Algorithm Analysis

Algorithm

- **An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:**

1. **Input: Zero or more quantities are externally supplied.**
2. **Output: At least one quantity is produced.**
3. **Definiteness: Each instruction is clear and unambiguous.**
4. **Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.**
5. **Effectiveness: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion3; it also must be feasible**

Design and Analysis of Algorithms

- ***Analysis:*** **predict the cost of an algorithm in terms of resources and performance**

- ***Design:*** **design algorithms which minimize the cost**

Algorithmic Performance

- **There are *two aspects* of algorithmic performance:**
- **Time**
    - **Instructions take time.**
    - **How fast does the algorithm perform?**
    - **What affects its runtime?**
- **Space**
    - **Data structures take space**
    - **What kind of data structures can be used?**
    - **How does choice of data structure affect the runtime?**

Why study algorithms and performance?

- **Algorithms help us to understand scalability.**
- **Performance often draws the line between what is feasible and what is impossible.**
- **Algorithmic mathematics provides a language for talking about program behavior.**
- **Performance is the currency of computing.**
- **The lessons of program performance generalize to other computing resources.**

The Execution Time of Algorithms

- **Each operation in an algorithm (or a program) has a cost.**
  - **Each operation takes a certain of time.**
- **`count = count + 1;` // take a certain amount of time, but it is constant**

- ***A sequence of operations:***

  `count = count + 1;` **Cost: c_1**

  `sum = sum + count;` **Cost: c_2**

  Total Cost = $c_1 + c_2$

The Execution Time of Algorithms

- ***Example: Simple If-Statement***

	Cost	Times
`if (n < 0)`	c1	1
`absval = -n`	c2	1
`else`		
`absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

The Execution Time of Algorithms

- ***Example: Simple Loop***

	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

- **Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5**
- **The time required for this algorithm is proportional to n**

The Execution Time of Algorithms

- ***Example: Nested Loop***

```
                                  Cost           Times
i=1;                               c1              1
sum = 0;                           c2              1
while (i <= n) {                   c3            n+1
      j=1;                         c4              n
      while (j <= n) {             c5        n*(n+1)
            sum = sum + i;         c6            n*n
            j = j + 1;             c7            n*n
     }
     i = i + 1;                    c8              n
}
```

- Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8
- The time required for this algorithm is proportional to n^2

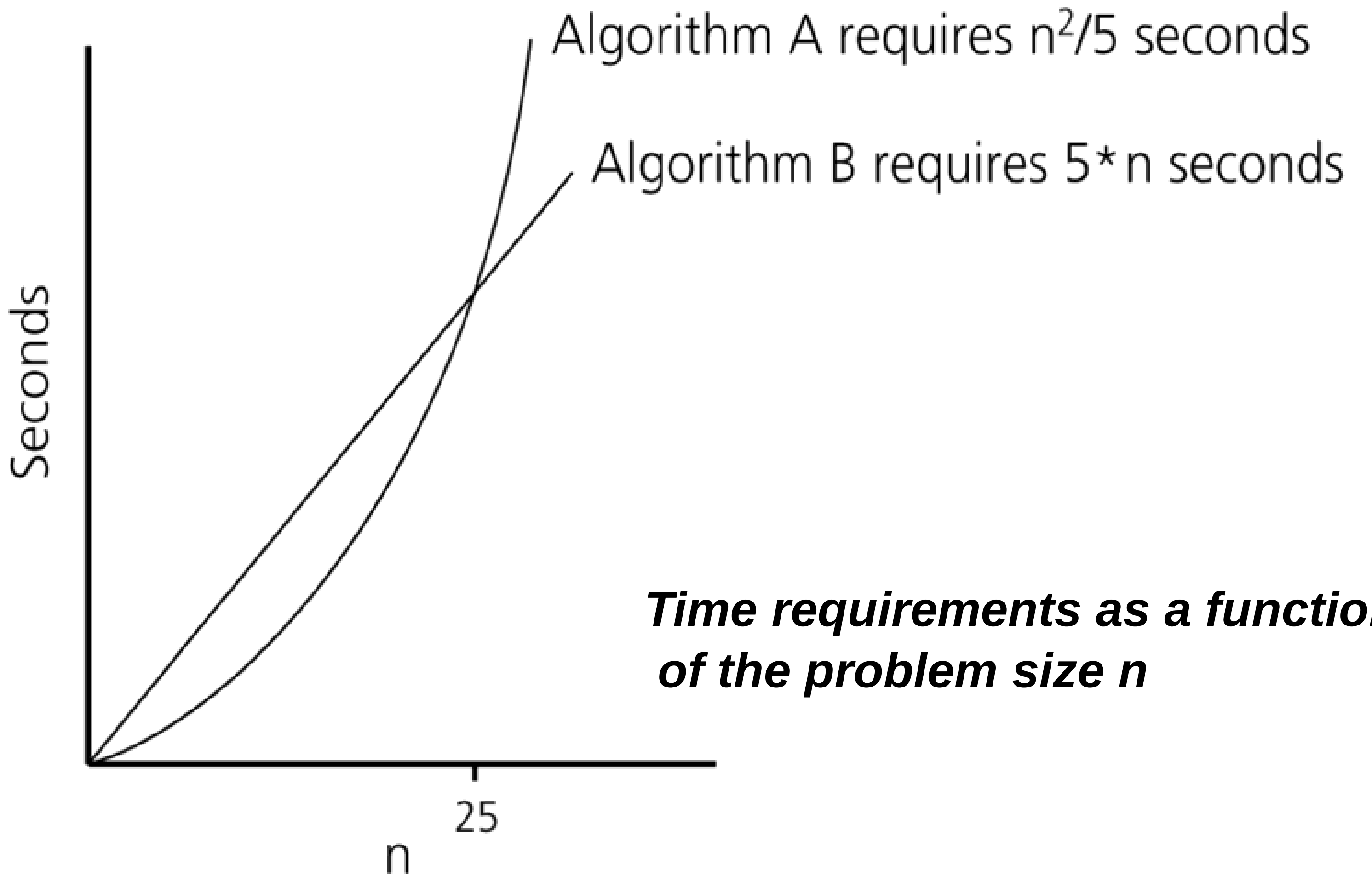
General Rules for Estimation

- **Loops**
  - **The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.**
- **Nested Loops**
  - **Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.**
- **Consecutive Statements**
  - **Just add the running times of those consecutive statements.**
- **If/Else**
  - **Never more than the running time of the test plus the larger of running times of S1 and S2.**

Algorithm Growth Rates

- **We measure an algorithm's time requirement as a function of the *problem size*.**
  - **Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number users for a social network search.**
- **So, for instance, we say that (if the problem size is n)**
  - **Algorithm A requires $5*n^2$ time units to solve a problem of size n.**
  - **Algorithm B requires 7*n time units to solve a problem of size n.**
- **The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.**
  - **Algorithm A requires time proportional to n^2.**
  - **Algorithm B requires time proportional to n.**
- **An algorithm's proportional time requirement is known as *growth rate*.**
- **We can compare the efficiency of two algorithms by comparing their growth rates.**

Algorithm Growth Rates

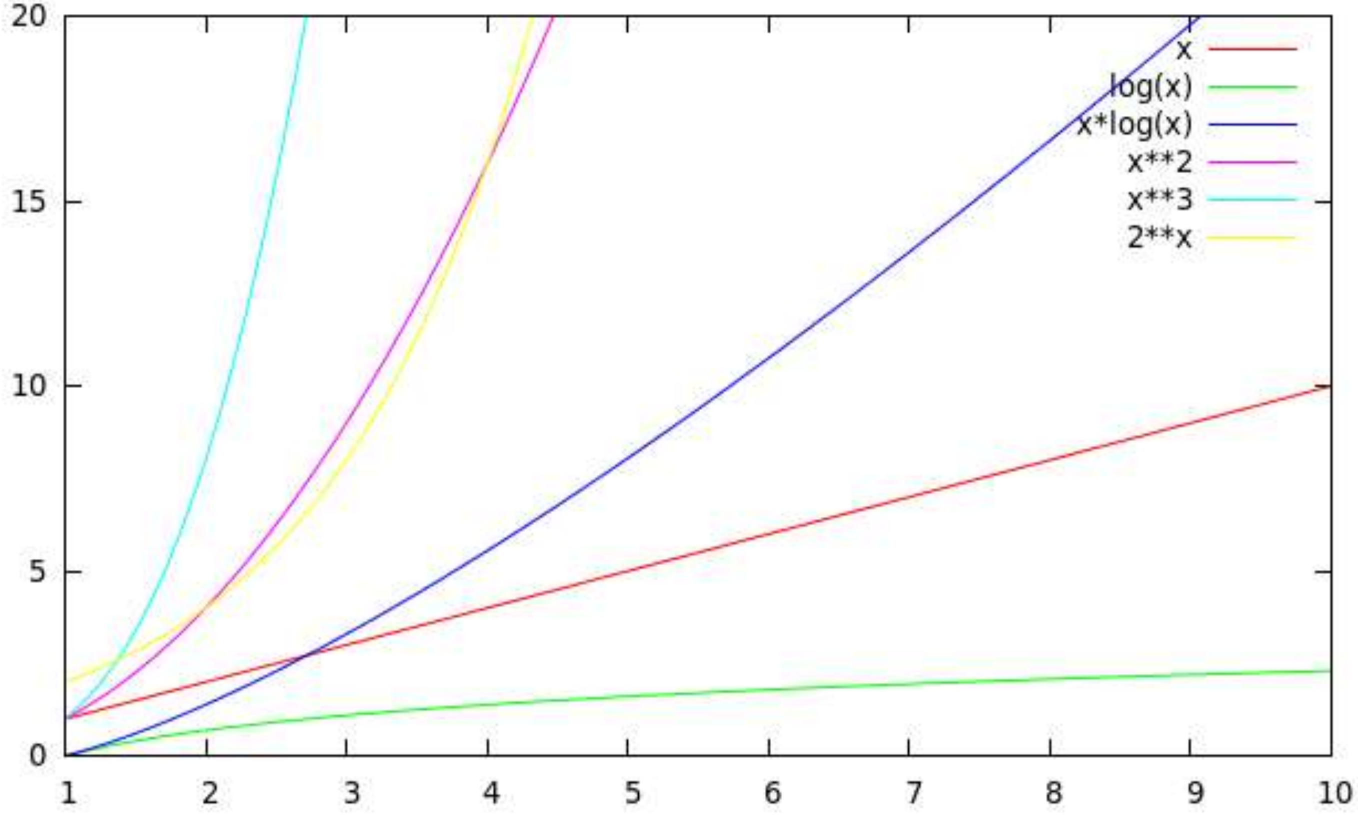


Time requirements as a function of the problem size n

Common Growth Rates

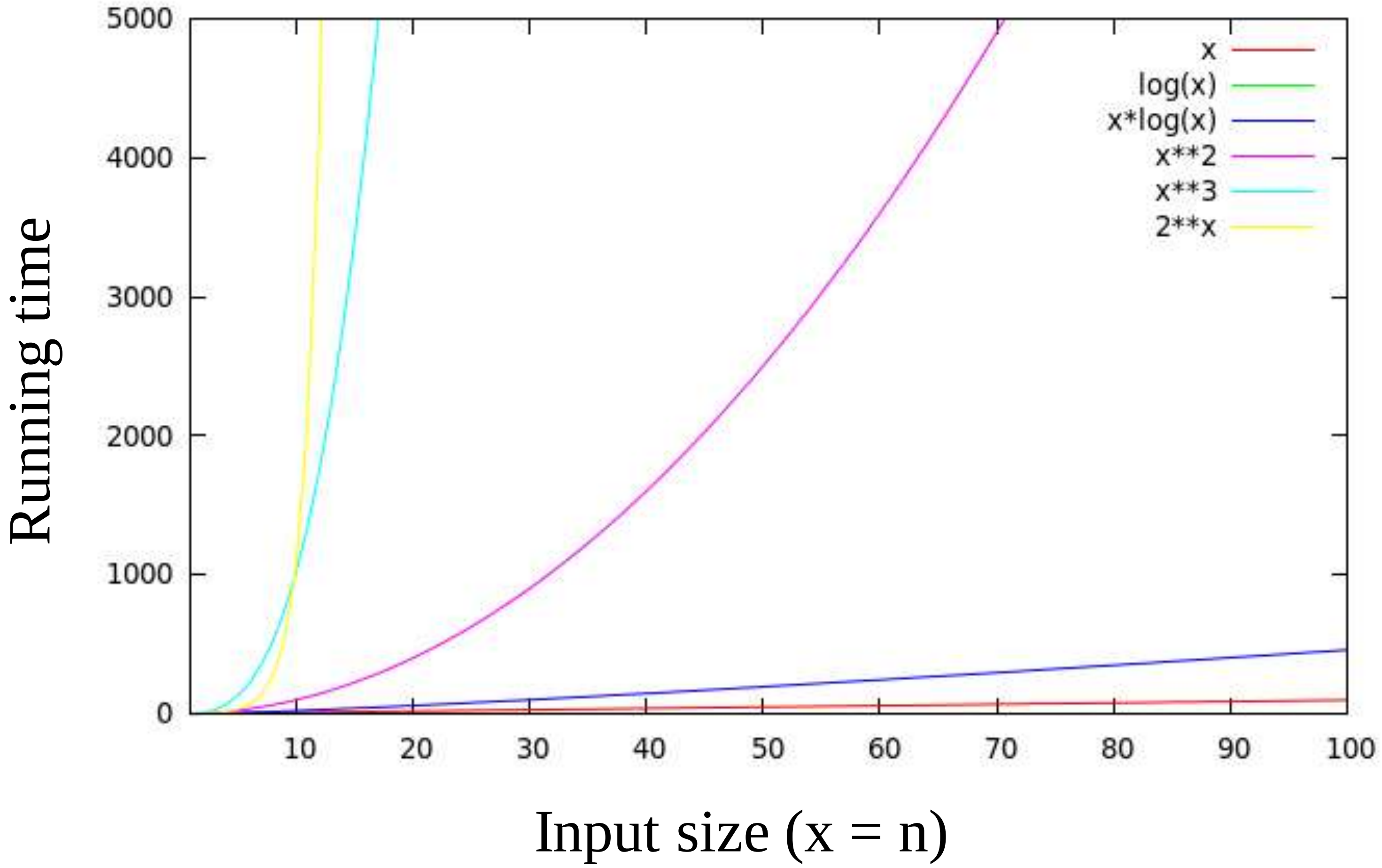
Function	Growth Rate Name
c	Constant
log N	Logarithmic
$\log^2 N$	Log-squared
N	Linear
N log N	Log-linear
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Running Times for Small Inputs



Input size (x = n)

Running Times for Large Inputs



Order-of-Magnitude Analysis and Big O Notation

- **If *Algorithm A requires time proportional to g(n),* Algorithm A is said to be order g(n), and it is denoted as O(g(n)).**
- **The function g(n) is called the algorithm's growth-rate function.**
- **Since the capital O is used in the notation, this notation is called the Big O notation.**
- **If Algorithm A requires time proportional to n^2, it is $O(n^2)$.**
- **If Algorithm A requires time proportional to n, it is O(n).**

Definition of the Order of an Algorithm

Definition:

Algorithm A is order g(n) – denoted as O(g(n)) – if constants k and n_0 exist such that A requires no more than k*g(n) time units to solve a problem of size n $\geq n_0$. ➔ f(n) ≤ k*g(n) for all n $\geq n_0$

- **The requirement of n $\geq n_0$ in the definition of O(f(n)) formalizes the notion of sufficiently large problems.**
  - **In general, many values of k and n can satisfy this definition.**

Order of an Algorithm

- **If an algorithm requires $f(n) = n^2 – 3*n+10$ seconds to solve a problem size n. If constants k and n_0 exist such that**

  $$k*n^2 \geq n^2 – 3*n+10 \qquad \text{for all } n \geq n_0 \,.$$

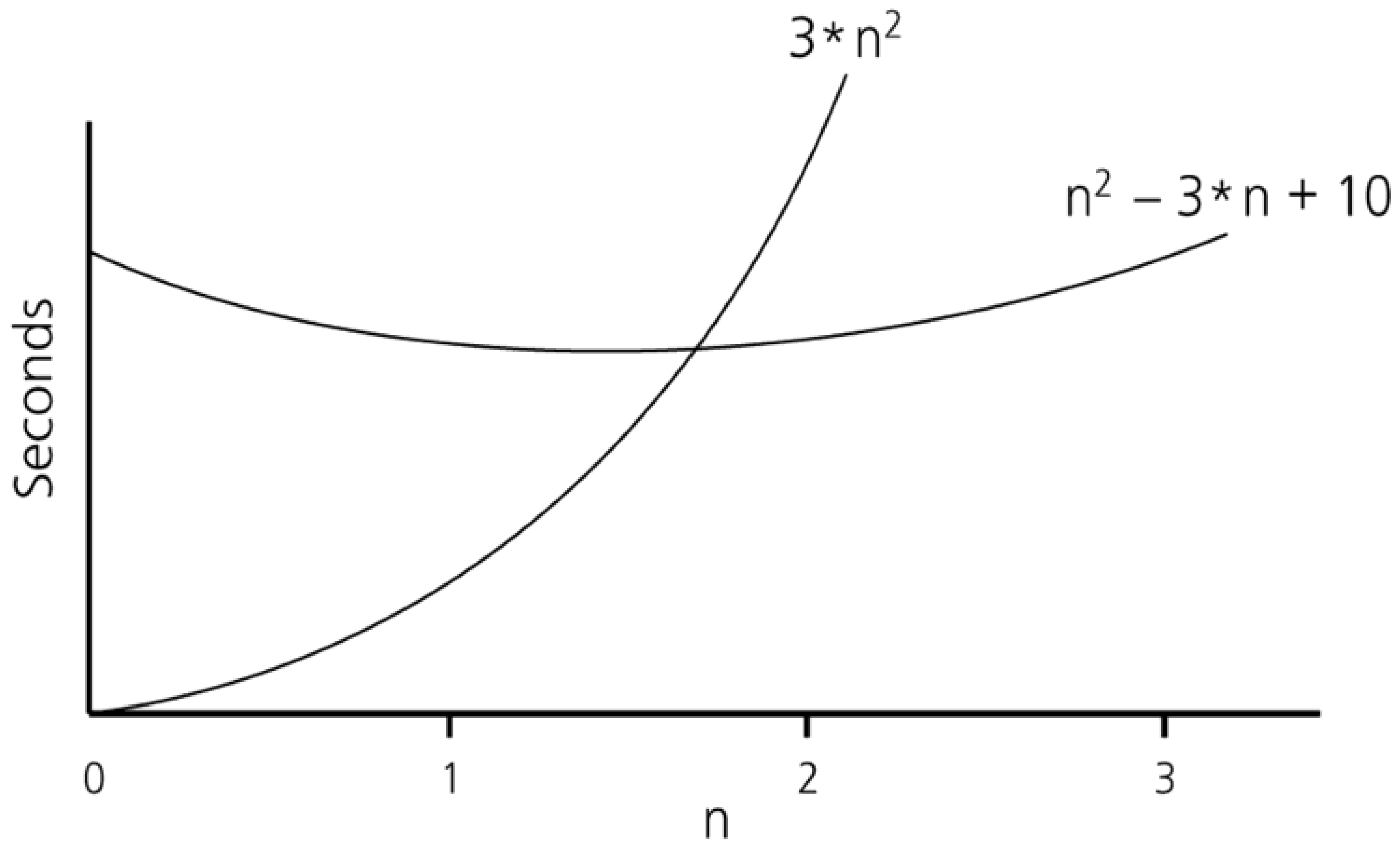
  the algorithm is order n^2 (In fact, k is 3 and n_0 is 2)

  $$3*n^2 \geq n^2 – 3*n+10 \text{ for all } n \geq 2 \,.$$

  Thus, the algorithm requires no more than $k*n^2$ time units for $n \geq n_0$,

  So it is $O(n^2)$

Order of an Algorithm



Order of an Algorithm

- **Show $2^x + 17$ is $O(2^x)$**

- **$2^x + 17 \leq 2^x + 2^x = 2*2^x$ for $x > 5$**
- **Hence $k = 2$ and $n_0 = 5$**

Order of an Algorithm

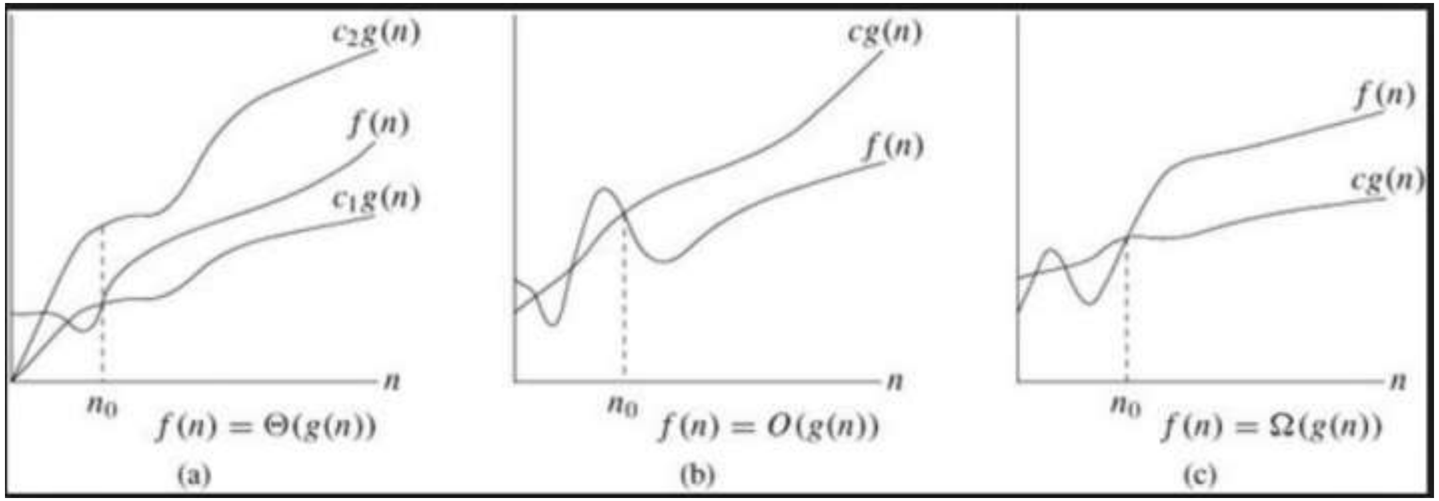
- **Show $2^x + 17$ is $O(3^x)$**

- **$2^x + \mathit{17} \leq k3^x$**
- **Easy to see that rhs grows faster than lhs over time → k=1**
- **However when x is small 17 will still dominate → skip over some smaller values of x by using $n_0 = 2$**
- **Hence $k = 1$ and $n_0 = 2$**

Definition of the Order of an Algorithm

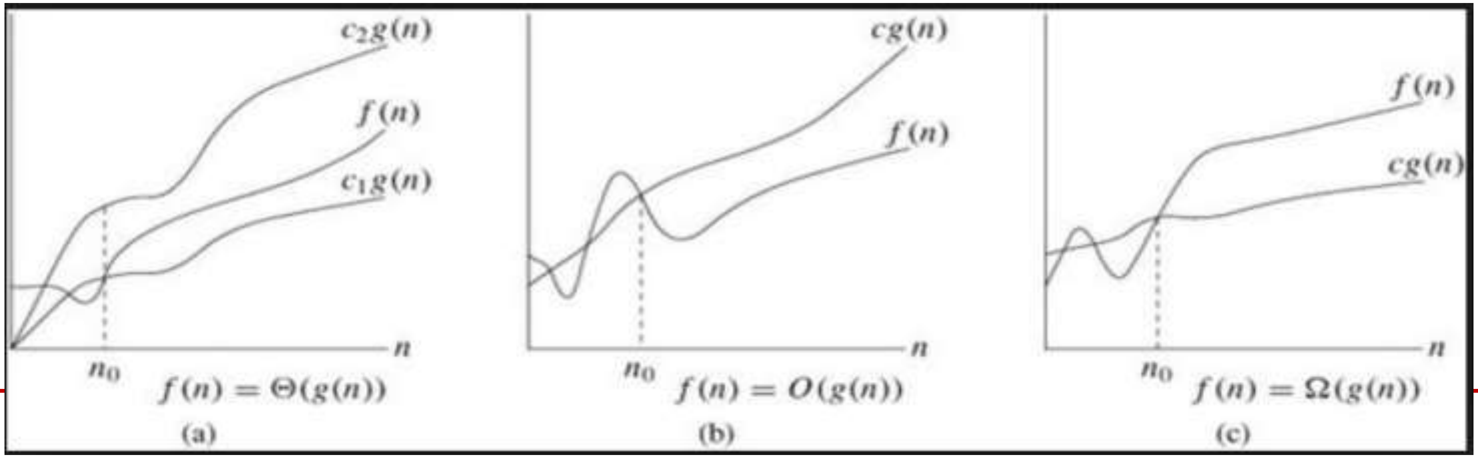
Definition:

Algorithm A is omega g(n) – denoted as $\Omega(g(n))$ – if constants k and n_0 exist such that A requires more than k*g(n) time units to solve a problem of size $n \geq n_0$. ➔ $f(n) \geq k*g(n)$ for all $n \geq n_0$



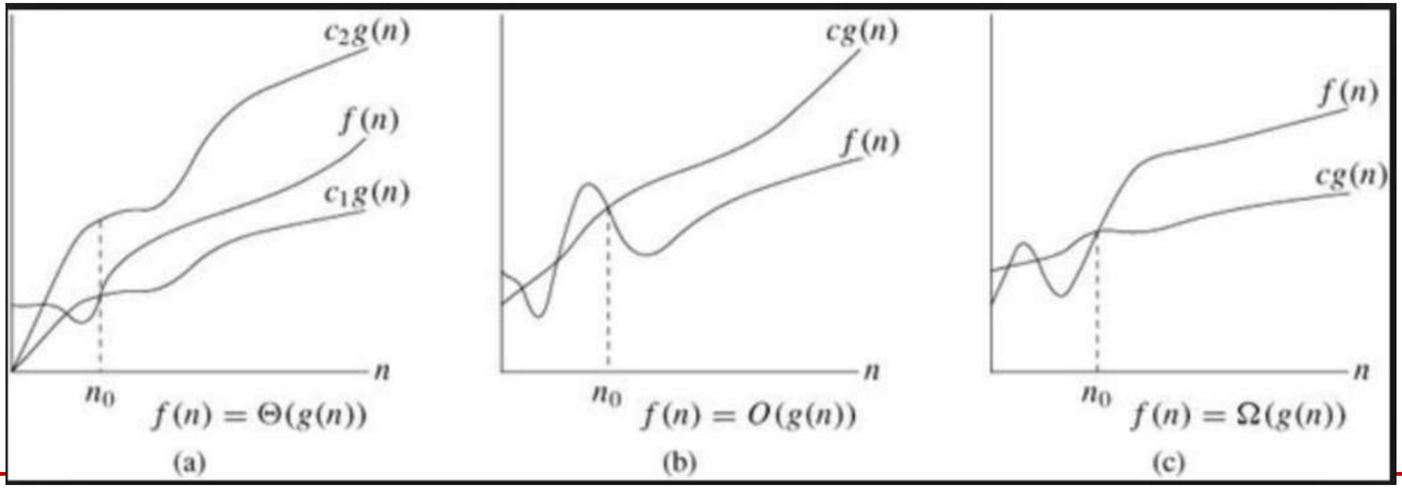
Definition of the Order of an Algorithm

- ***Definition:*** **The function f(n) = Θ(g(n)) if and only if there exist positive constants c1, c2, n0 such that c1g(n) ≤ f(n) ≤ c2g(n), ∀n ≥ n0. Theta can be used to denote tight bounds of an algorithm. i.e., g(n) is a lower bound as well as an upper bound for f(n).**
- **Note that f(n) = Θ(g(n)) if and only if f(n) = Ω(g(n)) and f(n) = O(g(n)).**



Order of an Algorithm

- **Show $f(n) = 7n^2 + 1$ is $\Theta(n^2)$**
  - **You need to show $f(n)$ is $O(n^2)$ and $f(n)$ is $\Omega(n^2)$**
  - **$f(n)$ is $O(n^2)$ because $7n^2 + 1 \leq 7n^2 + n^2\ \forall n \geq 1 \Rightarrow k_2 = 8\ n_0 = 1$**
  - **$f(n)$ is $\Omega\ (n^2)$ because $7n^2 + 1 \geq 7n^2\ \ \forall n \geq 0 \Rightarrow k_1 = 7\ n_0 = 0$**
  - **Pick the largest n_0 to satisfy both conditions naturally $\Rightarrow k_1 = 8$, $k_2 = 7$, $n_0 = 1$**



Growth-Rate Functions

$O(1)$ Time requirement is constant, and it is independent of the problem's size.

$O(\log_2 n)$ Time requirement for a logarithmic algorithm increases increases slowly as the problem size increases.

$O(n)$ Time requirement for a linear algorithm increases directly with the size of the problem.

$O(n*\log_2 n)$ Time requirement for a $n*\log_2 n$ algorithm increases more rapidly than a linear algorithm.

$O(n^2)$ Time requirement for a quadratic algorithm increases rapidly with the size of the problem.

$O(n^3)$ Time requirement for a cubic algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.

$O(2^n)$ As the size of the problem increases, the time requirement for an exponential algorithm increases too rapidly to be practical.

Properties of Growth-Rate Functions

1. ***We can ignore low-order terms in an algorithm's growth-rate function.***
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.
2. ***We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.***
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.
3. ***$O(f(n)) + O(g(n)) = O(f(n)+g(n))$***
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n^2)$ → So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

Growth-Rate Functions – Example1

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

$$T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*c5$$
$$= (c3+c4+c5)*n + (c1+c2+c3)$$
$$= a*n + b$$

➔ **So, the growth-rate function for this algorithm is O(n)**

Growth-Rate Functions – Example2

Code	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)

= a*n^2 + b*n + c

➔ So, the growth-rate function for this algorithm is $O(n^2)$

Growth-Rate Functions Recursive Algorithms

```
int fact(int n) {
  if (n ==0)
      return (1);
  else
      return (n * fact(n-1));
}
```

```
T(n) = T(n-1) + c
     = (T(n-2) + c) + c
```

- = (T(n-3) + c) + c + c
- = (T(n-i) + i*c)
- when i=n → T(0) + n*c → T(n) = O(n)

Sequential Search

```
int sequentialSearch(const int a[], int item, int n){
   for (int i = 0; i < n && a[i]!= item; i++);
   if (i == n)
       return -1;
   return i;
}
```

Unsuccessful Search: **➔ O(n)**

Successful Search:

Best-Case: ***item*** **is in the first location of the array ➔O(1)**

Worst-Case: ***item*** **is in the last location of the array ➔O(n)**

Average-Case: The number of key comparisons 1, 2, ..., n

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$ **➔ O(n)**

Binary Search

We can do binary search if the array is sorted:

```
int binarySearch(int a[], int size, int x) {
   int low =0;
   int high = size -1;
   int mid;      // mid will be the index of
             // target when it's found.
   while (low <= high) {
     mid = (low + high)/2;
     if (a[mid] < x)
        low = mid + 1;
     else if (a[mid] > x)
        high  = mid - 1;
     else
        return mid;
   }
   return -1;
}
```

Binary Search – Analysis

- **For an unsuccessful search:**
  - **The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$**

    ➔ $O(\log_2 n)$
- **For a successful search:**
  - ***Best-Case:* The number of iterations is 1. ➔ O(1)**
  - ***Worst-Case:* The number of iterations is $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$**
  - ***Average-Case:* The avg. # of iterations < $\log_2 n$ ➔ $O(\log_2 n)$**

0	1	2	3	4	5	6	← **an array with size 7**
3	2	3	1	3	2	3	← **# of iterations**

The average # of iterations = 17/7 = 2.4285 < $\log_2 7$

References

- Data Structure and Algorithms for Electrical Engineering by Yung Yi
- CENG707 - Data Structures and Algorithms by Yusuf Sahillioğlu