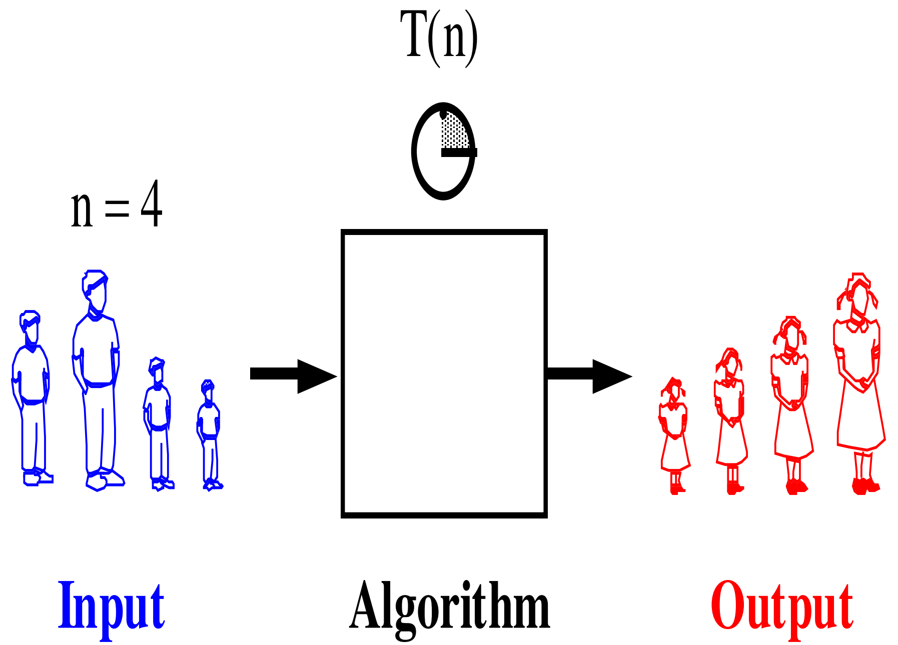
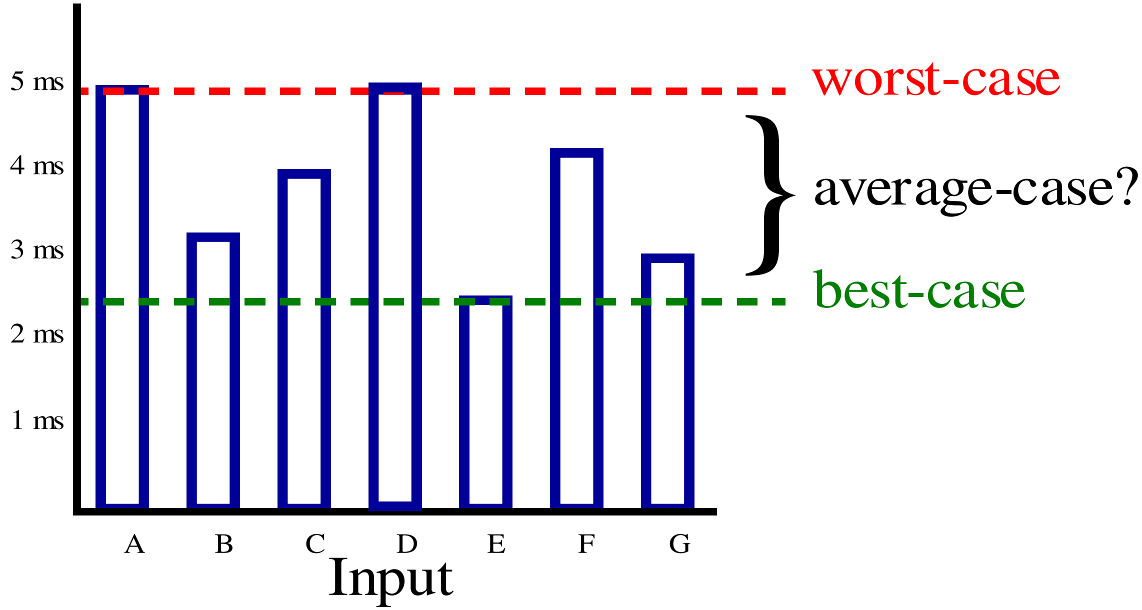
Analysis of Algorithms

- Running Time
- Pseudo-Code
- Analysis of Algorithms
- Asymptotic Notation
- Asymptotic Analysis
- Mathematical facts



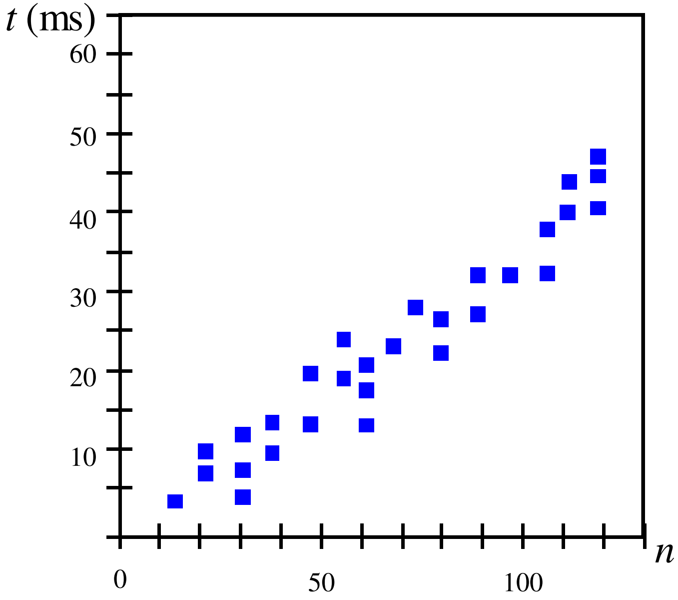
Average Case vs. Worst Case Running Timeof an algorithm

- An algorithm may run faster on certain data sets than on others.
- Finding the average case can be very difficult, so typically algorithms are measured by the worst-case time complexity.
- Also, in certain application domains (e.g., air traffic control, surgery, IP lookup) knowing the worst-case time complexity is of crucial importance.



Measuring the Running Time

- How should we measure the running time of an algorithm?
- Approach 1: Experimental Study
  - Write a program that implements the algorithm
  - Run the program with data sets of varying size and composition.
  - Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time.



Beyond Experimental Studies

- Experimental studies have several limitations:
  - It is necessary to implement and test the algorithm in order to determine its running time.
  - Experiments can be done only on a limited set of inputs, and may not be indicative of the running time on other inputs not included in the experiment.
  - In order to compare two algorithms, the same hardware and software environments should be used.

Beyond Experimental Studies

- We will now develop a general methodology for analyzing the running time of algorithms. In contrast to the "experimental approach", this methodology:
  - Uses a high-level description of the algorithm instead of testing one of its implementations.
  - Takes into account all possible inputs.
  - Allows one to evaluate the efficiency of any algorithm in a way that is independent from the hardware and software environment.

Pseudo-Code

- Pseudo-code is a description of an algorithm that is more structured than usual prose but less formal than a programming language.
- Example: finding the maximum element of an array.

Algorithm arrayMax(A, *n*):

Input: An array A storing n integers.

Output: The maximum element in A.

currentMax $\leftarrow$ A[0]

for $i \leftarrow$ **1 to** n **-1 do**

  if *currentMax* < A[i] **then** *currentMax* $\leftarrow$ A[i]

return *currentMax*

- Pseudo-code is our preferred notation for describing algorithms.
- However, pseudo-code hides program design issues.

What is Pseudo-Code ?

- A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm.

  -Expressions: use standard mathematical symbols to describe numeric and boolean expressions

  -Method Declarations: -**Algorithm** name(***param1*, *param2***)

  -Programming Constructs:
  - decision structures: **if ... then ... [else ...]**
  - while-loops: **while ... do**
  - repeat-loops: **repeat ... until ...**
  - for-loop: **for ... do**
  - array indexing: **A[i]**

  -Methods:
  - calls: object method(args)
  - returns: **return** value

Analysis of Algorithms

- **Primitive Operations:** Low-level computations independent from the programming language can be identified in pseudocode.
- Examples:
  - calling a method and returning from a method
  - arithmetic operations (e.g. addition)
  - comparing two numbers, etc.
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

Example:

Algorithm arrayMax(A, n):

Input: An array A storing n integers.

Output: The maximum element in A.

currentMax $\leftarrow$ A[0]

for *i* =1 **to** *n -1* **do**

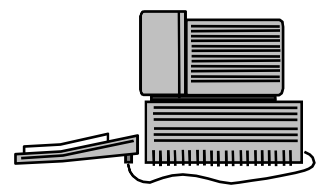
if *currentMax* < *A[i]* **then**

currentMax $\leftarrow$ *A[i]*

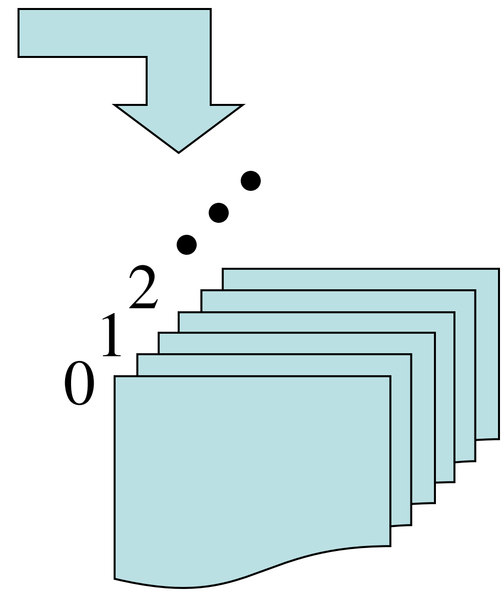
return *currentMax*

The Random Access Memory (RAM) Model

- A **CPU**



- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



- Memory cells are numbered and accessing any cell in memory takes unit time.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

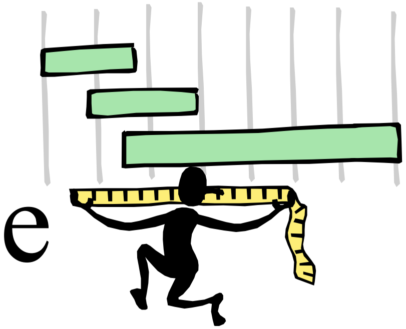
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return $currentMax$	1
Total	$8n - 2$

Estimating Running Time



- Algorithm ***arrayMax*** executes $8\boldsymbol{n} - 2$ primitive operations in the worst case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

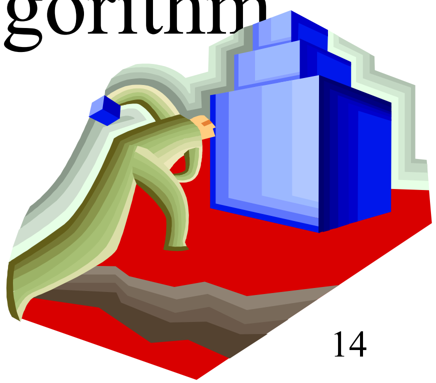
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***arrayMax*.** Then

$$\boldsymbol{a}\,(8\boldsymbol{n} - 2) \leq \boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{b}(8\boldsymbol{n} - 2)$$

- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two linear functions.
- **T(N)** as the number of such operations the algorithm performs given an array of length **N**.

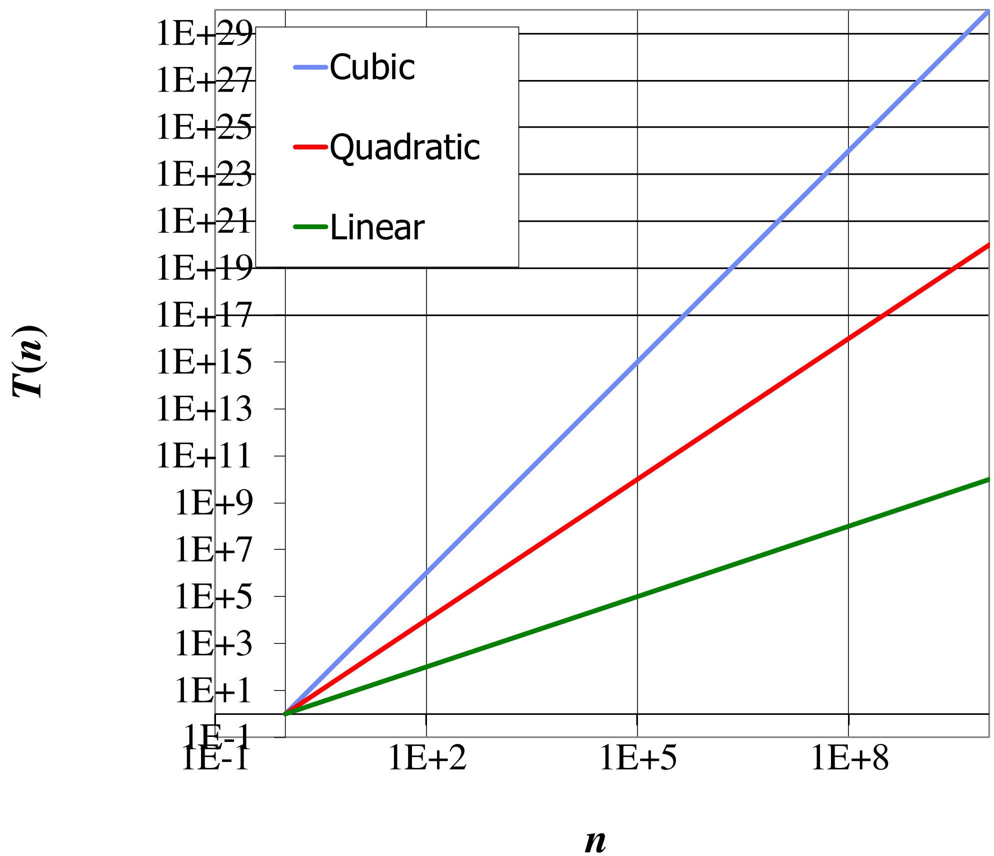
Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***



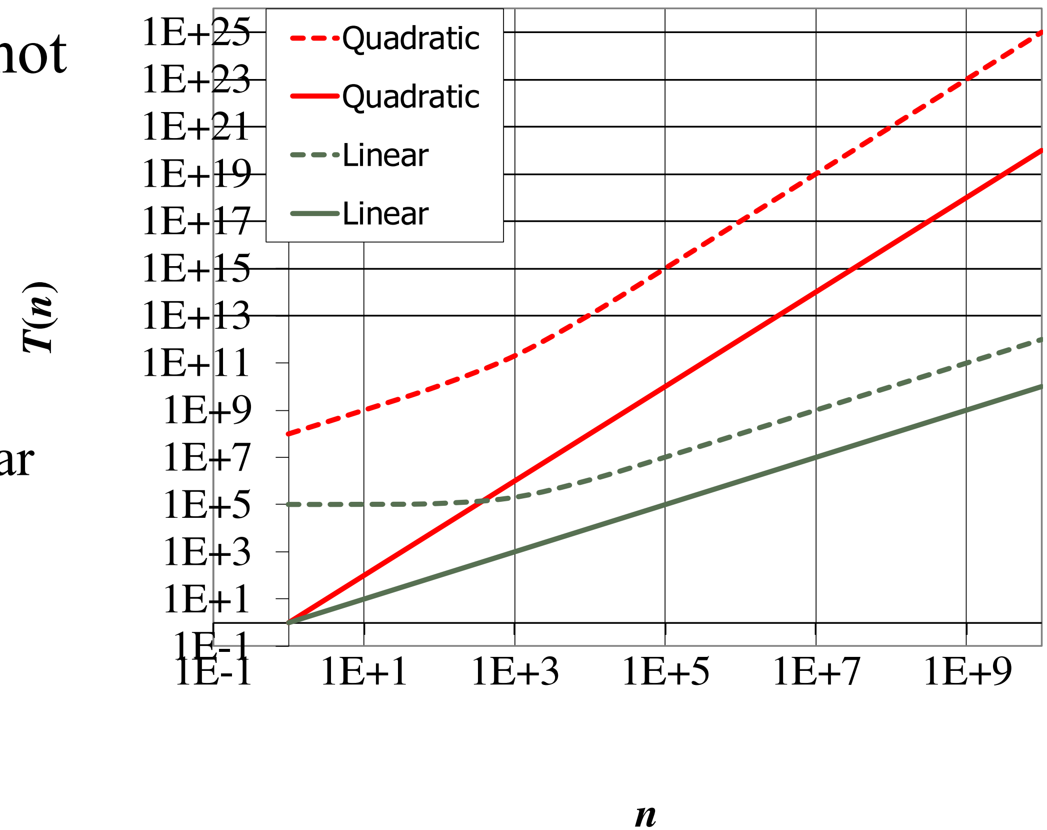
Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ ***1***
  - Logarithmic ≈ log ***n***
  - Linear ≈ ***n***
  - N-Log-N ≈ ***n*** log ***n***
  - Quadratic ≈ $\boldsymbol{n}^2$
  - Cubic ≈ $\boldsymbol{n}^3$
  - Exponential ≈ $\mathbf{2}^n$
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2\boldsymbol{n} + 10^5$ is a linear function
  - $10^5\boldsymbol{n}^2 + 10^8\boldsymbol{n}$ is a quadratic function

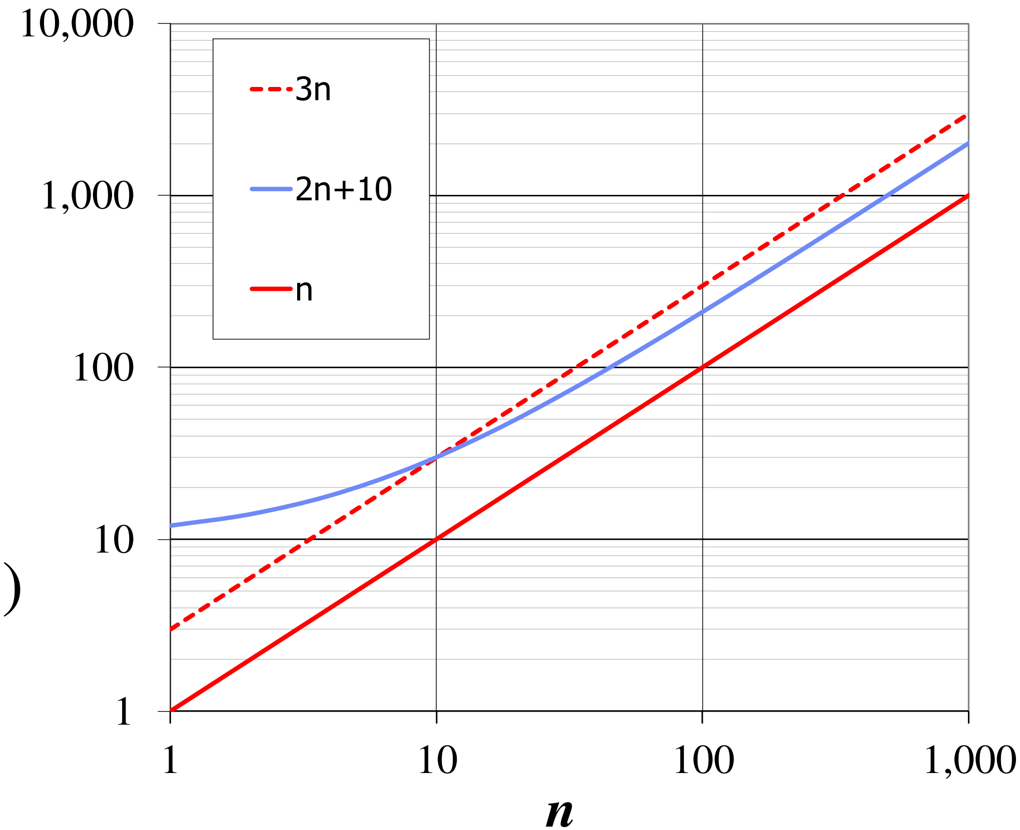


Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

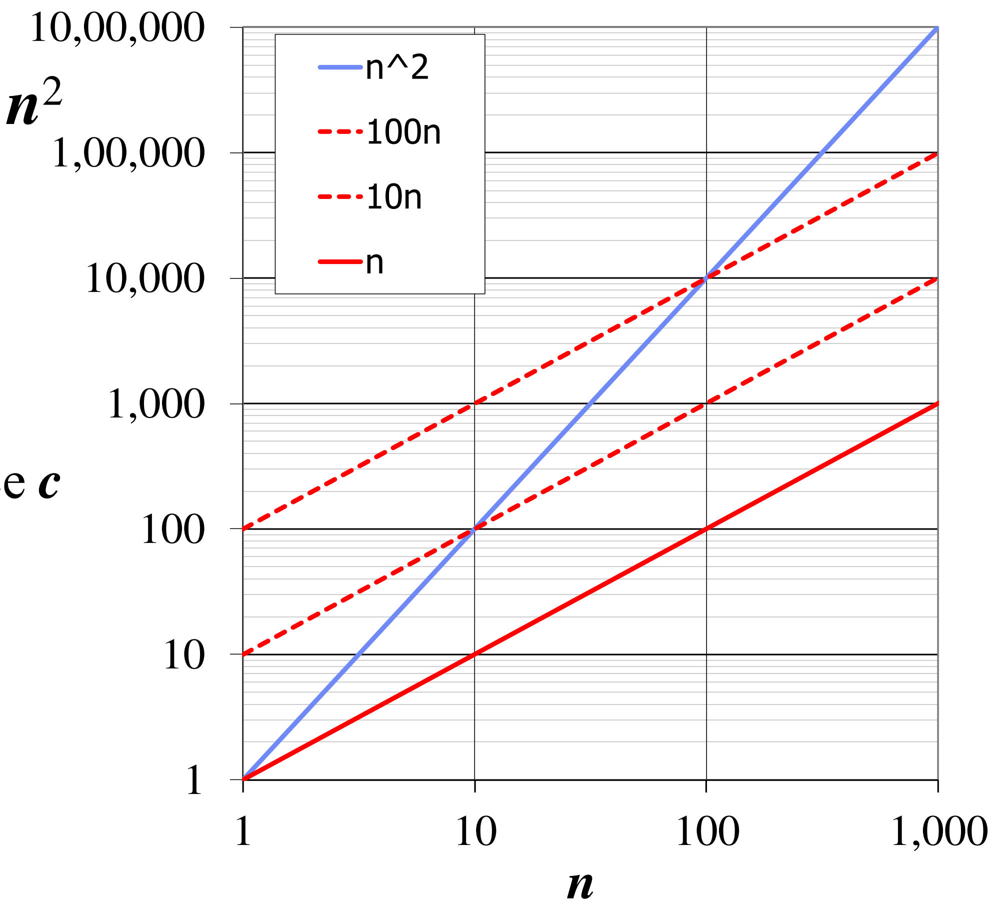
  $f(n) \le cg(n)$ for $n \ge n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

- 7n-2

  7n-2 is O(n)

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

  $Cn-7n \geq 2$ ⫤ $(c-7)\, n \geq 2$ ⫤ $n \geq 2 / (c-7)$ Therefore $c = 7$ and $n_0 = 1$

  //constant can be neglected.

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $8\boldsymbol{n} - 2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $\boldsymbol{O}(\boldsymbol{n})$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

O(1)

- void printFirstElementOfArray(int arr[])
- {
-     printf("First element of array = %d",arr[0]);
- }

- This function runs in O(1) time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

O(n)

```
void
printAllElementOfArray(i
nt arr[], int size)
{
   for (int i = 0; i < size;
i++)
   {
      printf("%d\n", arr[i]);
   }
}
```

- This function runs in O(n) time (or "linear time"), where n is the number of items in the array. If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

$O(n^2)$

```
void
printAllPossibleOrderedPairs(int
arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n",
arr[i], arr[j]);
        }
    }
}
```

- Here we're nesting two loops. If our array has n items, our outer loop runs n times and our inner loop runs n times for each iteration of the outer loop, giving us n2 total prints. Thus this function runs in O(n2) time (or "quadratic time"). If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

$O(2^n)$

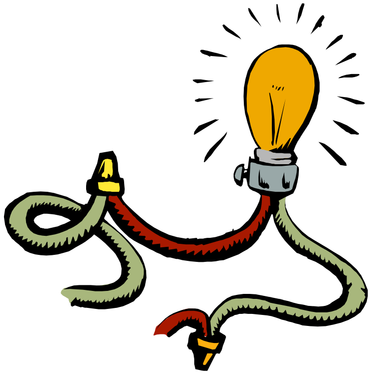
```
int fibonacci(int num)
{
    if (num <= 1)
return num;
    return
fibonacci(num - 2) +
fibonacci(num - 1);
}
```

- An example of an $O(2^n)$ function is the recursive calculation of Fibonacci numbers. $O(2^n)$ denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an $O(2^n)$ function is exponential - starting off very shallow, then rising meteorically.

Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation



Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

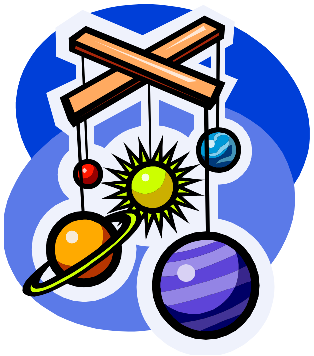
big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \leq c \bullet g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)

- Cheat with a slow algorithm that

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \qquad \text{[arithmetic series]}$$

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Is insertion sort a fast sorting algorithm?

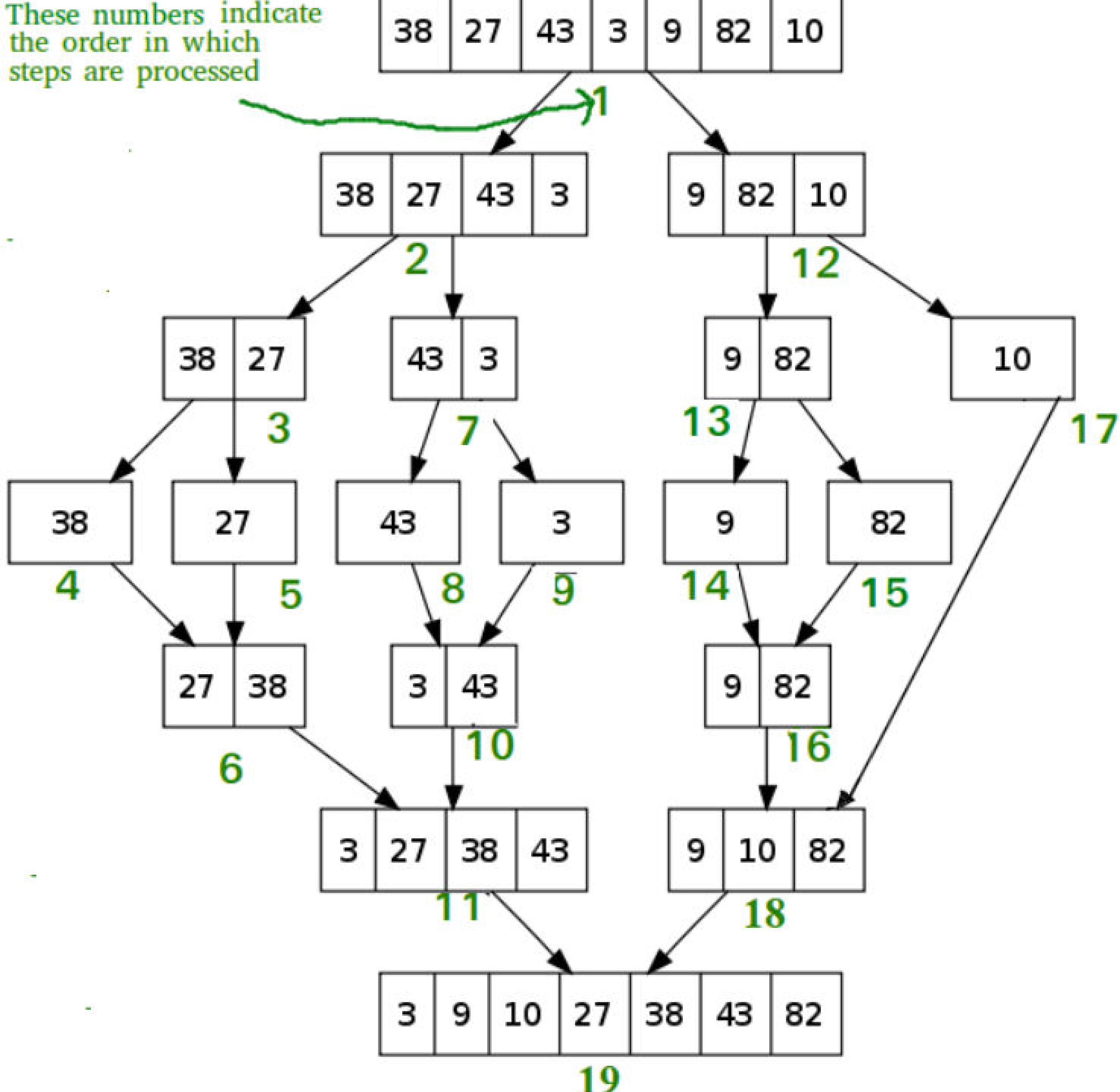
- Moderately so, for small n.
- Not at all, for large n.

Merge sort

MERGE-SORT $A[1 \,.\,.\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: MERGE



These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
4
5
8
9
14
15
27 38
3 43
9 82
6
10
16
3 27 38 43
9 10 82
11
18
3 9 10 27 38 43 82
19

Algorithm:
step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

```
if left > right
    return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

Analyzing merge sort

$T(n)$	**MERGE-SORT** $A[1\,..\,n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\,..\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,..\,n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

se

loppiness: Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$,
ut it turns out not to matter asymptotically.

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.

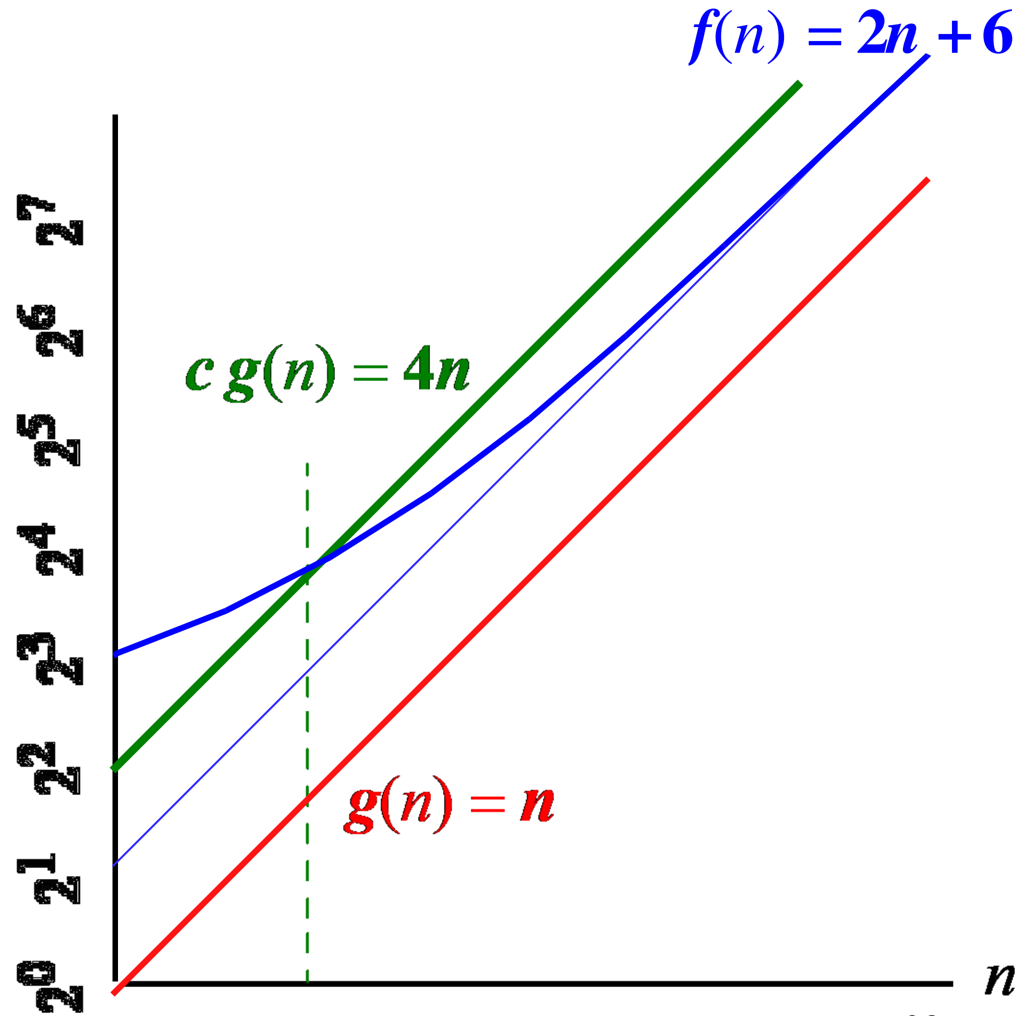
Time Complexity: $O(N \log(N))$, Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \theta(n)$$

Example

For functions ***f(n)*** and ***g(n)*** (to the right) there are positive constants ***c*** and n_0 such that:

$f(n) \leq c\, g(n)$ for $n \geq n_0$



conclusion:

2n+6 is O(n).

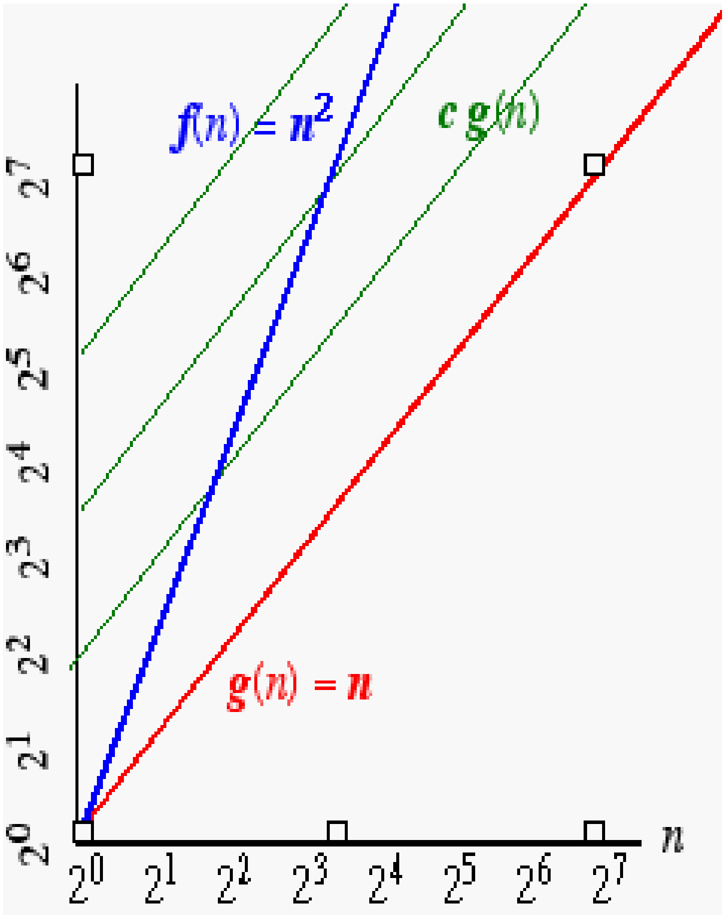
Another Example

On the other hand…

n^2 is not O(n) because there is no c and n_0 such that:

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

(As the graph to the right illustrates, no matter how large a c is chosen there is an n big enough that $n^2 > cn$)



Asymptotic Notation (cont.)

- Note: Even though it is correct to say "$7n - 3$ is $O(n^3)$", a better statement is "$7n - 3$ is $O(n)$", that is, one should make the approximation as tight as possible

- Simple Rule: Drop lower order terms and constant factors

  $7n-3$ is $O(n)$

  $8n^2\log n + 5n^2 + n$ is $O(n^2\log n)$

Asymptotic Notation *(terminology)*

- Special classes of algorithms:

  logarithmic: **O(log n)**

  linear: **O(n)**

  quadratic: **$O(n^2)$**

  polynomial: **$O(n^k)$, $k \geq 1$**

  exponential: **$O(a^n)$, $n > 1$**

- "Relatives" of the Big-Oh
  - Ω (f(n)): Big Omega--asymptotic *lower* bound
  - Θ (f(n)): Big Theta--asymptotic *tight* bound

Asymptotic Analysis of The Running Time

- Use the Big-Oh notation to express the number of primitive operations executed as a function of the input size.
- For example, we say that the arrayMax algorithm runs in $\boldsymbol{O}$(n) time.
- Comparing the asymptotic running time
  - -an algorithm that runs in $\boldsymbol{O}$(n) time is better than one that runs in $\boldsymbol{O}(n^2)$ time
  - -similarly, $\boldsymbol{O}$(log n) is better than $\boldsymbol{O}$(n)
  - -hierarchy of functions: $\sqrt{n} \ll \log n \ll n \ll n^2 \ll n^3 \ll 2^n$

  Caution! Beware of very large constant factors. An algorithm running in time 1,000,000 n is still $\boldsymbol{O}$(n) but might be less efficient on your data set than one running in time $2n^2$, which is $\boldsymbol{O}(n^2)$

Example 1

```
int main()
{
    int a = 0, b = 0;
    int N = 4, M = 4;
    for (int i = 0; i < N; i++) {              // This loop runs for N time
        a = a + 10;              }
for (int i = 0; i < M; i++) {                  // This loop runs for M time
        b = b + 40;              }
      cout << a << ' ' << b;
      return 0;
}
```

Explanation: The Time complexity here will be ***O(N + M)***. Loop one is a single for-loop that runs **N times** and calculation inside it takes ***O(1) time***. Similarly, another loop takes **M times** by combining both the different loops takes by adding them
is ***O(N + M + 1) = O(N + M)***.

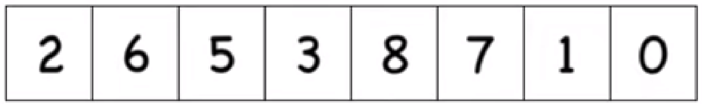
Example 2

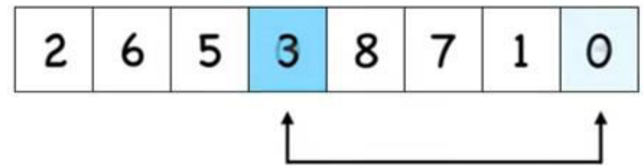
```
int main()
{
    int a = 0, b = 0;
    int N = 4, M = 5;

    // Nested loops
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            a = a + j;

            // Print the current
            // value of a
            cout << a << ' ';
        }
        cout << endl;
    }
    return 0;
}
```

- **outer** loop runs once, the inner will run **M times**, giving us a series as **M + M + M + M + M……….N times**, this can be written as **N * M**.

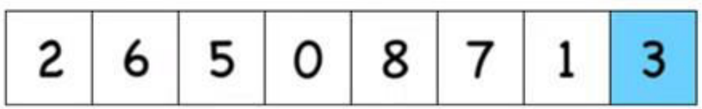
Quick Sort

1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

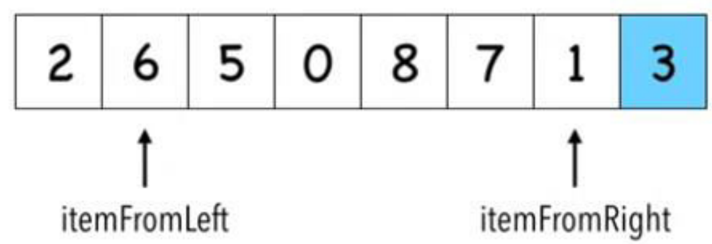




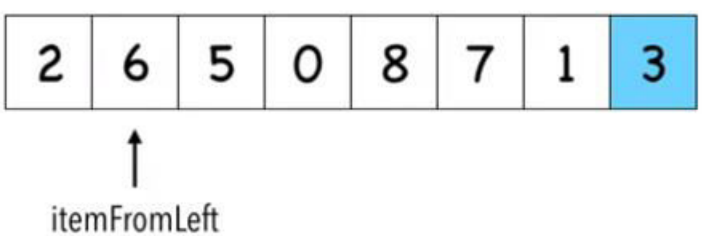
1. itemFromLeft that is larger than pivot



1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot

2	6	5	0	8	7	1	3

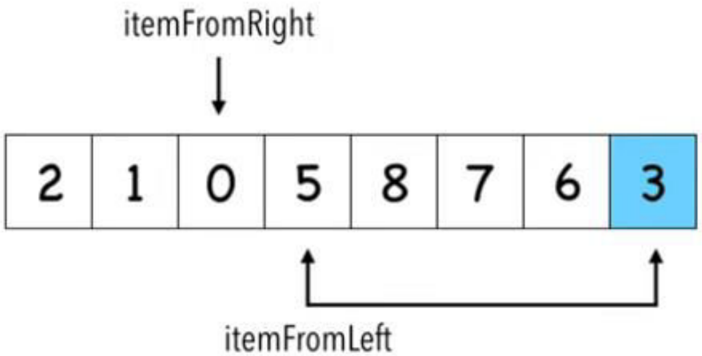
itemFromLeft itemFromRight

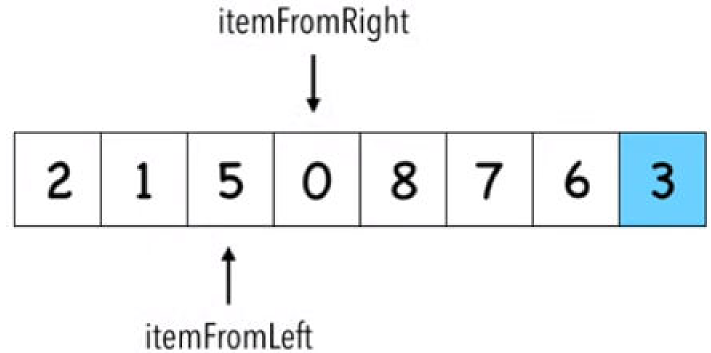
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot

2	1	5	0	8	7	6	3

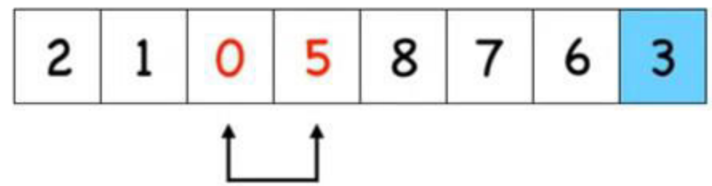
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot

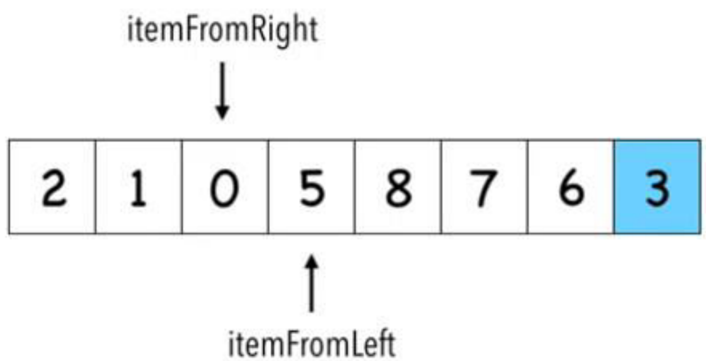
2	1	5	0	8	7	6	3





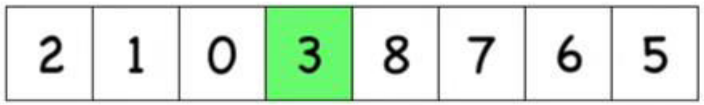
Swap itemFromLeft and pivot



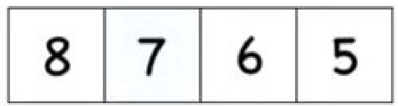


Stop when index of itemFromLeft > index of itemFromRight

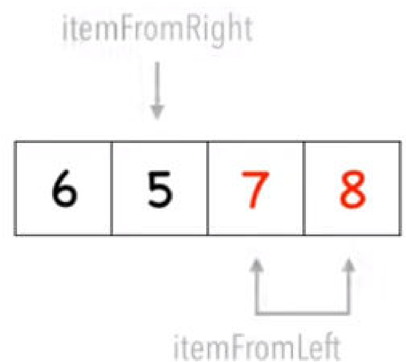
1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger



How to choose pivot:
Median position element:



1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



Worst Case Complexity: $O(n^2)$
Average Case Complexity: $\Theta(n \log n)$