac{7}{10}\right) + c \cdot n$$

Chapter 12

Parallel Algorithms

12.1 Inclusive Scan

This is a wonderful algorithm that can be used to magically build many other parallel algorithms.

12.2 Addition

.

Part V

Inefficient and Obscure Algorithms and Data Structures

.

Bibliography

[1] Robert Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.

[2] Chris Okasaki, *Purely Functional Data Structures*, FIXME: publisher, location YEAR.

[3] The author received an Undergraduate Education in Computer Science at Carnegie Mellon University in Pittsbugh, PA.