THE AMERICAN UNIVERSITY IN CAIRO
100 YEARS | SCHOOL OF SCIENCES AND ENGINEERING
Department of Computer Science and Engineering

Lecture Notes
on

Analysis and Design of Algorithms

Prof. Amr Goneid

CSCE 321/2202 Fall 2019

Part 0. Introduction

0.1 Algorithms

An Algorithm is a procedure consisting of a sequence of primitive steps to do a certain task. The word *Algorithm* comes from the name of *Abu Ja'afar Mohamed ibn Musa Al Khowarizmi* (c. 825 A.D.), a renowned scientist and mathematician who lived in the ninth century A.D.

An Algorithm is supposed to solve a general, well-specified problem. We may consider an algorithm as a function *f* that takes input *X* and maps it to an output *Y* with certain properties: $f : X \rightarrow Y$. The algorithm *f* is called *correct* if and only if for every possible input *X*, it outputs *Y* with the correct properties.

Generally, algorithms use two types of resources: Time and Space. An algorithm is called *efficient* if it uses as few as possible of such resources. We judge the efficiency of an algorithm or compare between the efficiencies of two different algorithms solving the same problem using *Algorithm Analysis*, i.e. the methods we use to analyze the resource usage of given algorithms (time and space).

In *Algorithm Design*, our goal is to design efficient algorithms. Here, we must be aware of two issues. The first issue is that designing an algorithm for a specific problem depends on the nature of the problem and, for it to be efficient; it must make advantage of the nature of the input and of the desired output. The second issue is that we do not have a "design manual"; rather, we can pose a set of design methodologies or guidelines and learn efficient designs by experience.

The reasons why we study this subject include:

1. Algorithm analysis provides a means to distinguish between what is practically possible and what is practically impossible.
2. Efficient algorithms lead to efficient programs that make better use of computer resources.
3. Efficient programs sell better.
4. Programmers who write efficient programs are preferred.

Part 1. Complexity Bounds

1.1 Algorithm Strategy:

An Algorithm is supposed to solve a general, well-specified problem. The solution must follow a strategy that would be implemented in the form of steps to reach the solution. The following examples show how a strategy can be mapped into an algorithm:

1.1.1 Sorting Problem:

In this problem, the input is a random sequence of elements $\{a_0 , a_1 , a_2 , \ldots, a_{n-1}\}$ and the desired output is a permutation (re-ordering) of the input, $\{a'_0 , a'_1 , \ldots, a'_{n-1}\}$ such that $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$. An *instance* of this problem might be sorting an array of names or sorting an array of integers. For an algorithm to be complete, it must be able to solve *all instances* of the problem. One possible strategy to solve this problem is as follows:

"From those elements that are currently unsorted, find the smallest and place it next in the sorted list".

Such strategy leads to the following algorithm:

ALGORITHM SORT (*a[0 .. n-1]*)
for i = 0 to n-2 do

- find smallest element in sub-array a_i to a_{n-1}
- swap that element with that at the start of the sub-array

This algorithm is called "Selection Sort"

1.1.2 The Greatest Common Divisor (GCD):

Reduction of fractions like 14/35 and 15/20 can be done by dividing both the numerator and the denominator by the GCD of the two. The *greatest common divisor* (GCD) of two integers *m* and *n* is the greatest integer that divides both *m* and *n* with no remainder. Thus, the problem is:
Given integers m and n such that $m \geq n > 0$, find the GCD of m and n.
A famous algorithm to solve this problem is *Euclid's algorithm***.** It is named after the ancient Greek mathematician Euclid who described it around 300 B.C. It is based on the following strategy:

"The GCD of two numbers does not change if the larger number is replaced by its difference with the smaller number. For example, 6 is the GCD of 42 and 12 and it is also the GCD of 30 and 12. Repeating this replacement gives successively smaller pairs of numbers until the two numbers become equal. When that occurs, they are the GCD of the original two numbers".

Such strategy leads to the following algorithm:

```
ALGORITHM GCD (m , n)
while n > 0 do
    r ← m mod n;
    m ← n;
    n ← r;
return m
```

The above algorithm can also be recursively implemented:

```
ALGORITHM RGCD (m , n)
if (m mod n == 0)  return n
        else  return gcd (n, m mod n)
```

According to Donald Knuth (The Art of Computer Programming, Vol. 2), *"The Euclidean algorithm is the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day"*.

1.1.3 Finding All Primes less than N:

An integer n is prime iff n ≥ 2 and n's only factors are 1 and itself. Our problem is to determine all the primes between a certain set of numbers. One possible solution to this problem is the Sieve of Eratosthenes, which is a method used to compute all primes less than N. It was invented back in around 240 BC by the Greek mathematician and astronomer *Eratosthenes of Cyrene (276 BC - 194 BC)*, who was also the former director of the famed Library of Alexandria *(Also see http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)*. A very recent article (Sept 28, 2016) on how this algorithm is still of great interest appears in: *http://www.sciencealert.com/an-ancient-greek-algorithm-could-be-the-key-to-finding-new-prime-numbers*

The strategy of the Sieve of Eratosthenes algorithm is as follow:

"Iteratively mark as non-prime all the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime".

To implement this strategy, we use an array of size N+1: a_0, a_1, a_2, ...a_i,....., a_N and the objective is to set $a_i = 1$ if i is a prime, and to zero if it is not. This leads to the following algorithm:

```
ALGORITHM SIEVE (a[0..N])
set a0 = a1 = 0
set all other a's to 1
for i = 2 to N/2 do
        for j = 2 to N / i do
                k = i * j;   set ak = 0;
for i = 1 to N do
        if ( ai ) display i ;
```

1.2 Cost of running an Algorithm:

1.2.1 Time and Space Cost

The running cost of an algorithm measures the extent of usage of computer resources to perform the desired task. Two main costs are considered; Time Resources and Space Resources.

Time cost depends on the type and number of operations executed and on the used hardware (machine speed). Space cost comes from the size of the data used as well as on the size of code of the algorithm.

Technological advances in space resource expansion (RAM, HDD, etc.) has proved to be much faster than expansion in machine speeds. That is why we focus primarily on time cost as far as algorithm analysis is concerned.

1.2.2 Number of Operations

In quantifying time cost of an algorithm, it is clear that different times will be consumed by the same algorithm if we run it on two machines with different hardware speeds. Instead, we can eliminate hardware properties by measuring time cost by the number of operations done by the algorithm. In this case, we are to evaluate a function *T(n)* representing the number of operations as a function of the problem size (*n*), i.e. the size of the data used or the number of iterations needed. For a given algorithm, we can evaluate *T(n)* for a given type of operations by adding all such operations executed by the algorithm. For example, *T(n)* may represent the number of comparisons, or the number of arithmetic operations done by the algorithm, etc.

1.2.3 Types of Operations to Count

We consider each of the “atomic” operation to be one operation. By ”atomic” operations we mean arithmetic operations like + , - , % , * , / etc., relational operations like > , <, >= , etc. and Boolean operations like ! , && , ||, etc. We also consider memory access [ ] as one operation. However, “sort”, selection constructs, repetition constructs and function calls are not considered as atomic and must be evaluated as composite processes.

1.2.4 Examples for Computing the Number of Operations

Example (1): Computing the Factorial

The factorial of a non-negative integer (*n*) is defined as:

$$n! = \begin{cases} 1 & \textbf{for } n = 0 \\ \prod_{i=1}^{n} i & \textbf{for } n > 0 \end{cases}$$

An algorithm to compute *n!* is:

```
ALGORITHM FACTORIAL (n)
f = 1;
if (n > 0)
   for i = 1 to n do
         f ← f * i ;
return f;
```

We want to find *T(n)* = number of multiplications done by the algorithm to compute *n!*. To do this, we observe that one multiplication is done inside the for loop and hence,

$$\boldsymbol{T(n)=\sum_{i=1}^{n}1=n}$$

We call this algorithm *"exact"* because *T(n)* is always equal to *n*.

Example (2): Location of Maximum Element in an Array

Consider a randomly ordered array *a[0..n-1]* of size (*n*) elements. An algorithm to find the location of the maximum element is:

```
ALGORITHM LOC_OF_MAX (a[0..n-1])
m = 0;
for i = 1 to n-1 do
         if (a_i > a_m) m = i ;
return m;
```

We want to find *T(n)* = number of comparisons done by the algorithm. To do this, we observe that one comparison is done inside the for loop and hence,

$$\boldsymbol{T(n)=\sum_{i=1}^{n-1}1=n-1}$$

We also call this algorithm *"exact"* because *T(n)* is always equal to *n - 1*.

We notice two important features from the above two examples:

1. In calculating the number of chosen operations, the loops iterating on the operation have loop indices incrementing by +1 from a lower bound (LB) to an upper bound (UB). This allows us to model the loop as a summation from the LB to the UB. This is also an example of the mathematical modeling of repetition constructs in algorithm analysis. More on such mathematical modeling of constructs will be discussed later.
2. Both algorithms are *"tightly bound"* in the sense that *T(n)* is *exact* and is independent of the data so long as the problem size is kept at a value *n*.

Example (3): Linear Search

Consider the problem of searching for a "target" element in a randomly ordered array *A[0..n-1]* of *n* elements. We want to find the location in the array of the element matching "target". The algorithm should return -1 in case there is no matching. Using linear search, the algorithm would be:

```
ALGORITHM LINEAR_SEARCH (A[0..n-1], target)
for i = 0 to n-1 do
        if (Ai == target) return i ;
 return -1;
```

We want to find *T(n)* = number of comparisons done by the above algorithm. Notice that in case of successful matching at location (*i*), the number of comparisons done is *T(n) = i + 1*, leading to a *"Best Case"* at location *i = 0*, with *T(n) = 1* comparison. Notice also that if successful matching occurs at the last location *i = n-1*, or if no matching occurs at all, then *T(n) = n* comparisons leading to a *"Worst Case"*. We now realize that the above algorithm is not exact, i.e., it is *"loosely bound"*, since *T(n)* can be any number between 1 and *n*.

1.3 Best Case, Worst Case and Average Case:

From the above examples, we may distinguish between algorithms as tightly bound (exact) in case *T(n) = f(n)* always, or as loosely bound when the cost *T(n)* for a fixed problem size (*n*) changes with the data values. In the latter case, *T(n) = f(n)* in the best case and *T(n) = g(n)* in the worst case, where $f(n) \neq g(n)$.

For some of the loosely bound algorithms, we may find it informative to compute an average case for *T(n)*. However, such average case analysis requires a knowledge of the probabilities of the different patterns of the data sets of a given size *n*. As an example, let us revisit the linear search algorithm given that the probabilities for successful match at all locations are known. A rather simple situation exists if the probabilities are all equal; in other words, the probability of successful match at location (*i*) is *p(i) = 1/n* so that the sum of all probabilities from *i = 0 to i = n-1* is equal to 1. Recalling that the number of comparisons done for a successful match at location (*i*) is *T(i) = i +1*, then

$$p(i) = 1/n \qquad 0 \le i \le n-1$$

$$\sum_{i=0}^{n-1} p(i) = 1$$

It follows that on the average

$$< T(n) > = \frac{\sum_{i=0}^{n-1} p(i)T(i)}{\sum_{i=0}^{n-1} p(i)} = (1/n)\sum_{i=0}^{n-1} T(i) = (1/n)\sum_{i=0}^{n-1}(i+1) = (1/n)\sum_{k=1}^{n} k = \frac{n+1}{2}$$

Notice that only in this special case, we could have reached the same result by taking the average between the best and worst case costs. This over-simplification would not have been possible if the probabilities were not equal over the whole range of locations.

1.4 Asymptotic Notations and Bounds

The number of operations $T(n)$ as a function of problem size (n) can generally be used to quantify the efficiency of an algorithm. However, in order to simplify the process of comparing the efficiencies of different algorithms, we use the algorithm "complexity" representing a functional form of $T(n)$. For this purpose, we use the asymptotic notation to express the complexity of algorithms as the problem size becomes very large (i.e. their limiting behavior):

$$\lim_{n\to\infty} T(n) = \Psi(f(n))$$

In the above notation, Ψ represents a type of a "*bound*" and $f(n)$ is some functional form (simpler than $T(n)$) like n , $\log n$, etc. The equal sign in the above relation is equivalent to saying "*is of complexity*" and the type of bound is typically one of the three bounds Θ , Ω , and O. Big-Θ represents the tight bound for exact algorithms, and for loosely bound algorithms, Big-Ω represents the lower bound (best case) while Big-O represents the upper bound (worst case).

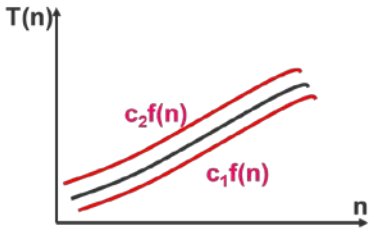
The use of the bounds Θ , Ω , and O allows us to easily compare between different algorithms solving the same problem. For example, algorithm A may be of complexity $O(n)$ and algorithm B has a complexity $O(n^2)$. Since $n < n^2$ for all $n > 1$, then we can judge that algorithm A is more efficient than algorithm B since it has lower complexity.

1.5 Formal Definition of Bounds

1.5.1 Tight Bound Θ

To express the complexity of exact algorithms, we say that $T(n) = \Theta(f(n))$. Formally this means that there exist positive constants c_1, c_2 and n_0 such that

$$c_1 f(n) \le T(n) \le c_2 f(n) \quad \text{for all } n \ge n_0$$



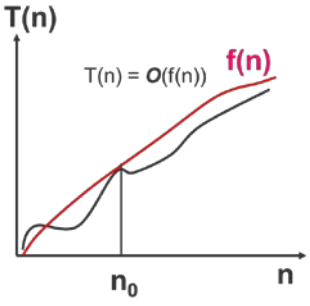
This actually means that $f(n)$ is both a lower bound and an upper bound of $T(n)$. Examples for such complexity are the algorithms for computing the factorial and finding the location of the maximum in an array. In both these examples, $c_1 = c_2 = n_0 = 1$ and $f(n) = n$. Hence $T(n) = \Theta(n)$.

1.5.2 Upper Bound O

The upper bound *Big-O* expresses the complexity in the worst case. Formally,
$T(n) = O(f(n))$ iff there exist positive constants c and n_0 such that

$$T(n) \le c\, f(n) \quad \text{for } n \ge n_0$$

This means that $T(n)$ grows <u>at most</u> as fast as $f(n)$. The linear search example given before is an algorithm with a worst case of $T(n) = n$ and hence its upper bound complexity is $T(n) = O(n)$.



Exercise:

Find positive constants c and n_0 to prove that

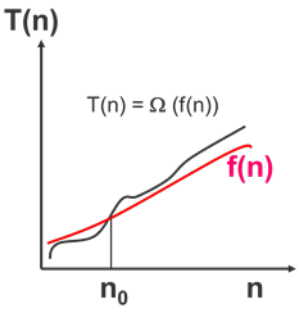
$T(n) = (n+1)^2 = O(n^2)$,

i.e. to show that $(n+1)^2 \leq cn^2$ for $n \geq n_0$

1.5.3 Lower Bound Ω

The lower bound *Big-* Ω expresses the complexity in the best case. Formally,
T(n) = Ω ***(f(n))*** iff there exist positive constants c and n_0 such that

$$T(n) \geq c\, f(n) \quad for\ n \geq n_0$$



This means that $T(n)$ grows <u>at least</u> as fast as $f(n)$. The linear search example given before is an algorithm with a best case of $T(n) = 1$ and hence its lower bound complexity is $T(n) = \Omega(1)$.

1.6 Constants do not matter

When the number of operations is $T(n) = c\, f(n)$ we still say that $T(n)$ *is* $O(f(n))$ or $\Omega\ (f(n))$ or $\Theta\ (f(n))$. This is because constants can be compensated by hardware and software properties. Notice that all bounds are expressed by a functional dependence $f(n)$ within a constant factor c. Different hardware, compilers, operating systems and languages produce different constant factors. That is why we always drop such constants when we derive the complexity from the number of operations. In this case, the complexity is meant to express the rate of growth of the algorithm cost with increasing problem size.
Examples are:

Number of Operations	Bound	Complexity Expression
$T(n) = 4$	*Best Case*	$T(n) = \Omega(1)$
$T(n) = 6\, n^2$	*Worst case*	$T(n) = O\,(n^2)$
$T(n) = 3\, n$	*Tight (Exact)*	$T(n) = \Theta(n)$

Recall that the equal sign in the expression for the number of operations means “equals”, while in the complexity expression it reads “is of complexity”. More examples are as follows:

1. $T(n) \leq 5.5\, n^3$ *for* $n \geq 2$ *hence* $T(n) = O(n^3)$
2. $T(n) \geq 30\, n$ *for* $n \geq 1$ *hence* $T(n) = \Omega(n)$
3. $T(n) = 6 \log_2 n$ *for* $n \geq 2$ *hence* $T(n) = \Theta(\log n)$

1.7 Important Remarks

1. In algorithm complexity comparisons, lower bound complexities (*Big-Ω)* are not very useful since they cannot express worst cases.

2. The *Big-O* complexity is usually used to describe the general behavior of an algorithm. We can still use it to represent the complexity even for tightly bound algorithms without any loss of generality. Also, we usually compare algorithms based on the upper bound (*Big-O*) complexity since it expresses the worst case behavior.

3. The functional form *f(n)* of worst case complexity obtained from the number of operations should be the ***least growing*** function satisfying the formal definition of the *Big-O*. For example, if the number of operations in the worst case is $T(n) = 5\,n$, then it is technically correct that $T(n) = O(n)$, $T(n) = O(n^2)$, and $T(n) = O(2^n)$, etc. This indicates that there is a whole set of functions satisfying the upper bound formal definition. However, the least growing function satisfying the upper bound is the first one, so that the proper complexity in this case is $T(n) = O(n)$.

Exercises

1. Consider a randomly ordered array *a[0..n-1]* of size (*n*) elements and the following algorithm:

```
ALGORITHM FUN (a[0..n-1])
x = a0 ;
for i = 1 to n-1 do
        if (ai < am) x = ai ;
return x;
```

   What does this algorithm do?
   Find *T(n)* = number of comparisons done by the algorithm in the best and worst cases.
   Is this algorithm tightly bound (exact) or loosely bound.

2. Find positive constants c and n_0 to prove that
   $T(n) = (n+1)^2 = O(n^2)$
   i.e. what are c and n_0 such that
   $(n+1)^2 \leq cn^2$ for $n > n_0$

3. The running times of certain algorithms are found to be as follows:
   $T(n) = 10$ *(best case)*
   $T(n) = 6 \log n^2$ *(worst case)*
   $T(n) = 5 n^3$ *(always)*
   What are the corresponding complexities of these algorithms?

4. The running times of certain algorithms are found to have the following bounds:
   $T(n) \leq 5$ *for* $n \geq 2$
   $T(n) \geq 2n$ *for* $n \geq 1$
   $T(n) = 6 \log n$ *for* $n \geq 2$
   What are the corresponding complexities of these algorithms?

5. The number of comparisons done by an algorithm is characterized by the asymptotic behavior:
   $$\lim_{n\to\infty} T(n) \leq 16n \leq 4n^2 \leq 4\,2^n \text{ for } n \geq 4$$
   What is the complexity of this algorithm?

Part 2. Types of Algorithm Complexities

2.1 Rules for Big-O

We recall that the *Big-O* complexity will be used to describe the general behavior of an algorithm and we can still use it to represent the complexity even for tightly bound algorithms without any loss of generality. In addition, we usually compare algorithms based on the upper bound (*Big-O*) complexity since it expresses the worst case behavior.

Since an algorithm may be a composite of different sub-algorithms with possibly different complexities, we need a set of rules to determine the overall complexity of an algorithm. The following is such a set of rules we apply for the *Big-O:*

1. Constant factors may be dropped:
   For a constant $k > 0$, $O(k\, f(n)) = O(f(n))$
   For example, both $2\, n^3$ and $5\, n^3$ are $O(n^3)$

2. Constant complexities are less than powers:
   $O(k) < O(n^k)$ *for all* $k > 0$. *Notice that by the first rule,* $O(k)$ *can be re-written as* $O(1)$.

3. The growth rate of a sum of terms is the growth rate of its fastest growing term:
   $O(f(n)) + O(g(n)) = max(O(f(n)), O(g(n)))$
   For example, $f(n) = 2n = O(n)$, $g(n) = 0.1\, n^3 = O(n^3)$,
   Hence $O(f(n)) + O(g(n)) = max(O(n), O(n^3)) = O(n^3)$

4. The product of big-O's is the big-O of the products:
   $O(f(n)) * O(g(n)) = O(f(n) * g(n))$
   For example, $T_1(n) = O(n)$, $T_2(n) = O(n^2)$ so that $T(n) = T_1(n) * T_2(n) = O(n^3)$

5. Logarithms grow slower than powers:
   $O(\log_a n) < O(n^k)$ *for all* $a > 1$ *and* $k > 0$
   For example, $O(\log n) < O(n)$

6. All logarithms grow at the same rate:
   $\log_a n = \Theta(\log_b n)$ *for all* $a, b > 1$
   For example, $\log_2 n = \Theta(\log_3 n)$.

7. The growth rate of a polynomial of degree m is $O(n^m)$
   For example $T(n) = 2 - 4n + 3n^2 + 2n^3 = O(n^3)$
   It follows that if $a < b$ then $O(n^a) < O(n^b)$

8. Exponential functions grow faster than powers
   $O(n^k) < O(b^n)$ *for all* $k \geq 0$ *and* $b > 1$
   For example, $O(n^3) < O(2^n)$

2.2 Some Proofs

All the statements of the above rules for the *Big-O* can be proved either by induction on n or by algebraic manipulation.

To prove a statement *S(n)* by induction on n, we follow two steps:

1. Base case: verify that *S(n)* is correct for the smallest possible value, i.e., *n = 1*.
2. Induction step:
   - Assume that *S(n)* holds for an arbitrary input of size *n* and then
   - Prove that it also holds for *n + 1 (Algebra)*

Consider Rule 5 above and take as an example *a = 2* and *k = 1* so that our statement is *O(log n) < O(n)*. To prove this statement, we take the base case of *n = 1* which verifies that the statement is correct since *0 < 1*. For the induction step, we know that *(n+1) ≤ 2 n* for all *n > 0*. Hence, *log (n+1) ≤ log (2n).* But *log (2n) = 1 + log n*, so that *log (n+1) ≤ log n + 1*. Assuming that our statement is correct for *n*, it follows that *log (n+1) ≤ (n + 1)*, proving the statement.

We now consider proving Rule 6 that all logarithms grow at the same rate. Take as an example proving that $log_2 n = \Theta(log_3 n)$. Since $log_2 n = log_3 n / log_3 2 = (Const)\ log_3 n$, then the rule is proved.

Finally, let us consider Rule 7 stating that the growth rate of a polynomial of degree *m* is $O(n^m)$. Such polynomial can be represented as:

$$T(n) = \sum_{i=0}^{m} a_i n^i \le \sum_{i=0}^{m} |a_i|\, n^i, \quad \text{But } \sum_{i=0}^{m} |a_i|\, n^i = n^m \sum_{i=0}^{m} |a_i|\, n^{i-m}$$

The largest value of $n^{i-m} = 1$ (when $i = m$) and $\sum_{i=0}^{m} |a_i| =$ constant (c)

Hence, $T(n) \le cn^m$ *and* $T(n) = O(n^m)$

Exercises

1. Prove by induction on *n* that $n^2 < 2^n$
2. Prove by induction on *n* that $2^n < n!$ for $n \ge 4$
3. Prove by induction on *n* that $S(n) = \sum_{i=1}^{n} (2i - 1) = n^2$ for $n > 0$
4. Prove by induction on *n* that $\sum_{i=1}^{n} i = n(n+1)/2$ for $n > 0$

2.3 Comparing Complexities

2.3.1 Dominance

When we compare complexities based on the Big-O or when Big-O's are summed, we usually look for the "dominant term". Consider two terms *f(n)* and *g(n)*, to determine who dominates who, we follow a simple procedure as follows:

1. If $lim_{(n->\infty)}\ f(n)/g(n) = \infty$ then *f(n)* dominates (i.e. grows faster)
2. If $lim_{(n->\infty)}\ f(n)/g(n) = 0$ then *g(n)* dominates.
3. If $lim_{(n->\infty)}\ f(n)/g(n) = constant$ then both grow at the same rate.

Examples are:

1. if $a > b$ then n^a dominates n^b since $lim_{(n->\infty)}\ n^a/n^b = lim_{(n->\infty)}\ n^{a-b} = \infty$
2. n^2 dominates $(3n+2)$ since $lim_{(n->\infty)}\ (3n+2)/n^2 = lim_{(n->\infty)}(3/n) + lim_{(n->\infty)}(2/n^2) = 0$
3. $(n \log n)$ and $(n \log n^3)$ grow at the same rate since $(n \log n^3) = (3\ n \log n)$ so that $lim_{(n->\infty)}\ (n \log n) / (3\ n \log n) = 1/3 = constant$

In finding the limit $lim_{(n->\infty)}\ f(n)/g(n)$ we sometimes need to use ***L'Hopital's*** rule which states that the asymptotic relationship of $f(n)$ to $g(n)$ is the same as the asymptotic relationship of the derivatives (with respect to n) of $f(n)$ and $g(n)$, i.e.,

$lim_{(n->\infty)}\ f(n)/g(n) = lim_{(n->\infty)}\ f'(n)/g'(n)$

For example, $f(n) = n \log n + n$ and $g(n) = n^2$.

$lim_{(n->\infty)}\ f(n)/g(n) = lim_{(n->\infty)}\ (n \log n + n)/ n^2 = lim_{(n->\infty)}\ (2 + \log n)/(2\ n) =$

$= lim_{(n->\infty)}(1/n) + lim_{(n->\infty)}(\log n /(2n)) = 0 + lim_{(n->\infty)}(1/(2n)) = 0 + 0 = 0$

Hence, $g(n)$ dominates $f(n)$

Exercises

1. Determine the dominant term in the expression $T(n) = n^2 \log n + n (\log n)^2$
2. Determine which grows faster:
   - $\log n$ or $n^{1/2}$
   - $n\ 2^n$ or $(n + \log n)\ 2^{n-1}$

2.3.2 Common Types of Complexities

The following table shows the common complexities in the order of their growth rate.

Complexity	***Meaning***
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n^a)$ for $a < 1$	Sub-Linear
$O(n)$	Linear
$O(n \log n)$	Super-Linear
$O(n^a)$ for $a > 1$	Polynomial
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(a^n)$ for $a > 1$	Exponential
$O(n!)$	Factorial
$O(n^n)$	Extremely Fast Growing

2.4 Complexities of Some Practical Algorithms

In the following, we highlight the complexities of some known practical algorithm:

1. Linear Search and Binary Search:
   Linear Search in an array has a linear complexity $O(n)$ while Binary Search is logarithmic, i.e., $O(\log n)$. Therefore, Binary Search is faster since $O(\log n) < O(n)$.
2. Sorting:
   Selection sort has a complexity of $O(n^2)$ while Quicksort is $O(n \log n)$. Quicksort is much faster since $O(n \log n) < O(n^2)$.
3. Brute–Force matrix multiplication has a cubic complexity $O(n^3)$
4. The number of sub-sets in a set of n items is 2^n. For example, in a set of 3 items$\{a,b,c\}$, the subsets are: $\{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}$. A Brute-Force algorithm to list all these sub-sets will have to do 2^n listing operations, with a complexity of $O(2^n)$, i.e., exponential complexity.
5. The number of permutations of a sequence of n items is $n!$. For example, for a sequence of 3 items $\{abc\}$, the possible permutations are: $\{abc\},\{acb\},\{bac\}, \{bca\}, \{cab\}$ and $\{cba\}$, i.e., 3! = 6 permutations. To list all such permutations, we require $O(n!)$ operations, i.e. factorial complexity.

2.5 Polynomial and Intractable Algorithms

An algorithm is said to have *Polynomial Time* complexity if it is $O(n^a)$ for some constant $a \geq 0$. Examples are $O(1), O(n), O(n \log n), O(n^{2.})$. Such algorithms can solve problems in reasonable time. However, some problems are *Intractable* because as the problem size becomes large, it is not possible to solve them in reasonable time. As examples, we mention algorithms of complexities $O(2^n)$, $O(n!)$ and $O(n^n)$.

In order to visualize *"unreasonable time"*, let us consider the problem of listing all subsets of a set of n items, which has a complexity of $O(2^n)$. For some moderate n, say $n = 100$ and using a fast machine that would do one listing operation in one nanosecond $(10^{-9}$ sec), we need a total time

$t = 2^{100} \times 10^{-9}$ seconds $= 1.267 \times 10^{21}$ seconds $= 4.03 \times 10^{13}$ Years!.

That needed time would be greater than several times of the whole age of the Universe!

Some complexities are even worse as $O(2^n) < O(n!) < O(n^n)$. For example, we show that: $O(2^n) < O(n!)$. Taking logs on both sides, we now compare $O(n)$ with $O(n \log n)$.
Since $\lim_{(n->\infty)} (n \log n) / n = \infty$, we conclude that $O(2^n) < O(n!)$.

Exercises

1. Prove by induction on n that:
   - $n^2 < 2^n$
   - $2^n < n!$ for $n \geq 4$

2. Prove by induction on n that for $n > 0$

$$S(n) = \sum_{i=1}^{n} (2i - 1) = n^2$$

$$S(n) = \sum_{i=1}^{n} i = n(n+1)/2$$

3. Prove that:
   - if $a > b$ then n^a dominates n^b
   - n^3 dominates $(3n^2 + 2n \log n)$
   - n^3 dominates $(n^2 \log n)$

4. Using *L'Hopital's* rule, determine which of the functions $f(n)$ and $g(n)$ grows faster (justify your answer fully):

$f(n) = n^{(k+\alpha)} + n^k (\log n)^2$	*or*	$g(n) = k\, n^{(k+\alpha)}$ *with the constants* $k, \alpha > 0$
$f(n) = n^{1/2} (\log n)^2$	*or*	$g(n) = (2 + n) \log n$
$f(n) = n\, 3^n$	*or*	$g(n) = (n^2 + n^{1/2})\, 2^n$

5. For the following pair of functions, find which grows faster:

$f(n) = n^3$	*or*	$g(n) = n \log n$
$f(n) = n^{0.001}$	*or*	$g(n) = \log n$
$f(n) = 2^{n+1}$	*or*	$g(n) = 2^n$
$f(n) = 2^n$	*or*	$g(n) = 2^{2n}$

6. For the following pair of functions, find which has smaller complexity:

$f(n) = 100\, n^4$	*or*	$g = n^5$
$f(n) = \log(\log n^3)$	*or*	$g = \log n$
$f(n) = n^2$	*or*	$g = n \log n$
$f(n) = 50\, n^5 + n^2 + n$	*or*	$g = n^5$
$f(n) = e^n$	*or*	$g = n!$

7. Arrange the following functions according to their order of growth from the lowest to the highest (show the steps used in the arrangement):
   $(n-2)!$, $5 \log(n + 100)^{10}$, 2^{2n}, $0.001n^4 + 3n^3 + 1$, $\ln^2 n$, $n^{1/3}$, 3^n.

8. Suppose that we have a chessboard of size $N \times N$ and we want to place N queens on the board such that no two queens occupy the same square. Show that the number of possible placements is $O((N^2)!)$

Part 3 Time Complexity Calculations

3.1 Algorithm Performance Measurement

In algorithm analysis, we are interested in quantifying the extent of usage of the computer time resources. This may be achieved by one of the following methods:

a) Clocking method by direct measurement of run time of the code, or
b) Counting method by counting the number of operations done by the algorithm as it actually runs, or
c) Mathematical modeling of the algorithm to obtain some functional form *g(n)* for the dependence of the number of operations *T(n)* on problem size *n*.

Methods (a) and (b) are empirical approaches to algorithm analysis. Method (a) may be more useful for real-time applications and requires coding and actual running of the algorithm on a machine to measure the time of execution. Method (b) also requires coding and actual running of the algorithm on a machine after inserting counters for relevant operations. However, using one of these two methods requires the following:

1. Choice of the bound (lower bound or upper bound)
2. Choice of clocking or counting domains in the algorithm (which part to clock or where to insert counters for the selected operations)
3. Choice of the data sets that reflect the bounds
4. Choice of the data sizes (what are the values of *n* for the data sets)

Such choices add constraints to the process of algorithm analysis as it sometimes becomes difficult to predict performance at problem sizes not included in the experiments. In the following, we give examples of methods (a) and (b).

3.1.1 Example of Clocking Method

Consider an algorithm ***A****(Data (n))*, using a chosen data set *Data(n)* of chosen size *n*. A clocking method would be:

```
ALGORITHM Clocking
Set repetition count m;
Set data size n;  Set data set Data (n)
tacc = 0;
Repeat on m {
        start time t1 = GetTime();
        Invoke A(Data(n));
        end time t2 = GetTime();
        tacc = tacc + (t2 - t1);
}
tav = tacc / m;
```

The result obtained is the running time of the algorithm, taken as an average over *m* experiments.

3.1.2 Examples of Counting Method

Example (1):
Consider an algorithm to compute the average value in a 2-D array *A[0..n-1][0..n-1]* of floating point elements with size *n x n.* The following counting code gives the number of floating point arithmetic operations *T(n)* done by the algorithm for a particular size *n*:

```
ALGORITHM Aver2D
int count = 0;
float s = 0.0;     float A[n][n];
for (i = 0; i < n; i++)
        for (j = 0, j < n; j++)
                { s = s + A[i][j];  count++;}
float Average = s / (n * n); count++;
```

Example (2):
Consider the Insertion Sort algorithm to sort an array *a[0..n-1]* of size *n* elements. The following code counts the number of data shifts *T(n)* as a function of size *n*:

```
ALGORITHM Insertsort (int a[ ], int n)
{
        int i , v , j ;
        for (i =1; i < n; i++)
        {
            v = a[i]; j = i;
            while (j > 0 && a[j-1] > v)
                { a[j] = a[j-1]; j--; count++;};
            a[j] = v;
        }
}
```

The following table shows the counting results obtained by running this code for *n* between 5 and 200. Three cases are considered, a best case where the input data is already sorted in ascending order ($T_1(n)$), an average case where the array is randomly ordered ($T_2(n)$) and the worst case when the array is already sorted in descending order ($T_3(n)$).

n	5	10	20	50	100	200
$T_1(n)$	0	0	0	0	0	0
$T_2(n)$	3	18	78	586	2609	10,052
$T_3(n)$	10	45	190	1225	4950	19,900
n(n-1)/2	10	45	190	1225	4950	19,900

The table shows that the number of data shifts is always zero as expected in the best case and the average is somewhat between the worst and best cases. What is significant is that some functional dependence *g(n) = n(n-1)/2* coincides exactly with the worst case for all *n* considered. If we want to find the worst case cost at say *n = 500*, then without knowledge of

g(n) we have to run the code at that size. However, with *g(n)* given, we can simply predict that at *n = 500*, the number of data shifts is exactly 500 x 499 / 2 = 124,750.

Another advantage of seeking a functional dependence of the number of operations is that when *T(n)* is directly proportional to some *g(n)*, and one timing experiment is conducted at a given size *N*, then it is possible to predict the time of execution of the algorithm at other times *n* without running the algorithm. To see this feature, consider the following example:

Example on run time prediction:

The number of operations of a sorting algorithm is directly proportional to $g(n) = n \log n$. Direct time measurement gives 1 ms to sort 1000 items. Find how long it will take to sort 1,000,000 items.

To predict the run time at $n = 1{,}000{,}000$, we know that $T(n) \alpha\ n \log n$, or, $T(n) = C\ n \log n$, where *C* is a constant. The value of *C* can be determined from the direct time measurement at $N = 1000$ since $C = 1\ ms / N \log N$. Hence, the time taken for size *n* is $(n/N)\ (\log n / \log N) = (10^6 / 10^3)\ (\log 10^6 / \log 10^3) = 2 \times 10^3$ ms = 2 seconds

Finding a functional dependence *g(n)* for the number of operations in an algorithm is the result of algorithm analysis using method (c) of mathematical modeling. Details of this modeling method are discussed in the following sections.

3.2 Mathematical Modeling of the Number of Operations

We have seen that the number of operations done by an algorithm gives a good measure of the extent of usage of the computer time resources. This enabled us to set up complexity bounds from which the asymptotic behavior of an algorithm can be described. Therefore, the starting step is to obtain a functional dependence of *T(n)* on the problem size *n* for different bounds.

3.2.1 A General Scheme

Given an algorithm in actual code, or preferably in a high-level description (pseudocode), the following describes a general scheme for the algorithm mathematical modeling:

1. Identify the problem size (*n*).
2. Choose type of operation to count (e.g., arithmetic, comparison, etc.)
3. Identify basic constructs in the algorithm such as sequential blocks, selection constructs, repetition constructs and functional constructs
4. Assign a number of chosen operations to each of these constructs
5. Form a mathematical description of the sum of operations *T(n)* in the different constructs
6. Obtain an explicit functional dependence *g(n)* for the number of operations *T(n)*

When we obtain *T(n) = g(n)* by this modeling process, it becomes possible to apply bound rules and to use the asymptotic notation to express the complexity of algorithms as the problem size becomes very large (i.e. their limiting behavior):

$$\lim_{n\to\infty} T(n) = \lim_{n\to\infty} g(n) = \Psi(f(n))$$

In the above notation, Ψ represents a type of the bound (Θ, Ω, O) and $f(n)$ is some functional form of the complexity. Generally, $f(n)$ is a simpler form of $g(n)$ and is derived from $g(n)$ by applying bound rules (e.g., rules for the *Big-O*)

3.2.2 Number of Operations for Different Constructs

a) Simple Statements
$T(n)$ = number of specified operations in the statement.
For example, in the statement $z = 2 * x + y$, $T(n) = 2$ arithmetic operations.

b) Code blocks
$T(n)$ = sum of sub-blocks $T(n)$'s

c) Function Calls
Treated as a whole algorithm.

d) if Statements
With a condition (C) and a statement block (S), let $T_c(n)$ and $T_s(n)$ be the number of operations for C and S, respectively. Now, consider the statement

if (C) S;

The best case (from the cost point of view) is when C is false, and the worst case when it is true. Hence

$T(n) = T_c(n)$ (best case);
$T(n) = T_c(n) + T_s(n)$ (worst case)

Also, for the statement

if (C) S1; else S2;

we have

$T(n) = T_c(n) + min\ (T_{s1}(n), T_{s2}(n))$ (best case)
$T(n) = T_c(n) + max\ (T_{s1}(n), T_{s2}(n))$ (worst case)

Example:

```
if ( x == fun(n))   call module1;  else call module2;
```

Given that *fun (n)* does n comparisons, module1 costs $n + 2$ comparisons and module2 costs $n^2 + 1$ comparisons, what will be the number of comparisons done in the best and worst cases?

Here, we find that:

$T(n) = 1 + n + n + 2 = 2n + 3 = \Omega(n)$ (best case)
$T(n) = 1 + n + n^2 + 1 = 2 + n + n^2 = O(n^2)$ (worst case)

e) while loops

Consider the statement:

while (C) S;

In this case, we have

T(n) = $T_c(n)$ + sum over the number of iterations of the loop of ($T_c(n)$ + $T_s(n)$)

Example (1):
Consider the following segment

```
i = 0;
while (i  <= n)
        { Fun(i,n); i++; }
```

We want to find the number of comparisons done by the above segment given that *Fun(i, n)* costs *(3i + log n)* comparisons.
We notice that the while loop will compare *i* with *n* and iterate for *i = 0,1,2…,n* before doing another comparison to exit the loop. In each of the *n+1* iterations, *Fun(i, n)* makes *3i + log n* comparisons. In this case,

$$\boldsymbol{T(n) = \sum_{i=0}^{n} (3i + \log n + 1) + 1 = (3/2)n(n+1) + (n+1)(\log n + 1) + 1 = O(n^2)}$$

Example (2):
Consider the following segment

```
D = n;
while (D > 0)
        { Fun(n); D = D/2; }
```

We notice that the loop will iterate $\lfloor \log n \rfloor + 1$ times before exit. Given that *Fun (n)* makes *n* arithmetic operations, then in each iteration the cost will be *(n + 1)*. Hence, the total number of such operations done by the loop will be,

T(n) = (n+1)($\lfloor \log n \rfloor$ + 1) = O(n log n)

3.2.3 Number of Operations with For Loops

1. Loop Variable Increment is Additive and Equals ±1

Consider a for loop with a loop variable (*i*) over a segment *F(i):*

```
for (i = 1; i ≤ n; i++)   F(i);
```

To compute the number of operations done by the loop, it is convenient to model it as a summation over the loop variable *(i)*. If the number of operations done by *F(i)* is *T(i)* then:

$$T(n) = \sum_{i=1}^{n} T(i)$$

Implicit in this model is that the loop variable increment (or decrement) is 1. For example, if in the above loop *F(i)* costs *T(i) = 2i* operations, then

$$T(n) = \sum_{i=1}^{n} T(i) = \sum_{i=1}^{n} 2i = n(n+1)$$

Exercise: Redo the above example for the loop

```
for (i = n; i ≥ 1; i--) F(i);
```

when *F(i)* costs *T(i) = 3i + n* operations.

2. Loop Variable Increment is Additive and Equals a Constant C

Consider the loop increment to be additive with a constant *C* with *|C| > 1*. An example with *C = +2* is:

```
for (i = 0; i ≤ n; i += 2) F(i);
```

In this case, *i = 0, 2, 4, 6, …n.* Here we cannot use the straight summation with +1 increment. Therefore, we transform the loop variable *(i)* into another variable *(k)* using the following transformation:

$k = i / C \leftrightarrow i = C\,k$

For the above example *C = +2* and *k = 0, 1, 2, 3, …,* $\lfloor n/2 \rfloor$. The loop is transformed to:

```
for (k = 0; k ≤ n/2; k++) F(i);
```

Notice that although now the loop variable *(i)* is transformed to the new variable *(k)*, the segment inside the loop is still dependent on the old variable *(i)***.** Therefore, we have to express that segment in terms of the new variable *(k)*. For example if *F(i)* costs *T(i) = 2i* operations, then in terms of the new variable *T(k) = 2(2k)*.

In this case, the number of operations done by the loop will be:

$$T(n) = \sum_{k=0}^{n/2} T(k) = \sum_{k=1}^{n/2} 2(2k) = 4\sum_{k=1}^{n/2} k = n(n/2+1)$$

Exercise: Redo the above example for the loop:

```
for (i =0; i < n; i + = 2) F(i);
```

when *F(i)* costs *T(i)* = constant number of operations (*B*).

3. Loop Variable Increment is Multiplicative by a Constant C

Consider the loop increment to be multiplicative by an integer constant $C > 1$.

Example (1):
Loop variable is incremented by multiplying by $C = 2$.

```
for (u = 1; u < n; u *= 2) F(u);
```

In this case, we assume that $n = 2^m$, where m is an integer and $m = \log n$. We then use the transformation:

$j = \log u \leftrightarrow u = 2^j$

For the above example, $u = 1, 2, 4, 8, 16 \ldots 2^{m-1}$
The transformation gives $j = 0, 1, 2, 3, \ldots m-1$, where $m = \log n$, and the loop is transformed to:

```
for (j = 0; j < m; j++) F(u);
```

For example, if $F(u)$ costs $T(u) = u$ operations, then in terms of the new variable the cost $T(j) = u = 2^j$. In this case, the number of operations done by the loop will be:

$$\boldsymbol{T(n) = \sum_{j=0}^{m-1} T(j) = \sum_{j=0}^{m-1} 2^j = 2^m - 1 = n - 1}$$

However, if $F(u)$ costs $T(u)$ = constant number of operations (B), then in terms of the new variable $T(j) = B$. In this case, the number of operations done by the loop will be:

$$\boldsymbol{T(n) = \sum_{j=0}^{m-1} T(j) = \sum_{j=0}^{m-1} B = mB = B \log n}$$

Example (2):
The loop variable is decremented by dividing by $C = 2$:

```
for (t = n; t > 1; t /= 2 ) F(t)
```

Using the transformation $i = \log t \leftrightarrow t = 2^i$, then for $t = n, n/2, n/4, \ldots 2$, the corresponding new variable will be $i = m, m-1, m-2, \ldots 1$. The loop is then transformed to:

```
for (i = m; i ≥1; i--) F(t);
```

For example, if $F(t)$ costs $T(t) = t$ operations, then in terms of the new variable the cost $T(i) = t = 2^i$. In this case, the number of operations done by the loop will be:

$$T(n) = \sum_{i=m}^{i=1} T(i) = \sum_{i=1}^{m} 2^i = 2^{m+1} - 2 = 2(n-1)$$

However, if *F(t)* costs *T(t)* = constant number of operations (*B*), then in terms of the new variable, *T(i) = B*. In this case, the number of operations done by the loop will be:

$$T(n) = \sum_{i=m}^{i=1} T(i) = \sum_{i=1}^{m} B = mB = B \log n$$

Exercise:

Consider computing the number of *double* arithmetic operations *T(n)* done by the following piece of code, assuming that $n = 2^m$:

```
for ( int t = n; t > 1; t /= 2 ){
        for ( int u = 1; u < n; u *= 2 ){
                for ( int v = 0; v < n; v += 2 ){
                    // constant number of double arithmetic operations (B)
                }
        }
}
```

Using the appropriate transformation of loop variables, show that:
$T(n) = (B/2)\, n\, (\log n)^2$

3.3 Examples on Modeling Some Practical Algorithms

Here, we present four examples of modeling some practical algorithms leading to the computation of the number of operations and the complexity of each. In the last two examples, we show that taking advantage of the nature of the problem can lead to a significant reduction of the algorithm complexity.

3.3.1 Uniqueness Test

Consider a simple algorithm to check whether all the elements in a given array are distinct. The input is an array *A[0…n-1]* of elements and the algorithm returns "true" if all the elements in *A* are distinct and "false" otherwise.

```
ALGORITHM UNIQUE_ELEMENTS (A[ 0.. n-1])
for i = 0 to n-2 do
        for j = i+1 to n-1 do
                if (A[i] == A[j]) return false;
return true;
```

The problem size is *n* and we take *T(n)* = number of comparisons. In the best case, the duplicate of *A[0]* is *A[1]* and one comparison is made so that $T(n) = \Omega(1)$. In the worst case, all elements are distinct and we have

$$T(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}(n-1-i)=\sum_{i=1}^{n-1}i=n(n-1)/2=O(n^2)$$

Hence, the above algorithm has quadratic complexity.

3.3.2 Insertion Sort

The general idea of the insertion sort is that for each element we find the slot where it belongs. The algorithm performs successive scans through the data. When an element is out of sequence (less than its predecessor), it is pulled out and then inserted where it should belong. Array elements have to be shifted to the right to make space for the insertion. A pseudocode for the insertion sort function is given as follows:

```
ALGORITHM InsertSort (A[0..n-1 ])
{    for i =1 to n-1 do
     {
          j = pointer to element Ai;
          v = copy of Ai;
          while( j > 0 && Aj-1 > v)
               { Move data right: Aj ← Aj-1
                 Move pointer left: j-- }
          Insert v at the last (j) location: Aj ← v;
      }
}
```

For this algorithm, we consider two types of operations, array data shifts and array elements comparisons. In the best case, the array is already sorted in ascending order. The while loop will do one comparison and will not iterate so that the number of data shifts is zero. Hence, in the best case,

$T_{comp}(n) = n-1 = \Omega(n)$ *and* $T_{shift}(n) = 0$.

In the worst case, the input array is sorted in descending order and the while loop iterates (i) times with one comparison and one shift each time. Therefore,

$$T_{comp}(n)=T_{shift}(n)=\sum_{i=1}^{n-1}i\left(1_{comp}\right)=\sum_{i=1}^{n-1}i=\frac{n(n-1)}{2}$$

$$\textbf{Hence } T(n)=T_{comp}(n)+T_{shift}(n)=(n^2-n)=O(n^2)$$

Therefore, the total complexity of insertion sort is linear in the best case and quadratic in the worst case.

3.3.3 Cosine Function Evaluation

The cosine of an angle x (radians) has the infinite expansion:

$$\cos(x) = \sum_{i=0}^{\infty} \frac{(-x^2)^i}{(2i)!}$$

A truncated series expansion can be used to obtain a good approximation to *cos(x)* in the form:

$$\cos(x) = 1 + \sum_{i=1}^{n} \frac{(-x^2)^i}{(2i)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... \frac{x^{2n}}{(2n)!}$$

A straightforward function to evaluate *cos(x , n)* using up to the n^{th} term is given below.

```
ALGORITHM cosine1 (double x, int n)
{
   double y = -x * x;  double sum = 1.0;
   for (int i = 1; i <= n; i++)
         sum = sum + pow (y,i) / fact (2*i);
   return sum;
}
```

To compute *T(n)* = number of arithmetic operations, we consider here the simple functions *pow(y,i)* to compute y^i and *fact (2*i)* to compute *(2i)!*. The function call *pow(y,i)* will do *(i-1)* multiplications and *fact(k)* uses *k* multiplications so that the call *fact(2*i)* will do *(2i+1)* multiplications. Then, inside the loop, we have to do *(i-1) + (2i+1) + 2 = 3i + 2* arithmetic operations. Notice that there is also one multiplication outside the loop. In this case,

$$T(n) = 1 + \sum_{i=1}^{n} (3i + 2) = 1 + (3/2)n(n+1) + 2n = 1 + 3.5n + 1.5n^2 = \Theta(n^2)$$

Therefore, the algorithm given above is quadratic in *n*. Such complexity is high for some applications with extensive computation of *cos(x)*. The question now is can we do better by taking into consideration the nature of the series?

Examining the above algorithm, we can readily see that the cause of the quadratic behavior is that computing the i^{th} term costs *O(i)* operations. If we can do it in constant time, the complexity would become *O(n)*, i.e. linear. The way to do this is to compute the i^{th} term from the $(i-1)^{th}$ term in constant time. Let the i^{th} term be $S_i = y^i / (2i)!$ and hence $S_{i-1} = y^{i-1} / (2(i-1))!$ so that, $S_i = y / [(2i-1)(2i)]\ S_{i-1}$ *for* $i > 0$ *with* $S_0 = 1$. The modified cosine function will be:

```
ALGORITHM cosine2 (double x, int n)
{
   double y = -x * x;  double S = 1.0; double sum = S ;
   for (int i = 1; i <= n; i++)
         int m = 2 * i ;
         S = y /((m-1) * m) * S;
         sum = sum + S;
   return sum;
}
```

In the above modified version of the algorithm, the loop body now costs only a constant number of operations (6 arithmetic operations) so that $T(n) = 1 + 6n = \Theta(n)$, i.e. linear complexity.

Exercise:
Implement an $O(n)$ algorithm to compute the exponential function e^x using the truncated series:

$$e^x = \sum_{i=0}^{n} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \frac{x^n}{n!}$$

3.3.4 The Maximum Subsequence Sum

Given a sequence of integers (possibly negative), $\{a_1, a_2, ..., a_n\}$, find a start index (i) and an end index (j), with $j \geq i$ that would maximize the sum $S_{max} = \sum_{k=i}^{j} a_k$. We consider the sum to be zero if all integers are negative. As an example, for the sequence -3, 10, -1, 7, -2, -5, we find the indices to be $i = 2$ and $j = 4$ with $S_{max} = 16$.

As an example of a practical application of this problem, we may consider a time series of daily stock market gains (positive values) and losses (negative values) over say 5 years. We want to mark the period in time over which the sum is maximum.

As a start, we consider the following straightforward algorithm:

```
ALGORITHM MAXSUBSUM1 (a [1..n])
Smax = 0;  im = 0;  jm = 0;
for i = 1 to n do
        for j = i to n do
                Sij = 0;
                for k = i to j do
                        Sij = Sij + a[k];
                if (Sij > Smax) { Smax = Sij;  im = i;   jm = j;}
return Smax, im, jm;
```

To analyze this algorithm, let $T(n)$ = number of arithmetic operations. In this case, we encounter one arithmetic operation (addition) inside the k-loop so that:

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 = \sum_{i=1}^{n} \sum_{j=i}^{n} (j - i + 1) = (1/2) \sum_{i=1}^{n} i(i+1)$$

With some algebra, $T(n) = (1/6)(n^3 + 3n^2 + 2n) = O(n^3)$

Therefore, the above algorithm has cubic complexity.

A faster version of the algorithm can be obtained when we realize that the sum *Sij* need not be calculated from the start every time, rather, we can accumulate it as the end index (*j*) increases:

```
ALGORITHM MAXSUBSUM2 (a [1..n])
Smax = 0;  im = 0;  jm = 0;
for i = 1 to n do
        Sij = 0;
        for j = i to n do
                Sij = Sij + a[j];
                if (Sij > Smax) { Smax = Sij;  im = i;   jm = j;}
return Smax, im, jm;
```

The above modification has removed the need for the innermost loop (the *k*-loop) and therefore:

$$\boldsymbol{T(n)=\sum_{i=1}^{n}\sum_{j=i}^{n}1=\sum_{i=1}^{n}(n-i+1)=\sum_{i=1}^{n}i=n(n+1)/2=O(n^2)}$$

We can now see that this small modification has reduced the complexity of the algorithm from cubic to quadratic. However, can we do better?

We can proceed by the observation that any negative subsequence cannot be the prefix of the maximum subsequence desired. This means that when *Sij* is negative, we can jump to a new start *i = j + 1* and proceed from there. The algorithm implementing this modification is:

```
ALGORITHM MAXSUBSUM3 (a [1..n])
i = 1; Smax = 0;  im = 0;  jm = 0; Sij =0;
for j = 1 to n do
        Sij = Sij + a[j];
                if (Sij > Smax) { Smax = Sij;  im = i;   jm = j; }
                        else if (Sij < 0) { i = j + 1;  Sij =0; }
return Smax, im, jm;
```

In this modification of the algorithm, we need only the end index loop (the j-loop). In the worst case, we find that:

$$\boldsymbol{T(n)=\sum_{j=1}^{n}2=2n=O(n)}$$

We can see now that taking advantage of the nature of the problem has reduced the complexity of the algorithm to become linear.

Exercises

1. Assume the processing time of an algorithm of *Big-O* complexity *O(f(n))* be directly proportional to *f(n).* Let three such algorithms *A, B,* and *C* have time complexity *O(n), O(n log n),* and $O(n^2)$, respectively. During a test, each algorithm spends 8 seconds to process 2^7 data items. Derive the time each algorithm should spend to process 2^{14} items.

2. Given two arrays *A* and *B* of *n* integers both of which are sorted in ascending order. Write an algorithm to check whether or not *A* and *B* have an element in common. Find the worst case number of array element comparisons done by this algorithm as a function of *n* and its *Big-O* complexity.

3. The discrete Fourier Transform of an image *f(x,y)* of size *N x N* pixels is computed as
$$F(u,v) = (1/N^2) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)\ exp\{-2\pi j (ux + vy)/N$$
where $u = 0, 1, \ldots, N-1$, and $v = 0, 1, \ldots, N-1$, and $j = sqrt(-1)$.
If *f(x,y)* is generally complex, and the evaluation of *exp{…}* costs one complex multiplication, how many complex multiplications are needed to compute the Fourier Transform of the whole image ?

4. The sine of an angle x (in radians) can be computed using the n-term expansion:
$sin(x) = \Sigma_{1 \le i \le n}\ (-1)^{i+1}\ x^{2i-1} / (2i-1)!$
Write an *O(n)* algorithm that uses the above expansion to compute *sin(x)* for arbitrary values of $n \ge 1$, with *x* of type float.

5. The exponential function e^x can be evaluated by the finite approximation:
$$e^x = f(x,n) = \sum_{i=0}^{n} x^i / i! = 1 + x + x^2/2! + \ldots + x^n/n!$$
Design an ***O(n)*** algorithm to compute the above function.

6. Suppose a sequence *A* of *n* integers should already be in ascending order. Thus for the sequence (6,2,9,5,8,7) there are 6 pairs that are out of sequence:
(6,2) , (6,5) , (9,5) , (9,8) , (9,7) , (8,7)
Implement an algorithm to return the number of these out-of-order pairs.
Determine *T(n)* = number of array element comparisons.
What is the *Big-O* of this algorithm?

7. A floating point array *P* contains the historical stock market prices for a given stock for days *0 .. n-1*. Design an algorithm to find days *i* and *j (i < j)* such that if you buy on day (*i*) and sell on day (*j*), you get the maximum possible profit. Find the number of floating point comparisons and arithmetic operations done by your algorithm in terms of *n*. Also, find the *Big-O* complexity of the algorithm.

Appendix (A)

Useful Mathematical Relations

Functional Relations

- $n^x = 2^{x \log n}$ • $(x^y)^z = x^{yz}$ • $x^y x^z = x^{y+z}$ • $\log(xy) = \log x + \log y$
- $2^{\log n} = n$ • $\log(n^x) = x \log n$ • $\log_b n = \log_a n / \log_a b$
- Stirling's Theorem: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large n
- $\log n! = c_1 + c_2 \log n + n \log n - c_3 n = O(n \log n)$
- $\sum_{i=1}^{n} \log i = \log n! = O(n \log n)$
- $\sum_{i=1}^{n} i \log i = \sum_{i=1}^{n} \log i^i \le n \log n^n = O(n^2 \log n)$

Finite Series

- **Arithmetic Series:** $\sum_{i=0}^{n-1} (a + b\,i) = (n/2)\{2a + (n-1)b\}$

  $\sum_{i=0}^{n-1}(n-i) = \sum_{i=1}^{n} i = n(n+1)/2$

- **Geometric Series:** $\sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x-1)$ for $x \ne 1$

  $\sum_{i=0}^{n-1} 2^i = 2^n - 1 \qquad \sum_{i=1}^{n-1} 2^i = 2^n - 2 \qquad \sum_{i=1}^{n} i\,2^{i-1} = (n-1)\,2^n + 1$

  $\sum_{i=0}^{n-1} i\,2^{n-i} = \sum_{i=1}^{n} (n-i)2^i = 2^{n+1} - 2(n+1)$

Sums of Powers of Natural Numbers

- $\sum_{i=1}^{n} i = n(n+1)/2$
- $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$
- $\sum_{i=1}^{n} i^3 = [n(n+1)/2]^2$

- $\sum_{i=1}^{n} i^4 = n(n+1)(2n+1)(3n^2 + 3n -1) / 30$
- $\sum_{i=1}^{n} i^5 = n^2 (n+1)^2 (2 n^2 +2n -1) / 12$

Sums of Odd Numbers and their Powers

- $\sum_{i=1}^{n} (2 i – 1) = n^2$
- $\sum_{i=1}^{n} (2 i – 1)^2 = n(4n^2 – 1)/3$
- $\sum_{i=1}^{n} (2 i – 1)^3 = n^2 (2n^2 – 1)$

Other Finite Series

- $\sum_{i=1}^{n} i (i + 1)^2 = (1/12) n(n+1)(n+2)(3n + 5)$
- $\sum_{i=1}^{n} i . i! = (n+1)! – 1$
- $\sum_{i=2}^{n} 1 /(i^2 -1) = (3/4) – (2n+1)/[2n(n+1)]$
- $\sum_{i=1}^{n} 1/ i \sim \gamma + \ln n + 1/(2n)$ for large n, $\gamma = 0.577$ = Euler’s constant
- $\sum_{i=1}^{n} \frac{1}{i\,(i+1)} = \frac{n}{n+1}$

Infinite Series and Products

- $\sum_{k=0}^{\infty} a^{kx} = 1 / (1 – a^x)$ for ($a > 1$ and $x < 0$) or ($0 < a < 1$ and $x > 0$)
- $\sum_{k=0}^{\infty} x^k / k! = e^x$
- $\sum_{k=0}^{\infty} x^k = 1 / (1-x)$ for $|x| < 1$

- $\sum_{k=0}^{\infty}$ (a + bk) x^k = a /(1-x) + b x /$(1 - x^2)$ for |x| < 1
- $\sum_{k=1}^{\infty}$ $(-1)^{k+1}$ / k = ln 2
- $\sum_{k=1}^{\infty}$ $(-1)^{k+1}$ / k^2 = π^2 /12
- $\sum_{k=1}^{\infty}$ 1 / $(2k - 1)^2$ = π^2 / 8
- $\sum_{k=1}^{\infty}$ 1 / (k 2^k) = ln 2
- $\sum_{k=1}^{\infty}$ 1 / (k^2 2^k) = π^2 /12 - $(\ln 2)^2$ /2
- $\sum_{k=0}^{\infty} \frac{1}{k!} = e = 2.71828$
- $\sum_{k=0}^{\infty} \frac{(-1)^k}{k!} = 1/e = 0.36787$
- $\sum_{k=1}^{\infty}$ k /(k+1)! = 1
- $\prod_{k=2}^{\infty}\left(1-\frac{1}{k^2}\right)=\frac{1}{2}$
- $\prod_{k=0}^{\infty}(1+x^{2^k})=\frac{1}{1-x}$ for |x| < 1

General Solution of 1st Order Linear Recurrences

$$T(n) = a_n\, T(n-1) + b_n \;, \quad \text{given } T(0) \text{ or } T(1)$$

Solution:

Successive substitution gives for the cases of T(0) given and T(1) given:

$$T(n)=T(0)\prod_{i=1}^{n} a_i + \sum_{i=1}^{n-1} b_i \prod_{j=i+1}^{n} a_j + b_n$$

$$T(n)=T(1)\prod_{i=2}^{n} a_i + \sum_{i=2}^{n-1} b_i \prod_{j=i+1}^{n} a_j + b_n$$

General D&Q Recurrence

$$T(n) = aT(n/c) + f(n)$$

With $m = \log_c n$, ***solutions are*** :

(1) $T(n) = T(1)a^m + \sum_{i=0}^{m-1} a^i f(n/c^i)$ ***for*** $n > 1$ ***with*** $T(1)$ ***given***

(2) $T(n) = T(c)a^{m-1} + \sum_{i=0}^{m-2} a^i f(n/c^i)$ ***for*** $n > c$ ***with*** $T(c)$ ***given***

For $f(n) = bn^x$, ***solutions are*** :

(3) $T(n) = a^m T(1) + bn^x \sum_{i=0}^{m-1} (\frac{a}{c^x})^i$ for T(1) given.

(4) $T(n) = a^{m-1} T(c) + bn^x \sum_{i=0}^{m-2} (\frac{a}{c^x})^i$ for T(c) given

When $a = c^x$, ***solutions are*** :

(5) $T(n) = T(1)a^m + bn^x \log_c n$ ***for*** $n > 1$ ***with*** $T(1)$ ***given***

(6) $T(n) = T(c)a^{m-1} + bn^x (\log_c n - 1)$ ***for*** $n > c$ ***with*** $T(c)$ ***given***