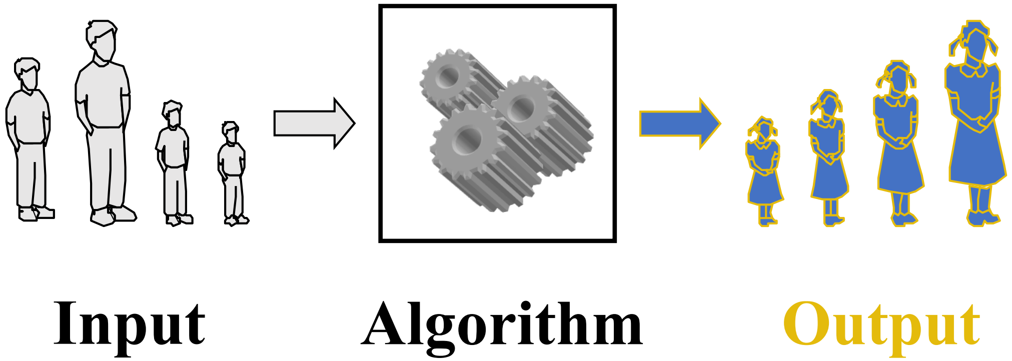
COMP9024: Data Structures and Algorithms

Analysis of Algorithms

Contents

- Big-oh notation
- Big-theta notation
- Big-omega notation
- Asymptotic algorithm analysis

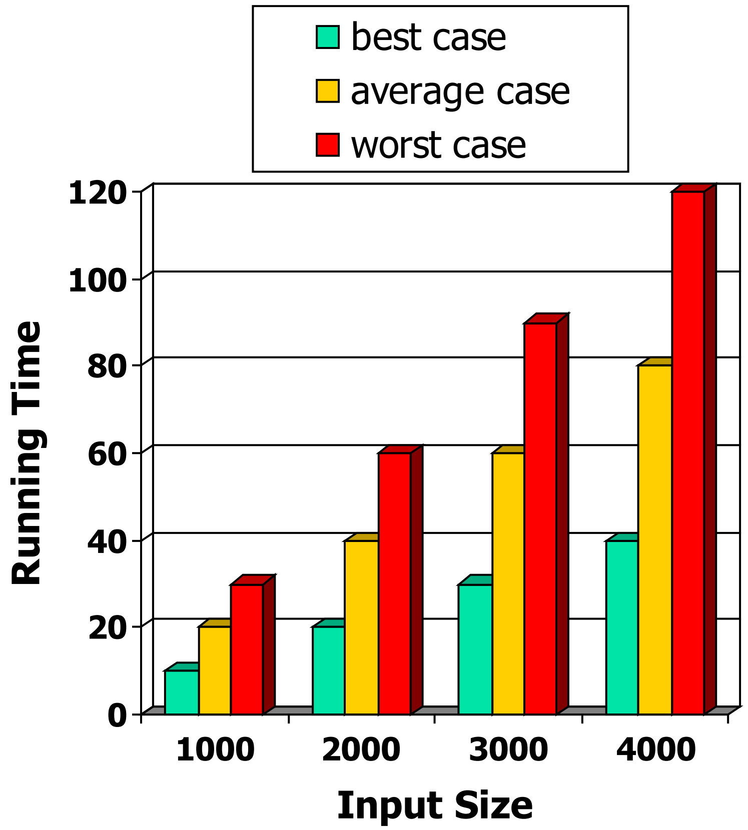
Analysis of Algorithms



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

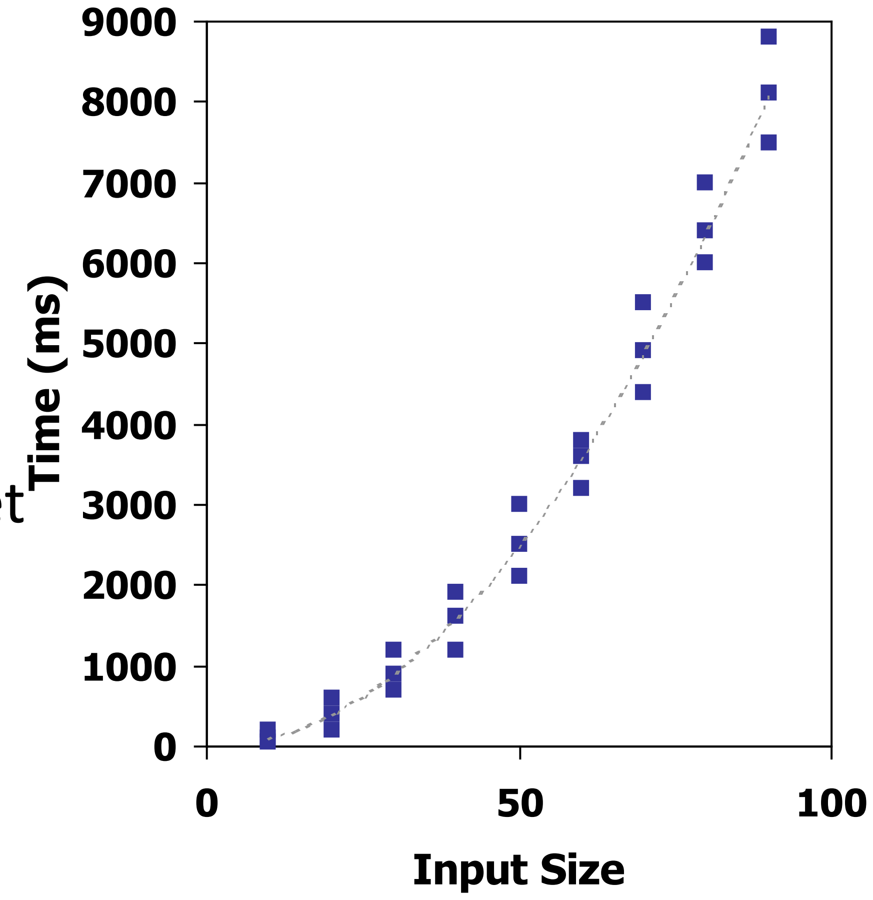
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



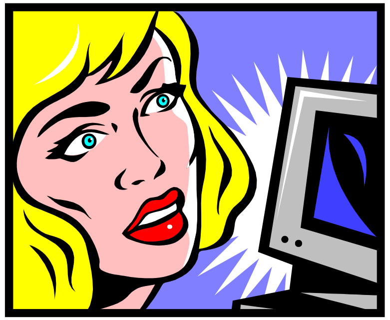
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results 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 must be used



Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

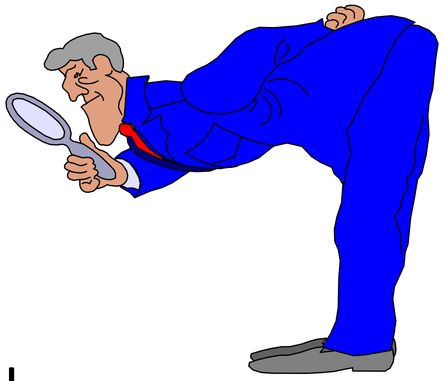
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax(A, n)
 {
   Input array A of n integers
   Output maximum element of A

   currentMax = A[0]
   for ( i=1; i<n; i++)
      if  A[i] > currentMax
         currentMax =A[i]
    return currentMax
 }
```

C-Like Pseudocode Details



- Control flow
  - **if** … [**else** …]
  - **while** …
  - **do** … **while** …
  - **for** …
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

- Method call

  var.method (***arg*** [, ***arg***…])

- Return value

  return ***expression***

- Expressions

  = Assignment

  = Equality testing

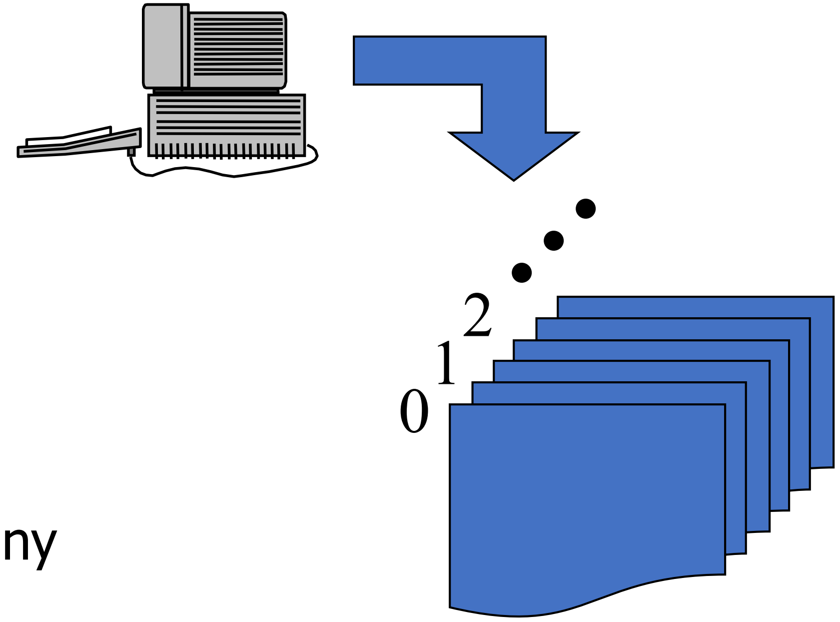
  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

- A **CPU**

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

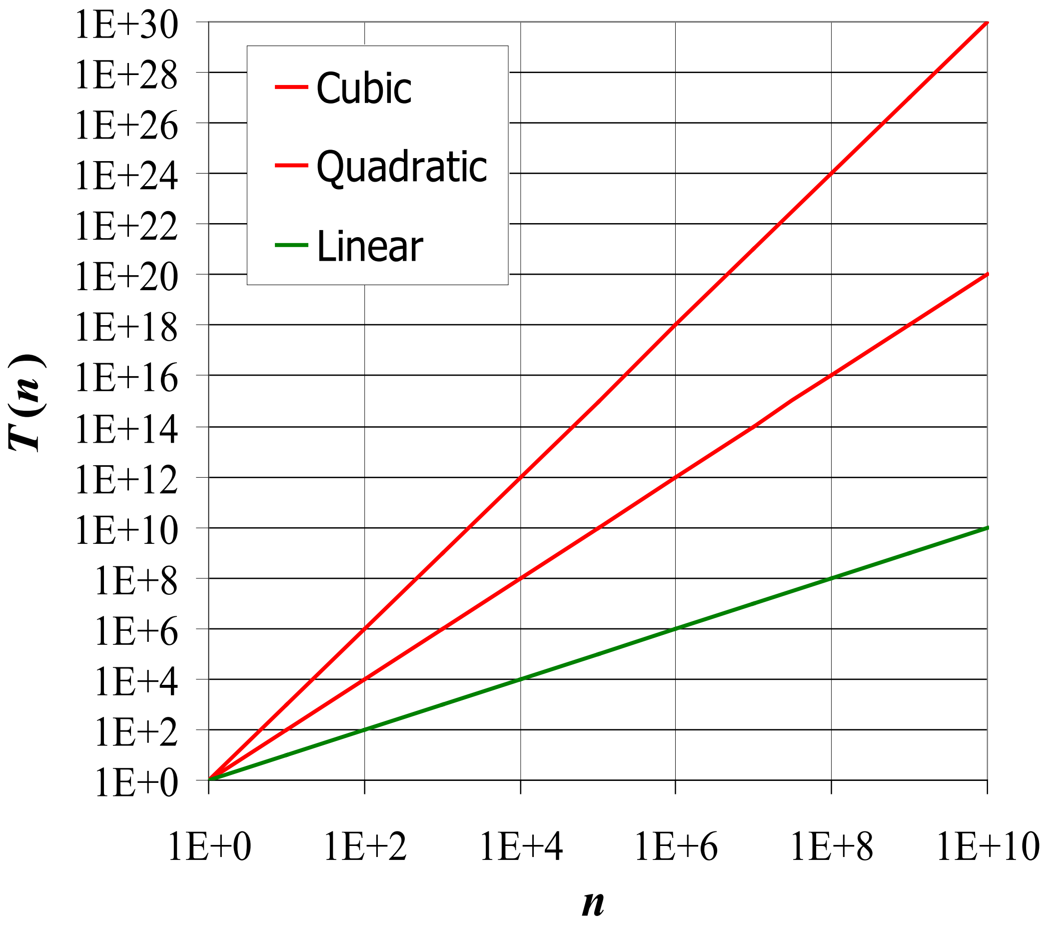
  - Memory cells are numbered and accessing any cell in memory takes unit time.



Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ $\boldsymbol{1}$
  - Logarithmic ≈ log $\boldsymbol{n}$
  - Linear ≈ $\boldsymbol{n}$
  - N-Log-N ≈ $\boldsymbol{n}$ log $\boldsymbol{n}$
  - Quadratic ≈ $\boldsymbol{n}^2$
  - Cubic ≈ $\boldsymbol{n}^3$
  - Exponential ≈ $\boldsymbol{2}^n$

- In a log-log chart, the slope of the line corresponds to the growth rate of the function



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

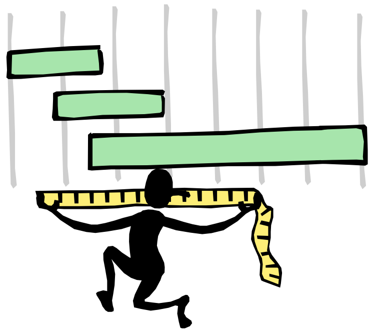
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

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 = *A*[0]	1
for (*i =1; i<n-1; i++*)	n
if *A*[*i*] > *currentMax*	$n - 1$
currentMax = *A*[*i*]	$n - 1$
// increment counter *i*	$n - 1$
return *currentMax*	1
}	Total $4n - 1$

Estimating Running Time



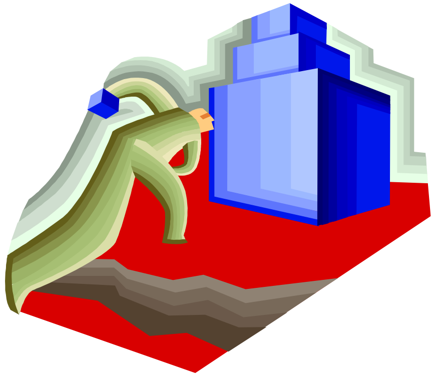
- Algorithm ***arrayMax*** executes $4n - 1$ primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of ***arrayMax.*** Then

$$a\,(4n - 1) \leq T(n) \leq b(4n - 1)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

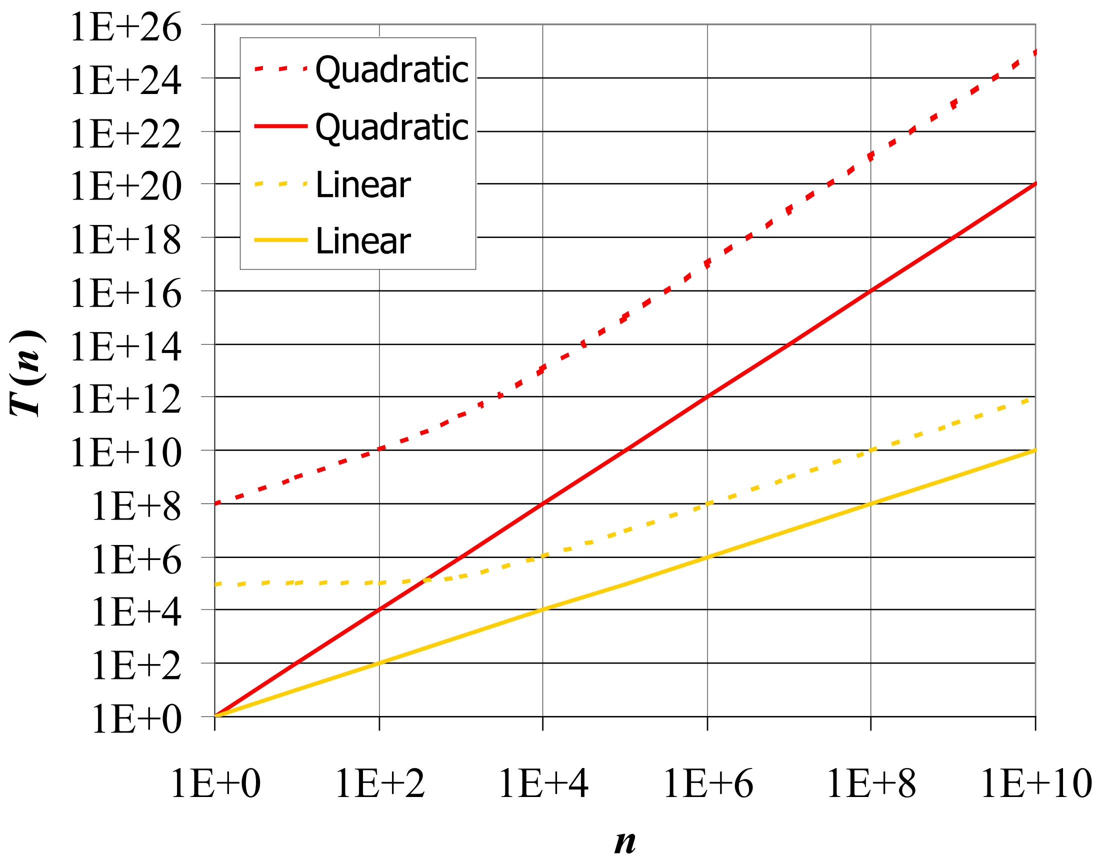
Growth Rate of Running Time

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



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

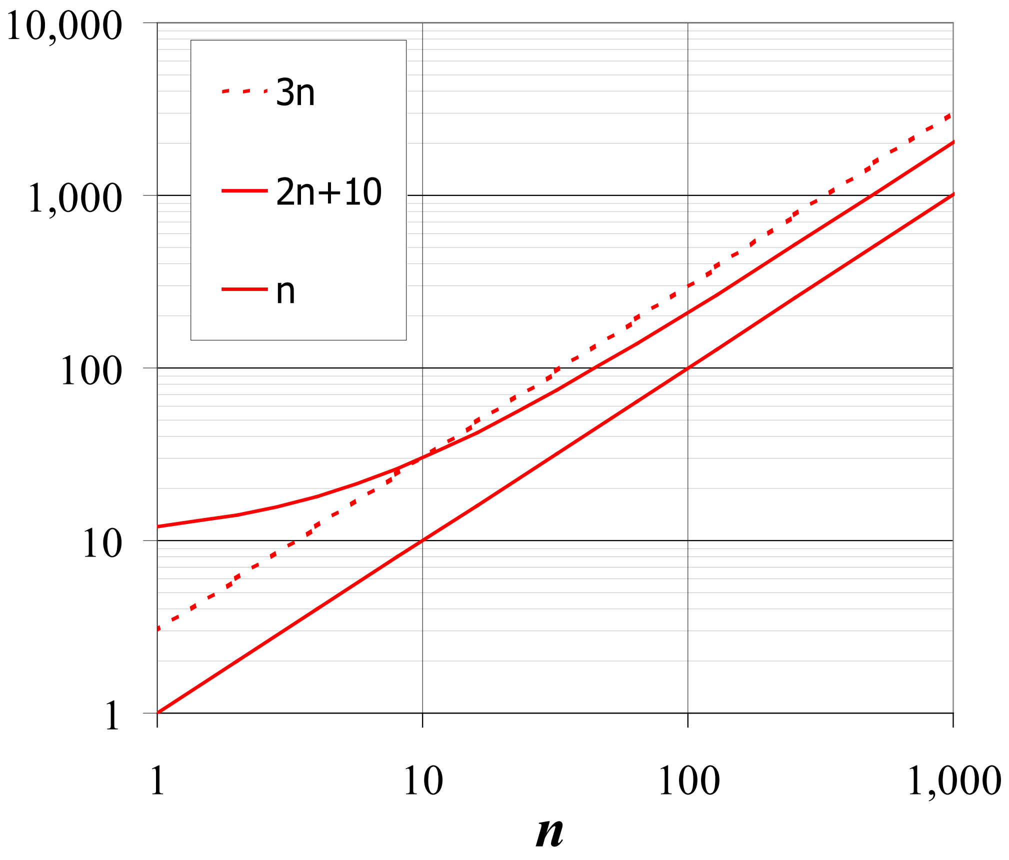


Big-Oh Notation

- 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

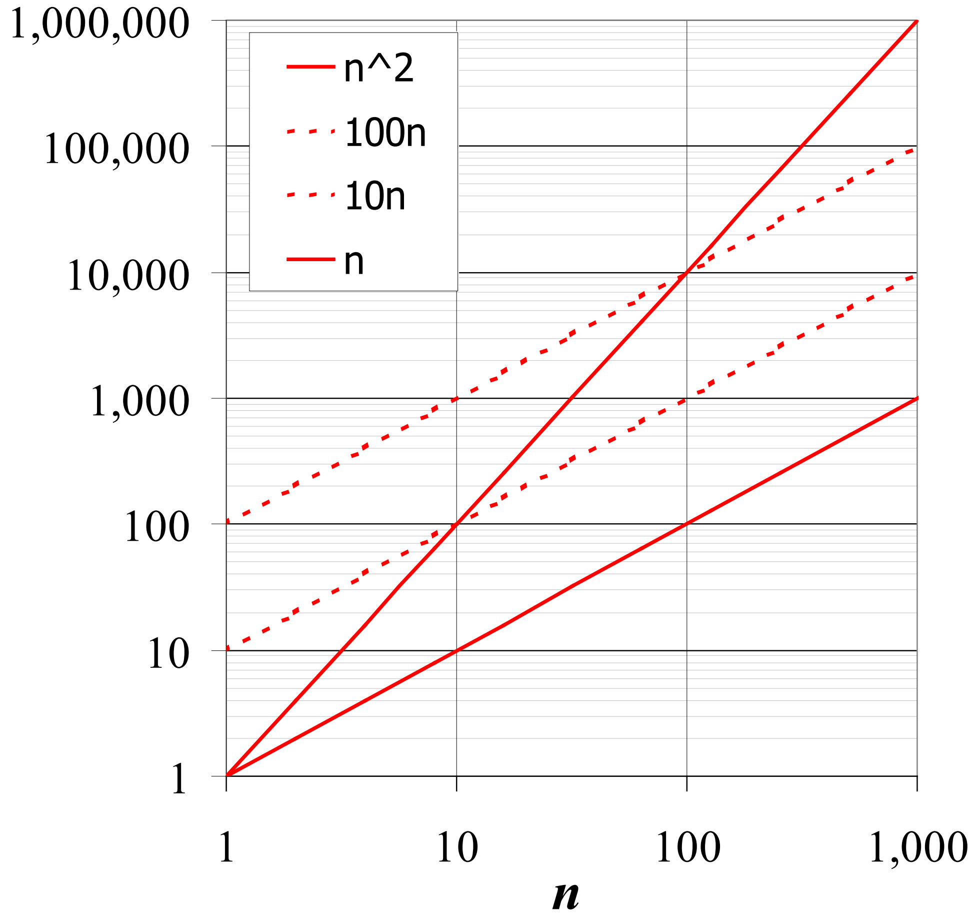
  $$f(n) \le cg(n) \quad \text{for } n \ge n_0$$

- 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 $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - 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 $7n-2 \leq c \cdot n$ for $n \geq n_0$

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

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

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

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

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

- 3 log n + 5

  3 log n + 5 is O(log n)

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

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 (1/3)

- 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 $O(3n)$"

Big-Oh Rules (2/3)

- $5n^5+20n^4-3n^3\log^2 n+10^7 n$ is $O(n^5)$

  Step 1: Drop lower-order terms: $5n^5$

  Step 2: Drop constant factors: n^5

  Therefore, $5n^5+20n^4-3n^3\log^2 n+10^7 n$ is $O(n^5)$.

- $10n^5+200n^4\log n-3n^3\log^2 n+5000n$

  Step 1: Drop lower-order terms: $10n^5$

  Step 2: Drop constant factors: n^5

  Therefore, $10n^5+200n^4\log n-3n^3\log^2 n+5000n$ is $O(n^5)$.

- $10*2^n+200n^{400}-3n^3\log^2 n+500n$

  Step 1: Drop lower-order terms: $10*2^n$

  Step 2: Drop constant factors: 2^n

  $10*2^n+200n^{400}-3n^3\log^2 n+500n$ is $O(2^n)$.

Big-Oh Rules (3/3)

- $1+2^3+3^3+\ldots+n^3$

  Step 1: Drop lower-order terms: n^3

  Step 2: Drop constant factors: n^3

  ~~Therefore, $1+2^3+3^3+\ldots+n^3$ is $O(n^3)$~~ ✕

- The drop-constant-factor rule is only applicable to an arithmetic expression with a constant number of terms.

- $1+2^3+3^3+\ldots+n^3 < n*n^3=O(n^4)$

Asymptotic Algorithm Analysis

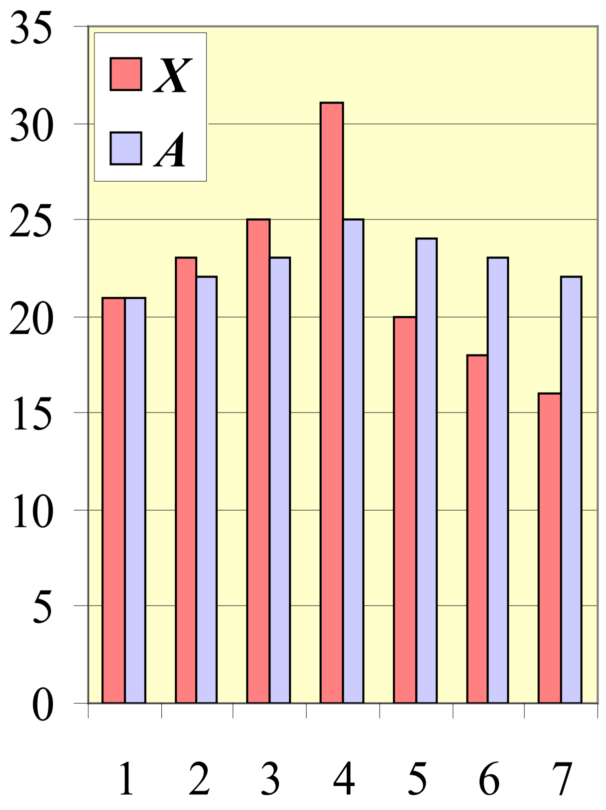
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation.
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $4n - 1$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The $\boldsymbol{i}$-th prefix average of an array $\boldsymbol{X}$ is average of the first $(\boldsymbol{i} + 1)$ elements of $\boldsymbol{X}$:

$$\boldsymbol{A}[\boldsymbol{i}] = (\boldsymbol{X}[0] + \boldsymbol{X}[1] + \ldots + \boldsymbol{X}[\boldsymbol{i}])/(i+1)$$

- Computing the array $\boldsymbol{A}$ of prefix averages of another array $\boldsymbol{X}$ has applications to financial analysis



Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm *prefixAverages1*(*X*, *n*)	
{	
Input array X of n integers	
Output array A of prefix averages of X	#operations
A = new array of n integers;	n
for ($i = 0$; $i < n$; i++)	$n+1$ **last unsuccessful compare also count**
{ $s = X[0]$;	n
for ($j = 1$; $j \le i$; j++)	$1 + 2 + \ldots + (n - 1)$
$s = s + X[j]$;	$1 + 2 + \ldots + (n - 1)$
$A[i] = s / (i + 1)$;	n
}	
return A;	1
}	

Arithmetic Progression

- The total number of primitive operations of **prefixAverages1** is

  $n+n+1+n+1 + 2 + \ldots + (n-1)+1 + 2 + \ldots + (n-1)+n+1$

  $=n^2+3n+2=O(n^2).$

- Thus, algorithm ***prefixAverages1*** runs in $\boldsymbol{O}(n^2)$ time, or we say the *time complexity* of ***prefixAverages1*** is $\boldsymbol{O}(n^2)$.

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

```
Algorithm prefixAverages2(X, n)
 { Input array X of n integers
   Output array A of prefix averages of X          #operations
   A = new array of n integers                            n
   s = 0                                                  1
   for ( i = 0;  i < n; i++ )                             n
     {  s = s + X[i]                                      n
        A[i] = s / (i + 1)                                n
     }
   return A                                               1
 }
```

- Algorithm ***prefixAverages2*** runs in $O(n)$ time

Binary Search (1/4)

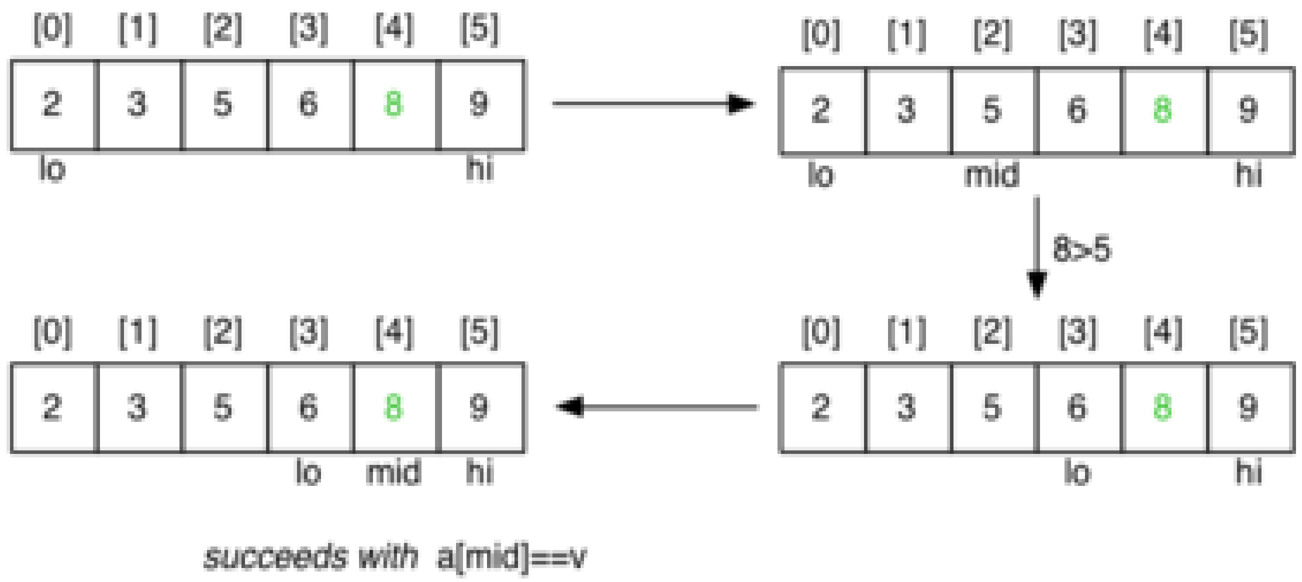
The following recursive algorithm searches for a value in a sorted array:

```
BinarySearch(v, a, lo, hi)
  Input  value v
         array a[lo..hi] of values
  Output true if v in a[lo..hi]
         false otherwise

   mid=(lo+hi)/2
   if lo>hi  return false
   if a[mid]=v  return true
   else if a[mid]<v
             return BiarySearch(v,a,mid+1,hi)
        else
             return BinarySearch(v,a,lo,mid-1)
```

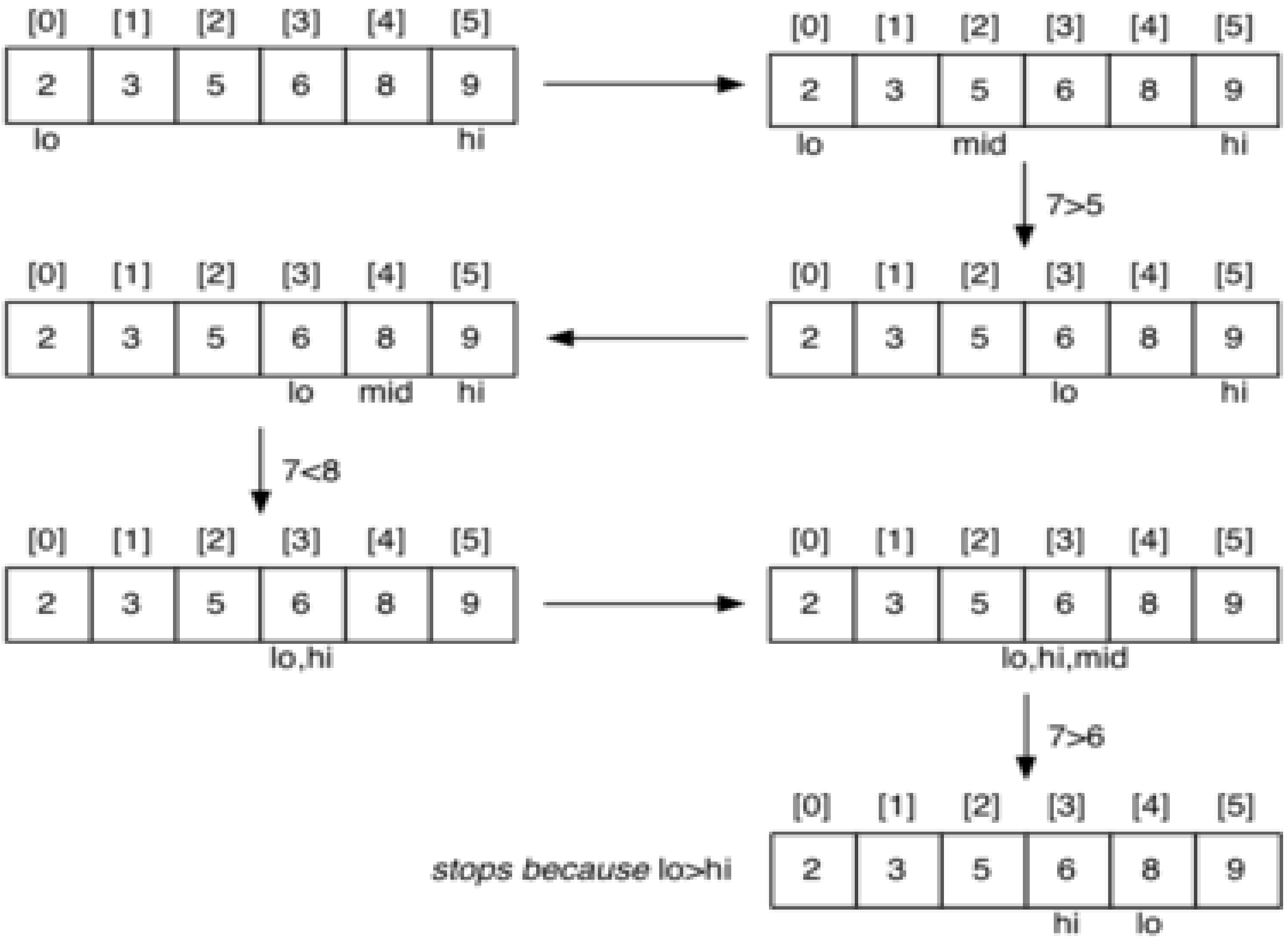
Binary Search (2/4)

Successful search for a value of 8:



Binary Search (3/4)

Unsuccessful search for a value of 7:

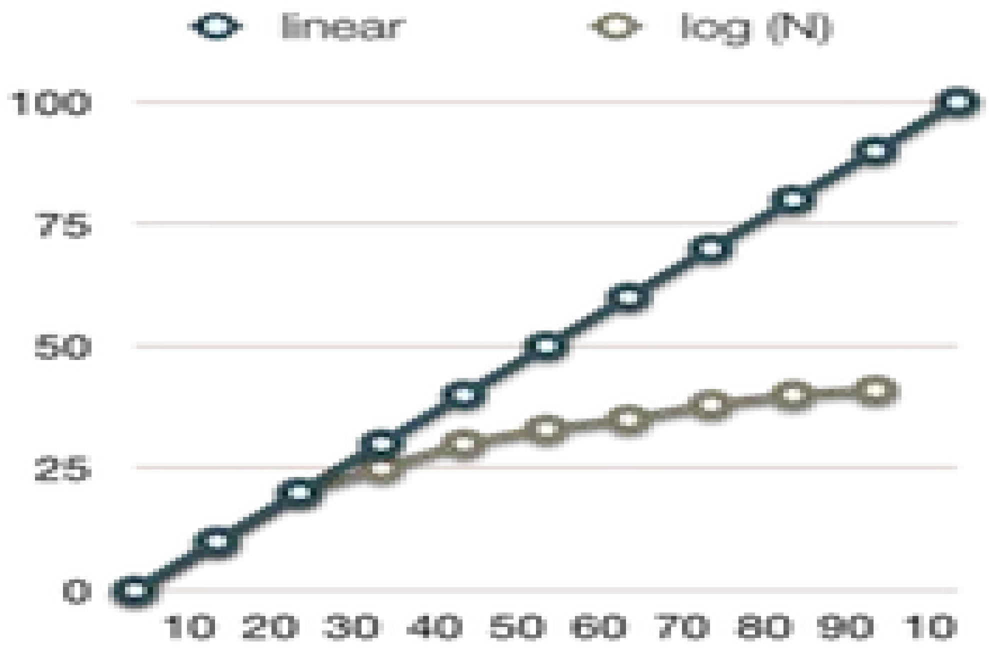


Binary Search (4/4)

Time complexity analysis:

- A single call of BinarySearch() takes O(1) time
- The number of calls of BinarySearch() is O(log n) in the worst case
- Therefore, the time complexity of the binary search is O(log n)

Linear Time vs Logarithmic Time



A logarithmic time algorithm is much faster than a linear time one

Computing Powers (1/3)

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to an power function that runs in O(n) time (for we make n recursive calls).
- We can do better than this, however.

Computing Powers (2/3)

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$

Computing Powers (3/3)

Time complexity analysis:

- Each call of Power() takes O(1) time
- There are O(log n) calls
- Time complexity: O(log n)

```
Algorithm Power(x, n)
 {
  Input : A number x and integer n = 0
  Output : The value x^n
   if  n = 0 return 1
   if  n is odd
      { y = Power(x, (n - 1)/ 2)
        return x*y*y }
   else
      { y = Power(x, n/ 2)
        return y*y }
 }
```

Computing Fibanacci Numbers (1/3)

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- As a recursive algorithm (first attempt):

Algorithm BinaryFib(k)
{ **Input :** Nonnegative integer k
Output : The kth Fibonacci number F_k
if (k =0 or 1) **return** k;
else
return BinaryFib(k - 1) + BinaryFib(k - 2);
}

Computing Fibanacci Numbers (2/3)

- Let n_k denote number of recursive calls made by BinaryFib(k). Then
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that the value at least doubles for every other value of n_k. That is, $n_k > 2^{k/2}$. It is exponential!
- Time complexity: $O(2^k)$

Computing Fibanacci Numbers (3/3)

To compute F_k's only once, we remember the value of each F_k:

```
Algorithm LinearFibonacci(k):
  {  Input : A nonnegative integer k
     Output : Pair of Fibonacci numbers (F_k, F_k-1)
      if  ( k = 1) return (k, 0);
      else
        {
          (i,  j)  =  LinearFibonacci(k - 1);
          return (i +j, i);
        }
   }
```

Time complexity: O(k)

* Space Complexity Analysis for Recursive Algorithms (1/9)

- In general, space complexity analysis is easier than time complexity analysis.
- The hard part in analyzing the space complexity of a recursive algorithm is in the stack space complexity.
- We need to understand how a recursive method is executed on computers.
- Key concept: stack frame or activation record.

* Space Complexity Analysis for Recursive Algorithms (2/9)

- A stack frame for a function stores the local variables, some parameters, the return address, and some other stuff such as values of some registers.
- A stack frame is created in the stack space whenever a function is called.
- A stack frame is freed when the function returns.
- The fame size of each frame can be determined at compile time.

* Space Complexity Analysis for Recursive Algorithms (3/9)

- A call graph is a weighted directed graph $G = (V, E, W)$ where
  - $V=\{v_1, v_2, \ldots, v_n\}$ is a set of nodes each of which denotes an execution of a function;
  - $E=\{v_i \rightarrow v_j: v_i \text{ calls } v_j\}$ is a set of directed edges each of which denotes the caller-callee relationship, and
  - $W=\{w_i\ (i=1, 2, \ldots, n): w_i \text{ is the frame size of } v_i\}$ is a set of stack frame sizes.
- The maximum size of stack space needed for method calls can be derived from the call graph.

* Space Complexity Analysis for Recursive Algorithms (4/9)

- How to compute the maximum size of stack space needed for a method call?

  Step 1: Draw the call graph.

  Step 2: Find the longest weighted path in the call graph.

  The total weight of the longest weighted path is the maximum stack size needed for the function calls.

* Space Complexity Analysis for Recursive Algorithms (5/9)

Assumptions:

- func3() is called 20 times
- Frame sizes (bytes):
  - main(): 10
  - func1(): 20
  - func2(): 60
  - func3(): 80
  - func4(): 10
  - func5(): 30

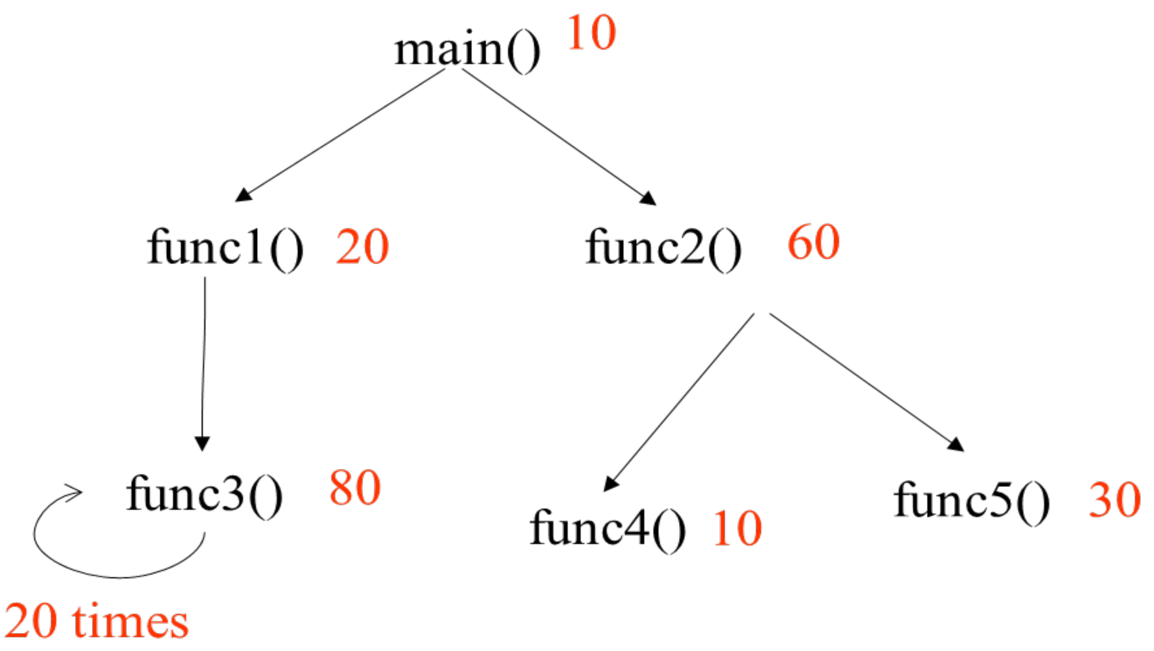
```
int main(void)
{ …
   func1();
   …
   func2();
}

void func1()
 { …
   func3();
   …
  }

void func2()
{ …
  func4();
  …
  func5();
 …
}

int func3()
{ …
   x=func3();
   …
}
```

* Space Complexity Analysis for Recursive Algorithms (6/9)



The longest path is main() → func1() → func3() … → func3() with a length (total weight) of 10+20+80*20=1630. So the maximum stack space needed for main() is 1630 bytes.

* Space Complexity Analysis for Recursive Algorithms (7/9)

The above approach can be generalized to recursive algorithms.

- The frame size of each algorithm is represented by big O.
- Compute the longest path length in terms of big O.

Consider the previous example.

- Assume that the frame sizes of all the methods except func4() are O(1), and the frame size of func4() is O(n). The space complexity of main() is O(n).
- Assume that the frame sizes of all the methods are O(1). The space complexity of main() is O(1).

* Space Complexity Analysis for Recursive Algorithms (8/9)

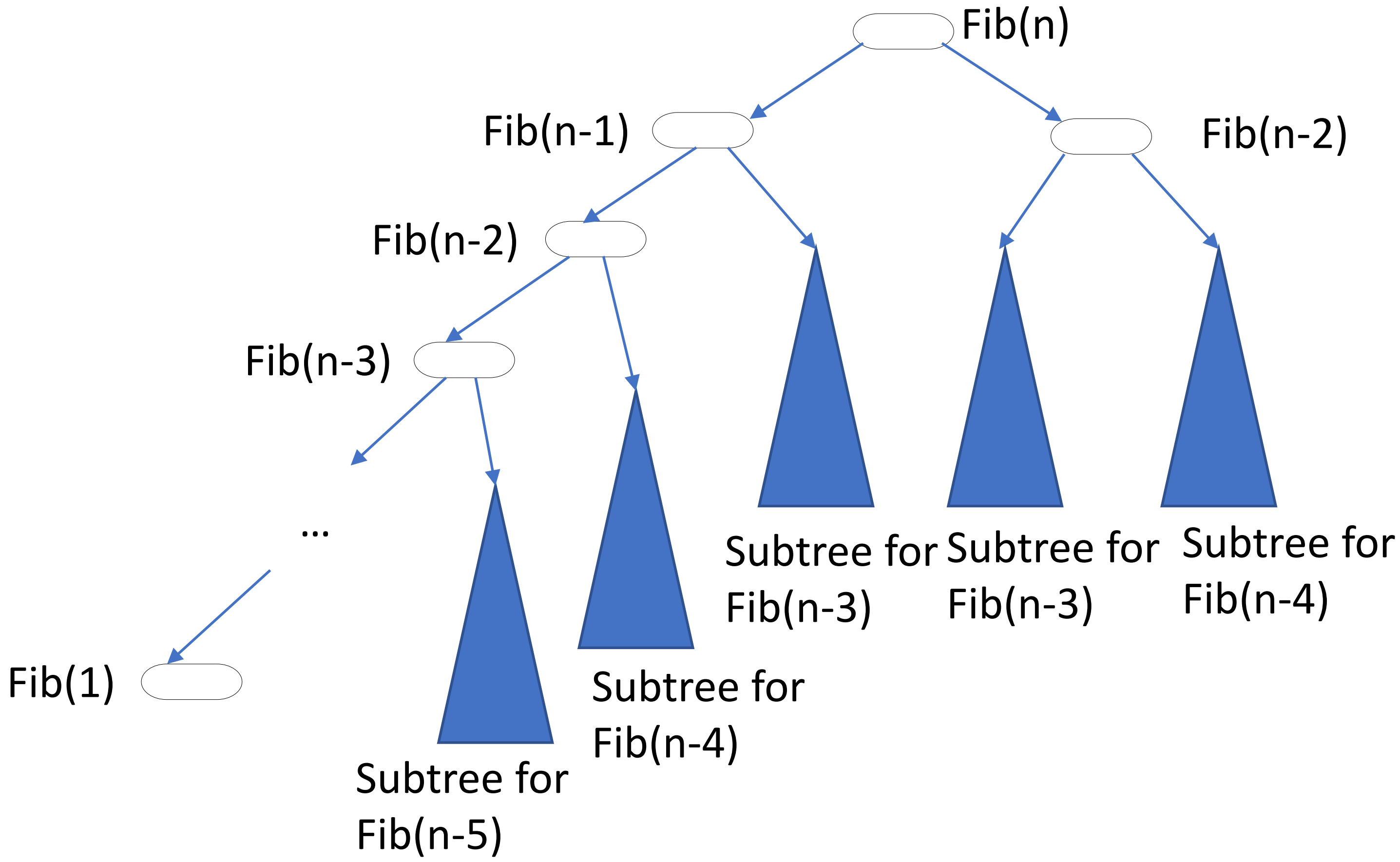
Recursive algorithm for Fibonacci numbers:

```
Algorithm Fib(k)
  { Input : Nonnegative integer k
    Output : The kth Fibonacci number F_k
     if ( k =0 or 1) return k;
     else
      return Fib(k - 1) + Fib(k - 2);
  }
```

What is the space complexity of Fib(n) in terms of big-O?

* Space Complexity Analysis for Recursive Algorithms (9/9)

What is the space complexity of BinaryFib(n) in terms of big-O?



The space complexity: the number of node on the longest path * frame size=n*c=O(n)

Math you need to Review

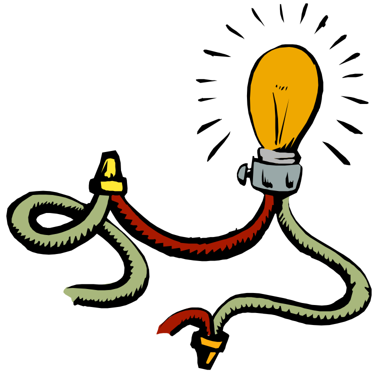
- Summations
- Logarithms and Exponents
  - **properties of logarithms:**
    - $\log_b(xy) = \log_b x + \log_b y$
    - $\log_b (x/y) = \log_b x - \log_b y$
    - $\log_b x^a = a\log_b x$
    - $\log_b a = \log_x a / \log_x b$
  - **properties of exponentials**:
    - $a^{(b+c)} = a^b a^c$
    - $a^{bc} = (a^b)^c$
    - $a^b/a^c = a^{(b-c)}$
    - $b = a^{\log_a b}$
    - $b^c = a^{c*\log_a b}$
- Proof techniques
- Basic probability

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 $f(n) \geq c \bullet g(n)$ 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 $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ 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)

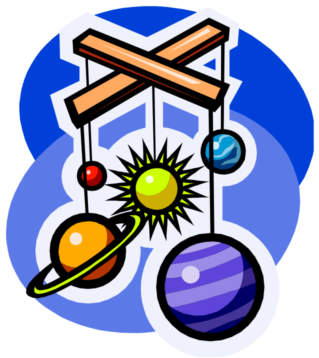
Big-Omega

- 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 $f(n) \geq c \cdot g(n)$ 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 $\mathrm{f}(n) \geq c \cdot g(n)$ 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 $\mathrm{f}(n) \leq c \cdot g(n)$ for $n \geq n_0$

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

Summary

- Big-Oh, big-theta and big-omega notations
- Asymptotic analysis of algorithms
- Examples of algorithms with logarithmic, linear, polynomial, exponential time complexity
- Suggested reading:
  - Sedgewick, Ch.2.1-2.4,2.6