BLM5106- Advanced Algorithm Analysis and Design

Asymptotic Notations and Basic Efficiency Classes

Asymptotic Notations

- $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

O big oh	upper bound
Ω big omega	lower bound
Θ big theta	average bound

O big oh

f(n) $\in$ O(g(n)) if $\exists$ (+) constant c and non negative integer n_0

$f(n) \le c * g(n) \; \forall \; n \ge n_0$

$f(n) = 2n*3$

$2n*3 \le ?$

$2n*3 \le 10n \quad n \ge 1$

f(n) c g(n)

$f(n) \in O(n)$

O big oh

- $2n*3 \le ?$ $7n$ $n \ge 1$ ok

  $2n+3n$

  $2n*3 \le$ $5n$ $n \ge 1$ ok

  f(n) c g(n) f(n) ϵ O(n)

- $2n*3 \le ?$ $4n^2+3n^2$

  $2n*3 \le$ $7n^2$ $n \ge 1$

  f(n) c g(n) f(n) ϵ O(n^2)

O big oh

- f(n) ϵ O(n) ok
- f(n) ϵ O(n^2) ok
- f(n) ϵ O(2^n) ok

upper bound

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$$

- f(n) ϵ O(log n) not ok

Ω big omega

f(n) $\in \Omega$ (g(n)) if $\exists$ (+) constant c and n_0 non negative integer

f(n) $\geq$ c * g(n) $\forall$ n $\geq n_0$

f(n) = 2n*3

2n*3 $\geq$ 1*n $\forall$ n $\geq$ 1

f(n) c g(n)

2n*3 $\geq$ log n $\forall$ n $\geq$ 1

f(n) $\in \Omega$ (n)

f(n) $\in \Omega$ $(\log n)$

f(n) $\in \Omega$ (n^2) not ok

Ω big omega

f(n) = 2n*3 f(n) ϵ $\Omega(n)$ ok

f(n) ϵ $\Omega(\log n)$ ok

lower bound

upper bound

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

Θ big theta

f(n) $\in \Theta$ (g(n)) if $\exists$ (+) constant c_1, c_2 and n_0 non negative int

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

f(n) = 2n+3

$1*n \leq 2n+3 \leq 5*n$

c_1 g(n) f(n) c_2 g(n)

average bound

upper bound

lower bound

f(n) = Θ(n)

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

Θ big theta

½ n(n-1) ϵ Θ (n^2) ?

Upper bound:

- ½ $n^2 - n/2 \le c_2$*g(n)

n^2

½ $n^2 - n/2 \le$ ½ $*n^2$ $n \ge 0$

c_2

Lower bound:

- ½ $n(n-1)$ = ½ n^2 - ½ $n \ge c_1$*g(n)

n^2

1 ? Not ok
2 ? Not ok
1/2 ? Not ok
1/4 ? Ok

C_1= 1/4
C_2= 1/2
n_0 ?

Θ big theta

- $1/4\ g(n) \le 1/2\ n^2 - 1/2\ n \le 1/2\ g(n)$

$n_0=1$? Not ok

$n_0=2$? Ok (better)

$n_0=3$? Ok

Θ big theta

- $f(n)=4n^2 +5n +4$

$4n^2 +5n +4 \geq n^2$ $\Omega\ (n^2)$

$4n^2 +5n +4 \leq 9n^2$ $O(n^2)$

$n^2 \leq 4n^2 +5n +4 \leq 9n^2$ $\Theta(n^2)$

g(n)

- $f(n)=n^2 \log n + n$

$n^2 \log n \leq n^2 \log n + n \leq 5\, n^2 \log n$

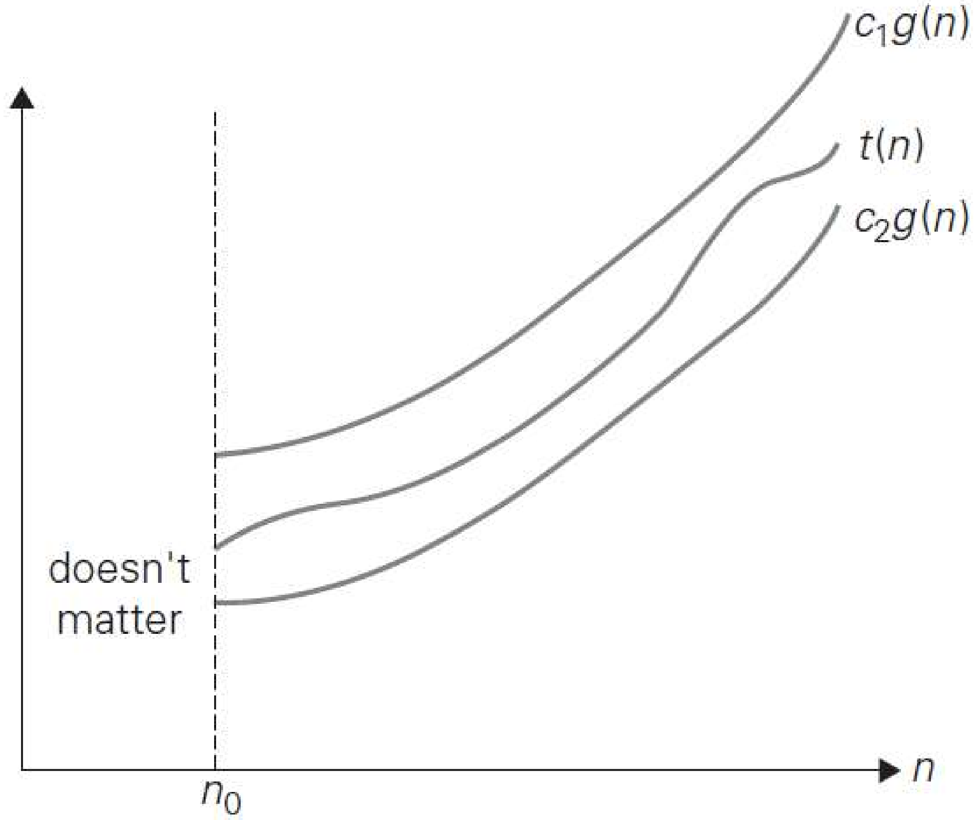
$g(n)$

$\Omega\,(n^2\ logn)$

$O(n^2\ logn)$ $\quad 1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

$\Theta(n^2\ logn)$

Asymptotic Notations



Analyzing algorithms that comprise two consecutively executed parts

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

PROOF a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1$$

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2$$

$$c_3 = \max\{c_1, c_2\} \qquad n \ge \max\{n_1, n_2\}$$

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2\max\{g_1(n), g_2(n)\} \end{aligned}$$

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Analyzing algorithms that comprise two consecutively executed parts

- Check whether an array has equal elements by a two-part algorithm:

  sort the array -> $2\ n(n-1)$ $O(n^2)$

  scan the sorted array -> n-1 $O(n)$

- $O(\max\{n2, n\}) = O(n2)$

- Algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part.
- What will be the space-efficiency class of the entire algorithm?

Using Limits for Comparing Orders of Growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

$t(n) \in \Omega(g(n))$

$t(n) \in O(g(n))$ $\quad$ $t(n) \in \Theta(g(n))$

Using Limits for Comparing Orders of Growth

Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n\to\infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n\to\infty} (1 - \frac{1}{n}) = \frac{1}{2}$$

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

- What about $\displaystyle\lim_{n\to\infty} \frac{n^2}{\frac{1}{2}n(n-1)}$

Using Limits for Comparing Orders of Growth

Compare the orders of growth of $n!$ and 2^n

Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n$$

$$\lim_{n\to\infty}\frac{n!}{2^n} = \lim_{n\to\infty}\frac{\sqrt{2\pi n}\,\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty$$

$n! \in \Omega(2^n)$ ok

- Can big-Omega notation preclude the possibility that *n*! and 2*n* have the same order of growth?

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Mathematical Analysis of Nonrecursive and Recursive Algorithms

Mathematical Analysis of Nonrecursive Algorithms

ALGORITHM *MaxElement(A*[0..*n* − 1]*)*

//Determines the value of the largest element in a given array

//Input: An array *A*[0..*n* − 1] of real numbers

//Output: The value of the largest element in *A*

maxval ←*A*[0]

for *i* ←1 **to** *n* − 1 **do**

if *A*[*i*]>*maxval*

maxval←*A*[*i*]

return *maxval*

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Analyzing the Time Efficiency of Nonrecursive Algoritms

1. Decide on a parameter (or parameters) indicating an input’s size.

2. Identify the algorithm’s basic operation.

3. Check whether the number of times the basic operation is executed depends only on the size of an input.

4. Set up a sum expressing the number of times the algorithm’s basic operation is executed.

5. Using standard formulas and rules of sum manipulation, either find a closedform formula for the count or, at the very least, establish its order of growth.

Properties of Logarithms

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_a x^y = y \log_a x$
4. $\log_a xy = \log_a x + \log_a y$
5. $\log_a \dfrac{x}{y} = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \dfrac{\log_b x}{\log_b a} = \log_a b \log_b x$

Combinatorics

1. Number of permutations of an n-element set: $P(n) = n!$
2. Number of k-combinations of an n-element set: $C(n, k) = \frac{n!}{k!(n-k)!}$
3. Number of subsets of an n-element set: 2^n

Important Summation Formulas

1. $\sum_{i=l} 1 = \underbrace{1 + 1 + \cdots + 1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \le u$); $\sum_{i=1} 1 = n$

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

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

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$ $(a \ne 1)$; $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)

8. $\sum^{n} \lg i \approx n \lg n$

Sum Manipulation Rules

1. $\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$

2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \le m < u$

4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Floor and Ceiling Formulas

The *floor* of a real number x, denoted $\lfloor x \rfloor$, is defined as the greatest integer not larger than x (e.g., $\lfloor 3.8 \rfloor = 3$, $\lfloor -3.8 \rfloor = -4$, $\lfloor 3 \rfloor = 3$). The *ceiling* of a real number x, denoted $\lceil x \rceil$, is defined as the smallest integer not smaller than x (e.g., $\lceil 3.8 \rceil = 4$, $\lceil -3.8 \rceil = -3$, $\lceil 3 \rceil = 3$).

1. $x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$

2. $\lfloor x + n \rfloor = \lfloor x \rfloor + n$ and $\lceil x + n \rceil = \lceil x \rceil + n$ for real x and integer n

3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$

4. $\lceil \lg(n + 1) \rceil = \lfloor \lg n \rfloor + 1$

Analyzing the Time Efficiency of Nonrecursive Algoritms

ALGORITHM *UniqueElements(A*[0*..n* − 1]*)*

//Determines whether all the elements in a given array are distinct
//Input: An array *A*[0*..n* − 1]
//Output: Returns "true" if all the elements in *A* are distinct
// and "false" otherwise

for *i* ←0 **to** *n* − 2 **do**
  for *j* ←*i* + 1 **to** *n* − 1 **do**
    if *A*[*i*]= *A*[*j*]
    return false
return true

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Analyzing the Time Efficiency of Nonrecursive Algoritms

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \qquad \sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n - 1 - i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Analyzing the Time Efficiency of Nonrecursive Algoritms

ALGORITHM *MatrixMultiplication(A*[0..*n* − 1*,* 0..*n* − 1]*, B*[0..*n* − 1*,* 0..*n* − 1]*)*

//Multiplies two square matrices of order *n* by the definition-based algorithm

//Input: Two *n* × *n* matrices *A* and *B*

//Output: Matrix *C* = *AB*

for *i* ←0 **to** *n* − 1 **do**

  for *j* ←0 **to** *n* − 1 **do**

    C[*i, j*]←0.0

    for *k*←0 **to** *n* − 1 **do**

      C[*i, j*]←*C*[*i, j*]+ *A*[*i, k*] ∗ *B*[*k, j*]

return *C*

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

Analyzing the Time Efficiency of Nonrecursive Algoritms

ALGORITHM *Binary(n)*

//Input: A positive decimal integer *n*

//Output: The number of binary digits in *n*'s binary representation

count ←1

while *n* > 1 **do**

count ←*count* + 1

n←*n*/2

return *count*

$$\lfloor \log_2 n \rfloor + 1$$

Mathematical Analysis of Recursive Algorithms

ALGORITHM *F(n)*

//Computes $n!$ recursively

//Input: A nonnegative integer n

//Output: The value of $n!$

if $n = 0$ **return** 1

else return $F(n - 1) * n$

Mathematical Analysis of Recursive Algorithms

$$F(n) = F(n-1) \cdot n$$

$$\overbrace{M(n) = \underset{\substack{\text{to compute} \\ F(n-1)}}{M(n-1)} + \underset{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}}{1}}^{\text{Recurrence Relation}} \qquad \text{for } n > 0$$

Number Of Multiplications

Method of Backward Substitutions

$M(n) = M(n-1) + 1$ substitute $M(n-1) = M(n-2) + 1$

$= [M(n-2) + 1] + 1 = M(n-2) + 2$ substitute $M(n-2) = M(n-3) + 1$

$= [M(n-3) + 1] + 2 = M(n-3) + 3$

$$M(n) = M(n-i) + i$$

Since initial condition is specified for *n* = 0, we have to substitute *i* = *n*

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n$$

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.

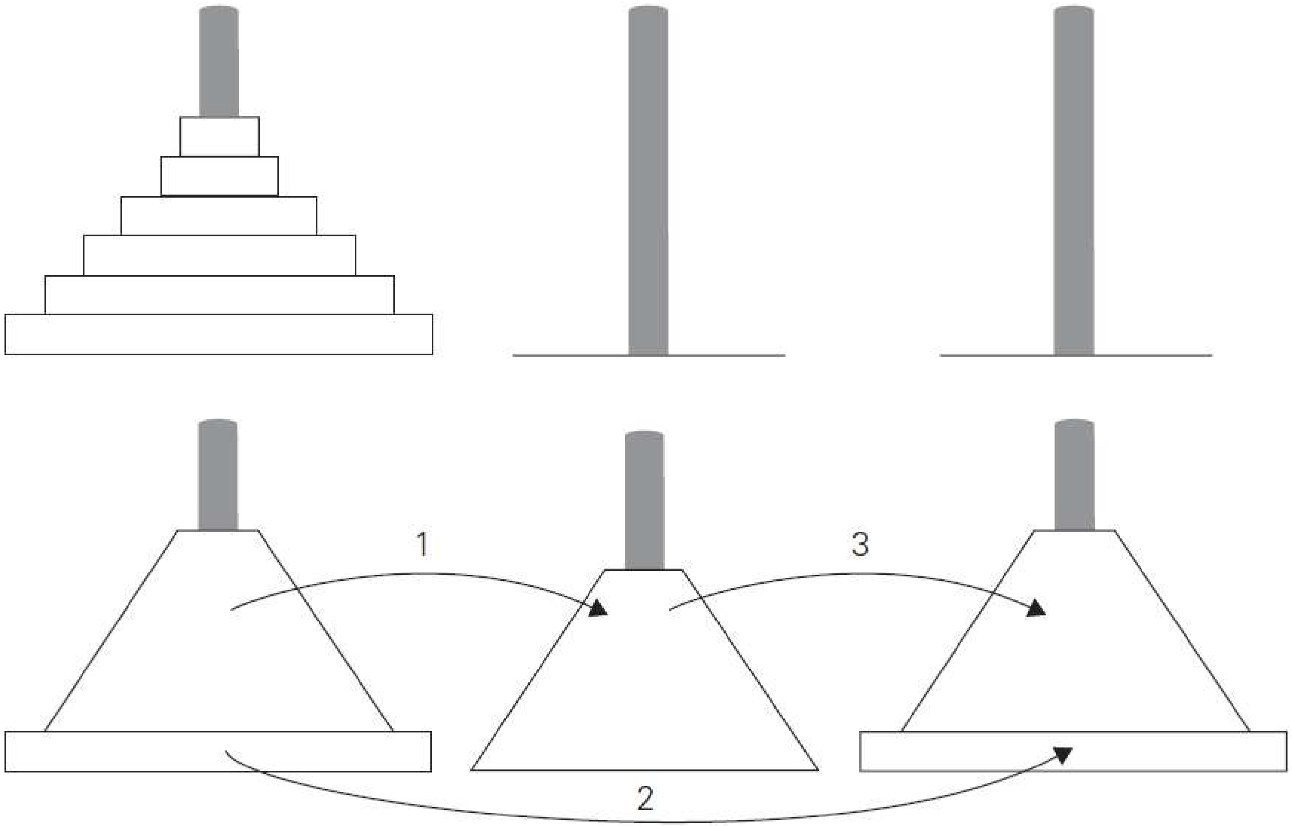
2. Identify the algorithm's basic operation

3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size

4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Tower of Hanoi



Tower of Hanoi

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1,$$
$$M(1) = 1$$

$M(n) = 2M(n-1) + 1$ sub. $M(n-1) = 2M(n-2) + 1$

$= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$

$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1.$

$$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^iM(n-i) + 2^i - 1$$

$n = 1$, which is achieved for $i = n - 1$

$$M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$$

Tree of recursive calls

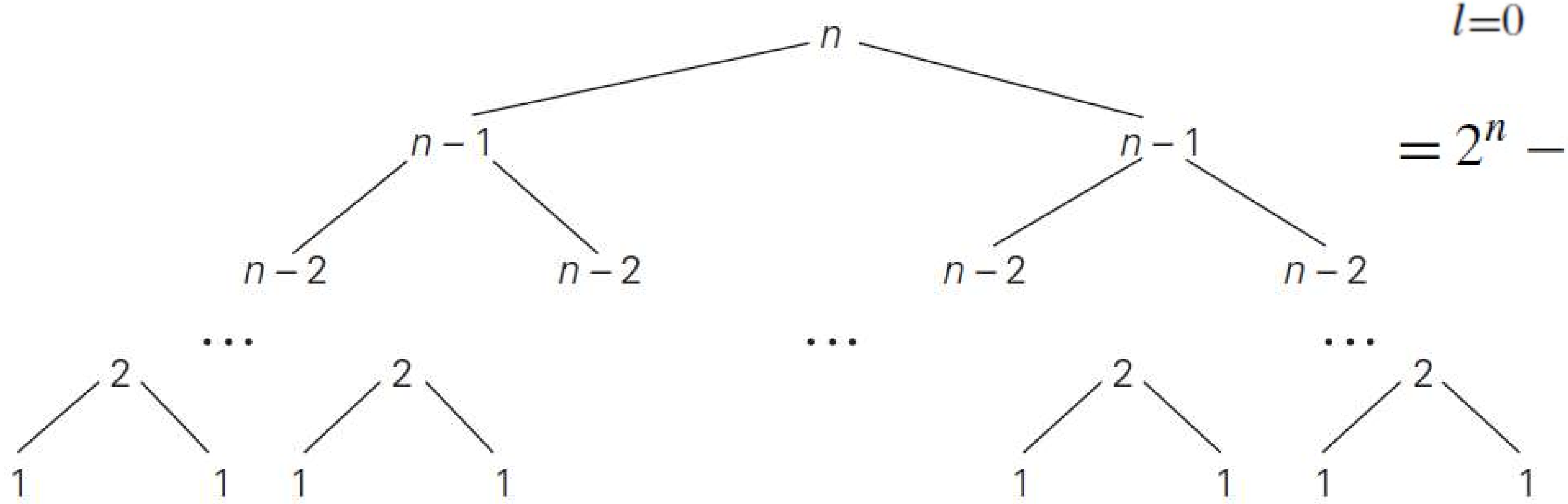
- Nodes correspond to recursive calls

Total number of calls

Level of tree

$$C(n) = \sum_{l=0}^{n-1} 2^l$$

$$= 2^n - 1$$



Binary Digits

ALGORITHM $BinRec(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

The number of additions made   increase the returned value by 1

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

when n is equal to 1 and there are no additions made

$$A(1) = 0$$

$$n = 2^k$$

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$

$$A(2^0) = 0.$$

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \ldots \\
&\ldots \\
&= A(2^{k-i}) + i \\
&\ldots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

$$A(2^k) = A(1) + k = k,$$

$$n = 2^k \quad k = \log_2 n,$$

$$A(n) = \log_2 n \in \Theta(\log n)$$

Fibonacci numbers

$$0,\quad 1,\quad 1,\quad 2,\quad 3,\quad 5,\quad 8,\quad 13,\quad 21,\quad 34,\ \ldots$$

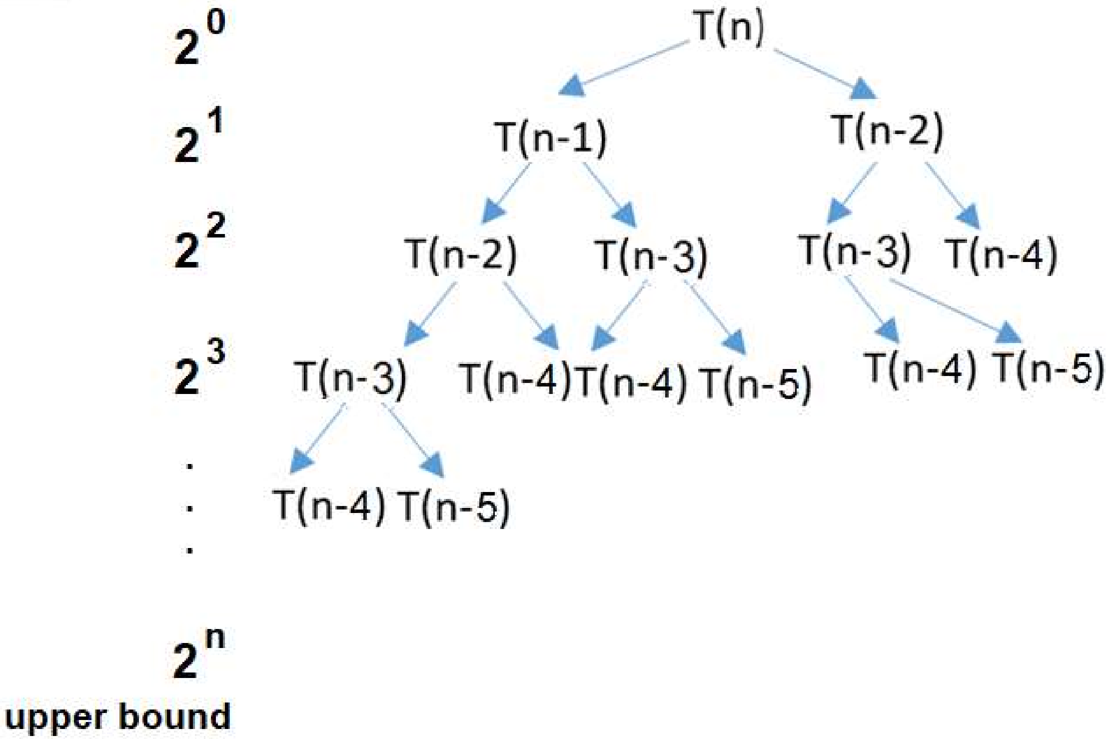
$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1$$

$$F(0) = 0, \qquad F(1) = 1.$$

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \le 1$ **return** n
else return $F(n-1) + F(n-2)$

Fibonacci numbers



$$T(n) = \sum_{i=0}^{n-1} 2^i = O(2^n)$$

upper bound

T(n)= Θ(golden_ratio^n)

$$\text{golden ratio} = \left(\frac{1+\sqrt{5}}{2}\right) \approx 1.618$$

$$\left(\frac{1+\sqrt{5}}{2}\right)^n < 2^n$$

Exercises

- Compare order of growths of the given functions
- $n(n + 1)$ and $2000n^2$? n^2 $2000\,n^2$ same
- $100n^2$ and $0.01n^3$? n^2 n^3 quadratic and qubic
- $\log_2 n$ and $\ln n$?

$$log_b a = \frac{log_x a}{log_x b}$$

$$log_2 n = \frac{lnn}{ln2}$$
$$= \frac{1}{ln2} \cdot lnn$$

$$log_2 n = \tfrac{1}{ln2} \cdot lnn \approx lnn$$

- $(n - 1)!$ and $n!$? $n! = n^* (n-1)!$ n! has a higher order of growth

Exercises

Find the order of growth of the following sums. Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

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

$$\sum_{i=1}^{n} i^k \approx \frac{1}{k+1} n^{k+1}$$

$$\begin{aligned}
\sum_{i=0}^{n-1}(i^2+1)^2 &= \sum_{i=0}^{n-1}(i^4+2i^2+1) \\
&\approx \sum_{i=0}^{n-1} i^4 = \sum_{i=1}^{n} i^4 + 0^4 - n^4 \\
&= \sum_{i=1}^{n} i^4 - n^4 \\
&\approx \frac{1}{4+1} n^{4+1} - n^4 \\
&= \frac{1}{5} n^5 - n^4 \\
&\approx \frac{1}{5} n^5 \in \Theta(n^5)
\end{aligned}$$

Exercises

ALGORITHM *Mystery(n)*
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
$S \leftarrow S + i * i$
return S

What does this algorithm compute?
What is its basic operation?
How many times is the basic operation executed?
What is the efficiency class of this algorithm?
Possible improvement?

Exercises

$$C(n) = \sum^{n} 1 = n$$

$$C(n) = n \in \Theta(n)$$

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

$$\Theta(1)$$

ALGORITHM *Secret*($A[0..n-1]$)

```
//Input: An array A[0..n − 1] of n real numbers
minval ← A[0]; maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] < minval
        minval ← A[i]
    if A[i] > maxval
        maxval ← A[i]
return maxval − minval
```

What does this algorithm compute?
What is its basic operation?
How many times is the basic operation executed?
What is the efficiency class of this algorithm?
Possible improvement?

Exercises

$$C(n) = \sum_{i=1}^{n-1} 2 = 2(n-1)$$

$$C(n) = 2(n-1) = 2n - 2 \approx 2n \in \Theta(n)$$

Exercises

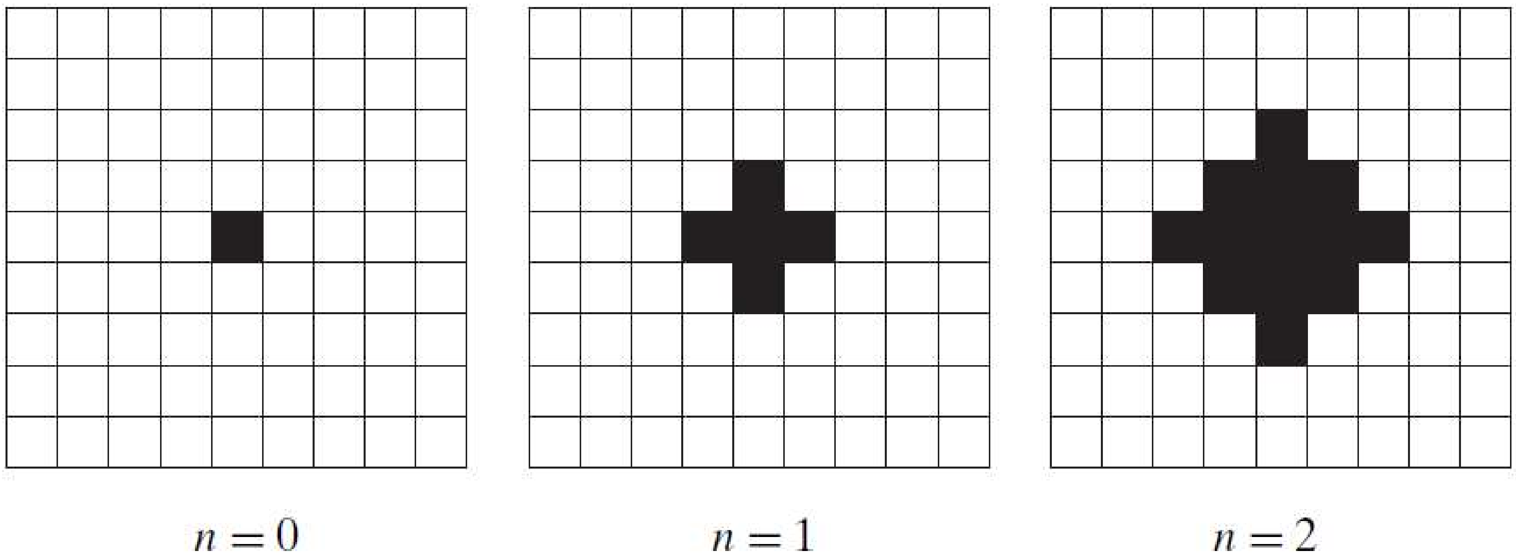
ALGORITHM *Enigma*($A[0..n-1, 0..n-1]$)
//Input: A matrix $A[0..n-1, 0..n-1]$ of real numbers
for $i \leftarrow 0$ **to** $n-2$ **do**
for $j \leftarrow i+1$ **to** $n-1$ **do**
if $A[i, j] = A[j, i]$
return false
return true

What does this algorithm compute?
What is its basic operation?
How many times is the basic operation executed?
What is the efficiency class of this algorithm?
Possible improvement?

Exercises

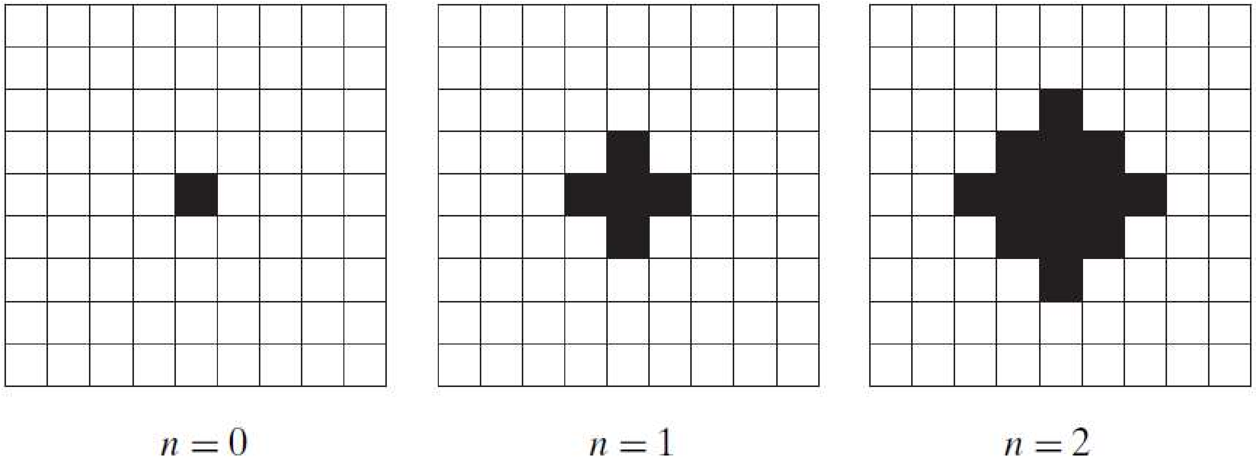
$$
\begin{aligned}
&= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{n-2} \Big[(n-1) - (i+1) + 1\Big] \\
&= \sum_{i=0}^{n-2} \Big[n - i - 1\Big] \\
&= (n-1) + (n-2) + ... + (n - (n-2) - 1) \\
&= (n-1) + (n-2) + ... + 1 \\
&= \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}
\end{aligned}
$$

Exercises



- How many one-by-one squares are there after *n* iterations?
- What about time complexity?

Exercises



- $C(0)=1$
- $C(1)=5=1+4=1+4*1=C(0)+4*1$
- $C(2)=13=5+8=5+4*2=C(1)+4*2$
- $C(3)=25=13+12=13+4*3=C(2)+4*3$

$C(n)=C(n-1)+4*n$, for all $n \geq 0$, $C(0)=1$

Exercises

$C(n)=C(n-1)+4*n$

$C(n)= C(n-2)+4*(n-1)+4*n$

$C(n)= C(n-3)+4*(n-2)+4*(n-1)+4*n$ ..

$C(n)=C(n-i)+4*(n-i+1)+4*(n-i+2)+..4*n$

$n-i=0$, $n=i$

$C(n)=C(0)+4*1+4*2+..+4*n$

$C(n)=1+4+4*2+..+4*n$

$C(n)=1+4(1+2+..+n)$

$=1+2n*(n+1)$

$=2n^2+2n+1$

$C(n-1)=C(n-2)+4(n-1)$

$C(n-2)=C(n-3)+4(n-2)$