Objectives

To understand basic techniques for analyzing the efficiency of algorithms
To know what searching is and understand the algorithms for linear and binary search.
To understand the basic principles of recursive definition and functions and be able to write simple recursive functions
To understand sorting in depth and know the algorithms for selection sort and merge sort
To appreciate how the analysis of algorithm can demonstrate that some problem are intractable and others are unsolvable.

1 Mathematical Preliminaries

When analyzing an algorithm, the amount of resources required is usually expressed as a function of the input size. A nontrivial algorithm typically consists of repeating a set of instructions either iteratively, e.g. by executing a **for** or **while loop**, or recursively by invoking the same algorithm again and again, each time reducing the input size until it becomes small enough, in which case the algorithm solves the input instance using a straightforward method. This implies that the amount of resources used by an algorithm can be expressed in the form of summation or recursive formula. This mandates the need for the basic mathematical tools that are necessary to deal with these summations and recursive formulas in the process of analyzing an algorithm. In this section we review some of the mathematical preliminaries and discuss briefly some of these mathematical tools that are frequently employed in the analysis of algorithms.

1.1 Logarithms

Let b be a positive real number greater than 1, x a real number, and suppose that for some positive real number a we have $a = b^x$. Then, x is called the logarithm of a to the base b, and we write this as

$$x = \log_b a$$

Here b is referred to as the base of the logarithm. For any real numbers x and y greater than 0, we have

$$\log_b xy = \log_b x + \log_b y$$

The log of a product equals the sum of the logs. similarly

$$\log_b x/y = \log_b x - \log_b y$$

The log of a quotient equals the difference of the logs and

$$\log_b(x^n) = n \log_b x,$$

if $x > 0$.
When $b = 2$, we will write $\log x$ instead of $\log_2 x$. To convert from one base to another, we use the chain rule.

$$\log_a x = \log_b \, x \, \log_a b$$

or

$$\log_b x = \frac{\log_a x}{\log_a b}$$

Unless otherwise stated, all logarithms in this manuscript are to the base 2. The natural logarithm of x will be denoted by $\ln x$.

1.2 Floor and Ceiling Functions

Let x be a real number. The floor of x, denoted by $\lfloor x \rfloor$, is defined as the greatest integer less than or equal to x.
More formally, the floor function F is defined by the rule
For each real number x,

$$F(x) = \lfloor x \rfloor = the\, unique\, integer\, n\, such\, that\, n \leq x < n + 1$$

. The ceiling of x, denoted by $\lceil x \rceil$, is defined as the least integer greater than or equal to x.
For example,

$$\lfloor \sqrt{2} \rfloor = 1$$
$$\lceil \sqrt{2} \rceil = 2$$
$$\lfloor -2.5 \rfloor = -3$$
$$\lceil -2.5 \rceil = -2.$$

1.3 Counting and Probability

Definition 1. *A sample space is the set of all possible outcomes of a random process or experiment. An event is a subset of a sample space*

In case an experiment has finitely many outcomes and all outcomes are equally likely to occur, the probability of an event (set of outcomes) is just the ratio of the number of outcomes in the event to the total number of outcomes.

Equally Likely Probability Formula

If S is a finite sample space in which all outcomes are equally likely and E is an event in S, then the probability of E, denoted $P(E)$, is

$$P(E) = \frac{the\, number\, of\, outcomes\, in\, E}{the\, total\, number\, of\, outcomes\, in\, S}.$$

. **Notation**

For any finite set A, $N(A)$ denotes the number of elements in A. With this notation, the equally likely probability formula becomes

$$P(E) = \frac{N(E}{N(S)}.$$

Summation

Let $S = a_1, a_2, \ldots a_n$ be any finite sequence of numbers. The sum $a_1 + a_2 + \ldots + a_n$ can be expressed compactly using the notation

$$\sum_{j=1}^{n} a_j$$

Counting the Elements of a List

Theorem 1. *The Number of Elements in a List If m and n are integers and $m \leq n$, then there are $n - m + 1$ integers from m to n inclusive.*

Counting Elements of a One-Dimensional Array Analysis of many computer algorithms requires skill at counting the elements of a one-dimensional array.

Let $A[1],]\ldots, n]$ be a one-dimensional array, where n is a positive integer. The array has the same number of elements as the list of integers from 1 through n. So by Theorem 1 it has n, or $n - 1 + 1$, elements.

Theorem 2. *Sum of the First n Integers*
For all integers $n \geq 1$,

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

Writing

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

expresses the sum $1 + 2 + 3 + \ldots + n$ *in closed form.*
For each positive integer n, the quantity **factorial** denoted $n!$, is defined to be the product of all the integers from 1 to n.

$$n! = n(n1) \ldots 321.$$

Zero factorial, denoted 0!, is defined to be 1 :

$$0! = 1$$

.

2 About Basic Algorithm Analysis

This course provides an introduction to mathematical modeling of computational problems. It covers the common algorithms, algorithmic paradigms, and data structures used to solve these problems. The course emphasizes the relationship between algorithms and programming, and introduces basic performance measures and analysis techniques for these problems.
We are going to cover the difference between the Big-Oh, the Omega notation and the Theta notation. We are also going to look as some simple looping algorithms to help us determine what the Big-Oh notation is of them. So let's start.

2.1 Algorithm Analysis

Why do we want to analyse an algorithm?
Let's say you have task of being in a city A, and you want to go to a another city B. Obviously, there are a lot of transportation options available to you: walk, bike, take a car, a train a city bus or even fly. some of the make sense

over the others. so for example, if you just need to cross the street to go city B, you can just walk over. That makes a lot more sense than flying over. But if city B was across the country, it probably make more sense to buy a flight ticket and flying over instead of riding your bike.
In computing, algorithms are the same thing. So let's take sorting for example. If you want to sort a number of elements from small to the largest, you have a lot of optional tools available to you such as Insertion sort, Selection sort, Quick sort, Bubble sort, buttom-up sort etc...
Our ultimate gaol in algorithm design is to complete the task we want to solve efficiently. By efficiently, we mean that it should be quick in terms of time and less space requirement because memory is limited in computer systems. We can have the space filled up quickly. Hence, efficiency is measured in terms of time and space.
By analyzing them, we can compare algorithms, and depend on the task we can pick the best one.
So what is running time analysis?
Well we want to determine how the running time increases related to the size of the input. So as the input gets bigger, we want to see exactly what that does to the running time.
Input is usually n values. So we can make up assumptions regarding n! In fact we can't assume that n is going to be small! It is probably the best when we always assume that n is going to be adequately large.
Running Time Analysis: without making assumptions for n make for a timeless concept that translate from old computing system for modern machines. So let's look at some definitions.

Definition 2 (Rate of Growth). *The rate at which the graph of a running time increases as a function of the input size*

Definition 3 (Lower Order Terms). *When given an approximate rate of growth of a function, we tend to drop the lower order terms as they are less significant to the higher order terms. For example, if given the following function*

$$f(n) = n^3 + n,$$

the lower order term is n.
Here, we observe that the larger the value of n the lesser the significance of the contribution of the lower order terms n.

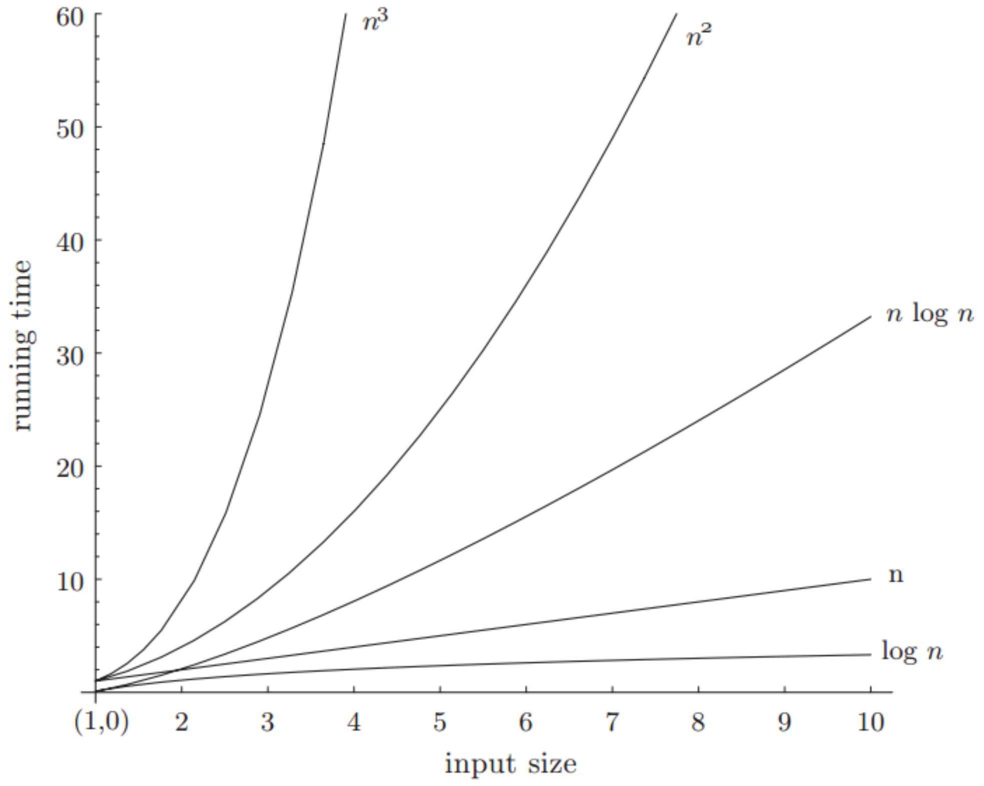

Figure 1: Growth of some typical functions that represent running times

Definition 4 (**Asymptotic Notation**). *Once we dispose of lower order term(s) and preceding constants from a function that represents the running time of an algorithm, we say that we are evaluating the asymptotic running time or using the more technical term "time complexity"of an algorithm. Let's take a look at the given example:*

$$f(n) = n^3 + n$$

n^3 is the higher order term and n is lower order term. When doing the asymptotic notation (e. g. using the big oh!) for simplicity sake we drop the lower order term n. Therefore, the function has $\mathcal{O}(n^3)$ for example.

Figure 1(a) on page 6 shows some functions that are widely used to express the running time of algorithms. They are called, respectively, *logarithmic, linear, quadratic and cubic.* Higher order, exponential and hyper-exponential functions are not shown in the figure. They are functions that can grow faster than the ones shown in the figure, even for a small size of n.

Definition 5 (**elementary operations**)**.** *We denote by an "elementary operation" any computational step whose cost is always upperbounded by a constant amount of time regardless of the input data or the algorithm used.*

Let us take, for instance, the operation of adding two integers. For the running time of this operation to be constant, we stipulate that the size of its operands be fixed no matter what algorithm is used. Furthermore, as we are now dealing with the asymptotic running time, we can freely choose any positive integer k to be the "word length" of our "model of computation". Incidentally, this is but one instance in which the beauty of asymptotic notation shows off; the word length can be any fixed positive integer. If we want to add arbitrarily large numbers, an algorithm whose running time is proportional to its input size can easily be written in terms of the elementary operation of addition. Likewise, we can choose from a large pool of operations and apply the fixed-size condition to obtain as many number of elementary operations as we wish. The following operations on fixed-size operands are examples of elementary operation.

- Arithmetic operations: addition, subtraction, multiplication and division.
- Comparisons and logical operations.
- Assignments, including assignments of pointers when, say, traversing a list or a tree.

2.2 $\mathcal{O}-$, Ω-, and Θ-Notations

In the followings, we present special mathematical notations widely used to formalise the notions of the order of growth and asymptotic running time of algorithms. These notations provide approximations that make it convenient to evaluate the large-scale differences in algorithm efficiency, while ignoring

differences of a constant factor and differences that occur only for small sets of input data.
The idea of the notations, $\mathcal{O}-$notation (read "big-Oh notation"), $\Omega-$notation (read "big-Omega notation") and $\Theta-$notation (read "big-Theta notation") is this. Suppose f and g are real-valued functions of real variables n.

1. If, for sufficiently large values of n, the values of $|f|$ are less than those of a multiple of $|g|$, then f is of order at most g, or $f(n)$ is $\mathcal{O}(g(n))$.

2. If, for sufficiently large values of n, the values of $|f|$ are greater than those of a multiple of $|g|$, then f is of order at least g, or $f(n)$ is $\Omega(g(n))$.

3. If, for sufficiently large values of n, the values of $|f|$ are bounded both above and below by those of multiples of $|g|$, then f is of order g, or $f(n)$ is $\Theta(g(n))$.

Definition 6 (**Big Oh! not**ation)**.** *Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to a set of nonnegative real numbers. $f(n)$ is said to be $\mathcal{O}(g(n))$ if there exists a natural number n_0 and a constant $c > 0$ such that $c|g(n)| \geq |f(n)|$ for all real numbers $n \geq n_0$.*

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \textit{ implies } f(n) = \mathcal{O}(g(n)).$$

Informally, this definition says that $f(n)$ grows no faster than the product of some constant times $g(n)$.
The $\mathcal{O}$-notation is sometimes used in equations as a simplification tool. For example, instead of writing

$$f(n) = 10n^3 + 14n^2 - 4n + 26,$$

we may write

$$f(n) = 10n^3 + \mathcal{O}(n^2).$$

This is helpful if we are not interested in the details of the lower order terms of the equation.

Definition 7 (**Big Omega not**ation)**.** *Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to a set of nonnegative real numbers. $f(n)$*

is said to be $\Omega(g(n))$ *if there exists a natural number* n_0 *and a constant* $c > 0$ *such that* $c|g(n)| \leq |f(n)|$ *for all real numbers* $n \geq n_0$*.*

Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ *exists, then*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \textit{ implies } f(n) = \Omega(g(n)).$$

Informally, this definition says that f *grows at least as fast as the product of some constant and* g*. It is obvious from the definition that*

$$f(n) \textit{ is } \Omega(g(n)) \textit{ if and only if } g(n) \textit{ is } \mathcal{O}(f(n)).$$

Definition 8 (**Big Theta not**ation)**.** *Let* $f(n)$ *and* $g(n)$ *be two functions from the set of natural numbers to a set of nonnegative real numbers.* $f(n)$ *is said to be* $\Theta(g(n))$ *if there exist two positive constants* c_1 *and* c_2 *and a natural number* n_0 *such that* $c_1|g(n)| \leq |f(n)| \leq c_1|g(n)|$ *for all real numbers* $n \geq n_0$*.*
Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ *exists, then*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \textit{ implies } f(n) = \Theta(g(n)),$$

where c *is a constant strictly greater than* 0*.*

An important result of the above definition is that

$$f(n) = \Theta(g(n)) \textit{ if and only if } f(n) = \Omega(g(n)) \textit{ and } f(n) = \mathcal{O}(g(n)).$$

Unlike the previous two notations, the $f(n) = \Theta(g(n))$-notation gives an exact picture of the rate of growth of the running time of an algorithm.
It may be helpful to think of $\mathcal{O}$ as similar to $\leq$, Ω as similar to $\geq$ and Θ as similar to $=$.

Type of Analysis

We can analyse or evaluate an algorithm in one of the three ways:

Table 1: Type of Analysis

WORST CASE	**BEST CASE**	**AVERAGE CASE**
Longest Time	Least Time	Average time
Big-Oh $\mathcal{O}(n)$	Omega $\Omega(n)$	Theta $\Theta(n)$

Most of the time we do use the Big-Oh notation because it is most practically used for one of the three. Best Case we can't always rely on because it is not always the best case, and Average Case is less useful than the worst case. So what we will always think about is the Worst Case about is to plan for the worst case scenario.

2.3 Determining the Big-Oh

The Big-Oh notation hope for the best, plan for the worst.

Let's look at a given function here:

$$f(n) = n^4 + 2n^2 + 100n + 500$$

Let's determine what the Big-Oh notation.

Let's define the highest order term as $g(n)$.

So what is $g(n)$ in the above equation?

$$g(n) = n^4$$

So the function

$$f(n) = \mathcal{O}(g(n))$$

is

$$f(n) = \mathcal{O}(n^4)$$

Is it the best to always use Big-Oh notation? Most of the time, Yes! However, if $f(n) = \mathcal{O}(n)$ and $f(n) = \Omega(n))$ then we tend to use $\Theta(n)$. Is just a formality and widely accepted practice...

So let us look at some examples of algorithms and look at what their Big-Oh notation is.

Example 1
A simple loop with a constant time operation $m = m + 2$ inside it.

```
1. for $i = 0$ to $n$
2.         $m = m + 2$
3. end for
```

If you look at the simple *for Loop* above, inside it is a constant time operation of simple addition we denote it by C. On the outside, the loop happens n times. So the function is a constant times n which is the amount of time that ever happens. Given as:

$$f(n) = C \times n$$

The leading constant is typically ignored! So our function has Big-Oh of n. Which is written as follows:

$$f(n) = \mathcal{O}(n)$$

Let us look at another slightly different example.
Example 2
A simple loop with a constant time operation $m = m + 2$ inside it.

```
1.for $i = 0$ to $n/2$
2.         $m = m + 2$
3.end for.
```

We observed that the constant operation is also the same. However, the loop happens $n/2$ times. Therefore

$$f(n) = C \times 1/2 \times n$$

$C \times 1/2$ are constants and n is the loop. Remember! the leading constants are usually ignored! Thus, our function running time is Big-Oh of n, denoted as

$$f(n) = \mathcal{O}(n)$$

.
Even though the loop only iterates half as much as the last example, they

are BOTH $\mathcal{O}(n)$!

Example 3 Nested loops

```
1. for $i = 1$ to $n$
2.     for $j = 1$ to $n$
3.         $k = k + 1$
4.     end for
5. end for
```

In the Listing above, we have constant operation inside the inner loop on line 3 that happens n times because the inner loop goes to n. That whole package inside the outer loop has inside loop that happens n times as well. So our function is:

$$f(n) = C \times n \times n$$

We can simplify this to have

$$f(n) = C \times n^2$$

so

$$f(n) = n^2$$

Example 4 (Consecutive Statement)

```
// consecutive statements (1)and (2) with a constant time operations
1. for j = 1 to n; j++)
2.     m = m + 1:
3. end for
4      for $i = 1$ to $n$
5          for j = 1 to n
6.             k = k + 1
7.         end for
8.     end for
```

Let' label the top (lines 1 - 3) as part (1) and the lower part lines 5 - 9 as (2). We see that the top one is a single loop operation and the bottom one is a nested loops. So writing our function we have

$$f(n) = C \times n + C \times n \times n$$

The $C \times n$ is the number (1) and the $C \times n \times n$ is the number (2). Therefore

$$f(n) = C \times n + C \times n^2$$

We know that the $C \times n^2$ is the highest order in this case, and that will take precedence. Thus,

$$f(n) = \mathcal{O}(n^2)$$

Furthermore, let's look at another example dealing with the *if-then-else* statement.

Example 5- if-then-else Statement]

```
// if-then-else statement
1. if(x+1¡5)
2.      return -1
3 else
4. for i = 1 to n
5.      for j = 1 to n
6.           k = k + 1
7.      end for
8. end for
9. return k
```

Let's build our function as we go through. So looking at the *if statement*, we have a constant operation ($x + 1 < 5$). We denote that as C_0. If this condition is met, then we have another constant operation (return -1. we denote it by C_1. So at this point

$$f(n) = C_0 + C_1$$

However with the *else statement* outer loop, the loops goes up to n and the inner loop goes to n also with a constant operation we denote that by C_2. Thus,

$$f(n) = C_0 + C_1 + n \times n \times C_2$$

. But when we group it

$$f(n) = C_0 + C_1 + C_2 \times n^2$$

$n^2$is the highest order here. Therefore

$$f(n) = \mathcal{O}(n^2)$$

Example 6

Consider Algorithm count, which consists of two nested loops and a variable count which counts the number of iterations performed by the algorithm on input n, which is a positive integer.

Algorithm count

Input: A positive integer n.

Output: count = number of times Step 5 is executed.

```
1. count$\leftarrow$ 0
2. for $i = 1$ to $n$
3.         $m \leftarrow \lfloor n/i \rfloor$
4.         for $j \leftarrow 1$ to $m$
5.                 $count \leftarrow count + 1$
6.         end for
7. end for
8. return count
```

The inner for loop is executed repeatedly for the following values of n:

$$n, \lfloor n/2 \rfloor, \lfloor n/3 \rfloor, \ldots, \lfloor n/n \rfloor$$

. Thus, the total number of times Step 5 is executed is

$$\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor.$$

Since

$$\sum_{i=1}^{n} (\frac{n}{i} - 1) \le \sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \le \sum_{i=1}^{n} \frac{n}{i},$$

we conclude that Step 5 is executed $\Theta(n \log n)$ times. (See Example 11). As the running time is proportional to count, we conclude that it is $\Theta(n \log n)$.

In general, let $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$. Then, $f(n) = \Theta(n^k)$. Recall that this implies that $f(n) = \mathcal{O}(n^k)$ and $f(n) = \Omega(n^k)$.

Example 7 Since

$$\lim_{n\to\infty} \frac{\log n^2}{n} = \lim_{n\to\infty} \frac{2 \log n}{n ln 2} = \frac{2}{ln 2} \lim_{n\to\infty} \frac{1}{n} = 0$$

(differentiate both numerator and denominator), we see that $f(n)$ is $\mathcal{O}(n)$, but not $\Omega(n)$. It follows that $f(n)$ is not $\Theta(n)$.

Example 8 Since
$\log n^2 = 2\log n$, we immediately see that $logn^2 = Theta(\log n)$. In general, for any fixed constant k, $\log n^k = \Theta(logn)$.
Example 9
Any constant function is $\mathcal{O}(1)$, $\Omega(1)$ and $\Theta(1)$.
Example 10
Consider the series $\sum_{j=1}^{n} \log j$. Clearly

$$\sum_{j=1}^{n} \log l \leq \sum_{j=1}^{n} \log n$$

That is

$$\sum_{j=1}^{n} \log j = \mathcal{O}(n \log n).$$

Also

$$\sum_{j=1}^{n} \log j \geq \sum_{j=1}^{\lfloor n/2 \rfloor} \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log(\frac{n}{2}) = \lfloor n/2 \rfloor \log n - \lfloor n/2 \rfloor$$

Thus,

$$\sum_{j=1}^{n} \log j = \Omega(n \ \log \ n)$$

It follows that

$$\sum_{j=1}^{n} \log j = \Theta(n\, log\, n)$$

Example 11
We want to find an exact bound for the function $f(n) = \log n!$. First, note that $\log n! = \sum_{j=1}^{n} \log j$. We have shown in Example 10 that $\sum_{j=1}^{n} \log j = \Theta(n \ \log \ n)$. It follows that $\log n! = \Theta(n \ \log \ n)$.
Example 12
It is easy to see that

$$\sum_{j=1}^{n} \frac{n}{j} = \sum_{j=1}^{n} \frac{n}{1} = \mathcal{O}(n^2).$$

In what follows, we list closed form formulas for some of the summations that occur quite often when analyzing algorithms. The proofs of these formulas

are left for the student as exercises.
The arithmetic series:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \Theta(n^2) \tag{1}$$

The sum of squares:

$$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3) \tag{2}$$

The geometric series

$$\sum_{j=1}^{n} c^j = \frac{c^{n+1}-1}{c-1} = \Theta(c^n),\ c \neq 1 \tag{3}$$

If $c = 2$, we have

$$\sum_{j=1}^{n} 2^j = 2^{n+1} - 1 = \Theta(2^n) \tag{4}$$

If $c = 1/2$, we have

$$\sum_{j=1}^{n} \frac{1}{2^j} = 2 - \frac{1}{2^n} < 2 = \Theta(1) \tag{5}$$

2.3.1 Complexity Classes and the o-notation pronounced as "little oh!

Let R be the relation on the set of complexity functions defined by fRg if and only if $f(n) = (g(n))$. It is easy to see that R is reflexive, symmetric and transitive, i.e., an equivalence relation The equivalence classes induced by this relation are called *complexity classes*. The complexity class to which a complexity function $g(n)$ belongs includes all functions $f(n)$ of order $\Theta(g(n))$. For example, all polynomials of degree 2 belong to the same complexity class n^2. To show that two functions belong to different classes, it is useful to use the o-notation(read "little oh") defined as follows.

Definition 9 (**little oh**). *Let $f(n)$ and $g(n)$ be two functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$ if for every constant $c > 0$ there exists a positive integer n_0 such that*

$f(n) < cg(n)$ for all $n \geq n_0$.
Consequently, if $\lim_{n\to\infty} f(n)/g(n)$ exists, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \, \textit{implies} \;\; f(n) = o(g(n)),$$

Informally, this definition says that $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity. It follows from the definition that

$$f(n) = o(g(n)) \Leftrightarrow f(n) = O(g(n)), \, \textit{but} \, g(n) \neq \mathcal{O}(f(n)).$$

For example, $n \log n$ is $o(n^2)$ is equivalent to saying that $n \log n$ is $\mathcal{O}(n^2)$ but n^2 is not $\mathcal{O}(n \log n)$.

3 Determining the Best, Worst and Average Cases for Algorithm Linear Search

Let $A[1...n]$ be a sequence of n elements. Consider the problem of determining whether a given element x is in A. This problem can be rephrased as follows. Find an index j, $1 \leq j \leq n$, such that $x = A[j]$ if x is in A, and $j = 0$ otherwise. A straightforward approach is to scan the entries in A and compare each entry with x. If after j comparisons,$1 \leq j \leq n$, the search is successful, i.e., $x = A[j]$, j is returned; otherwise a value of 0 is returned indicating an unsuccessful search. This method is referred to as sequential search. It is also called linear search, as the maximum number of element comparisons grows linearly with the size of the sequence. The algorithm is shown as Algorithm LINEARSEARRCH.

```
    Algorithm LINEARSEARRCH
Input: An array $A[1..n]$ of $n$ elements and an element $x$.
Output: $j$ if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.
1. $j \longleftarrow 1$
2. while ($j < n$) and ($x \neq A[j]$)
3.        $j \longleftarrow j + 1$
4. end while
5. if $x = A[j]$ then return $j$ else return 0
```

Obviously, the minimum number of comparisons is 1, and it is achievable when the element being searched for, x, is in the first position of the array, and that gives us the least running time (Best case). Therefore

$$C_{Best}(n) = \Omega(n) = \Omega(1)$$

To find the maximum number of comparisons, let us first consider applying the linear search on the array

$$A = [1, 4, 33, 7, 8, 17, 9, 10, 20, 12, 15, 22, 23, 27, 32, 18, 35].$$

If we search for 4, we need two comparisons,whereas searching for 8 costs five comparisons. Now, in the case of unsuccessful search, it is easy to see that searching for elements not in the array takes n comparisons because we need to search through all the element in the array. It is not difficult to see that, in general, the algorithm always performs the maximum number of comparisons whenever $x = A[n]$ or when x does not appear in the array at all. Thus in the worst case, the linear search algorithm is $\mathcal{O}(n)$. Hence,

$$C_{Worst}(n) = \mathcal{O}(n)$$

To find the average number of comparisons we consider a situation where the element x to searched exist in the array and that it is equally likely to occur in any position in the array. Accordingly, the number of comparison can be any of the numbers $1, 2, 3, ..., n$, and each occur with the probability $P[E] = 1/n$. Then

$$C_{Avg}(n) = [1 \times \frac{1}{n} + 2 \times \frac{1}{n} + ... + n \times \frac{1}{n}].$$

$$\begin{aligned} C_{Avg}(n) &= \frac{1}{n}[1 + 2 + 3 + ... + n] \\ &= \frac{1}{n}[\frac{n(n+1)}{2}] \\ &= [\frac{(n+1)}{2}] \end{aligned}$$

This agrees with our intuitive feeling that the average number of comparisons required to find an item is approximately equal to half the number of elements in the array.
Asymptotically, therefore

$$C_{Avg}(n) = \Theta(n)$$

4 Binary Search

Using the linear search approach for searching problem we see that intuitively, scanning all entries of $A[1 \dots n]$ is inevitable if no more information about the ordering of the elements in A is given. If we are also given that the elements in A are sorted, say in nondecreasing order, then there is a much more efficient algorithm. The following example illustrates this efficient search method.

Consider searching the array

$$A[1 \dots 14] = [1, 4, 5, 7, 8, 9, 10, 12, 15, 22, 23, 27, 32, 35].$$

In this instance, we want to search for element $x = 22$. First, we compare x with the middle element $A[\lfloor (1 + 14)/2 \rfloor] = A[7] = 10$. Since $22 > A[7]$, and since it is known that $A[i] \leq A[i + 1]$, $1 \leq i < 14$, x cannot be in $A[1 \dots 7]$, and therefore this portion of the array can be discarded. So, we are left with the sub-array

$$A[8 \dots 14] = [12, 15, 22, 23, 27, 32, 35].$$

Next, we compare x with the middle of the remaining elements $A[\lfloor (8 + 14)/2 \rfloor] = A[11] = 23$. Since $22 < A[11]$, and since $A[i] \leq A[i + 1]$, $11 \leq i < 14$, x cannot be in $A[11 \dots 14]$, and therefore this portion of the array can also be discarded. Thus, the remaining portion of the array to be searched is now reduced to

$$A[8 \dots 10] = [12, 15, 22].$$

Repeating this procedure, we discard $A[8 \dots 9]$, which leaves only one entry in the array to be searched, that is $A[10] = 22$. Finally, we find that $x = A[10]$, and the search is successfully completed.

In general, let $A[low \dots high]$ be a nonempty array of elements sorted in nondecreasing order. Let $A[mid]$ be the middle element, and suppose that $x > A[mid]$. We observe that if x is in A, then it must be one of the elements $A[mid + 1], A[mid + 2], \dots, A[high]$. It follows that we only need to search for x in $A[mid + 1 \dots high]$. In other words, the entries in $A[low \dots mid]$ are discarded in subsequent comparisons since, by assumption, A is sorted in nondecreasing order, which implies that x cannot be in this half of the array. Similarly, if $x < A[mid]$, then we only need to search for x in $A[low \dots mid - 1]$. This results in an efficient strategy which, because of its repetitive halving, is referred to as binary search. Algorithm BINARYSEARCH gives a more formal description of this method.

Algorithm BINARYSEARCH
Input: An array $A[1\dots n]$ of n elements sorted in nondecreasing order and an element x.
Output: j if $x = A[j]$; $1 \leq j \leq n$; and 0 otherwise.

```
1. $low \leftarrow 1$; $high \leftarrow n$; $j \leftarrow 0$
2. while ($low \leq high$) and ($j = 0$)
3.         $mid = \lfloor (low + high)/2 \rfloor$
4.         if $x = A[mid]$ then $j \leftarrow mid$
5.       else if $x < A[mid]$ then $high \leftarrow mid - 1$
6.         else $low \leftarrow mid + 1$
7. end while
8. return $j$
```

4.1 Analysis of the binary search algorithm

Henceforth, we will assume that each three-way comparison (if-then-else) counts as one comparison. Obviously, the minimum number of comparisons is 1, and it is achievable when the element being searched for, x, is in the middle position of the array.To find the maximum number of comparisons, let us First consider applying binary search on the array $[2, 3, 5, 8]$. If we search for 2 or 5, we need two comparisons, whereas searching for 8 costs three comparisons. Now, in the case of unsuccessful search, it is easy to see that searching for elements such as 1, 4, 7 or 9 takes 2, 2, 3 and 3 comparisons, respectively. It is not hard to see that, in general, the algorithm always performs the maximum number of comparisons whenever x is greater than or equal to the maximum element in the array. In this example, searching for any element greater than or equal to 8 costs three comparisons. Thus, to find the maximum number of comparisons, we may assume without loss of generality that x is greater than or equal to $A[n]$.
Example 13
Suppose that we want to search for $x = 35$ or $x = 100$ in

$$A[1\dots 14] = [1, 4, 5, 7, 8, 9, 10, 12, 15, 22, 23, 27, 32, 35].$$

In each iteration of the algorithm, the bottom half of the array is discarded until there is only one element:

$$[12, 15, 22, 23, 27, 32, 35,] \longrightarrow [27, 32, 35,] \longrightarrow [35].$$

Therefore, to compute the maximum number of element comparisons performed by Algorithm BINARYSEARCH, we may assume that x is greater than or equal to all elements in the array to be searched. To compute the number of remaining elements in $A[1 \ldots n]$ in the second iteration, there are two cases to consider according to whether n is even or odd. If n is even, then the number of entries in $A[mid+1 \ldots n]$ is $n/2$; otherwise it is $(n-1)/2$. Thus, in both cases, the number of elements in $A[mid+1 \ldots n]$ is exactly $\lfloor n/2 \rfloor$.
Similarly, the number of remaining elements to be searched in the third iteration is $\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/4 \rfloor$.
In general, in the jth pass through the while loop, the number of remaining elements is $\lfloor n/2^{j-1} \rfloor$. The iteration is continued until either x is found or the size of the subsequence being searched reaches 1, whichever occurs first. As a result, the maximum number of iterations needed to search for x is that value of j satisfying the condition

$$\lfloor n/2^{j-1} \rfloor = 1$$

. By the definition of the floor function, this happens exactly when

$$1 \leq n/2^{j-1} < 2.$$

or

$$2^{j-1} \leq n < 2^j,$$

or

$$j - 1 \leq \log n < j$$

. Since j is integer, we conclude that

$$j = \lfloor \log n \rfloor + 1.$$

Example 14
Suppose an array contains 1000000 elements.
Accordingly, using the binary search algorithm,on requires only about 20 comparisons to find the position of an item in the array.

4.1.1 Limitations of the Binary Search Algorithm

Since the binary search algorithm is very efficient (e.g. it requires only about 20 comparisons with an array of 1000000 elements) why would one want to

use any other search algorithm? This is because the algorithm requires two conditions: (1) the array must be sorted and (2) one must have direct access to the middle element in any sub-array. This means that one must essentially use a sorted array to hold the data. But keeping data in a sorted array is normally very expensive when there are many insertions and deletions. Accordingly, in such situation, one may use a different data structure, such as a binary search tree, to store the data.
The performance of the binary search algorithm can be described in terms of a *decision tree*, which is a binary tree that exhibits the behavior of the algorithm. Figure 2 shows the decision tree corresponding to the array given in Examples 13. by searching for the element $x = 22$. The darkened nodes are those compared against x.

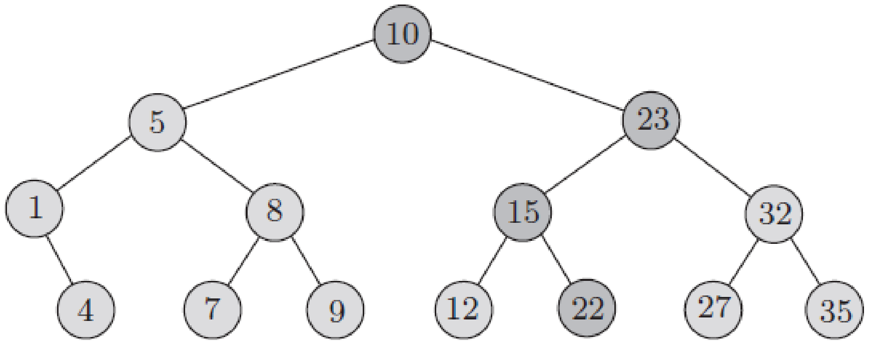

Figure 2: A decision tree that shows the behavior of binary search.

Note that the decision tree is a function of the number of the elements in the array only. Figure 3 on page 23 shows two decision trees corresponding to two arrays of sizes 10 and 14, respectively. As implied by the two figures, the maximum number of comparisons in both trees is 4. In general, the maximum number of comparisons is one plus the height of the corresponding decision tree. Since the height of such a tree is $\lfloor \log n \rfloor$, we conclude that the maximum number of comparisons is $\lfloor \log n \rfloor + 1$. We have in effect given two proofs of the following theorem:

Theorem 3. *The number of comparisons performed by Algorithm BINARY-SEARCH on a sorted array of size n is at most* $\lfloor \log n \rfloor + 1$.

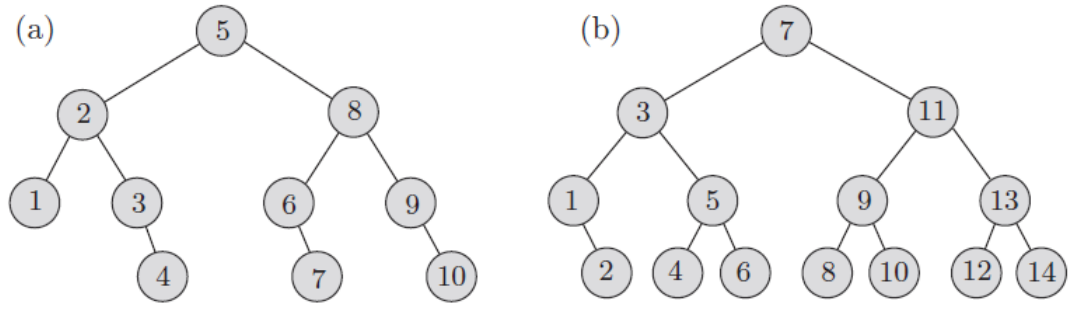

Figure 3: Two decision trees corresponding to two arrays of sizes 10 and 14.

5 Using recurrence relations

In recursive algorithms, a formula bounding the running time is usually given in the form of a recurrence relation, that is, a function whose definition contains the function itself, e.g. $T(n) = 2T(n/2) + n$. Finding the solution of a recurrence relation has been studied well to the extent that the solution of a recurrence may be obtained mechanically (See any Discrete mathematics book for a discussion on recurrence relations). It may be possible to derive a recurrence that bounds the number of basic operations in a non-recursive algorithm. For example, in Algorithm Binarysearch, if we let $C(n)$ be the number of comparisons performed on an instance of size n, we may express the number of comparisons done by the algorithm using the recurrence

$$C(n) \leq \begin{cases} 1, & \text{if } n = 1 \\ C(\lfloor n/2 \rfloor) + 1, & \text{if } n \geq 2 \end{cases} \tag{6}$$

The solution to this recurrence reduces to a summation as follows.

$$\begin{aligned} C(n) &\leq C(\lfloor n/2 \rfloor) + 1 \\ &= C(\lfloor \lfloor n/2 \rfloor /2 \rfloor) + 1 + 1 \\ &= C(\lfloor n/4 \rfloor) + 1 + 1 \\ &. \\ &. \\ &. \\ &= \lfloor \log n \rfloor + 1, \end{aligned}$$

That is, $C(n) \leq \lfloor \log n \rfloor + 1$. It follows that $C(n) = \mathcal{O}(\log n)$. Since the operation of element comparison is a basic operation in Algorithm Binarysearch, we conclude that its time complexity is $\mathcal{O}(\log n)$.

6 Divide and conquer: Merging two Sorted Lists

Suppose we have an array $A[1..m]$ and three indices p, q and r, with $1 \leq p \leq q < r \leq m$, such that both the sub-arrays $A[p..q]$ and $A[q+1..r]$ are individually sorted in nondecreasing order. We want to rearrange the elements in A so that the elements in the subarray $A[p..r]$ are sorted in nondecreasing order. This process is referred to as merging $A[p..q]$ with $A[q+1..r]$. An algorithm to merge these two sub-arrays works as follows. We maintain two pointers s and t that initially point to $A[p]$ and $A[q+1]$, respectively. We prepare an empty array $B[p..r]$ which will be used as a temporary storage. Each time, we compare the elements $A[s]$ and $A[t]$ and append the smaller of the two to the auxiliary array B,if they are equal we will choose to append $A[s]$. Next, we update the pointers: If $A[s] \leq A[t]$, then we increment s, otherwise we increment t. This process ends when $s = q+1$ or $t = r+1$. In the First case, we append the remaining elements $A[t..r]$ to B, and in the second case, we append $A[s..q]$ to B. Finally, the array B[p::r] is copied back to A[p::r]. This procedure is given in Algorithm MERGE.

Algorithm MERGE

Input: An array $A[1..m]$ of elements and three indices p, q and r, with $1 \leq p \leq q < r \leq m$, such that both the subarrays $A[p..q]$ and $A[q+1. : .r]$ are sorted individually in nondecreasing order.

Output: $A[p..r]$ contains the result of merging the two sub-arrays $A[p..q]$ and $A[q+1..r]$

```
1. comment: $B[p..r]$ is an auxiliary array.
2. $s \longleftarrow p$; $t \longleftarrow q+1$; $k \longleftarrow p$
3. while $s \leq q$ and $t \leq r$
4.         if $A[s] \leq A[t]$ then
5.                 $B[k] \longleftarrow A[s]$
6.                 $s \leftarrow s+1$
7.         else
```

```
8.              $B[k] \longleftarrow A[t]$
9.              $t \longleftarrow t+1$
10.         end if
11.          $k \longleftarrow k+1$
12.  end while
13.          if $s = q+1$ then $B[k..r] \longleftarrow A[t..r]$
14.          else $B[k..r] \longleftarrow A[s :: q]$
15.          end if
16.  $A[p..r] \longleftarrow B[p..r]$
```

Let n denote the size of the array $A[p..r]$ in the input to Algorithm MERGE, i.e., $n = r-p+1$. We want to Find the number of comparisons that are needed to rearrange the entries of $A[p..r]$. It should be emphasized that from now on when we talk about the number of comparisons performed by an algorithm, we mean element comparisons, i.e., the comparisons involving objects in the input data. Thus, all other comparisons, e.g. those needed for the implementation of the **while** loop, will be excluded.

Let the two sub-arrays be of sizes n_1 and n_2, where $n_1 + n_2 = n$. The least number of comparisons happens if each entry in the smaller sub-array is less than all entries in the larger sub-array. For example, to merge the two sub-arrays

$$[2\,3\,6] \; \textit{and} \; [7\,11\,13\,45\,57],$$

the algorithm performs only three comparisons. On the other hand, the number of comparisons may be as high as $n-1$. For example, to merge the two sub-arrays

$$[2\,3\;66] \; \textit{and} \; [7\,11\,13\,45\,57],$$

seven comparisons are needed. It follows that the number of comparisons done by Algorithm MERGE is at least n_1 and at most $n-1$.

Observation 1. *The number of element comparisons performed by Algorithm MERGE to merge two nonempty arrays of sizes n_1 and n_2, respectively, where $n_1 \leq n_2$, into one sorted array of size $n = n_1 + n_2$ is between n_1 and $n-1$. In particular, if the two array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons needed is between $\lfloor n/2 \rfloor$ and $n-1$.*

How about the number of element assignments (again here we mean assignments involving input data)? At first glance, one may start by looking at

the **while** loop, the **if** statements, etc. in order to find out how the algorithm works and then compute the number of element assignments.
However, it is easy to see that each entry of array B is assigned exactly once. Similarly, each entry of array A is assigned exactly once, when copying B back into A. As a result, we have the following observation:

Observation 2. *The number of element assignments performed by Algorithm MERGE to merge two arrays into one sorted array of size n is exactly $2n$.*

7 On some naive sorting: algorithms

Let $A[1 \dots n]$ be an array of n elements. A simple and straightforward algorithm to sort the entries in A works as follows. First, we find the minimum element and store it in $A[1]$. Next, we find the minimum of the remaining $n-1$ elements and store it in $A[2]$. We continue this way until the second largest element is stored in $A[n-1]$. This method is described in Algorithm SELECTIONSORT.

Algorithm SELECTIONSORT
Input: An array $A[1 \dots n]$ of n elements.
Output: $A[1 \dots n]$ sorted in nondecreasing order.

```
1. for $i \longleftarrow 1$ to $n-1$
2.         $k = i$
3.         for $j \longleftarrow i+1$ to $n$ {Find the ith smallest element.}
4.                 if $A[j] < A[k]$ then $k \longleftarrow j$
5.         end for
6.         if $k \neq i$ then interchange $A[i]$ and $A[k]$
7. end for
```

It is easy to see that the number of element comparisons performed by the algorithm is exactly

$$f(n) = \sum_{i=1}^{n-1}(n-i) = (n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

It is also easy to see that the number of element interchanges is between 0 and $n-1$. Since each interchange requires three element assignments, the number of element assignments is between 0 and $3(n-1)$.

Observation 3. *The number of element comparisons performed by Algorithm SELECTIONSORT is $n(n-1)/2$. The number of element assignments is between 0 and $3(n-1)$.*

7.1 Insertion Sort

As stated in Observation 1.1 above, the number of comparisons performed by Algorithm SELECTIONSORT is exactly $n(n-1)/2$ regardless of how the elements of the input array are ordered. Another sorting method in which the number of comparisons depends on the order of the input elements is the so-called ELECTIONSORT. This algorithm, which is shown below, works as follows. We begin with the sub-array of size 1, $A[1]$, which is already sorted. Next, $A[2]$ is inserted before or after $A[1]$ depending on whether it is smaller than $A[1]$ or not. Continuing this way, in the ith iteration, A[i] is inserted in its proper position in the sorted sub-array $A[1 \ldots i-1]$. This is done by scanning the elements from index $i-1$ down to 1, each time comparing $A[i]$ with the element at the current position. In each iteration of the scan, an element is shifted one position up to a higher index. This process of scanning, performing the comparison and shifting continues until an element less than or equal to $A[i]$ is found, or when all the sorted sequence so far is exhausted. At this point, $A[i]$ is inserted in its proper position, and the process of inserting element $A[i]$ place is complete. in its proper place is complete.

Algorithm INSERTIONSORT
Input: An array $A[1 \ldots n]$ of n elements.
Output: $A[1 \ldots n]$ sorted in nondecreasing order.

```
1. for $i \leftarrow 2$ to $n$
2.         $x \leftarrow A[i]$
3.         $j \leftarrow i-1$
4.         while $(j > 0)$ and $(A[j] > x)$
5. $A[j+1] \leftarrow A[j]$
6.                 $j \leftarrow j-1$
7.         end while
8.         $A[j+1] \leftarrow x$
9. end for
```

Unlike Algorithm SELECTIONSORT, the number of element comparisons done by Algorithm INSERTIONSORT depends on the order of the input elements.

It is easy to see that the number of element comparisons is minimum when the array is already sorted in nondecreasing order. In this case, the number of element comparisons is exactly $n-1$, as each element $A[i]$, $2 \lneq i \lneq n$, is compared with $A[i-1]$ only. On the other hand, the maximum number of element comparisons occurs if the array is already sorted in decreasing order and all elements are distinct. In this case, the number of element comparisons is

$$f(n) = \sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2},$$

as each element $A[i]$, $2 \leq i \leq n$, is compared with each entry in the sub-array $A[1..i-1]$. This number coincides with that of Algorithm SELECTION-SORT.

Observation 4. *The number of element comparisons performed by Algorithm INSERTIONSORT is between $n-1$ and $n(n-1)/2$. The number of element assignments is equal to the number of element comparisons plus $n-1$.*

Notice the correlation of element comparisons and assignment in Algorithm INSERTIONSORT. This is in contrast to the independence of the number of element comparison in Algorithm SELECTIONSORT related to data arrangement.

8 Bottom-up Merge Sorting

The two sorting methods discussed thus far are both inefficient in the sense that the number of operations required to sort n elements is proportional to n^2. That means that the number of comparison in these algorithms is proportional to the square of the size of the array. If the size of the array doubles, the number of comparison quadruples. If the size of array triples, the numbit will take nine times as long to finish. Computer scientist call this a quadratic on n^2 algorithm.
In this section, we describe an efficient sorting algorithm that performs much fewer element comparisons called *buttom-up merge sort algorithm.*
In the case of the buttom-up merge sort, we divide the array into two pieces and sorted the individual piece before merging them together. The real work

is done during the merge process when the elements in the sub-array are copied back into the original array.

Figure 4 on page 29 depicts the merging process to sort the array

$$A[1..8] = [3,\ 1,\ 4,\ 1,\ 5,\ 9,\ 2,\ 6]$$

.The dashed lines show the original array is continually halved until each element is its own array with the values shown at the bottom. The single-element array are then merged back up into the two item array to produce the values shown in the second level. The merging process continues up the diagram to produce the final sorted version of the array shown at the top.

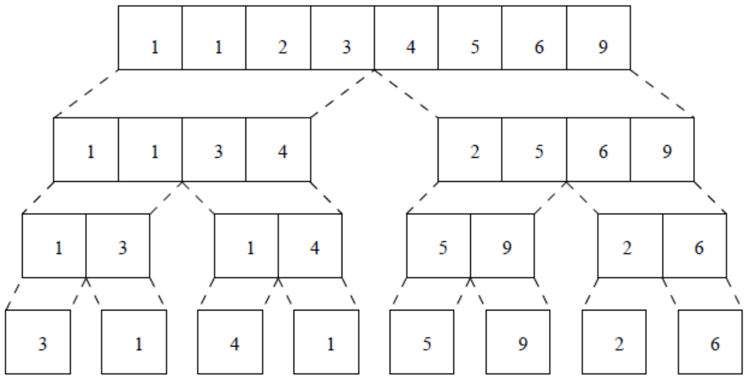

Figure 4: Merges required to sort $A[1..8] = [3,\ 1,\ 4,\ 1,\ 5,\ 9,\ 2,\ 6]$

Algorithm Bottomupsort

Input: An array $A[1..n]$ of n elements.

Output: $A[1..n]$ sorted in nondecreasing order.

```
1. $t \longleftarrow 1$
2. while $t < n$
3. $s \longleftarrow t$; $t \longleftarrow 2s$; $i \longleftarrow 0$
4.     while $i + t \leq n$
5.         MERGE($A$, $i+1$, $i+s$, $i+t$)
6.         $i \longleftarrow i + t$
7.     end while
8.     if $i + s < n$ then Merge($A$, $i+1$ $i+s$, $n$)
9. end while
```

The diagram makes analysis of the bottom-up merge sort easy. Starting at the bottom level, we have to copy the n elements in the second level. From the second to third, the n values. The only question left to is how many levels are there? This boils down to how many times an array of size n can be split in half. You already know from the analysis of binary search that this just $\log_2 n$. Therefore, the total work required to sort n elements is $n \log_2 n$. Computer scientists call this an $n \log n$ algorithm.

So which is going to be better, the n^2 sorting algorithm or the $n \log n$ bottom-up merge? If the input size is small, the selection sort might be a little faster because the algorithm is simpler and there is less overhead. What happens, though as n gets larger? we saw in the analysis of the binary search that the log function grows very slowly ($\log_2 16,000,000 \approx 24$) so $n(\log_2 n)$ will grow much slower than $n(n)$. (see Figure 1 on page 6 for more illustration).

9 Further Reading

1. Introduction to Algorithms by Thomas H Corman
2. Algorithms by Robert Sedgewick & Kevin Wayne.
3. The Algorithm Design Manual by Steve S
4. Algorithm for Interviews by Admin Aziz and Amit Prakash
5. Algorithm in Nutshell.
6. Algorithm Design by Kleinberg & Tardos
7. Introduction to Algorithms: A Creative Approach by Udi mamber
8. The Design and Analysis of Algorithms
9. Data Structures and Algorithms. Aho, Ullman & Hopcroft