Chapter 3
Algorithm Analysis

Algorithm analysis

- Studies computing resource requirements of different algorithms
- Computing Resources
  - Running time (Most precious)
  - Memory usage
  - Communication bandwidth etc
- Goal of algorithm analysis
  - Given alternative algorithms/solutions/ for a problem, pick up the one that is efficient

Reasons to perform algorithm Analysis

- Writing a working program is not good enough
  - The program may be inefficient, especially on a large data set!
- It enables us to:
  - Predict performance of algorithms
  - Compare algorithms.
  - Provide guarantees on running time/space of algorithms
  - Understand theoretical basis.
- Primary practical reason: **avoid performance bugs.**
  - client gets poor performance because programmer did not understand performance characteristics

How to Measure Efficiency/performance?

- Two approaches to measure algorithms efficiency/performance
  - Empirical
    - Implement the algorithms and
    - Trying them on different instances of input
    - Use/plot actual clock time to pick one
  - Theoretical
    - Determine quantity of resource (Execution time, memory space, etc.) required mathematically needed by each algorithms

Example- Empirical

Input size

N	time (seconds) †
250	0.0
500	0.0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

Actual clock time

Drawbacks of empirical methods

- It is difficult to use actual clock. Because clock time varies based on
  - Specific processor speed
  - Current processor load
  - Specific data for a particular run of the program
    - Input size
    - Input properties
  - Programming language (C++, java, python …)
  - The programmer (You, Me, Billgate …)
  - Operating environment/platform (PC, sun, smartphone etc)
- Therefore, it is quite machine dependent

Theoretical Approach:

- Efficiency of an algorithm is measured in terms of the ***number of basic operations*** it performs to process the input data.
  - Not based on actual clock time
  - Hence, it is a Machine independent analysis

- Factors affecting running time:
  - **System dependent effects.**
    - **Hardware**: CPU, memory, cache, …
    - **Software**: compiler, interpreter, garbage collector, …
    - **System**: operating system, network, other apps, …
  - **System independent effects**
    - **Algorithm.**
    - **Input data/ Problem size**

Cont..

- For most algorithms, running time depends on **"size"** of the input.
  - Size is often the number of inputs processed
  - Example:-
    - In sorting problem, size is the no of items to be sorted
- Running time is expressed as $T(n)$ for some function T on input size n.

Theoretical Approach - Complexity Analysis

- Complexity Analysis is the systematic study of **the cost of computation**, measured either in **time units** or in **operations performed**, or in the amount of **storage space** required.
  - ***The goal is to have a meaningful measure that permits comparison of algorithms independent of operating platform.***

- There are two things to consider:
  - **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n.
  - **Space Complexity**: Determine the approximate memory required to solve a problem of size n.

Theoretical Approach - Complexity Analysis

- Complexity analysis involves **two distinct phases**:
  1. **Algorithm Analysis**: Analysis of the algorithm or data structure to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.
  2. **Order of Magnitude Analysis**: Analysis of the function T (n) to determine the general complexity category to which it belongs.

Algorithm Analysis Rules:

- **OPTION 1 : Exact count**
  - There is no generally accepted set of rules for algorithm analysis. However, an exact count of operations is commonly used.

1. We assume that every basic operation takes constant time(Arbitrary time)

- **Examples of Basic Operations:**
  - Single Arithmetic Operation (Addition, Subtraction, Multiplication)
  - Assignment Operation
  - Single Input/Output Operation
  - Single Boolean Operation
  - Function Return
- Examples of Non-basic Operations are
  - Sorting, Searching.

Algorithm Analysis Rules:

2. **selection statement (if, switch)**:Running time is:

   the time for the **condition evaluation** + **the maximum of the running times for the individual clauses** in the selection.

3. **Loops**: Running time for a loop is equal to:

   the **running time for the statements inside the loop** * **number of iterations.**

Algorithm Analysis Rules:

4. **Nested loops**, the total running time of a statement inside a **group of nested loops** is (analyze inside out).

   the running time of the statements inside the loop multiplied by the product of the sizes of all the loops.

- Always assume, the loop executes the maximum number of iterations possible.

5. Running time of a function call is

   1 for setup + **the time for any parameter calculations** + the time required for **the execution of the function body**.

Complexity Analysis: Loops

- We start by considering how to count operations in **for**-loops.
  - We use integer division throughout.
- First of all, we should know the number of iterations of the loop; say it is **x**.
  - Then the loop condition is executed **x + 1 times**.
  - Each of the statements in the loop body is executed **x times**.
  - The loop-index update statement is executed **x times**.

Complexity Analysis: Loops (with <)

- In the following for-loops:

```
for(int i=k; i<n; i+=m)
{
        statement1;
        statement2;
 }
```

```
for(int i=n; i>k; i-=m)
{
        statement1;
        statement2;
 }
```

The number of iterations is: **(n – k) / m**

- The initialization statement, **i = k,** is executed **one** time.
- The condition, **i < n**, is executed **(n – k) / m + 1** times.
- The update statement, **i = i + m**, is executed **(n – k) / m** times.
- Each of **statement1** and **statement2** is executed **(n – k) / m** times.

Complexity Analysis : Loops (with <=)

- In the following for-loop:

```
for(int i=k; i<=n; i+=m)
{
        statement1;
        statement2;
 }
```

```
for(int i=n; i>=k; i-=m)
{
        statement1;
        statement2;
 }
```

The number of iterations is: **(n – k) / m + 1**

- The initialization statement, **i = k**, is executed **one** time.
- The condition, **i <= n**, is executed **(n – k) / m + 2** times.
- The update statement, **i = i + m**, is executed **(n – k) / m + 1** times.
- Each of **statement1** and **statement2** is executed **(n – k) / m + 1** times.

Complexity Analysis: Loops With Logarithmic Iterations

- In the following for-loop: (with <)

```
for (int i=k; i<n; i*=m){
        statement1;
        statement2;
 }
```

```
for (int i=n; i>k; i/=m){
        statement1;
        statement2;
 }
```

  - The number of iterations is: $\lceil (\mathbf{Log_m} \, (\mathbf{n} / \mathbf{k})) \rceil$

- In the following for-loop: (with <=)

```
for (int i=k; i<=n; i*=m){
        statement1;
        statement2;
 }
```

```
for (int i=n; i>=k; i/=m){
        statement1;
        statement2;
 }
```

  - The number of iterations is: $\lfloor (\mathbf{Log_m} \, (\mathbf{n} / \mathbf{k}) + \mathbf{1}) \rfloor$

Examples: Count of Basic Operations T(n)

- Sample Code

```
int count()
{
int k=0;
cout<< “Enter an integer”;
cin>>n;
for (i = 0;i < n;i++)
  k = k+1;
return 0;
}
```

Examples: Count of Basic Operations T(n)

- Sample Code

```
int count(int n)
{
int k=0;
cout<< “Enter an integer”;
cin>>n;
for (i = 0;i < n;i++)
   k = k+1;
return 0;
}
```

- Count of Basic Operations (Time Units)

- 1 for the assignment statement: int k=0
- 1 for the output statement.
- 1 for the input statement.
- In the for loop:
  - 1 assignment, n+1tests, and n increments.
  - n loops of 2 units for an assignment, and an addition.
- 1 for the return statement.
- **T (n) = 1+1+1+(1+n+1+n)+2n+1 = 4n+6**

Examples: Count of Basic Operations T(n)

```
int total(int n)
{
int sum=0;
for (int i=1;i<=n;i++)
     sum=sum+i;
return sum;
}
```

Examples: Count of Basic Operations T(n)

- Sample Code

```
int total(int n)
{
int sum=0;
for (inti=1;i<=n;i++)
      sum=sum+i;
return sum;
}
```

- Count of Basic Operations (Time Units)

- **1 for the assignment statement: int sum=0**
- In the for loop:
  - 1 assignment, n+1tests, and n increments.
  - n loops of 2 units (an assignment and an addition.)
- 1 for the return statement.
- **T (n) = 1+ (1+n+1+n)+2n+1 = 4n+4**

Examples: Count of Basic Operations T(n)

```
void func()
{
 int x=0;
 int i=0;
 int j=1;
 cout<< "Enter an Integer value";
 cin>>n;
 while (i<n){
        x++;
        i++;
 }
 while (j<n)
 {
     j++;
 }
}
```

Examples: Count of Basic Operations T(n)

Sample Code

```
void func()
{
 int x=0;
 int i=0;
 int j=1;
 cout<< "Enter an Integer value";
 cin>>n;
 while (i<n){
          x++;
          i++;
 }
 while (j<n)
 {
   j++;
 }
}
```

- Count of Basic Operations (Time Units)

  - 1 for the first assignment statement: x=0;
  - 1 for the second assignment statement: i=0;
  - 1 for the third assignment statement: j=1;
  - 1 for the output statement.
  - 1 for the input statement.
  - In the first while loop:
    - n+1tests
    - n loops of 2 units for the two increments (addition)
  - In the second while loop:
    - n tests
    - n-1 increments
  - **T (n) = 1+1+1+1+1+n+1+2n+n+n-1 = 5n+5**

Examples: Count of Basic Operations T(n)

- Sample Code

```
int sum (int n)
{
int partial_sum= 0;
for (int i = 1; i <= n; i++)
  partial_sum= partial_sum+ (i * i * i);
return partial_sum;
}
```

Examples: Count of Basic Operations T(n)

- Sample code

```
int sum (int n)
{
int partial_sum= 0;
for (int i = 1; i <= n; i++)
partial_sum= partial_sum+ (i * i * i);
return partial_sum;
}
```

- Count of Basic Operations (Time Units)

- 1 for the assignment
- 1 assignment, n+1tests, and n increments.
- n loops of 4 units for an assignment, an addition, and two multiplications.
- 1 for the return statement.

- $T(n) = 1+(1+n+1+n)+4n+1 = 6n+4$

Exercise

What is the Count of Basic Operations T(n) for the following code ?

```
i =0
While (i < n) {
    j = 0
    While (j < 3*n)
        j = j + 1
    j = 0
    While( j < 2*n)
        j = j + 1
    i = i + 1
}
```

Option 2: Formal Approach to Algorithm Analysis

Formal Approach to Algorithm Analysis

- Instead of exact count, formal approaches **uses simplified rules** to count the basic operations of an algorithm
- Formal approach focuses on the part of the algorithm that have the huge impact on the algorithms' behavior.
- Basically, the formal approach ignores ***initializations, loop control, and book keeping*** since their impact on the algorithms' behavior is less significant.

For loops: formally

- In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
```

$$\sum_{i=1}^{N} 1 = N$$

- Suppose we count ***the number of additions*** that are done. There is 1 addition per iteration of the loop, hence *N* additions in total.

Nested Loops: Formally

- Nested for loops translate into multiple summations, one for each for loop.

```
for (int i = 1; i <= N; i++) {
      for (int j = 1; j <= M; j++) {
            sum = sum+i+j;
      }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{M}2 = \sum_{i=1}^{N}2M = 2MN$$

- Again, ***count the number of additions***. The outer summation is for the outer for-loop.

Consecutive Statements: Formally

- Add the running times of the separate blocks of your code

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        sum = sum+i+j;
    }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

Conditionals: Formally

- If (test) **s1** else s2:
- Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
        sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) =$$

$$\max\left(N, 2N^2\right) = 2N^2$$

Example: Computation of Run-time from basic operations

- Suppose we have hardware capable of executing ***10^6 instructions per second***. How long would it take to execute an algorithm whose complexity function was ***$T(n) = 2n^2$*** on an input size of ***$n = 10^8$***?

The **total number of operations** to be performed would be **$T(10^8)$**:

$T(10^8) = 2*(10^8)^2 = 2*10^{16}$

The required number of seconds would be given by

$T(10^8)/10^6$ so:

Running time = $2*10^{16}/10^6 = 2*10^{10}$

The number of seconds per day is 86,400 so this is about 231,480 days (634 years).

2. Order of magnitude analysis

Complexity analysis

Remember, complexity analysis Involves two distinct phases

1. **Algorithm Analysis**: produces a function T (n) that describes the algorithm in terms of the operations performed in order to process input size n.
2. **Order of Magnitude Analysis**: Analysis of the function T (n) to determine the general complexity category to which it belongs.

Types of complexity analysis

- The ***worst-case runtime*** *complexity* of the algorithm is
  - The function defined by **the maximum number of steps** taken on any instance of size a.
  - Upper bound on cost.
  - Determined by “most difficult” input.
  - Provides a guarantee for all inputs.
- The ***best-case runtime*** *complexity* of the algorithm is
  - The function defined by **the minimum number of steps** taken on any instance of size a.
  - Determined by “easiest” input.
  - Provides a goal for all inputs.
  - Lower bound on cost.

Cont…

- The ***average case runtime*** *complexity* of the algorithm is
  - The function defined by **an average number of steps** taken on any instance of size a.
  - Need a model for “random” input.
  - Provides a way to predict performance.
- The ***amortized runtime*** *complexity* of the algorithm is
  - The function defined by a sequence of operations applied to the **input of size a and averaged over time**.

Example

- Let us consider an algorithm of sequential searching in an array of size n.

```
for(int i=0;i<n;i++)
{
    if(item==data[i])
    {
        cout<<"Found";
    }
}
```

- Its *worst-case runtime complexity* is O(n)
- Its *best-case runtime complexity* is O(1)
- Its *average case runtime complexity* is O(n/2)=O(n)

While average time seems to be the fairest measure, it may be difficult to determine.

Depends on distribution.

Assumption for above analysis: Equally likely at any position.

When is worst case time important?

algorithms for time-critical systems

Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)

  since the cost of the gold fish is insignificant when compared with cost of elephants

- Similarly, the low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

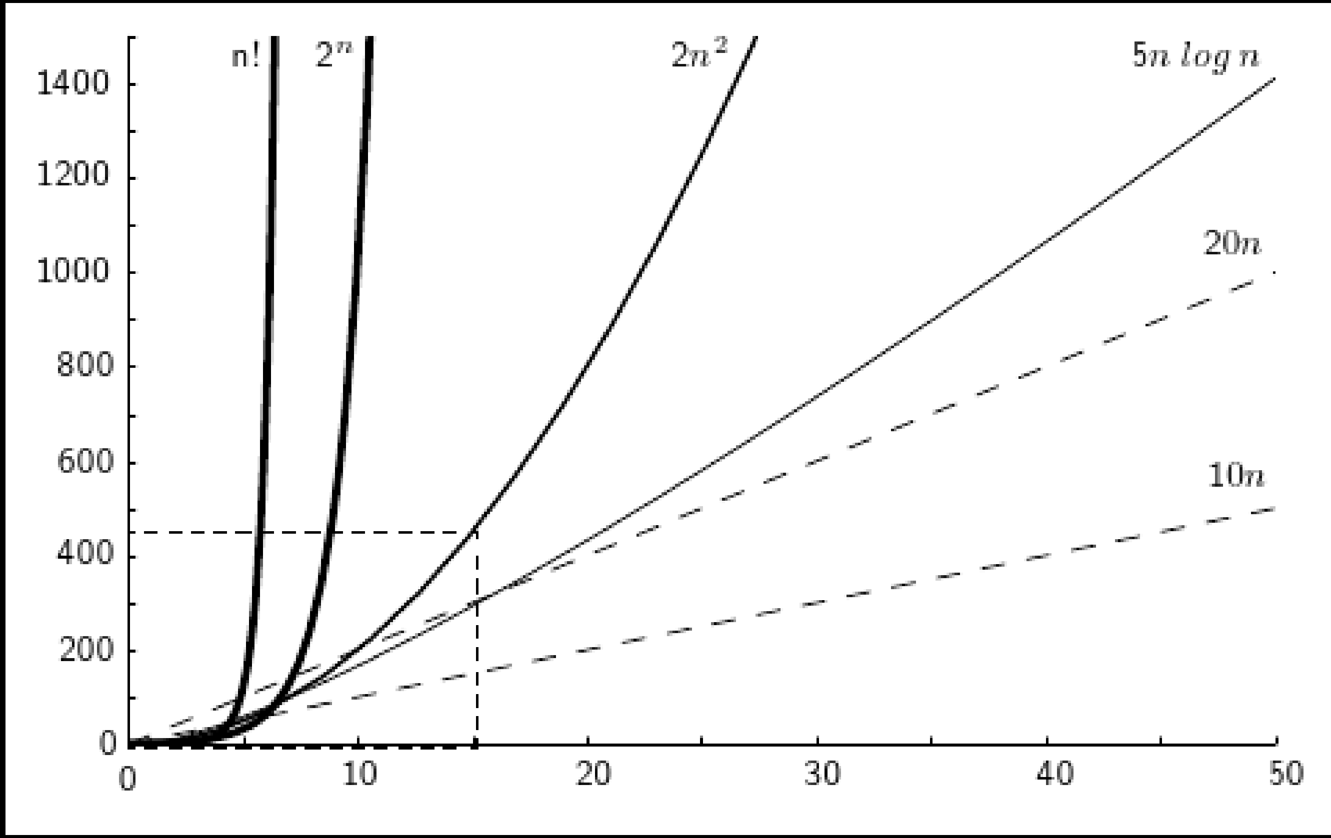
i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

More Examples: $f_B(n)=n^2+1 \sim n^2$

– $f_A(n)=30n+8 \sim n$

Growth rates

- The ***growth rate*** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.



Asymptotic analysis

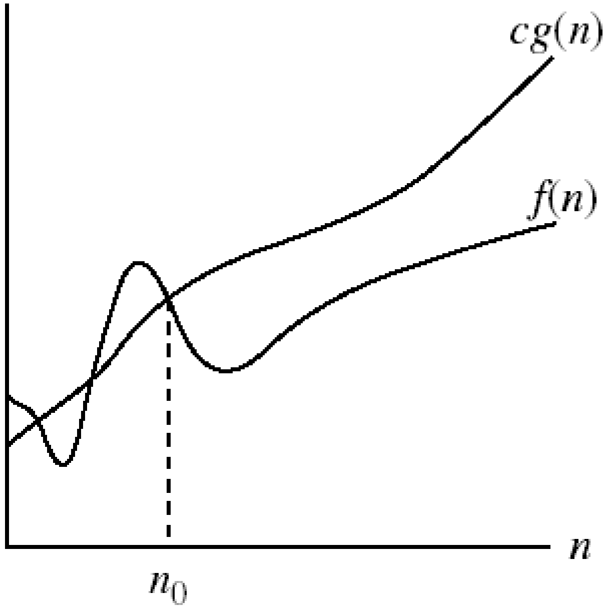
- Refers to the study of an algorithm as the input size "gets big" or reaches a limit.
- To compare two algorithms with running times *f(n)* and *g(n),* we need a **rough measure** that characterizes **how fast each function grows- growth rate.**
  - *Ignore constants [especially when input size very large]*
  - *But constants may have impact on small input size*
- Several notations are used to describe the running-time equation for an algorithm.
  - Big-Oh Notation (O) (<=)
  - Big-Omega Notation (Ω)(>=)
  - Theta Notation (Θ)(=)
  - Little-o Notation (o)(<)
  - Little-Omega Notation (ω)(>)

Big-Oh Notation

- Definition
  - For f(n) a non-negatively valued function, f(n) is in set O(g(n)) if there exist two positive constants c and n_o such that $f(n) \leq cg(n)$ for all $n > n_o$.
- Usage: The algorithm is in $O(n^2)$ in [best ,average, worst] case.
- Meaning: For all data sets big enough (i.e., n > no), the algorithm always executes in less than cg (n) steps [in best, average or worst case].

Big-Oh Notation - Visually

$O(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Proving Big-O

- Demonstrating that a function f(n) is in big-O of a function g(n) requires that we find specific constants **c** and **n_o** for which the inequality holds.

- The following points are facts that you can use for Big-Oh problems:
  - $1 <= n$ for all $n >= 1$
  - $n <= n^2$ for all $n >= 1$
  - $2^n <= n!$ for all $n >= 4$
  - $\log_2 n <= n$ for all $n >= 2$
  - $n <= n\log_2 n$ for all $n >= 2$

Examples

- $f(n) = 10n + 5$ and $g(n) = n$. Show that $f(n)$ is in $O(g(n))$.
  - To show that $f(n)$ is $O(g(n))$ we must show constants c and n_o such that

$$f(n) <= c.g(n) \text{ for all } n >= n_o$$

- $10n + 5 <= c.n$ for all $n >= n_o$

  $<= 10n+5n$ for all $n >= 1$

  $<= 15n$ for all $n >= 1$

  $<= cn$ for all $n>=1$

- Therefore:- $f(n)$ is in $O(g(n))$ for $c = 15$, and $n_o = 1$

Examples

- $2n^2 = O(n^3)$:

  $2n^2 \leq cn^3$

  Divide both sides by $n^2 \Rightarrow 2 \leq cn \Rightarrow c = 1$ and $n_0 = 2$

- $n^2 = O(n^2)$:

  $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

- $n = O(n^2)$:

  $n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

No Uniqueness

- $2n^2 = O(n^3)$:

$2n^2 \le cn^3$
Divide both sides by $n^2 \Rightarrow 2 \le cn \Rightarrow$ **$c = 1$** and **$n_0 = 2$**

- $2n^2 = O(n^3)$:

$2n^2 \le cn^3$
Divide both sides by $n^2 \Rightarrow 2 \le cn \Rightarrow$ **$c = 2$** and **$n_0 = 1$**

Big-O Theorems

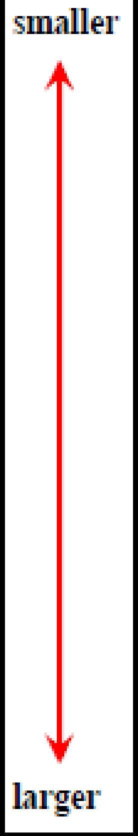
- For all the following theorems, assume that f(n) is a non-negative function of n and that K is an arbitrary constant.
- **Theorem 1:** K is O(1)
- **Theorem 2:** A polynomial is O(the term containing the highest power of n)
  - $f(n) = 7n^4 + 3n^2 + 5n + 1000$ is $O(7n^4)$
- **Theorem 3:** K*f(n) is O(f(n))
  - i.e., constant coefficients can be dropped
  - $g(n) = 7n^4$ is $O(n^4)$

Cont…

- **Theorem 4:** If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) is O(h(n)). **[transitivity]**
  - $f(n) = 6n^2 + 3n - 8$ is $O(6n^2) = O(g(n))$
  - $g(n) = 6n^2$ is $O(n^3) = O(h(n))$
  - $h(n) = n^3$
    - $f(n) = O(h(n))$
- **Theorem 5:** For any base b, $\log_b(n)$ is O(log(n)).
  - All logarithms grow at the same rate
  - $\log_b n$ is $O(\log_d n)$ where $b, d > 1$

Cont…

- **Theorem 6:** Each of the following functions is **strictly big-O of its successors:**
  - K [constant]
  - $\log_b(n)$ [always log base 2 if no base is shown]
  - n
  - $n \log_b(n)$
  - n^2
  - *n to higher powers*
  - 2^n
  - 3^n
    - larger constants to the n-th power
  - n! [n factorial]
  - n^n



Examples:

$f(n)= 3n\log_b n + 4 \log_b n+2$ is $O(n\log_b n)$ and (n^2) and $O(2^n)$

Cont…

- **Theorem 7:** In general, f(n) is big-O of the dominant term of f(n), where "dominant" may usually be determined from Theorem 6.
  - $f(n) = 7n^2 + 3n \log(n) + 5n + 1000$ is $O(n^2)$
  - $g(n) = 7n^4 + 3^n + 1000000$ is $O(3^n)$
  - $h(n) = 7n(n + \log(n))$ is $O(n^2)$

Important mathematics series formulas

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \qquad (1)$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} = \frac{n(2n+1)(n+1)}{6}. \qquad (2)$$

$$\sum_{i=1}^{\log n} n = n \log n. \qquad (3)$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ for } 0 < a < 1. \qquad (4)$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \text{ for } a \neq 1. \qquad (5)$$

As special cases to Equation 5,

$$\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}, \qquad (6)$$

and

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1. \qquad (7)$$

As a corollary to Equation 7,

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n+1} - 1 = 2n - 1. \qquad (8)$$

Finally,

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n}. \qquad (9)$$

Big-O Examples

The following run in constant time: **O(1)**

```
a = 1;
b = 2;
c = a + 5*b;
```

```
i := 0
while (i < 11 )
   i = i + 1
```

Big-O

What is the big-o of the following algorithms?

The following run in linear time: **O(n)**

```
i = 0;
While i < n Do
  i = i + 1
```

$f(n) = n$

$O(f(n)) =$ **O(n)**

```
i := 0
While i < n Do
  i = i + 3
```

$f(n) = n/3$

$O(f(n)) =$ **O(n)**

Big-O

What is the big-o of the following algorithms?

```
For (i = 0 ; i < n; i = i + 1)
    For (j = 0 ; j < n; j = j + 1)
```

- the above code run in quadratic time.
- $f(n) = n*n = n^2$, $O(f(n))$ = **$O(n^2)$**

Big-O

What is the big-o of the following algorithms?

```
For (i = 0 ; i < n; i = i + 1)
        For (j = i ; j < n; j = j + 1)
                //do-smtn
```

For a moment just focus on the second loop.
Since *i* goes from [0,n) the amount of looping
done is directly determined by what *i* is.
Remark that if ***i=0***, we do *n* work, if *i=1*, we do *n-1* work, if *i=2*, we do *n-2* work, etc…

So the question then becomes what is:
$(n) + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$?
Remarkably this turns out to be $n(n+1)/2$, so
$O(n(n+1)/2) = O(n^2/2 + n/2) = \mathbf{O(n^2)}$

Big-O

```
i := 0
While i < n Do
  j = 0
  While j < 3*n Do
    j = j + 1
  j = 0
  While j < 2*n Do
    j = j + 1
  i = i + 1
```

$f(n) = n * (3n + 2n) = 5n^2$

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

Big-O

- What is the big-O of the following problem?

Assume you want to move n items from one room to another room.

- **Operations** are Pick-up, forward moves, drops and reverse move.

Order of common functions

Notation	Name	Example
$O(1)$	Constant	Adding two numbers, c=a+b
$O(\log n)$	Logarithmic	Finding an item in a sorted array with a binary search or a search tree (best case)
$O(n)$	Linear	Finding an item in an unsorted list or a malformed tree (worst case);
$O(n \log n)$	Linearithmic	Performing a Fast Fourier transform; heap sort, quick sort (best case), or merge sort
$O(n^2)$	Quadratic	Multiplying two n-digit numbers by a simple algorithm; adding two n×n matrices; bubble sort (worst case or naive implementation), shell sort, quick sort (worst case), or insertion sort

Lower Bounds - Omega (Ω)

- We say that " *f*(*n*) is omega *g*(*n*)," which we write $f(n) = \Omega(g(n))$, if there exists an integer n_0 and a constant $C>0$ such that for all integers $n \geq n_0$, $f(n) \geq Cg(n)$.

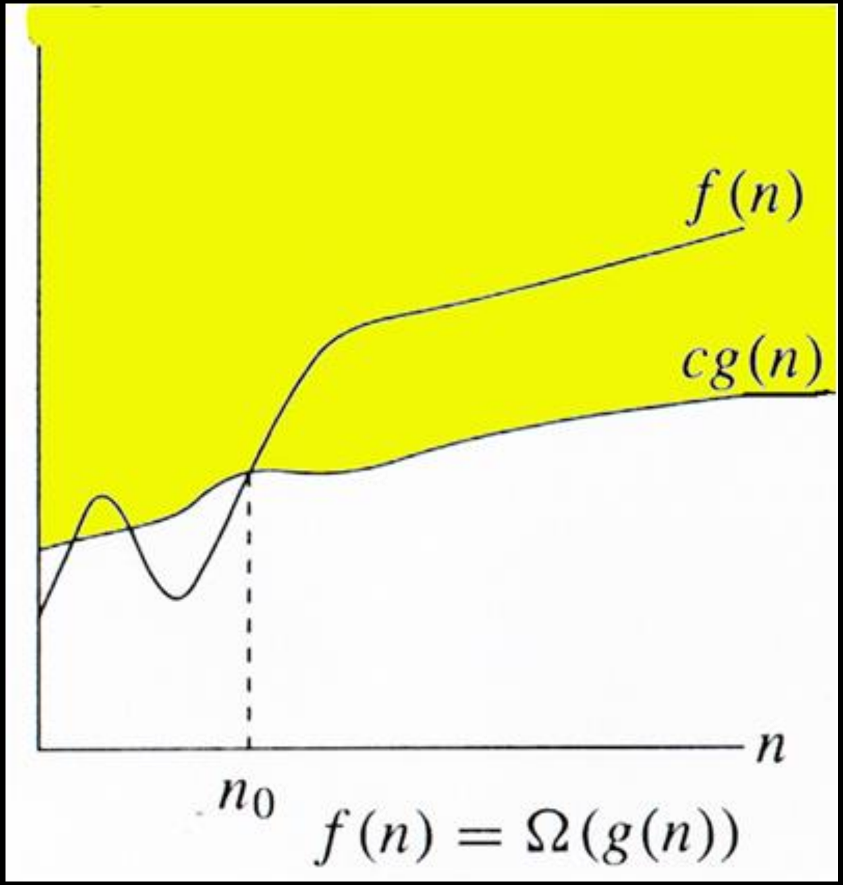
- The definition of omega is almost identical to that of Big O.

- The only difference is in the comparison—
  - for Big-O it is $f(n) \leq Cg(n)$;
  - for omega, it is $f(n) \geq Cg(n)$.

- All of the same conventions and caveats apply to omega as they do to Big O.

Omega (Ω)- Visually

- Just as Big O notation provides ***an asymptotic upper bound*** on a function, Ω notation provides ***an asymptotic lower bound.***



Example

- Consider the function $f(n)=5n^2-64n+256$.
- We wish to show that $f(n) = \Omega(n^2)$. Therefore, we need to find an integer n_0 and a constant $C>0$ such that for all integers $n \geq n_0$,

$$\mathbf{f(n) \geq C\, n^2.}$$

- Suppose we choose $C=1$. Then

$$f(n) \geq C\, n^2. \quad \Rightarrow 5n^2 - 64n + 256 \geq n^2$$

$$\Rightarrow 4n^2 - 64n + 256 \geq 0$$

$$\Rightarrow 4(n-8)^2 \geq 0$$

- Since $(n-8)^2 > 0$ for all values of $n \geq 0$, we conclude that $n_0 = 0$.
- So, we have that for $C=1$ and that $n_0 = 0$, $f(n) \geq C\, n^2$ for all integers $n \geq n_0$. Hence, $f(n) = \Omega(n^2)$. .

Theta Notation: Θ

- **Formal Definition**: A function f (n) is Θ (g(n)) if it is both *O(g(n))* and Ω *(g(n))*.
  - In other words, there exist constants c1, c2, and n_0 >0 such that c1.g (n)<=f(n)<=c2. g(n) for all n >= n_0
- If f(n)= Θ (g(n)), then g(n) is an asymptotically **tight bound** for f(n).
- In simple terms, f(n)= Θ (g(n)) means that f(n) and g(n) have **the same rate of growth.**

Theta Notation: Θ

Example:

1. If $f(n)=2n+1$, then $f(n) = \Theta(n)$

2. $f(n) = 2n^2$ then
   - $f(n)=O(n^4)$
   - $f(n)=O(n^3)$
   - $f(n)=O(n^2)$

- All these are technically correct, but the last expression is the best and tight one. Since $2n^2$ and n^2 have the same growth rate, it can be written as $f(n)= \Theta(n^2)$.

Little-o Notation

- Big-Oh notation may or may not be asymptotically tight, for example:
  - $2n^2 = O(n^2)$

    $=O(n^3)$
- $f(n)=o(g(n))$ means for all $c>0$ there exists some $n_0>0$ such that $f(n)< c.g(n)$ for all $n>=n_0$.
- Informally, $f(n)=o(g(n))$ means $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity.

Example: $f(n)=3n+4$ is $o(n^2)$

- In simple terms, $f(n)$ has less growth rate compared to $g(n)$.
- Function $g(n)= 2n^2$ is $g(n) =o(n^3)$ and $O(n^2)$. But $g(n)$ is not $o(n^2)$.

Little-Omega (ω notation)

- Little-omega (ω) notation is to big-omega (Ω) notation as little-o notation is to Big-Oh notation.
- We use ω notation to denote a lower bound that is not asymptotically tight.
- **Formal Definition**:
  - $f(n) = \omega(g(n))$ if there exists a constant $n_0 > 0$ such that $0 <= c.\ g(n) < f(n)$ for all $n >= n_0$.
- **Example**: $2n^2 = \omega(n)$ but it's not $\omega(n^2)$.

Next time !!

Ch4:Simple Searching and Sorting