Analysis of Algorithms

Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

Algorithm

- An ***algorithm*** is a set of instructions to be followed to solve a problem.
  - There can be more than one solution (more than one algorithm) to solve a given problem.
  - An algorithm can be implemented using different prog. languages on different platforms.

- Once we have a correct algorithm for the problem, we have to determine the efficiency of that algorithm.
  - How much ***time*** that algorithm requires.
  - How much ***space*** that algorithm requires.

- We will focus on
  - How to estimate the time required for an algorithm
  - How to reduce the time required

Analysis of Algorithms

- How do we compare the time efficiency of two algorithms that solve the same problem?

- We should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data*.

- To analyze algorithms:
  - First, we start counting the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

Analysis of Algorithms

- Simple instructions (+,-,*,/,=,if,call) take 1 step

  <,>

- Loops and subroutine calls are *not* simple operations

  (function)

  - They depend on size of data and the subroutine
  - “sort” is *not* a single step operation
  - Complex Operations (matrix addition, array resizing) are *not* single step

- We assume infinite memory

- We do not include the time required to read the input

The Execution Time of Algorithms

Consecutive statements

	Times
`count = count + 1;`	1
`sum = sum + count;`	1

Total cost = 1 + 1

➔ The time required for this algorithm is constant

Don’t forget: We assume that each simple operation takes one unit of time

assign + addition 2 op.

The Execution Time of Algorithms

If-else statements

```
                                  Times
if (n < 0){                         1
    absval = -n                     1
    cout << absval;                 1

}
else
    absval = n;                     1
```

Total Cost <= 1 + max(2,1)

1 for if comparison

The Execution Time of Algorithms

Single loop statements

```
                                        Times
i = 1;                                    1
sum = 0;                                  1
while (i <= n) {                         n+1     i = 1 ... n and one more comparison to exit loop
    i = i + 1;                            n
    sum = sum + i;                        n
}
```

Total cost = 1 + 1 + (n + 1) + n + n

➔ The time required for this algorithm is proportional to n

The Execution Time of Algorithms

Nested loop statements

```
                                    Times
i = 1;                                1
sum = 0;                              1
while (i <= n) {                     n+1
    j=1;                              n
    while (j <= n) {              n * (n+1)
        sum = sum + i;              n * n
        j = j + 1;                  n * n
    }
    i = i +1;                         n
}
```

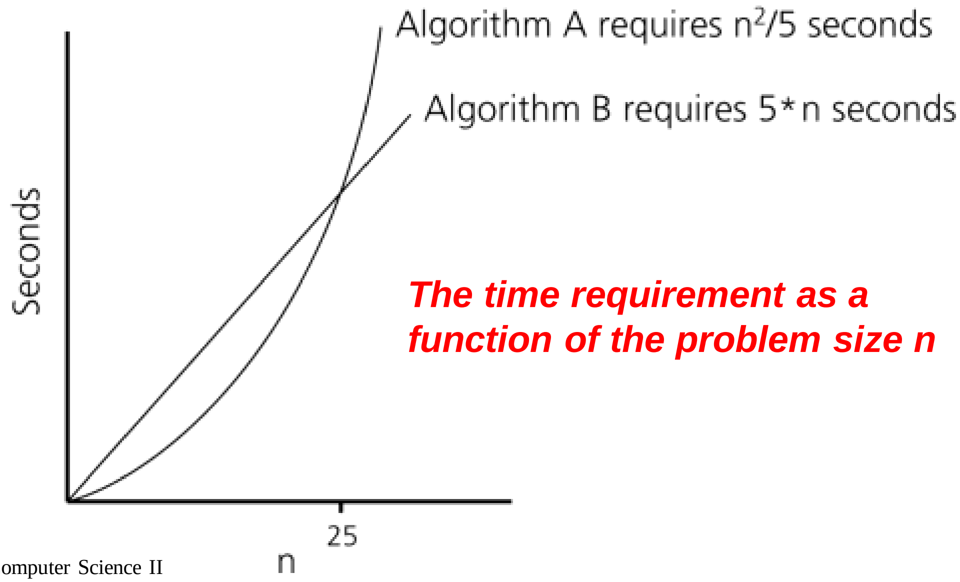
start from inside

Total cost = 1 + 1 + (n + 1) + n + n * (n + 1) + n * n + n * n + n

➔ The time required for this algorithm is proportional to n^2

Algorithm Growth Rates

- We measure the time requirement of an algorithm as a function of the *problem size*.
- The most important thing is to learn how quickly the time requirement of an algorithm grows as a function of the problem size.
- An algorithm’s proportional time requirement is known as ***growth rate***.
- We can compare the efficiency of two algorithms by comparing their growth rates.



The time requirement as a function of the problem size n

Order-of-Magnitude Analysis and Big-O Notation

- If *Algorithm A requires time at most proportional to f(n)*, it is said to be **order f(n),** and it is denoted as **O(f(n))**

- **f(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**

Big-O Notation

Definition:

$\underline{T(n) = O(f(n))}$ if there are positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$

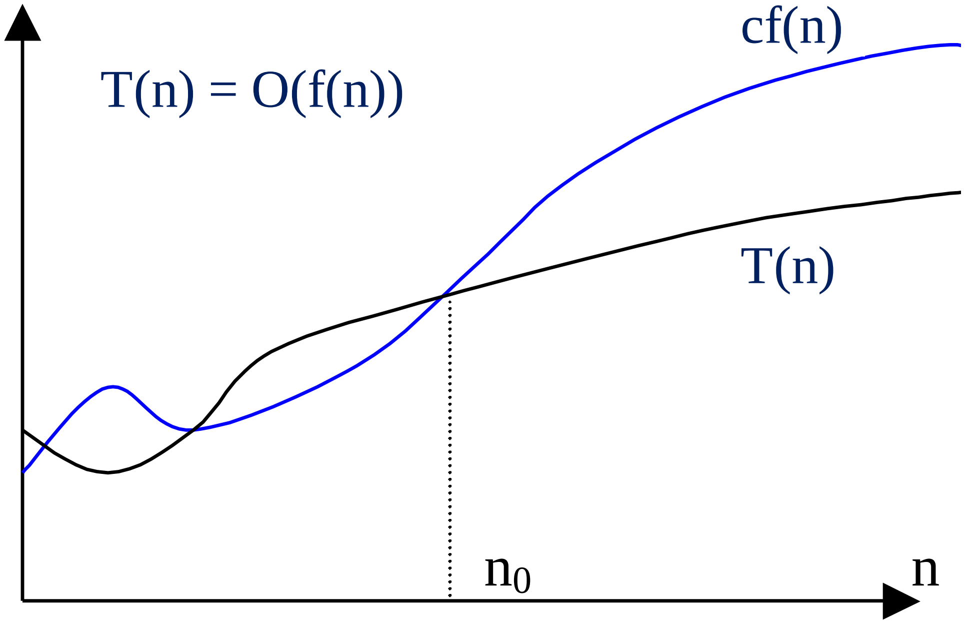
- Algorithm A is order of $f(n)$ if it requires no more than $c \cdot f(n)$ time units to solve a problem of size $n \geq n_0$
  - There may exist many values of c and n_0

- More informally, $c \cdot f(n)$ is an upper bound on $T(n)$

O-notation: Asymptotic upper bound

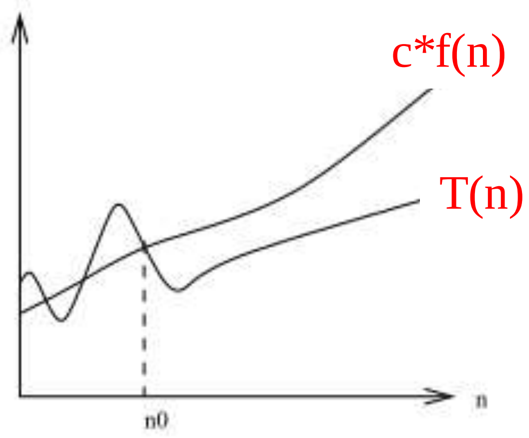
$T(n) = O(f(n))$ if $\exists$ positive constants c, n_0 such that

$$0 \le T(n) \le cf(n),\ \forall n \ge n_0$$



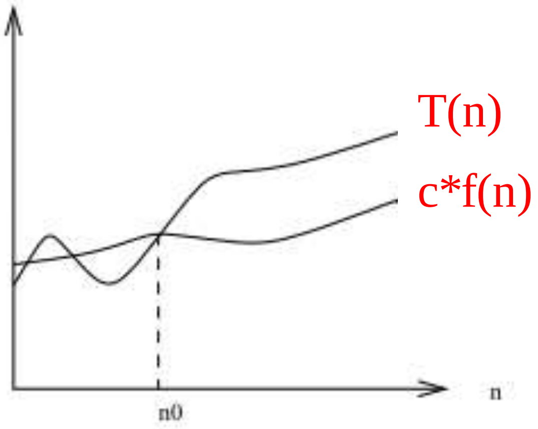
Asymptotic running times of algorithms are usually defined by functions whose domain are N={0, 1, 2, …} (natural numbers)

Big-O Notation



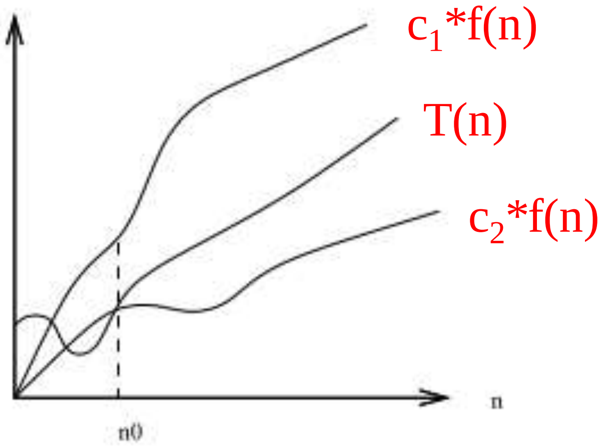
$O(f(n))$

upper bound



$\Omega(f(n))$

lower bound



$\Theta(f(n))$

tight bound

- Big-O definition implies: constant $\mathbf{n_0}$ beyond which it is satisfied

- We do not care about small values of n

Example

Show that $2n^2 = O(n^3)$

We need to find two positive constants: **c** and $\mathbf{n_0}$ such that:

$$0 \leq 2n^2 \leq cn^3 \quad \text{for all } n \geq n_0$$

Choose $c = 2$ and $n_0 = 1$

➔ $2n^2 \leq 2n^3$ for all $n \geq 1$

Or, choose $c = 1$ and $n_0 = 2$

➔ $2n^2 \leq n^3$ for all $n \geq 2$

Example

Show that $2n^2 + n = O(n^2)$

We need to find two positive constants: **c** and $\mathbf{n_0}$ such that:

$$0 \leq 2n^2 + n \leq cn^2 \text{ for all } n \geq n_0$$

$$2 + (1/n) \leq c \text{ for all } n \geq n_0$$

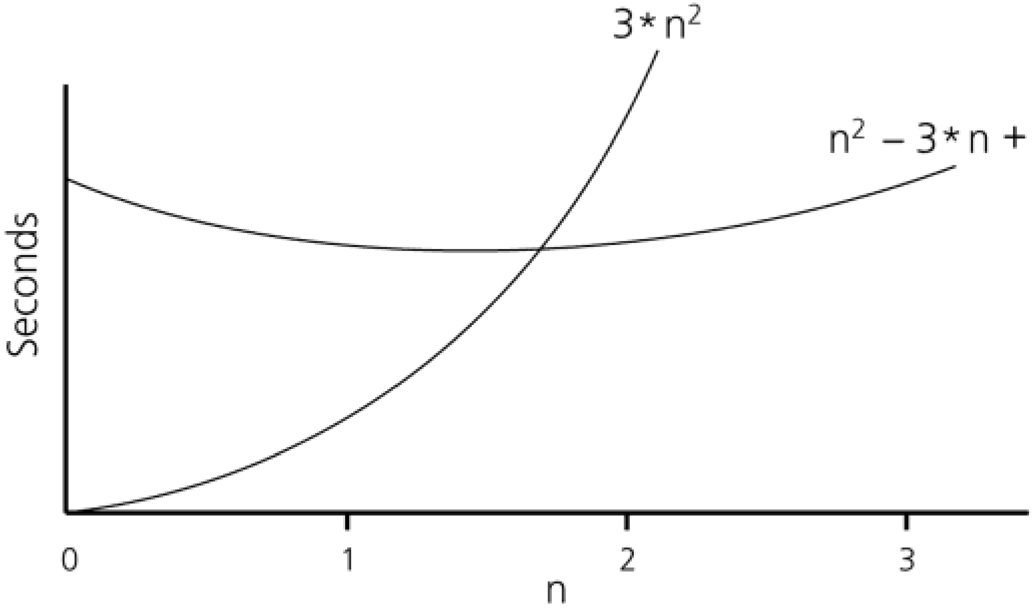
Choose $c = 3$ and $n_0 = 1$

➔ $2n^2 + n \leq 3n^2$ for all $n \geq 1$

Example

- Show that $f(n) = n^2 - 3 \cdot n + 10$ is order of $O(n^2)$
  - – Show that there exist constants *c* and n_0 that satisfy the condition

Try *c* = *3* and n_0 = 2



True or False?

$10^9n^2 = O(n^2)$

True

or bigger

Choose $c = 10^9$ and $n_0 = 1$

$0 \leq 10^9n^2 \leq 10^9n^2$ for $n \geq 1$

$100n^{1.9999} = O(n^2)$

True

Choose $c = 100$ and $n_0 = 1$

$0 \leq 100n^{1.9999} \leq 100n^2$ for $n \geq 1$

$\xcancel{10^{-9}n^{2.0001} = O(n^2)}$

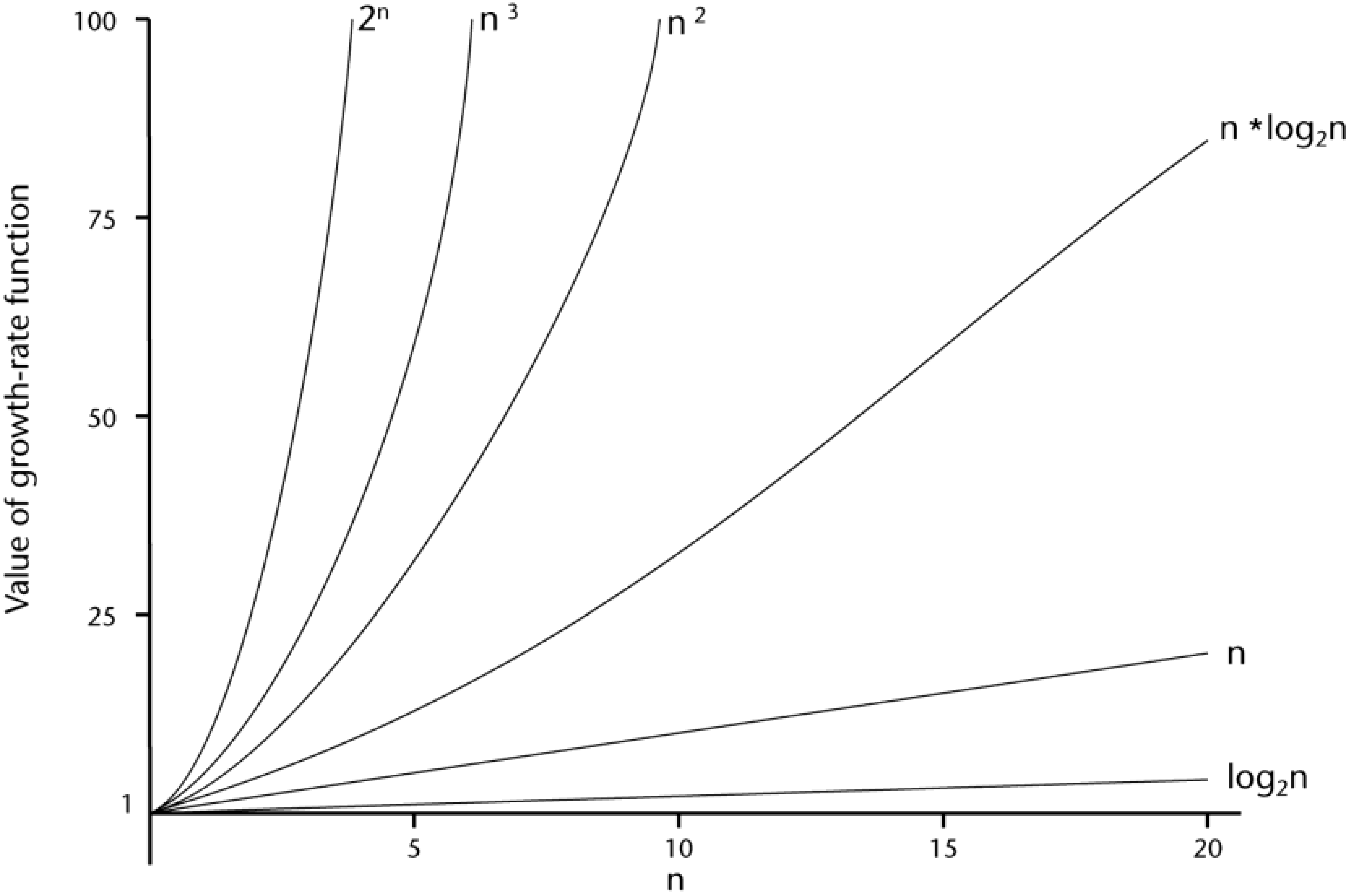
False

$10^{-9}n^{2.0001} \leq cn^2$ for $n \geq n_0$

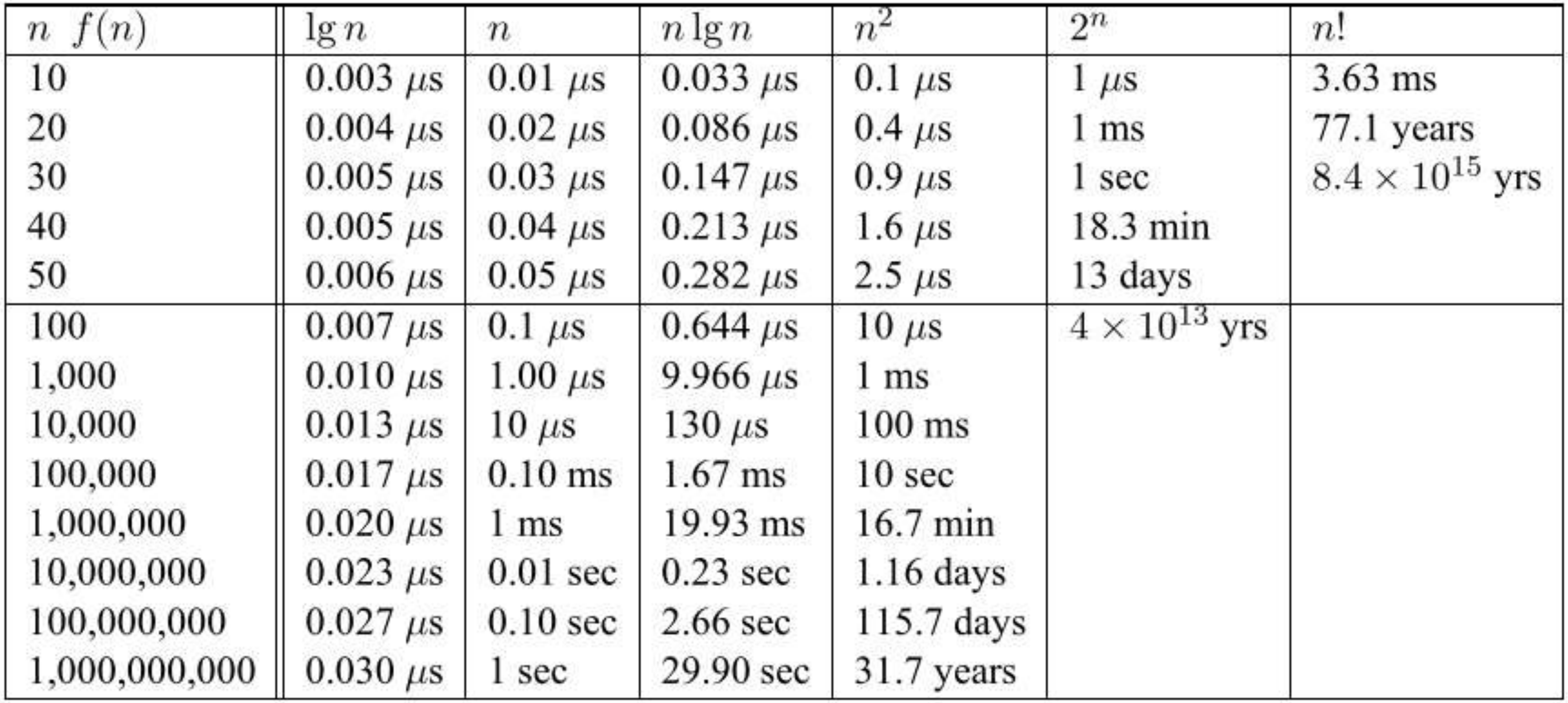
$10^{-9}\,n^{0.0001} \leq c$ for $n \geq n_0$

Contradiction

A Comparison of Growth-Rate Functions



A Comparison of Growth-Rate Functions



n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

A Comparison of Growth-Rate Functions

- Any algorithm with $n!$ complexity is useless for n>=20

- Algorithms with 2^n running time is impractical for n>=40

- Algorithms with n^2 running time is usable up to n=10,000
  - But not useful for n>1,000,000

- Linear time (n) and n log n algorithms remain practical even for one billion items

- Algorithms with log n complexity is practical for any value of n

Properties of Growth-Rate Functions

1. *We can ignore the low-order terms*
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$
   - Use only the highest-order term to determine its grow rate

2. *We can ignore a multiplicative constant in the highest-order term*
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$

3. $O(\ f(n)\ ) + O(\ g(n)\ ) = O(\ f(n) + g(n)\ )$
   - If an algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 +4n^2)$ ➔ So, it is $O(n^3)$
   - Similar rules hold for multiplication

Some Useful Mathematical Equalities

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

$$\sum_{i=1}^{n} i^2 = 1 + 4 + \ldots + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$$

Growth-Rate Functions

Remember our previous examples

```
                                          Times
i = 1;                                      1
sum = 0;                                    1
while (i <= n) {                          n + 1
    i = i + 1;                              n
    sum = sum + i;                          n
}
```

Total cost = $1 + 1 + (n + 1) + n + n = 3 * n + 3$

➔ The time required for this algorithm is proportional to n

➔ The growth-rate of this algorithm is proportional to O(n)

Growth-Rate Functions

```
                                                Times
i = 1;                                            1
sum = 0;                                          1
while (i <= n) {                                n + 1
    j=1;                                          n
    while (j <= n) {                          n * (n + 1)
        sum = sum + i;                          n * n
        j = j + 1;                              n * n
   }
   i = i +1;                                      n
}
```

Total cost = 1 + 1 + (n + 1) + n + n * (n + 1) + n * n + n * n + n

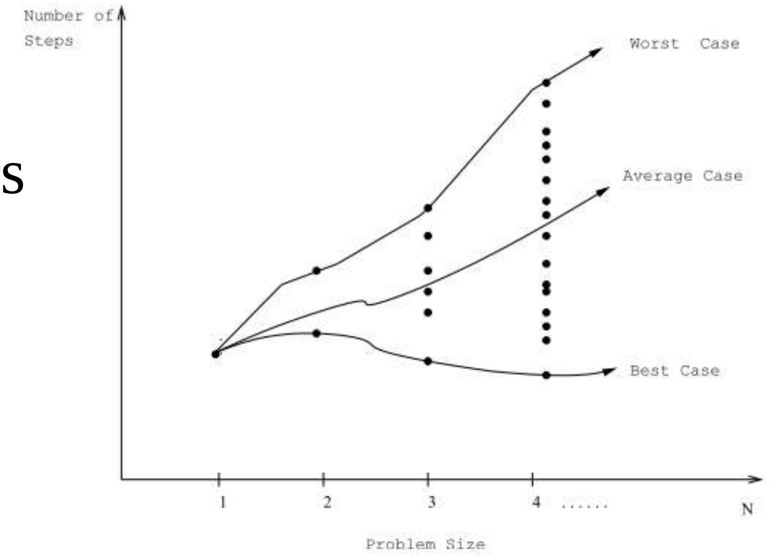
Total cost = 3 * n^2 + 4 * n + 3

➔ The time required for this algorithm is proportional to n^2

➔ The growth-rate of this algorithm is proportional to $O(n^2)$

What to Analyze

- *Worst-case performance*
  - It is an upper bound for any input
  - Its use is more common than the others



- *Best-case performance*
  - This is useless! Why?

- *Average-case performance*
  - It is valid if you can figure out what the "average" input is
  - It is computed considering all possible inputs and their distribution
  - It is usually difficult to compute

Consider the sequential search algorithm

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

Worst-case:

- *If the item is in the last location of the array or*
- *If it is not found in the array*

Best-case:

- *If the item is in the first location of the array*

Average-case:

- *How can we compute it?*

How to find the growth-rate of C++ codes?

Some Examples

Solved on the Board.

What about recursive functions?

Consider the problem of Hanoi towers

```
void hanoi(int n, char source, char dest, char spare) {
  if (n > 0) {
     hanoi(n - 1, source, spare, dest);
     move from source to dest
     hanoi(n - 1, spare, dest, source);
  }
}
```

http://www.cut-the-knot.org/recurrence/hanoi.shtml

How do we find the growth-rate of the recursive `hanoi` function?

- First write a recurrence equation for the `hanoi` function
- Then solve the recurrence equation
  - There are many methods to solve recurrence equations
    - **We will learn a simple one known as *repeated substitutions***

Let's first write **a recurrence equation** for the `hanoi` function

$$T(0) = \Theta(1)$$

$$T(n) = 2 \cdot T(n-1) + \Theta(1)$$

.

.

We will then solve it by using **repeated substitutions**

$$\begin{aligned}
T(n) &= 2 \cdot [2 \cdot T(n-2) + \Theta(1)] + \Theta(1) \\
&= 2 \cdot [2 \cdot [2 \cdot T(n-3) + \Theta(1)] + \Theta(1)] + \Theta(1) \\
&\vdots \\
&= 2^k \cdot T(n-k) + \sum_{i=0}^{k-1} 2^i \cdot \Theta(1) \\
&\vdots \\
&= 2^n \cdot T(n-n) + \sum_{i=0}^{n-1} 2^i \cdot \Theta(1) \\
&= 2^n \cdot T(0) + [2^n - 1] \cdot \Theta(1) \\
&= \Theta(2^n)
\end{aligned}$$

More examples

- Factorial function
- Binary search
- Merge sort – *later*