INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

Read along with Handout 3 (Brief Discrete Math Review)

Chapters 1 and 4.1 of GT

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 2016; distribution outside this group of students is NOT allowed.
Introduction

What is a Data Type? An Abstract Data Type? Data Structures? Algorithm?

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

What is a Data Model (DM)? It is an abstraction that describes how data are represented and used.

Primitive (built-in) vs composite data type. A programming language such as C++ consists of primitive data-types such as `int`, `char`, `double` and also composite data types that can be built on top of them such as array, struct and class. The whole set of data type and the mechanisms aggregation of them can be formed define the data model of C++.

What is an 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 abstract data type consisting of a collection of words on which a set of operations are defined such as Insert, Delete, Search.

What is a data-structure? A data structure is a representation of the mathematical model underlying an ADT, or, it is a systematic way of organizing and accessing data.

Does it matter what data structures we use? For the Dictionary ADT 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 as expressed by its operations. What data structure we use, 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 one or the other among those data structures, one operation is more efficient than the other.

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

An algorithm is a well-defined sequence of computational steps that performs a task by converting an (or a set of) input value(s) into an (or a set of) output value(s).

A (computational) problem defines an input-output relationship. An algorithm describes a specific procedure for achieving this relationship, i.e. for each problem we may have more than one algorithms.
Introduction
Examples: Data Structures and Algorithms

Data-structures and ADTs

Dictionary. A dictionary is an ADT with few operations defined for it: an Insert that inserts a word into the dictionary, a Search that searches for a word, and possibly, a Delete that deletes a word from the dictionary. A dictionary with only the the second operation might also be called a static dictionary; a “full-dictionary” is a dynamic data structure. Static data structures support “probing operations” (such as Search) that do not modify the data structure. Dynamic data structures change through “modifying operations” such as Insert and Delete that add to or delete elements from them.

A data structure for implementing the Dictionary ADT. There many ways one can implement a dictionary. CS 505-wise one could use an array, or a sorted array, a linked-list or a binary-search tree to implement a dictionary ADT. For an array Search would be a slow operation realized through LinearSearch; a sorted array can take advantage of a more efficient search operation known as BinarySearch.

Sequence: Order of elements is important, eg \( \langle 1, 2, 3 \rangle \neq \langle 2, 1, 3 \rangle \).

Set: Order of elements is NOT important, eg \( \{1, 2, 3\} = \{2, 1, 3\} \).

Computational Problem. The problem of sorting keys: Sort a sequence of keys in non-decreasing order (or sorted order in short). A more elaborate input/output relationship for this computational or combinatorial problem is given below.

Sorting Problem

Input. A sequence of \( n \) keys \( \langle a_1, a_2, \ldots, a_n \rangle \). (Eg.: \( \langle 10 \ 2 \ 1 \ 5 \rangle \))

Output. A permutation/reordering \( \langle a_{j_1}, a_{j_2}, \ldots, a_{j_n} \rangle \) of the input so that \( a_{j_1} \leq a_{j_2} \leq \ldots \leq a_{j_n} \). (Eg.: \( \langle 1 \ 2 \ 5 \ 10 \rangle \))

Example.
An input sequence like the \( \langle 10 \ 2 \ 1 \ 5 \rangle \) is an instance of the sorting problem. An instance of the problem consists of all the inputs needed to compute a solution to the problem.
An algorithm is correct if for every input instance it halts with the correct output.
We describe an algorithm in this course with pseudocode that looks like a procedural (aka imperative) language (eg. Pascal, C, C++, Java) but we are not concerned with its syntax, variable declaration/definition etc.
Algorithm Design

Incremental and Divide and Conquer

There are various methods one can use to design algorithms for various problems. Before we present such methods let us provide some useful definitions.

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 function \( g \) that invokes \( f \). A quite well-known recursive function from discrete mathematics is the Fibonacci function \( F_n \). The Fibonacci function is defined as follows.

\[
F_n = F_{n-1} + F_{n-2} \text{ if } n > 1
\]

where

\[
F_0 = 0 \text{ and } F_1 = 1
\]

There are various methods for computing the general solution \( F_n \).

Algorithm Design Method 1: The iterative approach whose design principle is known as the incremental algorithm design method.

Algorithm Design Method 2: The recursive approach that is known as the divide-and-conquer algorithm design method.

We show both of them below in pseudocode that may look like C/C++/Java.

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

```c
// Divide-and-conquer
FIBO-2 (n)
0. if (n==0) return (0);//F[0]
1. if (n==1) return (1);//F[1]
2. return(FIBO-2(n-1) + FIBO-2(n-2)); //F[n]
```

(c) Copyright A. Gerbessiotis. CS610 : Spring 2016 . All rights reserved.
Algorithm Design
Methods: Incremental vs Divide-and-Conquer

In the **incremental approach** the Fibonacci sequence $F_0, F_1, \ldots$ is computed one value at a time by an iterative algorithm implemented by an iterative function.

In the **divide-and-conquer approach**, in order to find $F_n$ we use the recursive function/definition of $F_n$ and thus split the task of finding $F_n$ into two subtasks: one of finding $F_{n-1}$, and one of finding $F_{n-2}$. We solve these subtasks recursively using the same function (FIBO-2) that we use to solve/find $F_n$. After solving the subtasks we solve our original problem of computing $F_n$ by combining the solutions of the subtasks (i.e. by adding $F_{n-1}$ and $F_{n-2}$).

Note that in general, the recursive **divide-and-conquer** solution is easier to design and implement. It may be, 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).

**Exercise 1:**
(a) Give an upper bound for $F_n$ by showing (by using induction) that $F_n \leq 2^n$. Also give a lower-bound by showing that $F_n \geq 2^{n/2-1}$.
(b) Find the exact solution for $F_n$ using linear recurrence techniques.
(c) Find the number of operations (additions) performed in establishing $F_n$ in both algorithms (i.e. assume the cost of each addition is 1).
(d) What is the number of function (recursive) calls in the recursive solution?
(e) Can you use constant space to find $F_n$?
(f) Can you find $F_n$ is less than linear time?
If one thing is obvious about the recursive divide-and-conquer approach is the waste of computational resources. We could have used a table to store intermediate values by computing each value ONCE and ONLY ONCE and retrieving a stored value if it needs to be accessed later on. This gives rise to the following modification of the algorithm.

```
0. int FIBO-2 (n)
1.    table int T[0..n];
2.    T[0]=0; T[1]=1; T[ 2<=i <=n] = UNDEFINED;
3.    return(TableFIBO-2(n,T));

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

Reviewing Induction (and strong induction)

Soln Exercise 1-1(a): Show \( F_n \leq 2^n \) for all \( n = 0, 1, \ldots \). Define \( P_n \) a proposition on an integer variable that is true or false.

\[ P_n : \quad F_n \leq 2^n \]

We thus want to show that \( P_n \) true for all \( n = 0, 1, 2, \ldots \).

**Base Case** \( n = 0 \): Show that the proposition is true for the smallest value of the integer variable (in our case \( n \)). The smallest value of \( n \) is 0 i.e. we need to show that \( P_0 \) is true i.e. show that \( F_0 \leq 2^0 = 1 \) is equivalently true.

**Base case Proof:** It is easy. Since \( F_0 = 0 \), it follows immediately that \( F_0 \leq 1 \). We have just proved that \( P_0 \) is true.

**Inductive Step:** If \( P_0, P_1, \ldots, P_n \) are true, then \( P_{n+1} \) will also be true. Equivalently, if \( F_i \leq 2^i \) for all \( i = 0, \ldots, n \), then \( F_{n+1} \leq 2^{n+1} \).

This is what we call **strong induction**. In strong induction we assume that all of \( P_0, \ldots, P_n \) are true and show then that \( P_{n+1} \) is also true. In **weak/ordinary induction** we just assume that \( P_n \) is true and then go on showing that \( P_{n+1} \) is also true. The assumption that \( P_0, \ldots, P_n \) are true for strong or \( P_n \) for weak induction is called the **inductive hypothesis**. The induction step is the proof that \( P_{n+1} \) is true.

**Inductive Step Proof:** If \( P_0, \ldots, P_n \) are true (inductive hypothesis), then \( P_n \) is true and so is \( P_{n-1} \). Therefore \( F_n \leq 2^n \), and \( F_{n-1} \leq 2^{n-1} \). We will show that \( F_{n+1} \leq 2^{n+1} \), i.e. we will show that \( P_{n+1} \) is true and thus prove and complete the inductive step.

\[ F_{n+1} = F_n + F_{n-1} \leq 2^n + 2^{n-1} \leq 2^n + 2^n = 2 \cdot 2^n = 2^{n+1} . \]

We have thus proved that \( F_{n+1} \leq 2^{n+1} \) thus showing that \( P_{n+1} \) is true. ■

**What does this mean?**

F1. We know that \( P_0 \) is true.
F2. We know that if \( P_0, \ldots, P_i \) are true then so is \( P_{i+1} \).
F3. Set \( i = 0 \). By F1 \( P_0 \) is true. By F2, if \( P_0 \) is true, so is \( P_1 \). Thus \( P_1 \) is true.
F4. Set \( i = 1 \). By F3 \( P_0 \) and \( P_1 \) are true. By F2, if \( P_0, P_1 \) are true so is \( P_2 \). Thus \( P_2 \) is true.
F5. Set \( i = 2 \). Similarly, \( P_3 \) is true.
F6. Set \( i = n - 1 \). We similarly show that \( P_n \) is true.
This proves that \( P_n \) is true for all \( n = 0, 1, 2, \ldots \)
Algorithm Design
Remarks: Pseudocode vs Java/C/C++ code

Some comments on our algorithm descriptions.

1. We don’t use actual Java/C/C++ code; it may look like Java/C/C++ but we do not bother about class or variable definitions or declarations. We call such code pseudocode.

2. We number the lines.

3. We are liberal with the syntax of the program (i.e. return values).

4. 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 as \( A[0...n-1] \), or \( A[1..n] \). Why? It causes confusion to say the fifth element of an array is \( A[4] \) rather than the more natural and less confusing \( A[5] \).

5. We do not bother about error-checking (e.g. what if \( n = -1 \) in the Fibonacci functions?)

6. DO NOT FORGET: our purpose is to introduce concepts of 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.
Algorithm Analysis

Computation Model and Problem Size

1. 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 they are 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, etc), 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.

2. We decide on the resource whose performance we are going to measure. Resources that are measured to establish algorithm performance include space (memory), communication bandwidth, logic gates etc. The two most important resources are time (how long an algorithm runs to completion) and space (how much memory/space it uses). As space tends to become quite inexpensive, time becomes the resource whose usage needs to be minimized.

3. We then find the size of the problem which is expressed in terms of input size. For Fibonacci sequences, problem size $n$ is the order of the number we wish to find. For a sorting problem, problem size $n$ is the number of keys to be sorted. For the multiplication of two integer numbers $a$ and $b$ problem size is not the number of multiplicands (i.e. two which is constant) but the number of bits needed to represent $a$ and $b$. Problem size and input size in most cases denote the same thing. Sometimes the may be different. For matrix multiplication problem size may be $n$ the dimension of the two matrices, but input size is the number of elements of the input matrices i.e. $n^2$ per matrix.

4. We then measure the resource in an execution of the program as a function of problem size $n$. Therefore time $T(n)$ may express the running time of an algorithm for a problem of size $n$, and $S(n)$ may express the space requirements for a problem of input size $n$. Note that for the multiplication of two numbers, if we had used as problem size the number of multiplicands (two), then $T(\cdot)$ would have been $T(2)$. But for a function $f(n)$, $f(2)$ is the value of the function at point $n = 2$, i.e. $f(2)$ is a constant. This would imply that the time it takes to multiply two numbers $a$ and $b$ is constant no matter what the their sizes are (i.e. multiplication of 2 and 3 takes as much time as the multiplication of two 2000-bit numbers since in both cases we multiply two numbers and the problem size is the same: 2).
Algorithm Analysis
Computation Model and Problem Size (continued)

5. In order to perform step 4 we need to be careful with the types of operations we count: we normally count the most
dominant/primitive operation(s), i.e. that/those operation(s) we deem more expensive that would dominate the running time
of the algorithm. In Fibonacci we only counted additions, neither assignments nor other operations nor the number of if
statements. In sorting we may count comparisons of keys.

Remark 5.a We make such a choice (of what operations to count) on the expectation that the most costly operation(s)
dominate all others and if say such an operation count is say $T(n) = 2n - 1$, then we expect all other operation counts
(that were ignored in establishing $T(n)$) to be proportional to that count $T(n)$, say by giving a total operation count of
$Total(n) = 10n + 20$. This also means that the constant 2 in the linear term $2n$ of $T(n)$ is meaningless, as a more careful
operation count of an algorithm may yield say a constant 10 in the linear term $10n$ (of $10n + 20$). This also implies that all
constant terms multiplicative or additive may be meaningless and the only important conclusion is the order of growth of
running time in terms of problem size $n$, i.e. linear in this particular case.

Remark 5.b. If on the other hand we had a case where $T(n) = 2n - 1$, and $Total(n) = n^2$, this would mean that our counted
operations were poorly chosen as they contribute to running time a linear term but the total operation count is quadratic in
$n$.

Remark 6: Review $2^{\lg n} = n$, $2^{c \lg n} = (2^{\lg n})^c = n^c$. $2^{a-b} = (2^a)^b = (2^b)^a$.

Remark 7: Review. $a^{\lg n} = (2^{\lg a})^{\lg n} = 2^{\lg a \cdot \lg n} = (2^{\lg n})^{\lg a} = n^{\lg a}$, $x^a + b = x^a \cdot x^b$. 
Introduction
The sorting problem: A solution

We present a solution to the sorting problem by presenting a sorting algorithm that is known as insertion sort. Insertion sort is an incremental algorithm. It solves the sorting problem, i.e., generates the output sequence one key at a time. Insertion sort 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.

Important Definition-1. 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.

Important Definition-2. A sorting algorithm is stable if the relative order of same valued keys remains the same in the input and output sequences.

Example. Consider the problem of Google sorting triplets of the form \(\langle \text{Doc-ID}, \text{word-ID}, \text{index} \rangle\) where Doc-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.
\[\langle 0, \text{algorithm}, 10 \rangle \langle 0, \text{data}, 20 \rangle \langle 0, \text{algorithm}, 30 \rangle \langle 1, \text{data}, 4 \rangle \langle 2, \text{data}, 20 \rangle \langle 2, \text{data}, 40 \rangle\]

Stable sorting: Sort by word-ID (sorting by index is for free) and use prior ordering by index/doc-ID
\[\langle 0, \text{algorithm}, 10 \rangle \langle 0, \text{algorithm}, 30 \rangle \langle 0, \text{data}, 20 \rangle \langle 1, \text{data}, 4 \rangle \langle 2, \text{data}, 20 \rangle \langle 2, \text{data}, 40 \rangle\]

Not stable sorting: Sort by word-ID but previous orderings might get lost
\[\langle 0, \text{algorithm}, 30 \rangle \langle 0, \text{algorithm}, 10 \rangle \langle 2, \text{data}, 40 \rangle \langle 1, \text{data}, 4 \rangle \langle 0, \text{data}, 20 \rangle \langle 2, \text{data}, 20 \rangle\]
Insertion Sort

Pseudocode

Insertion-Sort (A[0..n-1], n)  
<table>
<thead>
<tr>
<th>Cost</th>
<th>TimesExecuted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>n-1</td>
</tr>
<tr>
<td>1</td>
<td>t1+t2+...+tn-1</td>
</tr>
<tr>
<td>1</td>
<td>(t1-1)+..+(tn-1 -1)</td>
</tr>
<tr>
<td>1</td>
<td>(t1-1)+..+(tn-1 -1)</td>
</tr>
<tr>
<td>1</td>
<td>n-1</td>
</tr>
</tbody>
</table>

Questions.
Q1. How many times is line 1 executed?  
Answer: n
Q2. How many times is line 4 executed?  
Answer: It depends on the input sequence. For key $A[j]$ let $t_j$ be that number!

Input:  
Ordered  Reverse-Sorted  Random  
<1 2 3 4 5>  <5 4 3 2 1>  <2 5 1 4 3>

4 comparisons  10 comparisons  9 comparisons  

(c) Copyright A. Gerbessiotis. CS610: Spring 2016. All rights reserved.
The running time of an algorithm depends on the **input** and the **size of the input**.

A. If the input instance is already sorted then insertion sort is quite fast (best-case for insertion sort). Best-case running time is rarely used to assess the performance of an algorithm as it is a very optimistic measure. With reference to our previous example, the next key of the input sequence is only compared to the rightmost key of the output sequence and the test of line 4 fails because $A[i]$ is less than or equal to $key$. Therefore the number of comparisons per key is 1 for the best-case.

B. If the input instance is in decreasing order (reverse sorted) then insertion sort requires the longest running time for any input instance of the same size (worst case). In most cases, worst-case running time provides the most useful information as it gives an upper bound of the running time of the algorithm. In our example the $A[j]$ is compared to all $j$ output keys thus requiring $j$ comparisons!

C. If all the permutations of the input keys are equally likely to appear in the input we examine the average case of a sorting algorithm; if we average the running time for each such input, for every input size $n$, we get the average case running time as a function of input size $n$. $A[j]$ is on the average larger than half of the $j$ output keys and smaller than half of them. Therefore the number of comparisons performed to insert $A[j]$ in the output sequence of the $j$ sorted keys is $j/2 + 1$.

Based on these observations an informal way to determine the running time of insertion sort for all three cases is thus:

**Best-case running time is** : $\sum_{j=1}^{n-1} 1 = n - 1$. Or we can just write using asymptotic notation that will be introduced shortly, $\Theta(n)$. (Think of $\Theta$ as the asymptotic equivalent of $=$.)

**Worst-case running time is** : $\sum_{j=1}^{n-1} j = n(n - 1)/2$. Or in asymptotic notation $\Theta(n^2)$.

**Average-case running time is** : $\sum_{j=1}^{n-1} (j/2 + 1) = n(n - 1)/4 + (n - 1)$. Or in asymptotic notation $\Theta(n^2)$. This introduces the first problem with asymptotic notation. where as average-case running time is half of worst-case running time, asymptotically both measures exhibit quadratic performance: asymptotic description is inaccurate.

**Hint.** To derive the expression within $\Theta$ or any other asymptotic symbol do the following: convert any multiplicative constant into 1, eliminate low-order terms and then what is left is what you need. Thus $n(n - 1)/4 + (n - 1)$ becomes after turning $1/4$ into 1 and then $n(n - 1) + (n - 1)$ becomes $n^2$ as $-n$, $n$ and $-1$ are low-order term relative to the quadratic term (let alone the fact that $-n$ and $n$ cancel each other out).

We can now formally prove the claims for the Best, Worst and Average case running time of InsertionSort.
Introduction

Running time of Insertion-Sort

The running time \( T(n) \) of Insertion-Sort for an input of size \( n \) is given by the following expression if we multiply columns \textit{Cost} and \textit{TimesExecuted} on page 12.

\[
T(n) = n + (n - 1) + (n - 1) + (n - 1) + \sum_{j=1}^{n-1} t_j + \sum_{j=1}^{n-1} (t_j - 1) + \sum_{j=1}^{n-1} (t_j - 1)
\]

To derive a closed form answer for \( T(n) \) is difficult. We instead distinguish cases.

**Best-case.** For the best-case, it is \( t_j = 1 \) for all \( j \), i.e. the input is sorted. Therefore,

\[
T(n) = 5n - 4
\]

i.e. running time is linear in \( n \), or \( \Theta(n) \) by using the Hint of page 13.

**Worst-case.** For the worst-case, it is \( t_j = j + 1 \), i.e. the input is reverse-sorted, and therefore,

\[
T(n) = 3n^2/2 + 7/2n - 4
\]

i.e. running time is quadratic in \( n \), or \( \Theta(n^2) \). In this case, we used the fact that \( \sum_{j=1}^{n-1} j + 1 = n(n + 1)/2 - 1 \) and \( \sum_{j=1}^{n-1} j = n(n-1)/2 \).

**Average-case.** On the average, key \( A[j] \) is smaller (or bigger) than half of the \( j \) keys preceding it i.e. \( t_j = j/2 + 1 \).

It is worth noting that the important conclusion is the "linearity" and "quadracity" of the running time for the three cases. Lower-order terms are not important and we end-up with \( n \) and \( n^2 \). To signify that the running time (i.e. \( n \) and \( n^2 \)) is the result of this process we write respectively \( T(n) = \Theta(n) \) for the former and \( T(n) = \Theta(n^2) \) for the latter cases.

An input instance for InsertionSort might have values for \( t_j \) that lead to a running time between the best and worst-case running time. To say that the running time of InsertionSort is \( \Theta(n) \) is wrong (we know that a reverse-sorted sequence slows it down considerably). It also also wrong to say that it is \( \Theta(n^2) \) (i.e. always quadratic) as we know a sequence (already sorted) that requires only \( \Theta(n) \) time. Other sequences might yield running times between the two extremes of linear and quadratic. To say so we write that

The running time of insertion sort is \( O(n^2) \). The big-Oh notation is the asymptotic equivalent of \( \leq \). It indicates that for some input instances running time is or may be quadratic, and for some less than quadratic (e.g. linear).

The running time of insertion sort is also \( \Omega(n) \). The big-Omega notation is the asymptotic equivalent of \( \geq \). It indicates that the running time is always at least linear. This is so because a sorting algorithm needs to examine the \( n \) input keys, an operation that will take at least \( n \) steps.
Introduction
Divide-and-Conquer on Sorting

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 were such a case. Such algorithms follow what we call the divide-and-conquer approach, where a problem of some size $n$ is broken into similar smaller subproblems of size $m < n$, the subproblems are solved recursively first and then their solutions are combined to give the solution of the original problem.

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. problem size is 1).
3. Combine the subproblem solutions to get the solution of the original problem.

We present a new sorting algorithm that is known as merge-sort and whose growth rate is $n \log n$, i.e. it is slower than that of insertion sort that makes the algorithm faster than 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 sorted).
3. Combine Phase. Merge the two sorted sequences of size $n/2$ to produce the sorted result.
Introduction

D-and-C example: Mergesort

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

One-key sequence sorted

Merge into two-key sequences

Two-key sequences sorted

Merge into four-key

Four-key sequences sorted

Merge into eight-key

Eight-key sequence sorted

MergeSort(A){
    Running Time
    1. MergeSort(A,0,n-1);//if A[0..n-1] T(n)
}

MergeSort(A,l,r)
    Running Time
    //Sort numbers in A[l..r]
    1. if (l < r)
    2. m= floor((l+r)/2);
    3. Merge-Sort(A,l,m);
    4. Merge-Sort(A,m+1,r);
    5. Merge(A,l,m,r);
    //Merge(A,l,m,r) merges sorted sequences A[l..m] and A[m+1..r] using auxiliary space B[l..r]
**MergeSort**  

**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).

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

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).

\[
A \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
T(n) \quad cn \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
cn \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
cn \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
\]

\[
T(n/2) \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
T(n/2) \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
cn/2 \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
cn/2 \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
\]

\[
T(n/4) \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \\
T(n/4) \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
cn/4 \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
cn/4 \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
\]

\[
T(n) = cn \ (\lg n + 1) = cn\lg n + cn
\]
**Sorting**

**Other Sorting Algorithms**

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 $S_n = S$ be the sequence of input keys. In the $i$-th iteration of selection sort the maximum $m$ of $S$ is found and is placed in the output sequence in position $n - i$ and then $m$ is removed from $S$.

```
SelectionSort(A[0..n-1])
1. for (i=n; i > 0; i--) {
2.  index = MAX(A[0..i-1]); //find index of maximum key in A[0..i-1]
3.  exchange(A[index],A[i-1]);
}
```

BubbleSort is also known as odd-even transposition sort and can also be used for parallel sorting. The principle of the algorithm is as follows. It 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 bubble-sort based on this principle is shown below.

```
BubbleSort(A[0..n-1])
1. for(i=n; i > 1; i--){
2.  for(j=0; j < i-1; j++){
3.    if (a[j] > a[j+1]) exchange(A[j],A[j+1]);
}
}
```
**Introduction**

**Questions and Answers**

**Insertion-sort** sorts in-place. **Merge-sort** as presented in class DOES NOT sort in-place.

An algorithm is **stable** if same valued keys maintain their relative order in the input and output sequences.

Every sorting algorithm can be turned into a stable sorting algorithm, 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 stability problems are not possible.

Sort in place: Insertion, Selection, Bubble
QuickSort HeapSort (to be presented)

Do not sort in place: Mergesort (traditional algorithm)

<table>
<thead>
<tr>
<th>Stable</th>
<th>Textbook/Notes implementations of Insertion, Selection, Bubble. MergeSort is stable if Merge is done in a stable manner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Stable</td>
<td>QuickSort HeapSort (to be presented)</td>
</tr>
</tbody>
</table>

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 possible instances $I$ of $n$ keys: it is defined (or upper bounded) by its worst-case running time.

**Examples.** The best-case running time of Insertion-Sort is attained by a sorted sequence: Thus $T_{\text{InsSort}}(n, \text{Sorted}) = \Theta(n)$.
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 Insertion-Sort 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 3.