Introduction to Data Structures and Algorithms

Chapt 1,2,10 and Appendix A : Read along with Handout 3

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Data Types, Abstract Data Types (ADT), Data Model

1.1 Data-type. In a programming language, every variable has a data-type, which is the set of values the variable takes.

There are **built-in (or primitive or elementary)** data types and also **composite** data types. The primitive data types of programming language C++ include `char, int, double`. The composite data types can be built on top of the primitive ones and include arrays, structures, classes such as `struct, class , int[]`. The whole set of data types and the mechanisms that allow for their aggregation define the data model of C++. C++ is a **stronly-typed language**. When we write `int x` we define variable `x` to be of an integer data type. Some languages such as Python are **weakly-typed**. The data type of a variable depends on its value. Thus `x=10` in one line makes x to be of an integer data-type but `x='ab'` in the next line makes it to be a string value!

1.2 Data model (DM). It is an abstraction that describes how data are represented and used.

The data model offered by a programming language such as C++ includes characters, integers of different sizes, and floating-point numbers of different sizes BUT ALSO METHODS (eg. arrays, classes, structures) that allow for their aggregation and composition (eg templates). Note that the int data-type of C++ differs from the integer abstraction as we use it in mathematics; the latter is of unlimited precision.

1.3 Abstract data type (ADT). An abstract data type (ADT) is a mathematical model and a collection of operations defined on this model.

For example a Dictionary is an asbtract data type consisting of a collection of words on which a set of operations are defined such as Insert, Delete, Search. A priority queue is a collection of elements each of which has a priority. (Each element may also have a value or other information associated with it.) A set of operations defined for a priority queue are Insert and ExtractHigh. Operation ExtractHigh finds and removed the element with the highest priority. Whether priority 1 is high or low this is not a detail that needs to be addressed explicitly at this level. (Insert inserts an element into a priority queue; this includes not just the priority of the element but also all other relevant and auxiliary information. More often than not we will be inserting only priority values and ignoring other auxiliary info.)

Note 1. *Collection vs Sequence vs Set:* A collection can be a sequence or a set. In a set of elements the order of its elements does not matter. In a sequence order matters. Thus set $\{1, 2, 3\}$ is the same as $\{2, 1, 3\}$ but sequence $\langle 1, 2, 3\rangle$ is different from $\langle 1, 3, 2\rangle$.

Note 2. $\lg n$ vs $\log n$. The $\log m$ refers generically to the log base 10 of m i.e. $\log_{10} m$. The Neperian logarithm is $\log_e m$ and usually denoted by $\ln m$. In computing we use instead $\log_2 m$ denoted by $\lg m$. We denote by $\lg m$ the base-2 logarithm of m i.e. the logarithm base-two of m is the value $\lg m$ such $2^{\lg m}$ is equal to m.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

1.4 Data-structure(s). A data structure is a representation of the mathematical model underlying an ADT, **or**, *It is a systematic way of organizing and accessing data.*

If we try to represent an ADT in terms of the data-types and operations of a programming language we use data-structures which are collections of variables of different data-types connected in various ways. In essence, *a data-structure* is a systematic way of organizing and accessing data in a computer.

Does it matter what data structures we use? Yes, it matters if **economy of space** and **easiness of programming** are important. As **running/execution** time is paramount in some applications, we would like to access/retrieve/store data as fast as possible. For the Dictionary ADT previously defined we might use arrays, sorted arrays, linked lists, binary search trees, balanced binary search trees, or hash tables to represent the mathematical model of the ADT to realize its operations. So depending on the data structure used for implementing a Dictionary ADT one operation or another one is more efficient.

We call algorithms **the computational methods that we use** to operate on a data structure.

Note 3. *Recipe vs Algorithm:* How you cook or you brew coffee is a (cooking) recipe. An Algorithm is associated with a computational problem.

1.5 Computational Problem. A computational problem has an input (a set of values) and an output (set of values).

An algorithm establishes the input / output relationship of a computational problem.

A poorly defined computational problem is given below.
Input. Two numbers
Output. The sum of these numbers.

A better formulation follows. In such a formulation we do not need to worry how to deal with negative numbers, integer or real, nor with rational numbers, nor with complex numbers, etc.

Problem 1. Addition of bounded-size non-negative integers.
Input. Two n-bit non-negative integers a, b.
Output. An $(n+1)$-bit integer c such that $c = a + b$.

Note that c is obviously non-negative as well! We might have used digit instead of bit, as in n-digit and $(n+1)$-digit.

Note 4. The number of bits of integer m is proportional to $\lg m$. For a positive integer m the number of bits to represent it is $\lceil \lg (m+1) \rceil$ or equivalently $\lfloor \lg m \rfloor + 1$.

Informal Definition. An *Algorithm* is a "method" that transforms an input into an output. (We avoided intentionally the use of the word "recipe" !!)

1.6 Algorithm. An **algorithm** (for a computational problem) is a well-defined sequence of computational steps that performs a task by generating a set of output values from a set of input values

The input and output values are defined with respect to this computational problem. "Well-defined" means correct. A set of values can be empty, can include only one value or more than one values! An algorithm thus becomes a method for solving well-specified computational problems. A (computational) *problem* defines an input-output relationship. An *algorithm* describes a specific procedure for achieving this relationship. For each computational problem we may have more than one algorithms.

A **sorted sequence** is an increasing sequence or non-decreasing sequence or monotonically-increasing sequence such as $\langle 1, 2, 3, 3, 4\rangle$. A **reverse-sorted sequence** is a decreasing / non-increasing / monotonically-decreasing sequence such as $\langle 4, 3, 3, 2, 1\rangle$. The elements of a sequence are "comparable" i.e. a "total oder" is defined on them in the sense that antissymetry is applicable i.e. $a \leq b$ and $b \leq a$ implies $a = b$, transitivity is applicable i.e. $a \leq b$ and $b \leq c$ implies $a \leq c$ and totality is also applicable i.e. $a \leq b$ or $b \leq a$.

Problem 2. Key Sorting.
Input. A sequence of n keys $\langle a_1, a_2, \ldots, a_n\rangle$.
Output. A reordering i.e. a permutation of the n keys $\langle a_{j_1}, a_{j_2}, \ldots, a_{j_n}\rangle$ of the input sequence so that $a_{j_1} \leq a_{j_2} \leq \ldots \leq a_{j_n}$. (The output is a sorted sequence.)

Thus for an input such as $\langle 10,\ 2,\ 1,\ 5\rangle$ an algorithm for sorting would generate an output that looks like $\langle 1,\ 2,\ 5,\ 10\rangle$.

1.6.1 Input Instance. An input instance of the problem consists of all the inputs needed to compute a solution to the problem. For example, input sequence $\langle\ 10,\ 2,\ 1,\ 5\ \rangle$ is an **instance** of the sorting problem of four keys.

1.6.2 (Correct) Algorithm. An algorithm for a problem is an "Algorithm" if it works correctly for every input instance and halts with the correct output.

An algorithm is a sorting algorithm if and only if it sorts correctly all $n!$ permutations of n keys for all permissible (possible) values of n. A permissible value is determined by the available computational resources (eg memory).

(c) Copyright A. Gerbessiotis.

1.7. Dictionary. A dictionary is an ADT with data consisting of words (aka strings of characters) and few operations involving insertion, deletion and look-up of a word.

An **Insert** is an operation that inserts a word into the dictionary thus increasing its size by one, a **Search** is an operation that searches for a word in an already formed dictionary, and **Delete** an operation that deletes a specified word from the dictionary thus reducing its size by one.

1.7.1 Static Dictionary. For such a dictionary the number of its elements (words) remains the same. A static dictionary is one that does not support operations Insert and Delete. Only a Search or similar operations are allowed. (Think of a book dictionary that allows for look-ups only.)

1.7.2 Dynamic Dictionary. A "full-dictionary" is a *dynamic* data structure that supports operations Insert and Delete in addition to Search. Its size can thus change.

1.7.3 Probing Operations. Static data structures support *"probing operations"* that do not modify the data structure i.e. neither increase nor decrease the number of its elements nor change its structure. Search is one such probing operation. Others include Minimum and Maximum for finding the minimum over all the elements or respectively the maximum over all the elements of the data structure. A Predecessor and Successor of a given element x can also be defined. The Successor of x is the Minimum (smallest) element among those greater than x. Likewise the Predecessor of x is the Maximum (largest) element among those smaller than x.

1.7.4 Modifying Operations. Data structures get modified through "modifying operations" such as Insert and Delete that add to or delete elements from them. An operation ExtractMax could also be a modifying operation: it deletes the Maximum element. Sometimes ExtractMax is known as DeleteMax or RemoveMax or other similar names. On Binary Search Trees operations with the names of Single-Rotation or Double-Rotation do not add or delete elements, yet they both change the structure of the tree. They are considered modifying operations!

1.7.5 A data-structure for the Dictionary ADT. There are many ways one can implement a dictionary. CS505-wise one could use an array, a sorted array, a linked-list or a sorted linked-list, or a binary-search tree to implement a dictionary ADT. (A linked-list can be singly-linked or doubly-linked.) If an array is used, Search would be a slow operation if realized through **LinearSearch**; a sorted array can take advantage of a more efficient search operation known as **BinarySearch**. Thus the data-structure chosen affects the performance of the operations and also dictates the possible algorithms that can be used to realize them.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

1.8 Describing an Algorithm: Pseudocode. We describe an algorithm in this course with *pseudocode* that looks like a procedural (aka imperative) language such as Pascal, C, C++, or Java but we are not concerned with its syntax, variable declarations or definitions and other minute details.

1.8.1 Pseudocode: Including free-form English. We also use free-form English to describe more complex interactions. For example we say 'Find the maximum of three elements" instead of describing how to find the maximum of three elements.

1.8.2 Pseudocode: Black-Box algorithm invocation. If we have introduced an algorithm, for example QuickSort for sorting arrays or sequences of keys, we might invoke it in pseudocode without further details by issuing say a function call to QuickSort(B,m) to sort an array B of m keys with the implication being that at the completion of the function call all elements are sorted in B, even if originally B was not sorted.

1.8.3 Pseudocode: Array Indexing. We are liberal with the boundaries of an array if it fits our purposes (i.e it makes exposition of an algorithm simpler). Therefore we may declare an array of n elements more often as A[1...n] but occasionally as A[0..n-1]. Why ? It causes confusion to say the fifth element of an array is $A[4]$ rather than the more natural and less confusing $A[5]$. We do not bother about error-checking. So if we are to design an algorithm for adding two non-negative integers, we do not bother checking for negative-numbers or real numbers or complex numbers or strings. We assume that the input is correctly formed.

Our intent is to introduce concepts around algorithms, and the ideas behind them. We do not want to spend too much time on implementation issues and divert ourselves from the important design and analysis issues. Normally, it would be easy for a competent programmer to turn the pseudocode into actual code.

1.8.4 Pseudocode: Not necessarily compilable! We will intentionally leave a semicolon missing at the end of line 3 of the code for Fibonacci (you find out in which of the three programs this is so). Furthermore, no variables would be properly defined. And Python-like indentation will be used more often than adding well-balanced braces in statements!

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

1.9 Model of Computation: RAM We analyze the computational performance of an algorithm by first introducing the **computation model** that will be used. Our model is a generic uniprocessor machine called **Random-Access Machine (RAM)**. In a RAM instructions are executed one after the other in unit time. Each memory cell is wide enough to accommodate a word (e.g. for Fibonacci numbers, it can store each one of the numbers however big it is in one word) and each word can be retrieved in unit time. Note that we use the generic term word to refer to keys (for sorting problems), integers (short int, int, long int, etcs), floating point numbers (float, double), or arbitrary precision numbers or other data types. The computation model is as simplistic/optimistic (in resource usage) as possible. The intent is to show that if an algorithm uses A computation resources under the RAM it would also require at least A under more realistic models.

1.9.1 Other Models of Computation: WORD and BIT models. Later on when we discuss arithmetic problems we might want to differentiate between adding two integers and adding two relatively long integers. We would then talk about the **WORD** model where addition of integer a and integer b takes unit time i.e. one step, i.e. one operation (thus RAM is the Word model), and the **BIT** model where the addition of a and b would take time proportional to the number of bits of a and the number of bits of b.

1.9.2 Other Models of Computation: Straight Line Program. When we discuss the WORD and BIT model, we are going to define the Straight Line Program (SLP) model of computation with respect to the WORD model. In SLP we have no conditionals (if statements), no iterative statements (eg for/while loops), no function invocation, no arrays nor table lookups and no indirection (eg no pointers)!

1.10 Problem size. For a computational problem or algorithm for it, problem size or instance size is sometimes confusingly referred to as input size. It is the most important parameter of the input. It is this value that would be used in determining the running time and/or space resources used by the algorithm. Both space and time are to be expressed as functions of this value.

Traditionally we represent problem size with a variable n. It is n to indicate that it takes (usually) integer values and in fact it is a natural integer number (i.e. non-negative and non-zero). The functions for space and time are going to be denoted by $T(.)$ and $S(.)$ respectively. If n is the "problem size", they will become $T(n)$ and $S(n)$. Thus think of n **as being "the most important parameter of the problem; resources such as time or space as expressed as a function of it".**

1.11 Input size. If problem size sounds confusing, input size can refer to the number of elements of the input or the size of all of the input! (We are not going to say anything about the unit used for input size but it should be there!)

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Problem 1 and Problem 2 (pages 3 and 4). For both problems, problem size is n.

For Problem 1, n is the number of bits for each one of the two input integers a and b. Input size is an atrocious two integers (WORD model) or $2n$ bits (in the bit model). The fact that we utilize the name n in the expression $2n$ to express input size is enough to convince us that the most important parameter of the problem is n, its problem size. For Problem 2 things are easier. We have n input keys so problem size is indeed n. As far as input size is concerned this consists of n keys i.e. it is n keys as well.

Note that for input size we need to say something about the unit of measurement. It is (or was) integers, bit(s) or keys. And it is always there!

1.12 Resources. Computational resources expended during the execution of an algorithm are in the form of cpu-cycles used (or wasted) and amount of memory utilized (or expended). Performance measures established to describe the performance of an algorithm can associate with them directly. In other words **TIME** and **SPACE** are of paramount importance. In more complex problems other performance measures such as throughput, latency, communication bandwidth, logic gates, efficiency or speedup might also be defined and used. With **TIME** we mean how long an algorithm runs to completion with **SPACE** how much memory/space the algorithm uses.

We then find problem size and express those performance measures as a function of problem size. How do we measure space and time? We do not measure space in bytes (or megabytes). Rather in words or in multiples of the size of the elementary data type used (eg key, number). And for time forget nanoseconds or femtoseconds. In the best of cases it would be "operations" in general (mixing up additions, assignments, read/write operations) or if we can be more discriminating, we would identify the most fundamental operation (eg comparison of keys or addition of numbers) and express time in terms of those fundamental operations! This means what we measure as time and space for example is not a precise measure. If we count number of keys instead of bytes, we need to know how many bytes we use to store a key to be precise. Likewise if the cost of an addition is one operation, to get nanoseconds we need to know clock-cycles and other properties of a CPU on which execution is to take place. This is impossible to do in a generic fashion. Thus for such an analysis we do not aim for precision. Just for good enough accuracy!

Thus multiplicative constants and low-order terms becomes less important. And when we compare algorithms make sure that operations that are compared are the same. For example if an algorithm uses $3n-2$ multiplications and another one uses $4n$ additions, which of the two is faster?

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

1.13 Describing functions such as $T(n)$ and $S(n)$. With functions $T(n)$ and $S(n)$ details (low-order terms or multiplicative constants) are not important. Some of those details are hidden in the units (eg keys vs bytes, operations vs nanoseconds). But how do we remember all the constants if we need to write a $S(n) = 2n + 3$ or $S(n) = 2n$? The easiest thing to remember is to say that "$S(n)$ is linear to n". We would denote this as $S(n) = \Theta(n)$. Or if $S(n) \leq 3n$ to say that "$S(n)$ is upper bounded by a linear function to n" we would denote it by writing $S(n) = O(n)$. A $S(n) \geq n$ to say that "$S(n)$ is lower bounded by a linear function to n" would become $S(n) = \Omega(n)$. The symbols are known as Theta, Big-Oh and Big-Omega respectively. A $\Theta(1)$ means a constant!

1.13.1 Comparing Performance: Direct Comparison. Consider two algorithms one that requires n^2 operations and one that requires $n \lg n$ operations to sort n keys Here, $\lg n = \log_2 n$ **is the base two logarithm of** n i.e. $\lg 1 = 0$, $\lg 2 = 1$, $\lg 256 = 8$, $\lg 1024 = 10$, $\lg 65536 = 16$, and so on. Consider a machine that executes 10,000,000 such operations per second with a millisecond precision clock. The two algorithms solve instances of various sizes n in the following time.

```
        n      =  2            10^3        10^6        10^9
Alg A: n^2     =  <0.001sec    0.1sec      1day        300years
Alg B: n lgn   =  <0.001sec  0.001sec      2sec        1hour
```

Review. $2^{\lg n} = n$, $2^{c \lg n} = (2^{\lg n})^c = n^c$. $2^{a \cdot b} = (2^a)^b = (2^b)^a$.
Review. $a^{\lg n} = (2^{\lg a})^{\lg n} = 2^{\lg a \cdot \lg n} = (2^{\lg n})^{\lg a} = n^{\lg a}$, $x^{a+b} = x^a \cdot x^b$.

1.13.1 Comparing Performance: Asymptotic Comparison. By comparing n^2 and $n \lg n$ we draw the conclusion that for small values the performance of the two algorithms is indistinguishable. For large values it is not. In our example comparison was easy; there were no low-order terms nor multiplicative constannt involved. Consider for example the case where one algorithms requires n^2 operations and the other $20n \lg n + 1000n + 10^9$. Things become more complicated calculator-wise. To establish asymptotic performance and comparison we only need to now what happens as $n \Rightarrow \infty$. For this we just need to $\lim_{n \Rightarrow \infty} n^2 / n \lg n$. It is not difficult to find that the limit is infinity and thus the n^2 algorithm is too costly operation-wise (slower) than the $n \lg n$ algorithm. And the same applied to n^2 and $20n \lg n + 1000n + 10^9$.

(c) Copyright A. Gerbessiotis.

1.14. Design Techniques for Algorithms.
There are various approaches one can use to design algorithms for various computational problems. We call those approaches, **methods** or `techniques`. (Less often we might refer to them as principles.)

1.15 Incremental method. Sometimes the output(s) of a problem can be computed iteratively using an iteration approach – and this is implemented with a for or a while loop – that generates the output incrementally one-element at a time or by walking through the input and generating a solution one element of the input at a time `incrementally`. This lends the name to this method. It is known as the incremental method but several times it is referred to as the iteration or the iterative method. The iterative method name is more often used when the output is generated by (or in) the last iteration.

Problem 3. n-th Fibonacci Number (aka Fibonacci Sequence or Function).
Input. An integer $n \geq 0$.

Output. Integer F_n, the n-th Fibonacci number defined as follows: $F_n = F_{n-1}+F_{n-2}$ if $n > 1$ where $F_0 = 0$ and $F_1 = 1$

1.15.1 Incremental method and Fibonacci: Algorithm and Space resourses. The Fibonaci sequence definition lends itself to an incremental method solution as shown below! A simple for loop computes in the i-th iteration the i-th Fibonacci number for $i \geq 2$. The 0-th and 1-st numbers are precomputed. This naive solution computes not just the n-th Fibonacci number but also all previous Fibonacci numbers and stores them into array F. It looks as if it is wasting resourses. Amount of space used is $n+1$ elements for array F (storing the 0-th, 1-st throught n-th Fibonacci numbers plus space for n and index i. We can say $n+3$ or linear to n or proportional to n but $S(n)$ is described succinctly as $S(n) = \Theta(n)$ using the prior page discussion! Parameter n is problem size.

Question. How would you rewrite the algorithm so that $S(n) = \Theta(1)$. Is it possible ?

```
   F (n)    //Incremental
1. F[0]=0; F[1]=1;
2. for(i=2;i<=n;i++) {
3.  F[i]=F[i-1]+F[i-2]
4. }
5. return(F[n])
```

1.15.2 Incremental method and Fibonacci: Time resources. Finding the running time to completion of the algorithm does not make much sense if one count CPU cycles. How many cycles does it take to perform line 1? May be a constant but is it 1, 10 or 1000? How about line 4? Is it more complicated or of similar complexity? How about lines 2 and 3, let alone line 5? We could make an argument that executing ONCE each line takes c_i cycles where i takes values $i = 1, 2, 3, 4, 5$. How many times is line 1 executed? Obviously once. How many times is line 3 executed ? Obviously as many times as the inner loop is executed i.e. $n-1$ times. But how many times is the for-statement gets executed? This is one more than the inner-loop. This because $i \leq n$ is executed successfully $n-1$ times and execution proceeds with line 3, but at the last iteration i is set to $n+1$ and $i \leq n$ fails (evaluates to FALSE) and controls goes to the line after line 4, i.e. line 5 and the code terminates. A debated question is the execution of line 5 i.e. the right brace. The moment control hits the right brace variable `i` gets incremented and then the test $i \leq n$ takes place. You can consider it part of the for outter-loop or the inner-loop. It sends control back to line 2, a total of $n-1$ times. Thus let us summarize `T(n)` the running time of Fibonacci.

$$T(n) = 1 \cdot c_1 + n \cdot c_2 + (n-1) \cdot c_3 + (n-1) \cdot c_4 + 1 \cdot c_5$$

It is a relief to be able to say that $T(n) = \Theta(n)$.

1.15.3 Running time is worst-case running time bound. No matter what, for input n, each one of the lines of the five-line code would be executed as stated and eventually F_n would be output. There is no better or worse scenario. The running time is the worst-case running time (and there is no best-case running time either). Everything is worst-case.

1.15.4 Running time in terms of major operations performed: additions One thing that is clear is that the most important computational operation is the addition of two Fibonacci numbers. This is line 3. We could thus describe running time in terms of additions. If $A(n)$ is the number of additions to compute F_n we have $A(n) = n - 1$. If we do not want to remember that -1, we can also say $A(n) = \Theta(n)$. Thus whether we measure running time or just additions performed the conclusion is the same: a linear running-time operation involving this naive incremental approach for computing F_n.

1.16 Recursion and Recursive functions. A **recursive function** is a function that invokes itself. In **direct recursion** a recursive function f invokes directly itself, whereas in **indirect recursion**, function f invokes.

1.17 Algorithm Design Technique 2: Divide and Conquer. With Divide-and-Conquer, to solve a problem we first DIVIDE the problem into two or more sub-problems, CONQUER (i.e. solve) those sub-problems, and then combining those two solutions to get the solution for the original problem.

1.18 Fibonacci and the divide-and-conquer method. The solution is shown below. In order to solve a problem (i.e. compute $F(n)$, the n-th Fibonacci number) DIVIDE the problem into two or more sub-problems, those of computing the $(n-1)$-st and $(n-2)$-nd Fibonacci numbers, CONQUER those problems i.e. solve them by first solving the first by computing $F(n-1)$ and then solving the second by computing $F(n-2)$ and then combining those two solutions to get the solution for $F(n)$. Combining in this context means addition since $F(n) = F(n-1) + F(n-2)$. In 3 lines Divide and Conquer computes F_n through a call to $F(n)$.

```
    F(n)   // Divide + Conquer
1. if (n==0) return (0);//F[0]
2. if (n==1) rerurn (1);//F[1]
3. return(F(n-1) + F(n-2) ); //F[n]
```

1.18.1 Performance of divide-and-conquer method. The time $T(n)$ to compute $F(n)$ depends on the time $T(n-1)$ and $T(n-2)$ to compute $F(n-1)$ and $F(n-2)$ respectively. Thus the running time follows a Fibonacci-like time bound. (The difference being that $T(0) = T(1) = c$ for some constant $c > 0$. (This is the time of a line such as Line 1 or Line 2.) So in general, the recursive **divide-and-conquer** solution is easier to design and implement. It is, however, less efficient as recursive calls are costly in terms of space and time utilization: each call requires space on the program stack, and context switching from one to another recursive call is quite expensive (e.g. 100s of nanoseconds, even tens of microseconds). Furthermore its analysis requires the solution of a recursive function also known as a **recurrence** or **recurrence relation**.

Combining all the information derived so far, $T(n) = T(n-1) + T(n-2) + c'$. The c' reflect the contribution to the running time of the addition of Line 3 as well as the failed if statements of Lines 1 and 2. $T(n-1)$ and $T(n-2)$ account for the two recursive calls of line 3.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Exercise 1.19 (a) Show that $F_n \leq 2^n$ (or $F_n \leq 2^{n-1}$) by using induction (and also, $F_n \geq 2^{n/2-1}$).
(b) Find the exact solution for F_n using Discrete Math techniques.
(c) What is the number of function (recursive) calls in the recursive solution?
(d) Show that $T(n) \geq 2^{n/2-1}$.

Solution for 1.19(a). Show $F_n \leq 2^n$ for all $n = 0, 1, 2, \ldots$.

We call $P(n)$ the statement $F_n \leq 2^n$. We thus want to prove $P(n)$ true hopefully for all $n = 0, 1, 2, \ldots$.

Base Case $n = 0$**.** Show that $P(0)$ is true i.e. show that $F_0 \leq 2^0 = 1$. Since $F_0 = 0$, it follows immediately that $F_0 = 0 \leq 1$. We have just proved that $P(0)$ is true.

Inductive Step. If $P(0), P(1), \ldots, P(n-2), P(n-1)$ are all true, then $P(n)$ will also be true. Equivalently, if $F_i \leq 2^i$ for all $i = 0, \ldots, n-1$, then $F_n \leq 2^n$. The part $P(0), P(1), \ldots, P(n-2), P(n-1)$ i.e. the left side of the implication is also known as the induction hypothesis. This is what we call strong induction: all $P(0), P(1), \ldots, P(n-2), P(n-1)$ appear in the induction hypothesis. If only $P(n-1)$ appears it is known as (weak) induction.

Proof of the Inductive Step. If $P(0), \ldots, P(n-2), P(n-1)$ are all true, then $P(n-2)$ is true i.e. $F_{n-2} \leq 2^{n-2}$. If $P(n-1)$ is true then $F_{n-1} \leq 2^{n-1}$. Then

$$F_n \;=\; F_{n-1} + F_{n-2} \leq 2^{n-1} + 2^{n-2} \leq 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n.$$

We have thus proved that $F_n \leq 2^n$ thus showing that $P(n)$ is true.

What does this mean?

1. We have just proven that the implication is true $P(n-1), P(n-2) \Rightarrow P(n)$ along with the base cases(s) $P(0), P(1)$.
2. Since $P(0)$ and $P(1)$ are true, by way of the implication $\Rightarrow$, for $n = 2$, we conclude that $P(2)$ is true.
3. Since $P(0), P(1), P(2)$ are true, by way of $\Rightarrow$, for $n = 3$, we conclude that $P(3)$ is true.
4. Likewise this implies $P(0), \ldots, P(n-1), P(n)$ are all true. Thus $P(n)$ is true for all $n \geq 0$

Note. Sometimes the base case $P(0)$ cannot be (easily) proved. Then we pick another base case say $n = 3$ to show that $P(n)$ is true for all $n \geq 3$.

Solution for 1.19(d). Let us ignore the c' of the previous page (it is $c' \geq 0$). Then $T(n) \geq T(n-1) + T(n-2) \geq 2T(n-2)$. Likewise $T(n-2) \geq 2T(n-4)$. Thus $T(n) \geq 2T(n-2) \geq 2^2T(n-4) \geq 2^3T(n-6) \geq 2^iT(n-2i)$. Setting $i = n/2$ for n even or $i = (n-1)/2$ for n odd the result follows (and our result is a bit more forgiving the lower bound is $n/2 - 1$ and the worst of the two proven is $(n-1)/2$). Thus the Divided-and-Conquer is an exponential running time algorithm!

1.20 Algorithm Design Technique 3: The dynamic programming method (memoization). If one thing is obvious about the recursive divide-and-conquer approach is the waste of computational resources. The running time of the divide and conquer method is exponential to n whereas the running time of the incremental method is linear! n is much smaller than 2^n. Compute both functions for $n = 10, 100, 1000, \ldots$ or if take the limit $\lim_{n\to\infty} n/2^n = 0$. This is what we called before an **asymptotic comparison** of two functions (n vs 2^n here, $n \lg n$ and n^2 previously).

Recursion is a duplicating process. The number of times the recursive function is invoked with $F(1)$ or $F(2)$ is a Fibonacci number depending on n (check whether it is $F(n-1)$ or $F(n-2)$ or $\ldots$). We can eliminate duplication if we keep track of the Fibonacci numbers whose values have already been computed thus computing each value ONCE and only ONCE. Subsequently we do not recompute but retrieve a previously computed value or skip the computation altogether to avoid overhead costs. For this we need to store previously computed values in a storage area (array in memory). Thus we get

```
0.      F(n)    // Memoization
1.      table int T[0..n];
2.      T[0]=0; T[1]=1; T[ 2<=i <=n] = UNDEFINED;
3.      return(TableRecursive_Fn(n,T));

0.     int TableRecursive_Fn(n,T);
1.      if (n==0) return (0);//F[0]
2.      if (n==1) rerurn (1);//F[1]
3.      if T[n-1] != UNDEFINED left= T[n-1];
4.         else                left= T[n-1]= TableRecursive_Fn(n-1,T);
5.      if T[n-2] != UNDEFINED right= T[n-2];
6.         else                right=T[n-2]= TableRecursive_Fn(n-2,T);
7.      return(left + right);
```

This third algorithmic design technique is known as **dynamic programming**. Think about dynamic programming as divide and conquer without duplication of work that uses additional storage to retrieve previously computed results and avoid recomputing them. The term **memoization** refers to the additional storage required to making it work!

1.21 InsertionSort. We present a solution to the sorting problem by presenting a sorting algorithm that is known as *insertion sort*. InsertionSort is an **incremental** algorithm. It solves the sorting problem, i.e. generates the output sequence one key at a time. InsertionSort works as follows. We take each key of the input sequence in turn (starting from the leftmost one) and insert it into the correct position in the output sequence that is being formed. Below, in column Input/Output what is left of | is the output sequence being formed and what is on the right of | is the input sequence remaining. The * indicates the input key that is being inserted in the output sequence.

```
Example     Input                 Output                Input Output
Step 0.    5 2 4 6 1 3                                 |5  2  4  6  1  3
Step 1.      2 4 6 1 3           5*                    5| 2  4  6  1  3
Step 2.        4 6 1 3           2* 5                  2* 5| 4  6  1  3
Step 3.          6 1 3           2  4* 5               2  4* 5| 6  1  3
Step 4.            1 3           2  4  5  6*           2  4  5  6*|1  3
Step 5.              3           1* 2  4  5  6         1* 2  4  5  6| 3
Step 6.                          1  2  3* 4  5  6      1  2  3* 4  5  6|
```

1.21.1 Sorting in-place. A sorting algorithm **sorts in place**, if the same space is used for input and output and the extra memory used by the algorithm is constant. [There is a problem when one needs to assess a recursive sorting algorithm: its non-recursive version is being used to determine its in-place status.]

1.21.2 Stable Sort. A sorting algorithm **is stable** if the relative order of same valued keys remains the same in the input and output sequences. [if the input keys are all distinct, stability is NOT an issue.]

1.21.3 Example 1. Consider the problem of sorting student records, each record consisting of a college-year field (1:Freshman, 2:Sophomore, etc) and a name field. Somehow the input records are sorted by name, and we'll sort them by college-year.

```
Input: (sorted by name left to right)
 <<2,Pat>,<2,Tim>,<2,Tom>,<1,Vic>>        <<2,Pat>,<2,Tim>,<2,Tom>,<1,Vic>>
      Stable Algorithm gives                   Non-stable algorithm may give
 <<1,Vic>,<2,Pat>,<2,Tim>,<2,Tom>>        <<1,Vic>,<2,Tom>,<2,Pat>,<2,Tim>>
```

Both are legitimate sorted by college-year sequences. The former however retains the original sorted order for same valued keys (i.e. all sophomore students are in order).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

1.21.4 Example 2. Consider the problem of Google sorting triplets of the form $\langle \text{Doc} - \text{ID}, \text{word} - \text{ID}, \text{index} \rangle$ where $\text{Doc} - \text{ID}$ is the document id of a document (i.e. a URL turned into a number), word-ID is the word id of a given word (i.e. a word turned into a number) and index is the starting position of an instance of the word in the document indexed by Doc-ID.

When the Google indexer parses a single document it generates for that document triplets with the same Doc-ID values. Because a document is parsed beginning to end, the triplets are sorted by index for a given Doc-ID. All such triplets of all documents are collected (and thus are sorted based on Doc-ID as well). However the Google indexer wants to sort things based on word-ID but at the same time does not want to destroy the existing orderings based on doc-ID and also index.

```
Google parser: processes one document at a time: triplets sorted by Doc-ID and index.
<0,"algorithm",10><0,"data",20><0,"algorithm",30>  <1,"data",4>  <2,"data",20><2,"data",40>

Stable sorting: Sort by word-ID (sorting by index is for free)  ordering by index/doc-ID
<0,"algorithm",10><0,"algorithm",30><0,"data",20><1,"data",4><2,"data",20><2,"data",40>

Not stable sorting: Sort by word-ID  but previous orderings might get lost
<0,"algorithm",30><0,"algorithm",10><2,"data",40><1,"data",4><0,"data",20><2,"data",20>
```

1.21.5 Pascal Indexing (aka 1-to-n Indexing). A pseudocode for Insertion sort may look like this.

```
 InsertionSort (A[1..n])                                          Cost  TimesExecuted
1.  for (j=2;j<=n;j++)                                             1     n
2.     key= A[j];                                                  1     n-1
3.     i=j-1;// Insert A[j] in the sorted sequence A[1..j-1]       1     n-1
4.     while (i>0) and (A[i] > key) {                              1     t2+...+tn
//     Note                 ^^^ : This makes the algorithm stable
5.       A[i+1]= A[i]                                              1     (t2-1)+...(tn - 1)
6.       i=i-1                                                     1     (t2-1)+...(tn - 1)
       }
8.     A[i+1]=key;                                                 1     n-1
```

1.21.6 C/C++/Java Indexing (aka 0-to-$(n-1)$ indexing). An equivalent C/C++ oriented code follows.

```
 InsertionSort (A[0..n-1],n)                                    Cost  TimesExecuted
1. for (j=1;j<n;j++) {                                            1    n
2.   key= A[j];                                                   1    n-1
3.   i=j-1;// Insert A[j] in the sorted sequence A[0..j-1]        1    n-1
4.   while (i>= 0) and (A[i] > key) {                            1    t1+t2+...+tn-1
//     Note                ^^^ : This makes the algorithm stable
5.     A[i+1]= A[i];                                              1   (t1-1)+..+(tn-1 - 1)
6.     i=i-1;                                                     1   (t1-1)+..+(tn-1 - 1)
     }
8.   A[i+1]=key;                                                 1    n-1
9. }
```

1.22.1 Question 1. How many times is line 1 executed ? **Answer:** n

1.22.2 Question 2. How many times is line 4 executed ? **Answer:** It depends on the input sequence. For key $A[j]$ let tj be that number!

```
Input:   Sorted        Reverse-Sorted     Random
       <1 2 3 4 5>     <5 4 3 2 1>        <2 5 1 4 3>
       <1>             <5>                <2>
       <1 2>           <4 5>              <2 5>
       <1 2 3>         <3 4 5>            <1 2 5>
       <1 2 3 4>       <2 3 4 5>          <1 2 4 5>
       <1 2 3 4 5>     <1 2 3 4 5>        <1 2 3 4 5>
       4 comparisons   10 comparisons     9 comparisons ?
```

1.22.3 Question 3. How many (key) comparisons on a sorted sequence of n keys? **Answer.** $n-1$. The comparison $A[i] > key$ is performed exactly once.

1.22.4 Question 4. How many (key) comparisons on a reverse-sorted sequence of n keys? **Answer.** $n(n-1)/2$. The comparison $A[i] > key$ is performed $j-1$ times for `key=A[j]`.

1.22.5 Question 5. How many (key) comparisons on a random sequence of n keys? **Answer.** $n(n-1)/4$ on the average.

1.22.6 Question 6. How many (key) comparisons on n arbitrary keys? **Answer.** Arbitrary does not mean random! At most the worst number i.e. $\leq n(n-1)/2$.

1.22.7 Question 7. What comparisons are we talking about? **Answer.** Of the type $A[i] > key$ and nothing else i.e. comparisons between keys! That is $i \geq 0$ is NOT such a comparison! (An index "comparison" $5 > 4$ is a subtraction $5-4$ plus checking the sign of the result.)

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

In the remainder we talk about running time in general: it inludes key comparisons such as those of line 4 but also everything else!

1.23 Running time on an input instance. The running time of an algorithm depends on the **input** (i.e. input instance) and **problem size** and thus it is expressed as a function of n.

1.23.1 Running time of InsertionSort on some inputs. The running time of InsertionSort on n sorted keys is $T(n) = \Theta(n)$. (As the number of comparisons is $n-1$ this draws directly from 1.22.3. and the pseudocode.) The running time of InsertionSort on n reverse-sorted keys is $T(n) = \Theta(n^2)$. (As the number of comparisons is $n(n-1)/2$ this draws directly from 1.22.4. and the pseudocode.)

1.24 Running time of an algorithm (InsertionSort). The running time of InsertionSort is bounded by the worst-possible running time on an input instance. Thus $T(n) = O(n^2)$. This is because the running time is bounded above by the worst-case's running time. (It is also bounded below by the best-case's running time.) On an arbitrary input instance the running time is between the lower bound and the upper bound. The maximum (the upper bound) determines the running time of an algorithm.

1.25 What is the number of comparisons of InsertionSort. This number is bounded above by $n(n-1)/2$ and below by $n-1$. Thus $C(n)$ (if we denote by $C(n)$ the number of comparisons), it is $C(n) = O(n^2)$ but also $C(n) = \Omega(n)$. We can also say explicitly and more precisely and accurately that $n-1 \leq C(n) \leq n(n-1)/2$. The lower bound and the upper bound is exact (as far as comparisons are concerned). For running time they are asymptotic. This confirms that the running time of InsertionSort is determined by the operation count of its most important operation, the key comparison of line 4.

1.26 Is really every line of the pseudocode of cost 1? Let line i requires c_i operations instead of 1.

The running time $T(n)$ of InsertionSort for an input of size n is thus given by the following expression.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_7(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1)$$

1.27.1 Best case. For the best-case, it is $t_j = 1$ for all j. Therefore, a best-case is a case where the input is already sorted. The best-case running time is then.

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_7 + c_4)$$

Running time is linear in n, or $\Theta(n)$.
1.27.2 Worst case. For the worst-case, it is $t_j = j$ and therefore, a worst-case is a case where the input is reverse-sorted. The worst-case running time is then.

$$T(n) = (c_4 + c_5 + c_6)n^2/2 + (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7)n - (c_2 + c_3 + c_7 + c_4)$$

Running time is quadratic in n, or $\Theta(n^2)$. In this case, we used the fact that $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ and $\sum_{j=2}^{n}(j-1) = n(n-1)/2$.
1.27.3 Average case. On the average, key $A[j]$ is smaller (or bigger) than half of the $j-1$ keys already considered, i.e. $t_j = (j-1)/2 + 1$ (the one is for the first condition of the `while` that is always executed). Running time is "half" of that of the worst case, or asymptotically speaking $\Theta(n^2)$ as well.
1.27.4 Constants do not matter. It is worth noting that for 1.27.1 and 1.27.2 the important conclusions are the **"linearity"** and **"quadracity"** of the running time. We never assigned values to the c_i constants. We implicitly assume $1 + 1 = 1$ i.e. the sum of two constants is another constant.

MergeSort

Many algorithms are **recursive** in nature that is, in order to solve a given problem they call themselves recursively to solve similar smaller sized problems. The Fibonacci numbers gave rise to a recursive algorithm for their computation that followed **the divide-and-conquer** approach. Such a **divide-and-conquer** approach thus involves three steps.

1. **Divide** the problem into a number of independent subproblems.

2. **Conquer** the independent subproblems by solving them recursively. The recursion terminates when subproblem size is so small that it can be solved directly (e.g. n is 0 or 1 for Fibonacci).

3. **Combine** the subproblem solutions to get the solution of the original problem.

We present a new sorting algorithm that is known as **MergeSort** and whose growth rate is $n \lg n$, i.e. its comparison count is smaller than insertions sort and this makes the algorithm faster that insertion sort.

1. Divide Phase: Divide n numbers into two sequences of $n/2$ numbers each.

2. Conquer Phase. Recursively sort the two sequences (recursion terminates when a sequence of size 1 is to be sorteds).

3. Combine Phase. **Merge** the two sorted sequences of size $n/2$ to produce the sorted result.

For the Divide Phase to work nicely we may (theoretically assume) that n is divided by two. A simple theoretical assumption is that n is an integer power of two. Thus the half of the half of the half etc of n is still an integer!

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

An example plus Analysis

```
              A=<1 8 4 3 2 5 7 6>                Divide phase  starts
      <1 8 4 3>                   <2 5 7 6>
  <1 8>       <4 3>           <2 5>       <7 6>
<1>  <8>     <4> <3>         <2> <5>      <7> <6>  Divide phase  ends
                                                  Conquer phase starts
<1>  <8>     <4> <3>         <2> <5>      <7> <6>  One-key sequence sorted
                                                  Merge into two-key sequences
  <1 8>        <3 4>          <2 5>       <6 7>    Two-key sequences sorted
                                                  Merge into four-key
      <1 3 4 8>                   <2 5 6 7>        Four-key sequences sorted
                                                  Merge into eight-key
                <1 2 3 4 5 6 7 8>                 Eight-key  sequence sorted
```

```
 MergeSort(A){                                    Running Time
1. MergeSort(A,0,n-1);//if A[0..n-1]              T(n)
// MergeSort(A,1,n);  //if A[1..n]
 }
 MergeSort (A,l,r)                                Running Time
//Sort numbers in A[l..r]                         for l=0/1, r=n-1/n
1. if (l < r)                                     1
2.   m= floor((l+r)/2);                           1
3.   MergeSort(A,l,m);                           T(ceiling(n/2))
4.   MergeSort(A,m+1,r);                         T(floor(n/2))
5.   Merge(A,l,m,r);                             c4 n
//Merge(A,l,m,r) merges sorted sequences A[l..m] and A[m+1..r] using auxiliary space B[l...r]
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Running Time

Let $T(n)$ be the running time of MergeSort for sorting n keys, i.e. the running time of MergeSort(A,0,n-1).

Then $T(n)$ is given by the following expression (assuming n is a power of two to eliminate problems with floors and ceilings). Let $T(1) = 0$.

$$T(n) = 2T(n/2) + cn.$$

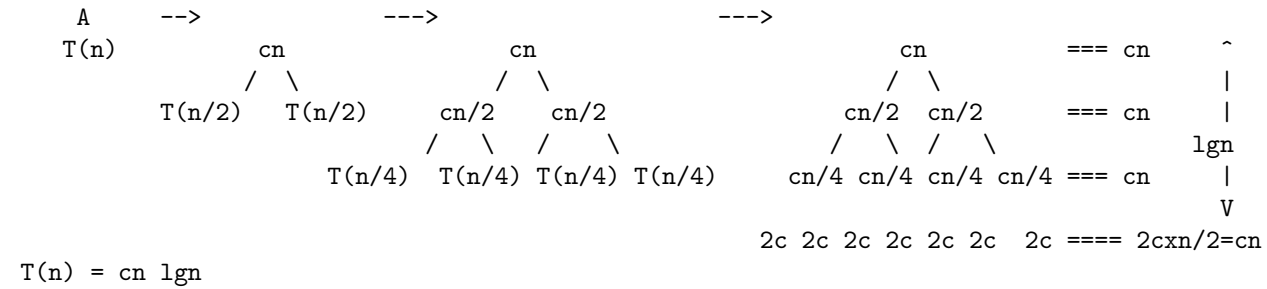
Since $n = 2^k$, for some integer k, we have by induction that $T(n) = dn \lg n$ for some constant d.

Question. How do we merge (in Step 5) in linear time?

Question. Does MergeSort sort in place (No! : Merge uses B[l..r]).

Question. Is MergeSort a **stable sorting** algorithm? Depends on Merge. If a key x from A[l..m] is compared to y from A[m+1..r] and they are equal, and x is placed in the output sequence first, then MergeSort is stable, if y is placed first then MergeSort is NOT stable. The latter however is a pathological case: in general merge sort is considered stable.

Remark. Note that for running time issues, lines of code from now are assumed to have cost 1 not c (a constant) and for simplicity $1 + 1 = 1$ (to mean $c_1 + c_2 = c_3$, i.e. the sum of two constants is a constant).



T(n) = cn lgn

1.10.1 MAXIMUM of n keys. Suppose we have n keys stored in an array $A[1..n]$, i.e. a sequence of n elements. Note that the first element of the array, $A[1]$ is indexed 1. We would like to find the MAXIMUM of those n elements as efficiently as possible. The word "efficiently" is being used now without a proper prior definition. We describe a solution that is indeed efficient in pseudocode. It can be easily turned into compilable C or C++ or even Java code: we choose not to do so!

```
    MAXIMUM( A[1..n], n)
1.  result = A[n];
2.  for(i=n-1 ; i >= 1 ; i--)  {
3.   if A[i] > result
4.     result = A[i];
5.  }
6.  return(result)
```

```
    MINIMUM( A[1..n], n)
1. results = A[1];
2. for(i=2; i <= n; i++) {
3.  if A[i] < result
4.    result = A[i];
5. }
6. return(result)
```

We find the maximum of the last element (obviously itself), then the maximum of the last two, three and so on. Every time a new maximum is found its value is stored into result. If we have found the maximum of the last $n-i$ elements that are stored in positions $A[n], \ldots, A[i+1]$ and this maximum is maintained in variable `result` we then go to the next element of the sequence which is $A[i]$. We do not need to compare this element with all previous $n-i$ elements, we just need to compare $A[i]$ with the maximum of the last $n-i$ elements already stored in `result`. Thus line 3 determines whether $A[i]$ is the new maximum of $A[n], \ldots, A[i+1], A[i]$. If this is the case `result` gets updated in line 4, otherwise it is not, and this (non-update) means that $A[i]$ is not the largest of the last $n-i+1$ elements! Either way at the end of line 4 the value of `result` correctly reflects the maximum of $A[n], \ldots, A[i]$. Likewise at the exit of the for loop of line 5 (or line 6) `result` reflects the maximum of $A[n], \ldots, A[1]$, i.e. of all the n elements. Note also that `result` is not updated if a repeat value is encountered; thus if all the elements are 10, result will never be updated in line 4, as it will be set to 10 in line 1 and the for loop will never encounter a larger value. This means that if we have 10 elements ($n = 10$) each one being a 5, the maximum stored in result will be the first one in position $i = 1$ stored through line 1. The subsequent occurrences of 5 will not cause line 4 to be executed.

We also show the code for MINIMUM.

Question. Is there a reason one scans the input left to right and the right to left? We can write the code for MAXIMUM one way or the other way. However if the function is to be used by a sorting algorithm to be called SelectionSort, for SelectionSort to be stable, if all keys are equal, the minimum must be the first (of equal) keys and the maximum the last.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

1.10.2 Running time of MAXIMUM. The running time of MAXIMUM (or MINIMUM for that matter) is not too complicated to derive. Line 1 is executed once. Line 2 is executed n times, of which $n-1$ succesfully that lead to decrementing i and once unsucessfuly when i is 0 (adjust accordingly for MINIMUM). Line 3 is executed $n-1$ times. However Line 4 can be executed anywhere from 0 to $n-1$ times. Adding up the contributions of all lines we conclude $T(n) = c_1 \cdot 1 + c_2 \cdot n + c_3(n-1) + L4$. (We denote c_i the execution time of line i, some constant.) $L4$ is the running time of Line 4. Thus $0 \leq L4 \leq c_4(n-1)$. Therefore $T(n) \leq Cn$ and $T(n) \geq Dn$ for some constant C, D dependents on the c_is. In other words $T(n)$ is a linear function to n and thus $T(n) = \Theta(n)$.

1.10.3 Number of comparisons of MAXIMUM. This is easier to compute. Line 3 is executed $n-1$ times exactly. Thus to find the maximum of n keys we need to perform $n-1$ key comparisons (such as those between $A[i]$ and `result`).

1.10.4 Can we find MAXIMUM with at most $n-2$ comparisons in a worst-case scenario? NO. $n-1$ is the best we can do. Every key that is not the maximum must be disqualified form being MAXIMUM. In every comparison we have one diqualification. We need to have $n-1$ disqualifications for the $n-1$ keys that will not be the MAXIMUM, thus we need at least $n-1$ comparisons. Think it also the following way. If there is an algorithm that 'finds' the MAXIMUM in $n-2$ comparisons it means that at most $n-2$ keys got disqualified, one was decided to be the MAXIMUM and (at least) one key never got disqualified. Change the value of that key to something larger than the MAXIMUM value and repeat the comparisons of the 'n-2' algorithm. All remaining keys have the same values as before so all previous steps/comparisons would be repeated. The only problem is that the wrong MAXIMUM would be reported now! (The previous MAXIMUM is not the MAXIMUM any more as a keys has a value greater than that.)

1.10.5 Is MAXIMUM time efficient? We just proved that MAXIMUM is comparison efficient. Thus it is also time efficient. In the worst-case we can do no better than $\Theta(n)$!

1.10.6 Is MAXIMUM space efficient? We are frugal in space allocation for MAXIMUM; other than A that holds the keys/elements, we only used i and `result` i.e. a constant number of variables (i.e. $\Theta(1)$ variables). So it is space efficient (asymptotically). No matter what you do you need one variable to hold the answer, and the difference between one and two is not asymptotically important!

1.10.7 Is MAXIMUM efficient? Yes because it is space and time efficient!

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

SelectionSort and BubbleSort

Both insertion sort and mergesort are what we characterize as **comparison-based** sorting algorithms. They sort by comparing two keys of the input sequence at any time and depending on the outcome of this comparison a decision is made to exchange the keys or not (note the delayed exchange in insertion sort for optimality purposes). Other comparison-based sorting algorithms are selection sort and bubble sort.

Selection sort is an incremental algorithm. Let $A_n = A$ be the sequence of input keys. In the i-th iteration of selection sort the minimum m of A is found and is placed in the output sequence in position i and then m is removed from A and thus A_{n-1} contains the remaining $n-1$ keys. This can become in-place if instead of swapping `A[index]` and `A[i]` we shift the keys to the right starting with the key at i and ending with the keys at `index-1` and then concluding moving the original `A[index]` into `A[i]`.

```
   SelectionSort(A[1..n])
1.  for(i=1;i< n ; i++) {
2.   index=MIN(A[i...n]);
3.   swap(A[i],A[index]);
    // To be stable this must be
    // 3 temp=A[index];
    // 4 ShiftR(A[i..index-1],A[i+1..index] ;
    // 5 A[i]=temp;
   }
```

```
   SelectionSort(A[0..n-1])
1. for(i=0;i < n-1 ; i++) {
2.  index=MIN(A[i..n-1]);
3.  swap(A[i],A[index]);
}
```

BubbleSort (an unoptimized version of it) is also known as odd-even transposition sort and can also be used for parallel sorting. The latter consists of rounds 1,2,3, etc. In an odd round i ($i = 1, 3, 5$, etc) odd-indexed keys $A[j]$ compare themselves to neighboring keys $A[j+1]$ and exchange their values if necessary. In an even round i, the same occurs with even indexed keys. A refined version of BubbleSort based on this principle is shown below.

```
  OddEvenTSort(A[1..n])
1. for (i=1;i<=n;i++) {
2.  if (odd(i))
3.   for (j=1;j<=n-1;j+=2)
4.     if A[j] > A[j+1]  swap(A[j],A[j+1]);
5.  else  // for even(i)
6.   for (j=2;j<=n-1;j+=2)
7.     if A[j] > A[j+1]  swap(A[j],A[j+1]);
```

```
  BubbleSort(A[1..n]) [CLRS version]
1. for(i=1;   i<= n-1 ; i++)
2.   for(j=n; j>= i+1 ; j--)
3.     if A[j-1] > A[j] swap(A[j-1],A[j])
```

```
  BubbleSort(A[1..n])
1. for(i=n-1; i>= 1 ; i--)
2.   for(j=1; j<= i ; j++)
3.     if (A[j] > A[j+1]) swap(A[j],A[j+1]);
```

```
  ShuttleSort(A[1..n])[Alg. 175;C.ACM 6(6),1963]
1. for(i=1;   i<= n-1; i++)
2.   for(j=i; j>=1   ; j--)
3.     if A[j] > A[j+1]  swap(A[j],A[j+1])
```

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Every sorting algorithm can be turned into a stable sorting algorithm, if it is not, if we modify the input sequence. If the i-th input key is k_i, and "augment the key" into (k_i, i), even if $k_i = k_j$ we have that $(k_i, i) \neq (k_j, j)$, as k_i and k_j are stored in different positions and thus $i \neq j$. Sorting the sequence $\langle k_1, k_2, \ldots \rangle$, is equivalent to sorting the sequence $\langle (k_1, 1), (k_2, 2), \ldots \rangle$; in the latter sequence however, no two keys are equal (i.e. they are all distinct) so stabibity problems are not possible.

```
 Referring to the implementations of Subjects/Class/Textbook
       Sort in place: InsertionSort, SelectionSort, BubbleSort, HeapSort
Do not sort in place: MergeSort, QuickSort [MergeSort uses extra array for merging]
                      [Non-recursive QuickSort  must use Omega(lgn) space]
       Stable       : InsertionSort,SelectionSort,BubbleSort, MergeSort
     Not Stable     : QuickSort, HeapSort
```

1.28 Comparison-based Sorting. All of the above sorting algorithms are comparison-based. A function `compare` compares two keys (such as A[i] and key in InsertionSort) and a negative, 0, or positive value (eg -1,0,+1) is output to indicate $A[i]$ is less, equal or greater than x.

1.29 Distribution-based Sorting. There exist some sorting algorithms that do not use key comparisons. Instead they inspect the key values because they have prior knowledge of those key values. Such algorithms are known with the names of CountSort, BucketSort and RadixSort.

For a sorting algorithm A, and an input instance I of n keys,
Running Time (single instance): $T_A(n, I)$ is the running time of A for an input instance I of n keys, and
Running Time Bound (over all instances of same input size): $O(\max_I T_A(n, I))$ is the (bound on the) running time of algorithm A over all posible instances I of n keys: it is defined (or upper bounded) by its worst-case running time.
Examples. The best-case running time of InsertionSort is attained by a sorted sequence: Thus $T_{\text{InsSort}}(n, Sorted) = \Theta(n)$. Its worst-case running time is attained by a reverse sorted sequence: $T_{\text{InsSort}}(n, \text{ReverseSorted}) = \Theta(n^2)$.

The running time bound of InsertionSort is $O(\max_I T_{\text{InsSort}}(n, I)) = O(n^2)$. This is because $\max_I T_{\text{InsSort}}(n, I) = \Theta(n^2)$ as the maximum is the running time of a reverse sorted sequence. When we say that the running time of InsertionSort is $O(n^2)$ we mean that its running time can be $\Theta(n^2)$ for some inputs, $\Theta(n)$ for some other inputs, or anything in between for some of the other inputs. More on asymptotic notation in Subject 2.

Sets as well as operations on sets are of paramount importance in computer science. Although in a mathematical context a set is a static concept, in computer science a set becomes a dynamic entity whose size can change during the course of the execution of a program/algorithm. Such sets are called **dynamic** to distinguish them from **static** ones. For a dynamic set we need to introduce the operations that operate on these sets and describe algorithms that implement these operations. Such operations can be **modifying operations** or **probing operations**. Modifying are operations such as Insert, Delete, ExtractMax or DeleteMax, ExtractMin or DeleteMin, that modify a set. Probing operations are operations such as Search, Min or FindMin, Max or FindMax, Predecessor and Successor that do not modify the set but just probe it.

Issues related to the efficiency of these operations need to be dealt with as well. Foremost and related to such issues is the problem of the representation and storage of a set in the memory of a computer.

An abstract data type that is a **dictionary** supports operations like insertion of a word, deletion of a word and search of a word in the dictionary. It can be used to maintain a set of words (in the dictionary).

In the following discussion U is a universe of elements, $S \subseteq U$ denotes a set and $x \in U$ is an element of U. Each x has associated with it a field **key** with value k. Other information may also be associated with x. Without loss of generality we shall ignore this additional information as for the operations that we shall describe for a set S, this additional information does not affect the description of the operations.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Search(S,k)	Given a set S and a key value k, decide whether there exists an element x of S whose key is k. If there exists such an x return a pointer to x, otherwise return NULL.
Insert(S,x)	Given a set S, x is inserted into S.
Delete(S,x)	Given a set S, and a pointer to an element x of S, element x is deleted.
Min(S)	Given a set S find the element of x with the minimum key value and return the value (or the element with the min value).
Max(S)	Given a set S find the element of x with the maximum key value and return the value (or the element with the max value).
Successor(S,x)	Given x and S the element of S is returned with value the next larger to the value k of x or NULL.
Predecessor(S,x)	Given x and S the element of S is returned with value the next smaller to the value k of x or NULL.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Stacks and queues are dynamic sets where the element that is being deleted by a DELETE operation is prespecified. In a **stack** this element is the most recently inserted into the stack. In a **queue** it is the element least recently inserted. For this reason the two structures are also known as LIFO (last-in first-out) and FIFO.

For a stack, the Insert and Delete operations are more often called **Push** and **Pop**.

An implementation of a stack in terms of an array is immediate. The leftmost end of the array is the bottom of the stack, the stack grow to the right and the top of the stack is pointed by a pointer **top**. A stack with no elements is an empty stack. If an operation tries to reach an element past the left end an **underflow** has occurred. If it tries to reach an element past the right end an **overflow** is flagged.

For a queue, the Insert and Delete operations are more often called **Enqueue** and **Dequeue**. A queue has a **head** and a **tail**. A dequeue operation removes the element of the head, and an enqueue operation adds an element at the tail of the queue.

An implementation of a queue in terms of an array is easy if we use a floating pointer to denote the head and the tail of the queue. Initially head and tail point to the beginning of the queue. If head=tail+1, the queue is full. When they are equal the queue is empty. Head points to the first element and tail next to the last element (i.e. the first available position). When tail or head reaches n (size of array) the value is rolled over to 1, that is, we visualize the array as a circular one.

```
EmptyStack(S)            PUSH(S,x)             POP(S)
1. if top(S)==0           top(S)=top(S)+1;     if EmptyStack(S)
2.  return(TRUE);         S[top(S)] = x;         return(UNDERFLOW);
3. else                                        else
4.  return(FALSE);                               top(S)=top(S)-1;
5.                                               return(S(top(S)+1));

1. Enqueue(Q,x)                  Dequeue(Q)
2.  Q[tail(Q)]=x;                 x=Q[head(Q)];
3.  if (tail(Q) == length(Q)      if head(Q)==length(Q)
4.   tail(Q)=1;                     head(Q)=1;
5.  else                          else
6.   tail(Q)++;                     head(Q)++;
```

(c) Copyright A. Gerbessiotis.

An **array** is a random access data structure. We can access the i-th element of the array in unit time. Whether we acess the 1st, i-th or last element, the time is the same. **Random Access** means the same as the term Random Access Memory (RAM).

A **linked-list** is a data structure whose objects are arranged in a linear order. Random access to the i-th element of the list is not supported (as it was, in the case of an array): it takes i steps for access the i-th element of a linked list. A pointer structure is used however to represent a linked list.

A **doubly linked list** is a list whose objects are arranged in a linear and reverse linear order. Pointers prev and next point to the following element in the list in each of these two orders (forward and backward). A doubly linked list allows for constant time deletion of an element pointed by a pointer/reference.

In a singly linked list only one pointer is available (next). If the list is sorted according to the value of the keys of its elements, the linear order of the list is the linear order of its keys.

In a **circular** list next of last element (tail) is the first (head).

Important operations in a linked list: Search, Insert, Delete.

What is a **Sentinel?** We may avoid the use of null pointers by making such elements (pointing to null) to point to a sentinel element (a default-non-valid-list-element) that points to the head of the list.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

If we have say a doubly-linked list with two pointers next, prev, and key, we can represent such a list by three arrays one for each of next,prev, and key. A pointer value then becomes an index value of the array.

Sometimes it makes sense to represent a list in a single array. Then, an element of the list will occupy space m which is large enough to store prev, next, and key (for the first two an array index is stored for each prev,next). A pointer points to the first position of size m allocated to the pointed linked-list element.

As a linked-list is a dynamic data structure in both representations elements are added to and deleted from the list. If the arrays/array are/is of size n a naive approach of implementing linked-list operations on these structures may cause problems after the first n operations.

We need to some how record the freed positions (after a delete operation) and reallocate these positions whenever an Insert is issued. The **garbage-collector** is responsible for such allocation/deallocation operations.

To allow this we keep in the **free list** all available memory (ie it is a linked list). Initially the whole memory belongs to the free list. Whenever a portion of the array is deleted it is added to the top of the free list and whenever an element needs to be allocated it is allocated from the top (ie it is a stack). Every element is either in the list L or in the free store.

This is one reason why it is advised that a malloc(A), malloc(B) in the C programming language be followed by a free(B) and free(A). **Why is it so?**

The methods we used to represent lists extend to representing any homogeneous data structure including trees.

We can represent a binary tree by using three fields, **left and right** to point to the children of the node/element of the tree and p to point to its **parent**. An element with both left and right NULL is a leaf. An element with p NULL is the root pointed to by $root(T)$.

Suppose that a tree node has unlimited degree (number of children). Then left(x) points to the left child y_1 of x and right (y_1) points to its sibling y_2 (second child of x) and so one until the last one y_k points to nil (no more siblings).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.