3. Algorithms

We will study algorithms to solve many different types of problems such as

- finding the largest of a sequence of numbers
- sorting a sequence of numbers
- finding the shortest path between two given points
- finding gcd (greatest common divisor) of two integers

3. Algorithms

We will study algorithms to solve many different types of problems such as

- finding the largest of a sequence of numbers
- sorting a sequence of numbers
- finding the shortest path between two given points
- finding gcd (greatest common divisor) of two integers

What matters?

- How fast do we solve the problem?
- How much computer resource do we need?

3.1 Algorithm

Definition:

An *algorithm* is a finite set of precise step by step instructions for performing a computation or for solving a problem.

3.1 Algorithm

Definition:

An *algorithm* is a finite set of precise step by step instructions for performing a computation or for solving a problem.

e.g. Find the largest number in a finite sequence of integers:

23,56,67,43,32,42,56,33,65,58,12,26, -56,23,56,43,62,59.

e.g. Find the largest number in a finite sequence of integers:

23,56,67,43,32,42,56,33,65,58,12,26, -56,23,56,43,62,59.

Algorithm:

1. Set a temporary maximum, `tmax`, equal to the first integer in the sequence.
2. Compare next integer in the sequence; if it is larger than `tmax`, set `tmax` equal to this integer.
3. Repeat step 2 until reaching the end of sequence.
4. Stop; `tmax` is the largest of integer sequence.

C (or Java) Code:

```
int max(int a[], int n)
{
    int tmax;

    for (i=1, tmax = a[0]; i<n; i++){
        if (tmax < a[i]){
            tmax = a[i];
        }
    }
    return tmax;
}
```

Properties that algorithms generally share:

1. Input	
2. Output	
3. **Definiteness**:	Each step should be defined precisely
4. **Correctness**:	Should produce correct output for input values
5. **Finiteness**:	Should produce desired output after a finite number of steps
6. **Effectiveness**:	Each step should be performed exactly in a finite amount of time
7. **Generality**:	Should be applicable to any case

Searching Algorithms:

- Locate x in a sequence a_0, a_1, ..., a_{n-1} or determine x is not in the sequence.
- Return index i or -1 if not found.

Searching Algorithms:

- Locate x in a sequence $a_0, a_1, \ldots, a_{n-1}$ or determine x is not in the sequence.
- Return index i or -1 if not found.

Linear Search Algorithm:

1. Set index to 0, $i = 0$
2. Compare x and a_i,
3. If $x \neq a_i$, i++ and go to step 2
   stop whenever $x = a_i$ or $i = n$

C (or Java) Code:

```
int linear_search(int a[],int x,int n)
{
    int i, location;
    i = 0;

    while (i<n && a[i]!=x)
        i = i+1;

    if (i<n) location = i;
    else location = -1;

    return location;
}
```

Pseudocode:

```
function linear_search(int a[ ],int x,int n)
{
    i = 0

    while (i<n and a[i]≠ x)
        i = i+1

    if (i<n)
        location = i
    else
        location = -1

    return location
}
```

Remark: Pseudocode is a simplified way of writing algorithms in computer language. Your textbook uses its own way of writing pseudocodes. You can also use it. The above pseudocode is different than that and looks more like a C or Java code.

Binary Search Algorithm:

Consider a search problem s.t. the sequence is an ordered list (say increasing).

e.g.

Search 15 in 1, 3, 5, 6, 9, 11, 15, 19

Binary Search Algorithm:

Consider a search problem s.t. the sequence is an ordered list (say increasing).

e.g.

Search 15 in 1, 3, 5, 6, 9, 11, 15, 19

split into two:

1, 3, 5, 6 9, 11, 15, 19

check

$15 \leq 6$ NO

Binary Search Algorithm:

Consider a search problem s.t. the sequence is an ordered list (say increasing).

e.g.

Search 15 in 1, 3, 5, 6, 9, 11, 15, 19

split into two:

1, 3, 5, 6 9, 11, 15, 19

check

$15 \leq 6$ NO

split upper block into two:

9, 11 15, 19

check

$15 \leq 11$ NO

split upper block into two:

15 19

$15 \leq 15$ YES

check

$15 \overset{?}{=} 15$ YES

Search x in the list $a_0, a_1, \ldots, a_{n-1}$ where $a_0 < a_1 < \ldots < a_{n-1}$.

```
1. Compare $x$ with the middle term of the sequence, $a_m$, where $m = \lfloor (n-1) / 2 \rfloor$.
2. If $x > a_m$, search $x$ on the second half $\{a_{m+1}, a_{m+2}, \ldots a_n\}$
   else
      search $x$ on the first half $\{a_1, a_2, \ldots a_m\}$
3. Repeat the first two steps until a list with one single term is obtained.
4. Determine whether this one term is $x$ or not.
```

C (or Java) Code:

```
int binary_search(int x, int a[], int n)
{
        int i, j, location;
        int m;
        i = 0;
        j = n-1;

        while (i < j) {
            m = (i + j) / 2;
            if (x > a[m]) i = m+1;
            else j = m;
        }

        if (x == a[i])   location = i;
        else   location = -1;

        return location;
}
```

3.2 & 3.3 The Growth of Functions & Complexity of Algorithms

We will now address the *algorithm complexity* issue.

Examples:

- Which algorithm is more efficient, binary search or linear search?
- Consider sorting n integers:
  How long will it take to sort? (Bubble sort or insertion sort; see your textbook)

We can time how long it takes a computer

- But what if the computer is doing other things?
- And what happens if you get a faster computer?
  - A 3 Ghz Windows machine chip will run an algorithm at a different speed than a 3 Ghz Mac

We can time how long it takes a computer

- But what if the computer is doing other things?
- And what happens if you get a faster computer?
  - A 3 Ghz Windows machine chip will run an algorithm at a different speed than a 3 Ghz Mac

We need more platform independent measures!

We can time how long it takes a computer

- But what if the computer is doing other things?
- And what happens if you get a faster computer?
  - A 3 Ghz Windows machine chip will run an algorithm at a different speed than a 3 Ghz Mac

We need more platform independent measures!

An efficient algorithm on a slow computer will *always* beat an inefficient algorithm on a fast computer (given sufficiently large inputs!)

A better way is to count basic computer operations involved,

- A comparison, an assignment, an addition, etc.
- Regardless of how many machine instructions it translates into (different CPUs will require different amount of machine instructions for the same algorithm)

A better way is to count basic computer operations involved,

- A comparison, an assignment, an addition, etc.
- Regardless of how many machine instructions it translates into (different CPUs will require different amount of machine instructions for the same algorithm)

We might come up with something like:

# of operations: $f(n) = 100n\log(n) + 25n + 9$

A better way is to count basic computer operations involved,

- A comparison, an assignment, an addition, etc.
- Regardless of how many machine instructions it translates into (different CPUs will require different amount of machine instructions for the same algorithm)

We might come up with something like:

# of operations: $f(n) = 100n\log(n) + 25n + 9$

$\longrightarrow$

We will say:

$f(n)$ is $O(n\log(n))$.

Worst-case computational complexity analysis

Linear search:

1 initial assignment operation

5 operations (2 comparisons, 1 addition, 1 assignment, and 1 logical) per iteration:

5n operations inside the loop

1 additional comparison if x is not in the list

1 comparison and 1 assignment outside the loop

Totally $5n + 4$ operations in the worst case.

```
int linear_search(int a[],int x,int n){
    int i, location;
    i = 0;

    while (i<n && a[i]!=x)
        i = i+1;

    if (i<n) location = i;
    else location = -1;

    return location;
}
```

The linear search takes *n* “steps”

Let’s say the linear search takes the following number of instructions on specific CPUs:

- Intel Pentium IV CPU: 58**n* /2
- Motorola CPU: 84.4*(*n* + 1)/2
- Intel Pentium V CPU: 44**n*/2

Notice that each has an “*n*” term

- As *n* increases, the other terms will drop out
- As processors change, the constants will always change
- The exponent on *n* (if there is any) will not

```
int binary_search(int x, int a[], int n)
{
       int i, j, location;
       int m;
       i = 0;
       j = n-1;

       while (i < j) {
          m = (i + j) / 2;
          if (x > a[m]) i = m+1;
          else j = m;
       }
       if (x == a[i])   location = i;
       else   location = -1;

       return location;
}
```

5 operations (3 assignments, 1 subtraction, 1 comparison) outside the loop.

7 operations (2 additions, 1 division, 2 comparisons, 2 assignments) at each loop run.

```
int binary_search(int x, int a[], int n)
{
        int i, j, location;
        int m;
        i = 0;
        j = n-1;

        while (i < j) {
            m = (i + j) / 2;
            if (x > a[m]) i = m+1;
            else j = m;
        }
        if (x == a[i])   location = i;
        else   location = -1;

        return location;
}
```
5 operations (3 assignments, 1 subtraction, 1 comparison) outside the loop.
7 operations (2 additions, 1 division, 2 comparisons, 2 assignments) at each loop run.
1 comparison to terminate the loop

Binary search:

Let $f(n)$ be # of operations required for n-element list.

Let $n = 2^k$ (simplifying assumption), p: # of operations per loop ($p = 7$).

Then $f(n) = p + f(n/2)$

$= p + p + f(n/4)$

.

.

.

$= k\,p + f(n/2^k)$

need to stop at $n = 2^k$

$\therefore\ k = \log n$

$f(n) = p \log n + f(1)$

$= 7 \log n + 6$

Binary search:

Let $f(n)$ be # of operations required for n-element list.

Let $n = 2^k$ (simplifying assumption), p: # of operations per loop ($p = 7$).

Then
$$\begin{aligned} f(n) &= p + f(n/2) \\ &= p + p + f(n/4) \\ &\quad\vdots \\ &= k\,p + f(n/2^k) \end{aligned}$$

need to stop at $n = 2^k$

$\therefore\ k = \log n$

$$\begin{aligned} f(n) &= p \log n + f(1) \\ &= 7 \log n + 6 \end{aligned}$$

for $2^{k-1} < n < 2^k$, easy to see that $f(n) = 7\lceil \log n \rceil + 6$

Linear search *vs.* Binary search

Consider $f(n)$: # of operations (in the worst case)

Linear search: $f(n) = 5n + 4$

We will say $f(n)$ is $O(n)$; has linear complexity

Binary search: $f(n) = 7\log n + 6$

We will say $f(n)$ is $O(\log n)$; has logarithmic complexity

Linear search *vs.* Binary search

Consider $f(n)$: # of operations (in the worst case)

Linear search: $f(n) = 5n + 4$

We will say $f(n)$ is $O(n)$; has linear complexity

Binary search: $f(n) = 7\log n + 6$

We will say $f(n)$ is $O(\log n)$; has logarithmic complexity

Binary search is more efficient than linear search!

What actually matters: How many times does the loop execute in the worst case?

Linear search *vs.* Binary search

Consider $f(n)$: # of operations (in the worst case)

Linear search: $f(n) = 5n + 4$

We will say $f(n)$ is $O(n)$; has linear complexity

Binary search: $f(n) = 7\log n + 6$

We will say $f(n)$ is $O(\log n)$; has logarithmic complexity

Binary search is more efficient than linear search!

Note: Logarithms in computer science is almost always in base 2. But in the Big-Oh notation, the base of the logarithm does not matter. Why?

$\log_b(x) = \log_c(x) / \log_c(b)$

Big-O Notation

Let f and g be functions from the set of integer (or real) numbers to the set of real numbers.

We say:

$f(x)$ is O(g(x)) iff $\exists C, k$ real constants such that $|f(x)| \leq C\,|g(x)|\ \ \forall x > k$.

"$f(x)$ is big-oh of g(x)"

<u>**Big-O Notation**</u>

Let f and g be functions from the set of integer (or real) numbers to the set of real numbers.

We say:

$$f(x) \text{ is } O(g(x)) \text{ iff } \exists C, k \text{ real constants such that } |f(x)| \le C\,|g(x)| \quad \forall x > k.$$

"$f(x)$ is big-oh of $g(x)$"

Note: (C, k) pair is <u>never unique</u> if it exists.

Note: One can always find a <u>positive</u> (C, k) integer pair if a pair exists.

<u>Remark</u>: Big-O notation gives an idea about the growth of a function.

Example: Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

$?\exists C, k$ such that $|\, x^2 + 2x + 1 \,| \leq C\,|\, x^2| \quad \forall x > k$

<u>Example</u>: Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

$?\exists C,k$ such that $|\, x^2 + 2x + 1 \,| \leq C\,|\, x^2| \quad \forall x > k$

Let $x > 1$ $\quad \Rightarrow 0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$ since $x^2 > x$

$\Rightarrow |f(x)| \leq 4|x^2|$ whenever $x > 1 \;\Rightarrow\; (C = 4,\ k = 1)$

$\therefore\quad f(x)$ is $O(x^2)$.

<u>Example</u>: Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

$?\exists C,k$ such that $| x^2 + 2x + 1 | \leq C\, | x^2| \quad \forall x > k$

Let $x > 1$ $\quad \Rightarrow 0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$ since $x^2 > x$

$\Rightarrow |f(x)| \leq 4|x^2|$ whenever $x > 1 \;\Rightarrow\; (C = 4,\ k = 1)$

$\therefore \quad f(x)$ is $O(x^2)$.

Note that there are infinitely many other pairs that satify this inequality such as (C = 5, k =1), (C = 6, k =1), (C = 3, k =2), …

For example, let $x > 2$ $\quad \Rightarrow 0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$

$\Rightarrow \quad C = 3,\ k = 2$ also satisfy the inequality

$\therefore$ (C,k) is **not** unique if exists.

Example: Show that $f(x) = 7x^2$ is $O(x^3)$.

?$\exists C,k$ such that $|7x^2| \leq C\,|x^3|$ whenever $x > k$

<u>Example:</u> Show that $f(x) = 7x^2$ is $O(x^3)$.

$?\exists C,k$ such that $|7x^2| \leq C\,|x^3|$ whenever $x > k$

$\forall x > 7, \quad 0 < 7x^2 < x^3 \;\Rightarrow\; |7x^2| < |x^3|$

$\therefore \quad C = 1, k = 7$ and $7x^2$ is $O(x^3)$.

How about x^3? Is it $O(x^2)$?

$?\exists C,k \quad |x^3| \leq |C\,x^2|$ whenever $x > k$

How about x^3? Is it $O(x^2)$?

$?\exists C,k$ positive $\quad 0 < x^3 \leq C\,x^2 \quad$ whenever $x > k$

$\Rightarrow\ x \leq C \quad$ whenever $x > k$

No such (C, k) exists $\therefore$ x^3 is not $O(x^2)$

<u>Recall</u>: One can always find a <u>positive</u> (C, k) integer pair if a pair exists.

Growth of Polynomials

Theorem:

Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, where a_i are real numbers.

Then $f(x)$ is $O(x^n)$.

Growth of Polynomials

Theorem:

Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, where a_i are real numbers.

Then $f(x)$ is $O(x^n)$.

Proof: Use triangle inequality.

$$
\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0| \\
&\le |a_n x^n| + |a_{n-1} x^{n-1}| + \ldots + |a_1 x| + |a_0| \quad \text{(by triangle inequality)} \\
&\le |a_n| x^n + |a_{n-1}| x^{n-1} + \ldots + |a_1| x + |a_0| \\
&\le x^n (|a_n| + |a_{n-1}|/x + \ldots + |a_0|/x^n) \\
&\le x^n (|a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0|) \ \text{ when } x > 1
\end{aligned}
$$

$\Rightarrow C = |a_n| + |a_{n-1}| + \ldots + |a_0|, \quad k = 1$

$\therefore\ f(x)$ is $O(x^n)$

Complexity of an algorithm is usually expressed as a discrete function $f(n)$, where n is a positive integer $n > 1$. Some examples:

e.g. $f(n) = 1 + 4 + 9 + \ldots + n^2$

Give a big-O estimate for this complexity function?

e.g. $f(n) = 1 + 4 + 9 + \ldots + n^2$

$0 < 1 + 4 + 9 + \ldots + n^2 \leq n^2 + n^2 + \ldots + n^2 = n^3 \quad \forall n > 1$

$\Rightarrow C = 1, k = 1$

$\therefore$ $(1 + 4 + 9 + \ldots + n^2)$ is $O(n^3)$

e.g. $f(n) = 1 + 4 + 9 + \dots + n^2$

$0 < 1 + 4 + 9 + \dots + n^2 \leq n^2 + n^2 + \dots + n^2 = n^3 \quad \forall n > 1$

$\Rightarrow C = 1, k = 1$

$\therefore \quad (1 + 4 + 9 + \dots + n^2)$ is $O(n^3)$

Note: If $f(n)$ is non-negative, then you can get rid of the absolute signs in the big-O definition for the sake of simplicity (complexity functions are always non-negative valued).

e.g. $f(n) = n!$?

e.g. $f(n) = n!$?

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdots n \\ &\leq n \cdot n \cdot n \cdots n = n^n \qquad \forall n > 0 \end{aligned}$$

$\Rightarrow C = 1,\ k = 0$

$\therefore$ $n!$ is $O(n^n)$

e.g. $f(n) = \log n!$?

e.g. $f(n) = \log n!$?

$\log n! \leq n\log n \quad \forall n > 0$

$\Rightarrow C = 1, k = 0$

$\therefore$ $\log n!$ is $O(n\log n)$

e.g.

$f(n) = 3n\log(n!) + (n^3+3)\log n \quad n > 0$

big-O estimate?

The Growth Combinations of Functions:

Theorem:

Let $f_1(x)$ be $O(g_1(x))$ and $f_2(x)$ be $O(g_2(x))$.

Then, $f_1(x)f_2(x)$ is $O(g_1(x)g_2(x))$.

The Growth Combinations of Functions:

Theorem:

Let $f_1(x)$ be $O(g_1(x))$ and $f_2(x)$ be $O(g_2(x))$.

Then, $f_1(x)f_2(x)$ is $O(g_1(x)g_2(x))$.

Proof:

$\exists C_1, k_1$ such that $|f_1(x)| \leq C_1|g_1(x)|$ whenever $x > k_1$,

$\exists C_2, k_2$ such that $|f_2(x)| \leq C_2|g_2(x)|$ whenever $x > k_2$,

$$\begin{aligned}\Rightarrow |f_1(x)f_2(x)| &\leq C_1\,|g_1(x)|\;C_2\,|g_2(x)| \quad \forall x > \max(k_1, k_2) \\ &\leq C_1C_2\,|g_1(x)g_2(x)|\end{aligned}$$

Choose $C = C_1C_2$ and $k = \max(k_1, k_2)$.

$\therefore$ $f_1(x)f_2(x)$ is $O(g_1(x)g_2(x))$.

Theorem:

Suppose $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.

Then $f_1(x) + f_2(x)$ is $O(\max(|g_1(x)|, |g_2(x)|))$.

Theorem:

Suppose $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.

Then $f_1(x) + f_2(x)$ is $O(\max(|g_1(x)|, |g_2(x)|))$.

Proof:

Suppose $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ with C_1, C_2, k_1, k_2

$\exists C_1, k_1$ such that $|f_1(x)| \le C_1|g_1(x)|$ whenever $x > k_1$,

$\exists C_2, k_2$ such that $|f_2(x)| \le C_2|g_2(x)|$ whenever $x > k_2$,

$$\begin{aligned}\Rightarrow |\, f_1(x)+f_2(x) \,| &\le |f_1(x)| + |f_2(x)| < C_1\, |g_1(x)| + C_2\, |g_2(x)|, \quad \forall x > \max(k_1, k_2)\\ &< C_1\, |g(x)| + C_2\, |g(x)| = (C_1 + C_2)\, |g(x)|\end{aligned}$$

where $g(x) = \max(|g_1(x)|, |g_2(x)|)$.

Choose $C = C_1 + C_2$ and $k = \max(k_1, k_2)$

$\therefore\ f_1(x)+f_2(x)$ is $O(\max(|g_1(x)|, |g_2(x)|)\,)$.

Corollary:

Let $f_1(x)$, $f_2(x)$ be both $O(g(x))$, then $f_1(x) + f_2(x)$ is also $O(g(x))$.

e.g.

$f(n) = 3n\log(n!) + (n^3+3)\log n \quad n > 0$

big-O estimate?

$\log(n!)$	is $O(n\log n)$
$3n$	is $O(n)$

$\therefore\ 3n\log(n!)$ is $O(n^2\log n)$

(n^3+3)	is $O(n^3)$
$\log n$	is $O(\log n)$

$\therefore\ (n^3+3)\log n$ is $O(n^3\log n)$

$\therefore\ f(n)$ is $O(n^3\log n)$

Big-Omega Notation:

We say $f(x)$ is $\Omega(g(x))$

iff $\exists C,k$ positive constants s.t. $|f(x)| \geq C\,|g(x)|$ whenever $x > k$

Remark: $f(x)$ is $\Omega(g(x)) \leftrightarrow g(x)$ is $O(f(x))$

Big-Omega Notation:

We say $f(x)$ is $\Omega(g(x))$

iff $\exists C,k$ positive constants s.t. $|f(x)| \geq C\,|g(x)|$ whenever $x > k$

Remark: $f(x)$ is $\Omega(g(x)) \leftrightarrow g(x)$ is $O(f(x))$

Big-Theta Notation:

We say $f(x)$ is $\Theta(g(x))$ (or $f(x)$ is of order $g(x)$)

iff $\exists C_1, C_2, k$ positive constants s.t. $C_1\,|g(x)| \leq |f(x)| \leq C_2\,|g(x)|$ whenever $x > k$

Remark: $f(x)$ is $\Theta(g(x)) \quad \leftrightarrow \quad f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.

Example: Show that $3x^2 + 8x \log x$ is $\Theta(x^2)$.

Example: Show that $3x^2 + 8x \log x$ is $\Theta(x^2)$.

For this we have to show that

i. $3x^2 + 8x \log x$ is $O(x^2)$

ii. x^2 is $O(3x^2 + 8x \log x)$ (or equivalently $3x^2 + 8x \log x$ is $\Omega(x^2)$)

Example: Show that $3x^2 + 8x \log x$ is $\Theta(x^2)$.

For this we have to show that

- **i.** $3x^2 + 8x \log x$ is $O(x^2)$
- **ii.** x^2 is $O(3x^2 + 8x \log x)$ (or equivalently $3x^2 + 8x \log x$ is $\Omega(x^2)$)

i. $\forall x \geq 1,\ 0 \leq 8x \log x \leq 8x^2$

$\Rightarrow \forall x > 1\ |3x^2 + 8x \log x| \leq 11|x^2| \Rightarrow$ Choose $C = 11$, $k = 1$.

$\therefore 3x^2 + 8x \log x$ is $O(x^2)$

Example: Show that $3x^2 + 8x \log x$ is $\Theta(x^2)$.

For this we have to show that

i. $3x^2 + 8x \log x$ is $O(x^2)$

ii. x^2 is $O(3x^2 + 8x \log x)$ (or equivalently $3x^2 + 8x \log x$ is $\Omega(x^2)$)

i. $\forall x \geq 1,\ 0 \leq 8x \log x \leq 8x^2$

$\Rightarrow \forall x > 1\ |3x^2 + 8x \log x| \leq 11|x^2| \Rightarrow$ Choose $C = 11$, $k = 1$.

$\therefore 3x^2 + 8x \log x$ is $O(x^2)$

ii. $\forall x > 1\ |x^2| \leq |3x^2 + 8x \log x| \Rightarrow$ Choose $C = 1$, $k = 1$.

$\therefore x^2$ is $O(3x^2 + 8x \log x)$

(i) and (ii) $\Rightarrow 3x^2 + 8x \log x$ is $\Theta(x^2)$

e.g. Show that $f(n) = 1+2+\ldots+n$ is $\Omega(n^2)$

$1+2+\ldots+n = n(n+1)/2= n^2/2+n/2$

This is equivalent to show that n^2 is $O(n^2/2+n/2)$

$\forall n>1 \quad n^2 \leq 2\ (n^2/2+n/2)$ hence choose $C = 2$, $k = 1$

$\therefore\ f(n)$ is $\Omega(n^2)$

Note that $f(n)$ is non-negative, so you can get rid of the absolute signs in the big-O definition.

e.g. Show that 1+2+…+n is $\Theta(n^2)$

$\forall n>1 \quad 1+2+\ldots+n \quad \leq n + n + \ldots + n = n^2$

$\therefore\ f(n)$ is $O(n^2)$ with C=1, k=1

Together with the above exercise, $f(n)$ is $\Theta(n^2)$.

Worst-case computational complexity analysis

Examples:

Linear search:

Totally $5n+4$ operations in the worst case:

Linear search algorithm has complexity $O(n)$.

Finding maximum in n-element list:

Totally $2(n-1) + 1 = 2n - 1$ *comparison* operations in the worst case;

has complexity $O(n)$.

Binary search:

Totally $(p\log n + r)$ operations in the worst case:

has complexity $O(\log n)$.

Worst-case computational complexity analysis

Examples:

Linear search:
Totally $5n+4$ operations in the worst case:
Linear search algorithm has complexity $O(n)$.

Finding maximum in n-element list:
Totally $2(n–1) + 1 = 2n – 1$ comparison operations in the worst case;
has complexity $O(n)$.

Binary search:
Totally $(p\log n + r)$ operations in the worst case:
has complexity $O(\log n)$.

Note that p and r are some constant factors and do not affect complexity.
What actually matters is: How many times does the loop execute in the worst case?

Average-case computational complexity analysis

Linear search:

Average # of *comparison* operations, **assuming that** x is in the list:

$$\frac{3+5+7+\cdots+(2n+1)}{n}:$$

```
int linear_search(int a[],int x,int n)
{
    int i, location;
    i = 0;

    while (i<n && a[i]!=x)
        i = i+1;

    if (i<n) location = i;
        else location = -1;
    return location;
}
```

Average-case computational complexity analysis

Linear search:

Average # of *comparison* operations, **assuming that** x is in the list:

$$\frac{3+5+7+\cdots+(2n+1)}{n}=\frac{2(1+2+3+\cdots+n)+n}{n}=\frac{2\left[n(n+1)/2\right]}{n}+1=n+2$$

$\therefore$ The average complexity of linear search is O(n).

```
int linear_search(int a[],int x,int n)
{
    int i, location;
    i = 0;

    while (i<n && a[i]!=x)
        i = i+1;

    if (i<n) location = i;
        else location = -1;
    return location;
}
```

Terminology:

O(1)	constant complexity
O(log n)	logarithmic complexity
O(n)	linear complexity
O($n$log n)	$n$log n complexity
O(n^b)	polynomial complexity
O(b^n), $b > 1$	exponential complexity
O(n!)	factorial complexity

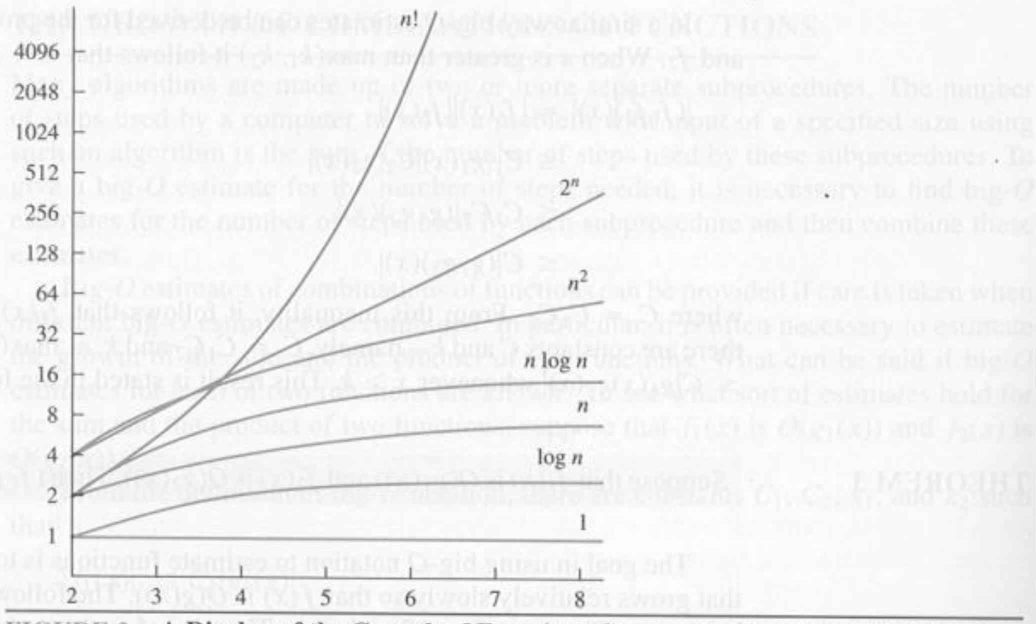

FIGURE 3 A Display of the Growth of Functions Commonly Used in Big-*O* Estimates.

Terminology:

O(1)	constant complexity
O(log n)	logarithmic complexity
O(n)	linear complexity
O($n$log n)	$n$log n complexity
O(n^b)	polynomial complexity
O(b^n), $b > 1$	exponential complexity
O($n!$)	factorial complexity

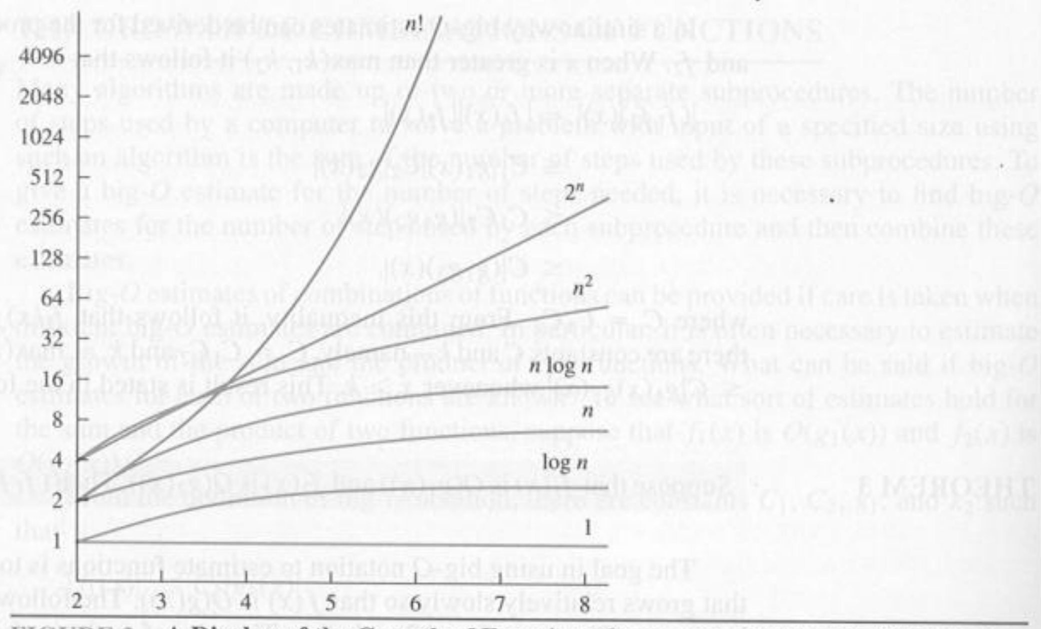

FIGURE 3 A Display of the Growth of Functions Commonly Used in Big-*O* Estimates.

If a problem is solvable with polynomial worst-case complexity, it is called *tractable*.
If a problem cannot be solved with polynomial worst-case complexity, it is *intractable*.

Problems that no algorithm exists for solving them are called *unsolvable* problems (e.g., The Halting Problem).

Complexity	# Steps (for input size $n = 1000$)	
$O(1)$	1	
$O(\log n)$	≈ 10	
$O(n)$	10^3	
$O(n \log n)$	$\approx 10^4$	
$O(n^2)$	10^6	
$O(n^3)$	10^9	
$O(n^4)$	10^{12}	
$O(n^c)$	10^{3*c}	c is a consant
2^n	$\approx 10^{301}$	
$n!$	$\approx 10^{2568}$	
n^n	10^{3000}	

Complexity Classes (A one-slide introduction)

P Class: Problems that can be solved in polynomial-time (tractable problems)

NP Class: Non-deterministic polynomial-time class problems. There are many solvable problems that are believed no algorithm with O(n^b) polynomial complexity solves them. But given a candidate solution, it can be checked in polynomial time. These are called NP class problems. Many optimization problems are in this class. Many people believe (not proven yet) that P $\neq$ NP, which means there are NP problems that are not solvable with any polynomial-time algorithm. Note that P $\subseteq$ NP.

NP–Complete Class:
Class of NP problems such that if any of these problems can be solved with polynomial O(n^b) complexity then all NP problems can be solved with O(n^b). It is accepted, though not proven yet, that no NP–complete problem can be solved in O(n^b).

Note that NP class may not contain all problems (e.g., ones for which even the time to verify the solution is not polynomial, such as some NP-hard problems).

Remark: To fully understand these concepts, you need to learn *Turing machines*.