Chapter 1

Introduction

1.1 What are Algorithms

An ***algorithm*** is an unambiguous specification of how to solve a class of problems. It is any well-defined computational procedure that takes some value or set of values as ***input*** and provides some value or set of values as ***output***. Put in other words, an algorithm is a sequence of computational steps that transform an input to an output, or, a tool for solving well-specified computational problems.

A computer program can be viewed as an elaborate algorithm trying to explain the steps to be taken by a computer to solve a problem. The audience in computer program is a computer, hence, the algorithm is finally expressed in a language that the computer understands. We will see later on, however, that there are other 'languages' we could use in stating an algorithm depending on the audience we are trying to communicate with.

The concept of algorithm has existed for centuries; however, a partial formalization of what would become the modern algorithm began with attempts to solve the Entscheidungsproblem (the "decision problem") posed by David Hilbert in 1928. Subsequent formalizations were framed as attempts to define "effective calculability" or "effective method"; those formalizations included the GdelHerbrandKleene recursive functions of 1930, 1934 and 1935, Alonzo Church's lambda calculus of 1936, Emil Post's "Formulation 1" of 1936, and Alan Turing's Turing machines of 19367 and 1939. Giving a formal definition of algorithms, corresponding to the intuitive notion, remains a challenging problem.

The **problem statement/ statement of a problem** specifies the desired input/output relationship. Algorithms, therefore, describes the specifics for achieving this IO relationship. Two examples are presented here to illus-

trate this concept of problem statement for which students are assumed to know at least one solution algorithm.

Example 1: Searching problem
Input: A sequence of n numbers $\{a_1, a_2, a_3...a_n\}$ and a number $\boldsymbol{b}$ to search for
Output: A index $\boldsymbol{i}$ if $a_i = b$ or $\boldsymbol{-1}$ if there exists no such $\boldsymbol{i}$ where $0 \leq i < n$
Example 2: Sorting problem
Input: A sequence of $\boldsymbol{n}$ numbers $\{a_1, a_2, a_3...a_n\}$
Output: A permutation (reordering) $\{a_i, a_{ii}, a_{iii}, ..., a_m\}$ of the input sequence such that $\{a_i \leq a_{ii} \leq a_{iii} \leq ... \leq a_m\}$

An example input for the searching problem could be $\{93, 29, 40, 14, 26, 84\}$ and 14, accordingly the output of a searching algorithm will be 3 (with a zero based indexing scheme). An example input sequence for the sorting problem could be 93, 29, 40, 14, 26, 84 and the output/result of a sorting algorithm for this input will be 14, 26, 29, 40, 84, 93. Both of these inputs are called ***instances*** of their respective problems.

In general an instance of a problem consists of the input (satisfying whatever constraints problem statement imposes). An algorithm is said to be correct if and only if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem.

Incorrect algorithms either do not halt at all or will halt with the wrong output. Incorrect algorithms might be at times useful if we can control the error rate. E.g., A ***heuristic*** is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimal, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.

The notion of Algorithm its relationship with the notion of functions is often times confused and many use one to mean the other. However, the two notions are completely different. Function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output. An example is the function that relates each real number x to its square x_2. The output of a function f corresponding to an input x is denoted by *f(x)* (read "f of x"). In this example, if the input is -3, then the output is 9, and we may write $f(-3) = 9$. Likewise, if the input is 3, then the output is also 9, and we may write $f(3) = 9$. (The same output may be produced by more than one input, but each input gives only one output.) The input variable(s) are sometimes referred to as the

argument(s) of the function.

Algorithm on the other hand refers to the steps taken to produce the desired output. Hence, there could be more than one algorithm per function. In our previous example *f(x)*, which takes in a number as an input and produces the squared value of the number as an output could be implemented in a number of ways (i.e., algorithms). One possible algorithm is to add the input value x , x times and return the absolute value of the sum found. Another possible implementation is to multiply the input value x by itself once. Both of this algorithms are correct(we will define what a correct algorithm is later on), however one could be more efficient than the other. One should note that both of these assertion are valid and testable if we define the a certain computational model (discussed later)

1.2 Characteristics of Algorithms

- **Finiteness**: An algorithm must always terminate after a finite number of steps.

- **Definiteness/Precision**: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.

- **Input** An algorithm has zero or more inputs, i.e, quantities which are given to it initially before the algorithm begins.

- **Output**: An algorithm has one or more outputs i.e, quantities which have a specified relation to the inputs.

- **Effectivenes**: An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

- **Uniqueness**: Results of each step are uniquely defined and only depend on the input and the result of the preceding steps.

- **Generality**: The algorithm applies to a set of inputs

1.3 Expression of Algorithms

Flowcharts

A flowchart is a type of diagram that represents an algorithm, work-flow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution model to a given problem. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Pseudo Code

Pseudo Code is an informal high-level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading. Pseudo Code typically omits details that are essential for machine understanding of the algorithm, such as variable declarations, system-specific code and some subroutines. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation. The purpose of using pseudo code is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm. It is commonly used in textbooks and scientific publications that are documenting various algorithms, and also in planning of computer program development, for sketching out the structure of the program before the actual coding takes place.

Programming Languages

Since programmers can read source code of high level programming languages with ease one could use any programming language that the intended audience knows and understands.

Drakon-Charts

DRAKON is an algorithmic visual programming language developed within the Buran space project following ergonomic design principles. The language provides a uniform way to represent flowcharts of any complexity that are easy to read and understand.

Control Tables

Control tables are tables that control the control flow or play a major part in program control. There are no rigid rules about the structure or content of a control table its qualifying attribute is its ability to direct control flow in some way through "execution" by a processor or interpreter. The design of such tables is sometimes referred to as table-driven design (although this typically refers to generating code automatically from external tables rather than direct run-time tables). In some cases, control tables can be specific implementations of finite-state-machine-based automata-based programming. If there are several hierarchical levels of control table they may behave in a manner equivalent to UML state machines

In this course

In this course we will primary use pseudo code and programming language source code to express algorithms. In both cases we will adopt a C-based language.

1.4 Classification

There are various ways to classify algorithms, each with its own merits.

1.4.1 By implementation

Recursion

A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition (also known as termination condition) matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, towers of Hanoi [1] is well understood using recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice-versa.

[1] Detailed explaniation of Tower of Hanoi is found here: `https://en.wikipedia.org/wiki/Tower_of_Hanoi`

Logical

An algorithm may be viewed as controlled logical deduction. This notion may be expressed as: Algorithm = logic + control.[61] The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the logic programming paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the axioms has a well-defined change in the algorithm. Algorithms can be specified in English (i.e., Natural Language), as computer program, or even as a hardware design as long as it specifics precise description of the computational procedures to be followed. Most of the time we avoid using natural language for expressing algorithms due to its ambiguity, especially for complex algorithms. Instead when communicating an algorithm with other people we use one of the following *'languages'*

Serial, parallel or distributed

Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms or distributed algorithms. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network. Parallel or distributed algorithms divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together. The resource consumption in such algorithms is not only processor cycles on each processor but also the communication overhead between the processors. Some sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.

Deterministic or non-deterministic

Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithms solve problems via guessing although typical guesses are made more accurate through the use of heuristics.

Exact or approximate

While many algorithms reach an exact solution, approximation algorithms seek an approximation that is closer to the true solution. Approximation can be reached by either using a deterministic or a random strategy. Such algorithms have practical value for many hard problems. One of the examples of an approximate algorithm is the Knapsack problem. The Knapsack problem is a problem where there is a set of given items. The goal of the problem is to pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number X. So, we must consider weights of items as well as their value.

Quantum Algorithm

They run on a realistic model of quantum computation. The term is usually used for those algorithms which seem inherently quantum, or use some essential feature of quantum computation such as quantum superposition or quantum entanglement.

1.4.2 By design paradigm

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories include many different types of algorithms. Some common paradigms are:

Brute-force or exhaustive search

This is the naive method of trying every possible solution to see which is best.

Divide and conquer

A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments. A simpler variant of divide and conquer is called a decrease and conquer algorithm, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple

subproblems and so the conquer stage is more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is the binary search algorithm.

Search and Enumeration

Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

Randomized algorithm

Such algorithms make some choices randomly (or pseudo-randomly). They can be very useful in finding approximate solutions for problems where finding exact solutions can be impractical (see heuristic method below). For some of these problems, it is known that the fastest approximations must involve some randomness.Whether randomized algorithms with polynomial time complexity can be the fastest algorithms for some problems is an open question known as the P versus NP problem. There are two large classes of such algorithms:

1. Monte Carlo algorithms return a correct answer with high-probability. E.g. RP is the subclass of these that run in polynomial time.

2. Las Vegas algorithms always return the correct answer, but their running time is only probabilistically bound, e.g. ZPP.

Reduction of Complexity

This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as transform and conquer.

1.4.3 Optimization Problems

For optimization problems there is a more specific classification of algorithms; an algorithm for such problems may fall into one or more of the general

categories described above as well as into one of the following:

Linear programming

When searching for optimal solutions to a linear function bound to linear equality and inequality constraints, the constraints of the problem can be used directly in producing the optimal solutions. There are algorithms that can solve any problem in this category, such as the popular simplex algorithm. Problems that can be solved with linear programming include the maximum flow problem for directed graphs. If a problem additionally requires that one or more of the unknowns must be an integer then it is classified in integer programming. A linear programming algorithm can solve such a problem if it can be proved that all restrictions for integer values are superficial, i.e., the solutions satisfy these restrictions anyway. In the general case, a specialized algorithm or an algorithm that finds approximate solutions is used, depending on the difficulty of the problem.

Dynamic programming

When a problem shows optimal substructures meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called dynamic programming avoids recomputing solutions that have already been computed. For example, FloydWarshall algorithm, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that subproblems are more or less independent in divide and conquer, whereas subproblems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When subproblems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex problems. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

The Greedy Method

A greedy algorithm is similar to a dynamic programming algorithm in that it works by examining substructures, in this case not of the problem but of a given solution. Such algorithms start with some solution, which may

be given or have been constructed in some way, and improve it by making small modifications. For some problems they can find the optimal solution while for others they stop at local optima, that is, at solutions that cannot be improved by the algorithm but are not optimum. The most popular use of greedy algorithms is for finding the minimal spanning tree where finding the optimal solution is possible with this method. Huffman Tree, Kruskal, Prim, Sollin are greedy algorithms that can solve this optimization problem.

The Heuristic Method

In optimization problems, heuristic algorithms can be used to find a solution close to the optimal solution in cases where finding the optimal solution is impractical. These algorithms work by getting closer and closer to the optimal solution as they progress. In principle, if run for an infinite amount of time, they will find the optimal solution. Their merit is that they can find a solution very close to the optimal solution in a relatively short time. Such algorithms include local search, tabu search, simulated annealing, and genetic algorithms. Some of them, like simulated annealing, are non-deterministic algorithms while others, like tabu search, are deterministic. When a bound on the error of the non-optimal solution is known, the algorithm is further categorized as an approximation algorithm.

1.4.4 By Complexity

Algorithms can be classified by the amount of time they need to complete compared to their input size:

- Constant time: if the time needed by the algorithm is the same, regardless of the input size. E.g. an access to an array element.
- Linear time: if the time is proportional to the input size. E.g. the traverse of a list.
- Logarithmic time: if the time is a logarithmic function of the input size. E.g. binary search algorithm.
- Polynomial time: if the time is a power of the input size. E.g. the bubble sort algorithm has quadratic time complexity.
- Exponential time: if the time is an exponential function of the input size. E.g. Brute-force search.

  Some problems may have multiple algorithms of differing complexity, while other problems might have no algorithms or no known efficient

algorithms. There are also mappings from some problems to other problems. Owing to this, it was found to be more suitable to classify the problems themselves instead of the algorithms into equivalence classes based on the complexity of the best possible algorithms for them.

1.4.5 By Field of Study

Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, computational geometric algorithms, combinatorial algorithms, medical algorithms, machine learning, cryptography, data compression algorithms and parsing techniques.

Fields tend to overlap with each other, and algorithm advances in one field may improve those of other, sometimes completely unrelated, fields. For example, dynamic programming was invented for optimization of resource consumption in industry, but is now used in solving a broad range of problems in many fields.

1.5 Problems That Could be Solved by Algorithms

Obviously sorting and searching are not the only areas where algorithms play great role. The following are examples of areas whereby algorithms serve invincible function.

1.5.1 Human Genome Project

The Human Genome Project has made great progress toward the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. Although the solutions to the various problems involved are beyond the scope of this book, many methods to solve these biological problems use ideas from several of the chapters in this book, thereby enabling scientists to accomplish tasks while using resources efficiently. The savings are in time, both human and machine, and in money, as more information can be extracted from laboratory techniques.

1.5.2 The Internet

The Internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the Internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data will travel, and using a search engine to quickly find pages on which particular information resides.

1.5.3 E-Commerce

Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements. The core technologies used in electronic commerce include public-key cryptography and digital signatures, which are based on numerical algorithms and number theory.

1.5.4 Manufacturing

Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met. An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming.

1.5.5 Fourier Transform

Not every problem solved by algorithms has an easily identified set of candidate solutions. For example, suppose we are given a set of numerical values representing samples of a signal, and we want to compute the discrete Fourier transform of these samples. The discrete Fourier transform converts the time domain to the frequency domain, producing a set of numerical coefficients, so that we can determine the strength of various frequencies in the sampled signal. In addition to lying at the heart of signal processing, discrete

Fourier transforms have applications in data compression and multiplying large polynomials and integers.

1.5.6 Longest subsequence problem

Given two ordered sequences of symbols we wish to find the longest common subsequence of these sequences. The length of the longest common subsequence of the two lists gives one measure of how similar these two sequences are. For example, if two sequences are base pairs in DNA strands, then we might consider them similar if they have a long common subsequence. For lists X (with m symbols) and Y (with n symbols), there exist 2^m & 2^n subsequences of X and Y, respectively. All these problems have a couple of common characteristics shared by many inserting algorithmic problems. These characteristics are

- The problems have many candidate solutions, most of which are incorrect (i.e., do not solve the problem) and finding those that can is often quite challenging. Finding the "best" from those that solve the problem is an even greater challenge.
- They have a practical application

1.6 Related Issues

1.6.1 Data structure

Data structure is a way to store and organize data in order to facilitate access and modification. No single data structure works well for all types of problems. When solving problems with algorithms, the underlying data structure often determines the applicability, efficiency and correctness of algorithms that try to solve a problem. Hence, choosing the right data structure is as important as choosing the right algorithm to solve a problem.

1.6.2 Parallelism

The computing world is moving to multiple processing cores rather than a single powerful CPU due to physical limitation. Algorithms, therefore, should be designed with this fact in mind, hence, considering parallel computing. A parallel algorithm, as opposed to a traditional serial algorithm, is an algorithm which can be executed a piece at a time on many different

processing devices, and then combined together again at the end to get the correct result.

Algorithms vary significantly in how parallelizable they are, ranging from **easily parallelizable** to **completely unparallelizable**. Further, a given problem may accommodate different algorithms, which may be more or less parallelizable. Some problems are easy to divide up into pieces in this way these are called **embarrassingly parallel problems**. For example, splitting up the job of checking all of the numbers from one to a hundred thousand to see which are primes could be done by assigning a subset of the numbers to each available processor, and then putting the list of positive results back together.

Some problems cannot be split up into parallel portions, as they require the results from a preceding step to effectively carry on with the next step these are called inherently serial problems. Examples include iterative numerical methods, such as Newton's method, iterative solutions to the three-body problem, and most of the available algorithms to compute pi (π).

1.6.3 Hard Problems

Most of this course is about efficient algorithms. Our usual measure of efficiency is speed, i.e., how long an algorithm takes to produce its result. There are some problems, however, for which no efficient solution is known. Chapter 34 of the text book studies an interesting subset of these problems, which are known as NP-complete.

Why are NP-complete problems interesting? First, although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist. In other words, no one knows whether or not efficient algorithms exist for NP-complete problems. Second, the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing. Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. Computer scientists are intrigued by how a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

You should know about NP-complete problems because some of them arise surprisingly often in real applications. If you are called upon to produce an efficient algorithm for an NP-complete problem, you are likely to spend a lot of time in a fruitless search. If you can show that the problem is NP-complete, you can instead spend your time developing an efficient algorithm

that gives a good, but not the best possible, solution.

As a concrete example, consider a delivery company with a central depot. Each day, it loads up each delivery truck at the depot and sends it around to deliver goods to several addresses. At the end of the day, each truck must end up back at the depot so that it is ready to be loaded for the next day. To reduce costs, the company wants to select an order of delivery stops that yields the lowest overall distance traveled by each truck. This problem is the well-known "traveling-salesman problem," and it is NP-complete. It has no known efficient algorithm. Under certain assumptions, however, we know of efficient algorithms that give an overall distance which is not too far above the smallest possible.

1.7 Few Example Problems & Algorithms

The following are few of the simple problems that can be solved with relatively very simple algorithms. Students are assumed to be familiar with most of them, hence, they are presented here to show that the notion of algorithm is not that new to students.

1.7.1 GCD and LCM

Greatest Common Divisor

The greatest common divisor (GCD) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers and is often denoted as GCD(a,b) . It's also known as the greatest common factor (GCF), highest common factor (HCF), greatest common measure (GCM), or highest common divisor.

Problem Statement

Input: Two integers of a and b

Output: An integer i such that $i \leq min(a, b)$ and i is the biggest number that can divide both a and b

Euclidean algorithm

The Euclidean algorithm, or Euclid's algorithm, is an efficient method for computing the greatest common divisor (GCD) of two numbers, the largest number that divides both of them without leaving a remainder. It is named after the ancient Greek mathematician Euclid, who first described it in Euclid's Elements (c. 300 BC).

Pseudocode implementation

```
procedure gcd(a, b)
    while b      0
        t = b
        b = a % b
        a = t
    end while
    return a
end procedure
```

Least Common Multiple

The least common multiple, lowest common multiple, or smallest common multiple of two integers a and b, usually denoted by LCM(a, b), is the smallest positive integer that is divisible by both a and b. Since division of integers by zero is undefined, this definition has meaning only if a and b are both different from zero. However, some authors define LCM(a,0) as 0 for all a, which is the result of taking the LCM to be the least upper bound in the lattice of divisibility.

Problem Statement

Input: Two integers of a and b

Output: An integer i such that i is the smallest number that is divisible by both a and b

Example Applications

- A salesman goes to New York every 15 days for one day and another every 24 days, also for one day. Today, both are in New York. After how many days both salesman will be again in New York on same day?
- A bell rings every 18 seconds, another every 60 seconds. At 5.00 pm the two ring simultaneously. At what time will the bells ring again at the same time?
- Ato Abebe has 120 crayons and 30 pieces of paper to give to his students. What is the largest number of students he can have in his class so that each student gets equal number of crayons and equal number of paper.
- Mekdes has two pieces of cloth. One piece is 72 inches wide and the other piece is 90 inches wide. She wants to cut both pieces into strips of equal width that are as wide as possible. How wide should she cut the strips?

1.7.2 Searching

The Problem

The search problem is one of the most frequent in computer science. A search algorithm is any algorithm which solves the search problem[2], namely, to retrieve information stored within some data structure, or calculated in the search space of a problem domain. Examples of such structures include but are not limited to a linked list, an array data structure, or a search tree. The appropriate search algorithm often depends on the data structure being searched, and may also include prior knowledge about the data. Searching also encompasses algorithms that query the data structure, such as the SQL SELECT command.

In this section we will limit our discussion to linear data structures such as array and linked list when to define our search problem. Hence our definition of the searching problem will become, given a a list of items and an item to look for in the list, called *key*, the task is to go through the list and find/return the location of the key with in the collection, if it exists. If the key is not found in the list the algorithm should return a value that clearly indicates that the key was not found, often this is by returning -1 since it is assumed the location of items in the collection is to mean the index of the items within the collection which normally starts from zero and increments by one upto the last item.

Linear Search

Linear search or sequential search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

Linear search runs in at worst linear time and makes at most n comparisons, where n is the length of the list. If each element is equally likely to be searched, then linear search has an average case of $n/2$ comparisons, but the average case can be affected if the search probabilities for each element vary. Linear search is rarely practical because other search algorithms and schemes, such as the binary search algorithm and hash tables, allow significantly faster searching for all but short lists. However, since linear search is does not assume the input to be in any particular sorting order it entertains a larger problem domain than the alternative algorithms. What's more, it is easy for users to understand linear search as the steps in linear search are common in many real life searching scenarios.

[2] A more formal definition of the search problem can be found here `https://en.wikipedia.org/wiki/Search_problem`

Problem Statement

Input: A collection of $\boldsymbol{n}$ numbers $\{a_0, a_1, a_2...a_{n-1}\}$ and a single item key

Output: The first index i such that $a_i = key$. If no such item is found with in the collection return -1

Linear search sequentially checks each element of the list until it finds an element that matches the target value. If the algorithm reaches the end of the list, the search terminates unsuccessfully.

Basic algorithm

Given a list A of n elements with values or records $\{a_0, a_1, a_2...a_{n-1}\}$, and target value K, the following subroutine uses linear search to find the index of the target K in A

Pseudocode implementation

```
procedure linearSearch(A,K)
  location = -1
  for i = 0 to n do
    if A[i] == K then
        location = i
        break
    end if
  end for
  return location
end procedure
```

Binary Search

Binary Search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a **sorted** list. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the collection. Even though the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation.

Binary search is generally more efficient than linear search but requires the input to be sorted. If the input is completely sorted the algorithm will fail to find the target item.

Problem Statement

Input: A sorted collection of $\boldsymbol{n}$ numbers $\{a_1, a_2, a_3...a_{n-1}\}$ and a single item key

Output: The first index i such that $a_i = key$. If no such item is found with in the collection return -1

Procedure

1. Set L to 0 and R to $n1$.
2. If $L > R$, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the $floorof(L + R)/2$, which is the greatest integer less than or equal to $(L + R)/2$.
4. If $a_m < K$, set L to $m + 1$ and go to step 2.
5. If $a_m > K$, set R to $m1$ and go to step 2.
6. Now $a_m = K$, the search is done; return m.

This iterative procedure keeps track of the search boundaries with the two variables L and R. The procedure may be expressed in pseudocode as follows, where the variable names and types remain the same as above, floor is the floor function, and unsuccessful refers to a specific variable that conveys the failure of the search.

Pseudocode implementation

```
procedure binarySearch(A,K)
        L = 0
    R = n       1
    location = -1
        while L <= R
        m = floor((L + R) / 2)
        if A[m] < K
            L = m + 1
        else if A[m] > K:
            R = m - 1
        else
            location = m
            break
    end while
    return location
end procedure
```

1.7.3 Optimum Finding

Searching for a given item with in a list is an interesting problem in computer science as described in the previous section. However, the aforementioned form of searching is not the only version that is relevant in computer science. In fact, we have a number of variations that could help in addressing related problems such as sorting. A few of these are presented here.

Find Max

The first variation of the search problem defined earlier is the problem of finding maximum. Put simply, given a list of elements (numbers for simplicity) as an input the task is to find the element with highest value.

Problem Statement

Input: A collection of $\boldsymbol{n}$ numbers $\{a_0, a_1, a_2...a_{n-1}\}$

Output: The first index j such that $a_j \geq a_i \forall i \in 1..n$.

The problem could be modified to search for the smallest item by altering the problem statement such that the $a_j \geq a_i \forall i \in 1..n$ becomes $a_j \leq a_i \forall i \in 1..n$

Algorithm

The algorithm in the following pseudocode starts by assuming that the first element is the maximum and traverses through the list of items to check if there exists an item that is greater than the initial assumption, and if so, will update the assumption by saving this new item.

Pseudocode implementation

```
procedure findMax(A)
  max = 0
  for i = 1 to n do
    if A[i] > A[max] then
        max = i
    end if
  end for
  return max
end procedure
```

Find maximum and minimum

As modification to the Find Max problem the Find maximum and minimum problem is defined by the following problem statement.

Problem Statement

Input: A collection of $\boldsymbol{n}$ numbers $\{a_0, a_1, a_2...a_{n-1}\}$

Output: The first index j such that $a_j \geq a_i \forall i \in 1..n$ and the first index k such that $a_k \leq a_i \forall i \in 1..n$

Algorithm

The algorithm in the following pseudocode is similar to the Find Max algorithm but instead of iterating to find a single value it instead checks two conditions with two assumptions.

Pseudocode implementation

```
procedure findMaxMin(A)
  maxLocation = 0
  minLocation = 0
  for i = 1 to n do
    if A[i] > A[maxLocation] then
        maxLocation = i
    end if
    if A[i] < A[minLocation] then
        minLocation = i
    end if
  end for
  return maxLocation, minLocation
end procedure
```

Find the top two maximums $Max_1 and Max_2$

This problem is an extension of find max problem discussed earlier. In this problem our objective is to find the top two elements instead of just on top element as we did in the previous example.

Problem Statement

Input: A collection of $\boldsymbol{n}$ numbers $\{a_0, a_1, a_2...a_{n-1}\}$

Output: The first indices j and k such that $a_j \geq a_i \forall i \in 1..n$ and $a_k \geq a_k \forall k \in 1..nexceptj$.

Algorithm

The algorithm in the following pseudocode starts by assuming that the first element is the two maximums we are looking for and traverses through the list of items to check if there exists an item that is greater than the initial assumption, and if so, will update the assumption by saving this new item. As opposed to the findmax algorithm, in this solution we also track the second maximum number

Pseudocode implementation

```
procedure find2Max(A)
  max1 = 0
```

```
  max2 = 0
  for i = 1 to n do
    if A[i] > A[max1] then
        max2 = max1
        max1 = i
    else if A[i] > A[max2] then
        max2 = i
    end if
  end for
  return max1,max2
end procedure
```

1.7.4 Sorting

Sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

- The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order);
- The output is a permutation (reordering but with all of the original elements) of the input.

Further, the data is often taken to be in an array, which allows random access, rather than a list, which only allows sequential access, though often algorithms can be applied with suitable modification to either type of data.
Problem Statement

Input: A sequence of $\boldsymbol{n}$ numbers $\{a_1, a_2, a_3...a_n\}$

Output: A permutation (reordering) $\{a_i, a_{ii}, a_{iii}, ..., a_m\}$ of the input sequence such that $\{a_i \leq a_{ii} \leq a_{iii} \leq ... \leq a_m\}$

Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for

the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass

(**5 1** 4 2 8) (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 **5 4** 2 8) (1 **4 5** 2 8), Swap since 5 ¿ 4

(1 4 **5 2** 8) (1 4 **2 5** 8), Swap since 5 ¿ 2

(1 4 2 **5 8**) (1 4 2 **5 8**), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass

(**1 4** 2 5 8) (**1 4** 2 5 8)

(1 **4 2** 5 8) (1 **2 4** 5 8), Swap since $4 > 2$

(1 2 **4 5** 8) (1 2 **4 5** 8)

(1 2 4 **5 8**) (1 2 4 **5 8**)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass

(**1 2** 4 5 8) (**1 2** 4 5 8)

(1 **2 4** 5 8) (1 **2 4** 5 8)

(1 2 **4 5** 8) (1 2 **4 5** 8)

(1 2 4 **5 8**) (1 2 4 **5 8**)

Pseudocode implementation

```
procedure bubbleSort(A)
  for i = n−1 down to 0 do
    for j = 0 to i do
      if A[j+1] < A[j] then
        tmp = A[j+1]
```

```
        A[j+1] = A[j]
        A[j] = tmp
      end if
    end for
  end for
end procedure
```

Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Jon Bentley shows a three-line C version, and a five-line optimized version.
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(nk)$ when each element in the input is no more than k places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount $O(1)$ of additional memory space
- Online; i.e., can sort a list as it receives it

When people manually sort cards in a bridge hand, most use a method that is similar to insertion sort. Example: The following listing shows the steps for sorting the sequence 3, 7, 4, 9, 5, 2, 6, 1. In each step, the key under consideration is underlined. The key that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1
3 4 7 **9** 5 2 6 1
3 4 **5** 7 9 2 6 1
2 3 4 5 7 9 6 1
2 3 4 5 **6** 7 9 1
1 2 3 4 5 6 7 9

Pseudocode implementation

```
procedure insertionSort( A : list of sortable items )
 for j = 2 to length [A]
   do key = A[j]
      // Insert A[j] into the sorted sequence A[1... j-1]
      i= j      1
      while i>0 and A[i]>key
         do A[I + 1]   A [i]
         j = i   1
      end while

      A[i+1] = key
   end for
end procedure
```

Selection Sort

selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Step-by-step example

```
Sorted sublist == ( )
Unsorted sublist == (11, 25, 12, 22,64)
Least element in unsorted list == 11

Sorted sublist == (11)
Unsorted sublist == (25, 12, 22, 64)
Least element in unsorted list == 12

Sorted sublist == (11, 12)
Unsorted sublist == (25, 22, 64)
Least element in unsorted list == 22

Sorted sublist == (11, 12, 22)
Unsorted sublist == (25, 64)
Least element in unsorted list == 25

Sorted sublist == (11, 12, 22, 25)
Unsorted sublist == (64)
Least element in unsorted list == 64

Sorted sublist == (11, 12, 22, 25, 64)
Unsorted sublist == ( )
```

Pseudocode implementation

```
procedure selectionSort( A : list of sortable items )
        for j = 0 to n-1 do
                iMin = j;
            for  = j+1 to n do
                if A[i] < A[iMin]
                        iMin = i;
                end if
        end for
        if iMin != j
                swap(A[j], A[iMin]);
                end if
        end for
end procedure
```

1.7.5 Peak Finding

In a given list of items a peak is defined as an element which is larger or equal to both the elements on its sides. **Problem Statement**

Input: A sequence of $\boldsymbol{n}$ numbers $\{a_1, a_2, a_3 ... a_n\}$

Output: An element a such that $b \leq a \geq c$ where *bandc* are the elements in the list which are immediately to the left and right side of element a, respectively.

Sample Solution

One simple is solution is presented in the pseudocode below. A more efficient solution is presented later in this course.

Pseudocode implementation

```
procedure peakFindingNaive(A)
        peak = -1
        for i = 0 to n-1 do
                left = max(0, i-1)                    // boundary case
                right = min(n-1, i+1)                 // boundary case
                if left >= A[i] and right <= A[i]
                        peak = A[i]
                        break
                end if
        end for
        return peak;
end procedure
```

1.7.6 Summary

In this lecture we discussed what algorithms are and their role in the modern world of computing. We highlighted the important characteristics of algorithms that we want to uphold especially

- **Correctness**: in all input instances
- **Efficiency**: especially for large input sizes

1.7.7 Review Questions

1. It is often required to count the number of occurrences of an element within a list. Write the problem statement and an algorithm to the count problem.
2. Write the problem statement and algorithm of the Ethiopian Multiplication Algorithm presented in class.
3. Write the problem statement and algorithm by modifying the find max problem presented earlier. Modify it such that instead of finding the maximum or the top two elements as we did in the examples before, a number m is provided as an input and the task is to find the top m values in the list.

4. Another modification to the find max problem is to find the m^{th} maximum value. As opposed to the previous question the task is to find a single number which is greater than all elements in the list except the top $m - 1$ elements.

5. Other than speed, what other measures of efficiency might one use in a real-world setting?

6. Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

7. Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

1.7.8 Further Reading

- In the first chapter of the book Algorithm Design Jon Kleinberg and Eva Tardos present a good example for the need to study Algorithms.

- Algorithm does not have a generally accepted formal definition. Researchers are actively working on this problem. Algorithm characterizations are attempts to formalize the word algorithm. There is a wikipedia article presenting some of the "characterizations" of the notion of "algorithm" in more detail here: `https://en.wikipedia.org/wiki/Algorithm_characterizations`.