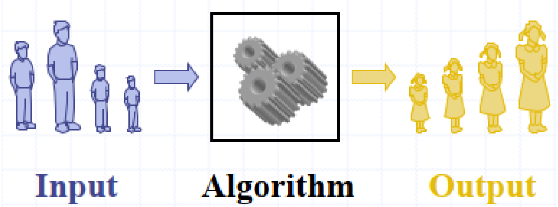
Analysis of an Algorithm

Slides based on

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein: Introduction to Algorithms
- R. Sedgewick and K. Wayne: Algorithms

Algorithm

- What is an algorithm?
  - A step-by-step procedure for solving a problem in a finite amount of time.



- Questions of interest while executing the program/code of an algorithm:
  - How long will the program take to produce the output?
    - Will it produce an output? (Will it halt/terminate?)
    - Will the output be correct?

    Design of Algorithm
  - Will the execution run out of memory?

Aim for analyzing an algorithm

- *What is the goal of analysis of algorithms?*
  - To measure the resources required to run an algorithm (on a computer)
  - To compare algorithms mainly in terms of **running time** but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- *What do we mean by running time analysis?*
  - Determine how running time increases as the $\underbrace{\text{size}}_{\text{input size}}$ of the problem increases.

Input Size

- Input size (number of elements in the input)

  n is reserved for input size

  - size of an array
  - degree of a polynomial
  - # of elements in a matrix
  - # of bits in the binary representation of the input
  - vertices and edges in a graph

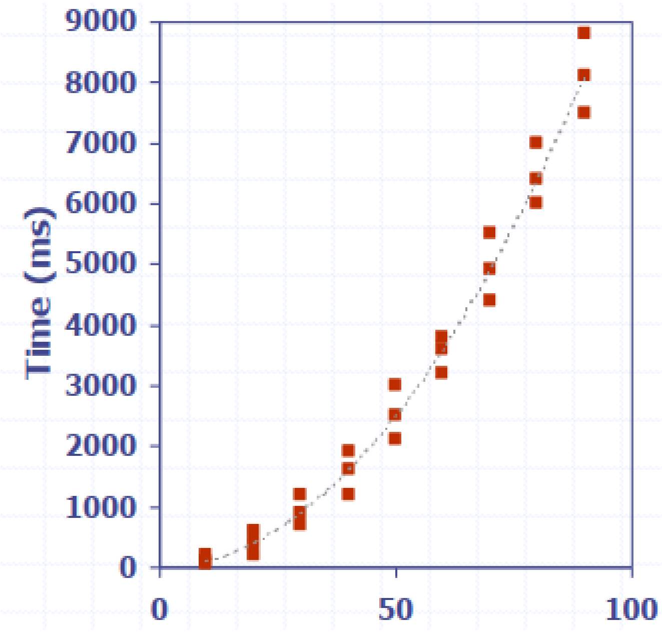
Helps come up with an objective measure to analyse an algorithm.

Experimental/Scientific Analysis

Observe -> Hypothesize -> Predict -> Verify -> Validate -> hypothesize -> predict …

Write a program for the algorithm

- Run the program with inputs of varying size and composition
- Use a method to get an accurate measure of the actual running time
- Plot the result



$T(n) \approx 0.9n^2$

Experimental Analysis - Example

Problem: 3-SUM Given n distinct integers, how many triples sum to zero?

Brute force algorithm: Assume that the input is an array A of size n.

```
Triples ← 0
For i = 1 to n-2
    For j = i+1 to n-1
        For k = j+1 to n
            If A[i]+A[j]+A[k] = 0 then
                Triples ← Triples + 1
Print Triples
```

Instance: 30, -40, -10, -20, 40, 0, 10, 5

o/p: 4

Tuple #	Tuple	Sum
1	30, -40, -10	-20
2	30, -40, -20	-30
3	30, -40, 40	30
4	30, -40, 0	-10
5	30, -40, 10	0
6	30, -40, 5	-5
7	30, -10, -20	0
8	30, -10, 40	60
…		
	-40, 40, 0	0
…		
	-10, 0, 10	0
…		
56	0, 10, 5	15

Experimental Analysis - Example

A Java Program for the algorithm:

```
Public class ThreeSum
{
Public static int Triples(int[], A)
{
int n = A.length;
int Triples = 0;
For (int i=0; i<n-3; i++)
   For (int j=i+1; j<n-2; j++)
     For (int k=j+1; k<n-1; k++)
        If (A[i]+A[j]+A[k] == 0)
           Triples++;
Return Triples;
}
```

```
public static void
Main(string[] args)
{
In in=new In(args[0]);
int[]A=in.readAllInts();
Stdout.Println(Triples(A));
}
}
```

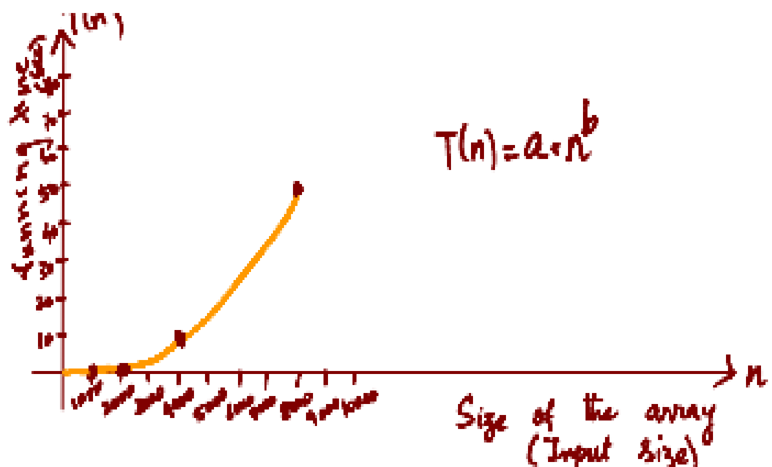
Experimental Analysis - Example

On a 2.8 GHz Intel PU-226 with 64GB DDR E3 memory and 32MB L3 cache, running Oracle Java 1.7.0_45-b18 on Springdale Linux v.6.5

Observations

n	Time (seconds)
250	0
500	0
1000	0.1
2000	0.8
4000	6.4
8000	51.1

Data analysis



Hypothesis: $T(n) = an^b$

Estimating a, b using straight line fit (linear regression) of the log-log plot yields

$\log_2(T(n)) = 2.999\log_2(n) +(-33.21)$

So, running time;

$T(n) = 1.006 \times 10^{-10} \times n^{2.999}$ seconds

Predictions:

For n=8000, it would take 51.0 seconds

For n=16000, it would take 408.1 seconds

Validation: Observations

n	Time (seconds)
8000	51.1
8000	51.0
8000	51.1
16000	410.8

Validates the hypothesis

Estimating a, b using input size doubling technique of the power-law yields

$b = 3$, $a = 0.998 \times 10^{-10}$

So, $T(n) = 0.998 \times 10^{-10} \times n^3$ seconds

Experimental Algorithmics

System independent effects: algorithm, input (determines b in an^b)

System dependent effects: hardware, software, system (determines a in an^b)

==

Good news: experiments are easier and cheaper than other sciences

Bad news: sometimes difficult to get accurate measurements,

in order to compare two algorithms, the same hardware, software environments have to be used,

results may not be indicative of the running time on other inputs not included in the experiment,

it is necessary to code/implement the algorithm, which may be difficult

Number of statements as a measure

- Write a program for the algorithm
- Count the number of statements executed for each input

- **Not good**: number of statements vary with the programming language as well as the style of the individual programmer

Theoretical/Mathematical Analysis

- *Uses a high-level description (pseudocode) of the algorithm [No programming/coding]*

- *Takes into account all possible inputs – makes way for best case, worst case, average case analysis*

- *Allows us to evaluate the speed of an algorithm independent of the hardware/software environment*

- *Based on primitive operations*

Types of Analysis

- **Worst case**
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- **Best case**
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$\boxed{\text{Lower Bound} \le \text{Running Time} \le \text{Upper Bound}}$$

- **Average case**
  - Provides a prediction about the running time
  - Assumes that the input is random

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in the pseudocode
- Largely independent of the programming language
- Exact definition is not important
- Assumed to take a constant amount of time
- Example:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method/procedure/function
  - Returning from a method/procedure/function

Analysis using Primitive Operations

- *By inspecting the pseudocode, determine the (maximum) number of primitive operations executed by the algorithm as a function of the input size.*
- **Example:**

Algo1:	# of operations
A[1]←0	1
A[2]←0	1
…	…
A[n-1]←0	1
A[n]←0	1

Total = n

Algo2:	# of operations
For i=1 to n	2n
A[i]←0	n

Total = 3n

Which will run faster?
How to code Algo1?

Analysis using Primitive Operations

- *Estimate the running time by associating a cost/time taken for each primitive operation.*
- ***Example:** Let c_1 be the cost for assignment, c_2 be the cost for addition/increment by 1*

Algo1:	**cost|frequency**
A[1]←0	c_1|1
A[2]←0	c_1|1
…	…
A[n-1]←0	c_1|1
A[n]←0	c_1|1

Total = nc_1

Algo2:	**cost|frequency**
For i=1 to n	(c_1+c_2)|n
A[i]←0	c_1|n

Total = $2c_1n+c_2n$

- Which algorithm runs faster?
- If c_1=7 for machine1 and c_1=1 and c_2=2 for machine2, Algo1 runs on machine1 and Algo2 on machine2 then?
- What is the rate of growth of the two funtions?

Analysis using Primitive Operations

- **Example:** worst case analysis

Assuming that 'f' is the time taken by the fastest primitive operation and 's' is the time taken by the slowest primitive operation, for

Algo2 $3nXs <= T(n) <= 3nXf$

where T(n) is the worst case running time.

Thus, the running time for the algorithm is bounded by two linear functions in n.

Asymptotic analysis

Problem: **1-SUM** Given n distinct integers, how many of them are zero?

Brute force algorithm: Assume that the input is an array A of size n.

```
Single←0
For i=1 to n
    If A[i]=0 then
        Single←Single + 1
Print Single
```

Operation	Cost	Frequency
Initialization	C_1	1
Loop counter	C_2	n
Condition check	C_3	n
Increment	C_4	$\in\{0, 1, \ldots, n\}$

$T(n) = C_1+nC_2+nC_3+nC_4 = C_1+nC'$
$= 1/5 + (4/5)n$, if $C_1=1/5$, $C_2=2/5$, $C_3=1/5$, $C_4=1/5$

$T(n) \sim (4/5)n$
$T(n) = O(n)$

Asymptotic analysis

Problem: 2-SUM Given n distinct integers, how many pairs sum to zero?

Brute force algorithm: Assume that the input is an array A of size n.

Operation	Cost	Frequency
Initialization	C_1	1
Loop counter i	C_2	n
Loop counter j	C_2	$(n-1)+\ldots+1=n(n-1)/2$
Condition check	C_3	$(n-1)+\ldots+1=n(n-1)/2$
Increment	C_4	$\in\{0, 1, \ldots, n(n-1)/2\}$

```
Doubles←0
For i=1 to n-1
 For j=i+1 to n
  If A[i]+A[j]=0 then
  Doubles←Doubles + 1
Print Doubles
```

$T(n) = C_1+(n+n(n-1)/2)C_2 + (n(n-1)/2)C_3 + (n(n-1)/2)C_4 = C_1 + (n(n+1)/2)C_2 + (n(n-1)/2)(C_3+C_4)$

$= 1/5 + (2/5)n^2$, if $C_1=1/5$, $C_2=2/5$, $C_3=1/5$, $C_4=1/5$

$T(n) \sim (2/5)n^2$ $\qquad$ $T(n) = O(n^2)$

Asymptotic analysis

Problem: **3-SUM** Given n distinct integers, how many triples sum to zero?

Brute force algorithm: Assume that the input is an array A of size n.

```
Triples←0
For i=1 to n-2
For j=i+1 to n-1
For k=j+1 to n
If A[i]+A[j]+A[k]=0 then
Triples←Triples+1
Print Triples
```

Operation	Cost	Frequency
Initialization	C_1	1
Loop counter i	C_2	$n-2$
Loop counter j	C_2	$(n-2)(n-1)/2$
Loop counter k	C_2	$(n-3) + 2(n-4) + 2(n-5) +\ldots+ 6 + 4 + 1=(n-3)^2+1$
Condition check	C_3	$n(n-1)(n-2)/(3!)$
Increment	C_4	$\epsilon\{0, 1, \ldots, n(n-1)(n-2)/3!\}$

$T(n) \sim C'n^3$ $\quad\quad$ $T(n) = O(n^3)$

Asymptotic Analysis

System independent effects: algorithm, input

(determines frequency of primitive operations)

System dependent effects: hardware, software, system

(determines cost of primitive operations)

==

Tilda approximation: discard lower order terms

Order-of-growth: discard the leading coefficient too

Rationale:

- When n is large, lower order terms are negligible
- When n is small, we don't care

Asymptotic Analysis

- *To compare two algorithms with running times f(n) and g(n), we need a* ***rough measure*** *that characterizes* ***how fast each function grows.***
- *Use rate of growth*

  (i.e., the shape of the graph of the two functions)
- *Compare functions in the limit, that is,* ***asymptotically!***

  (i.e., for large values of *n*)

Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)

- The low order terms in a function are relatively insignificant for **large** n

  $$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

  i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Asymptotic Notation

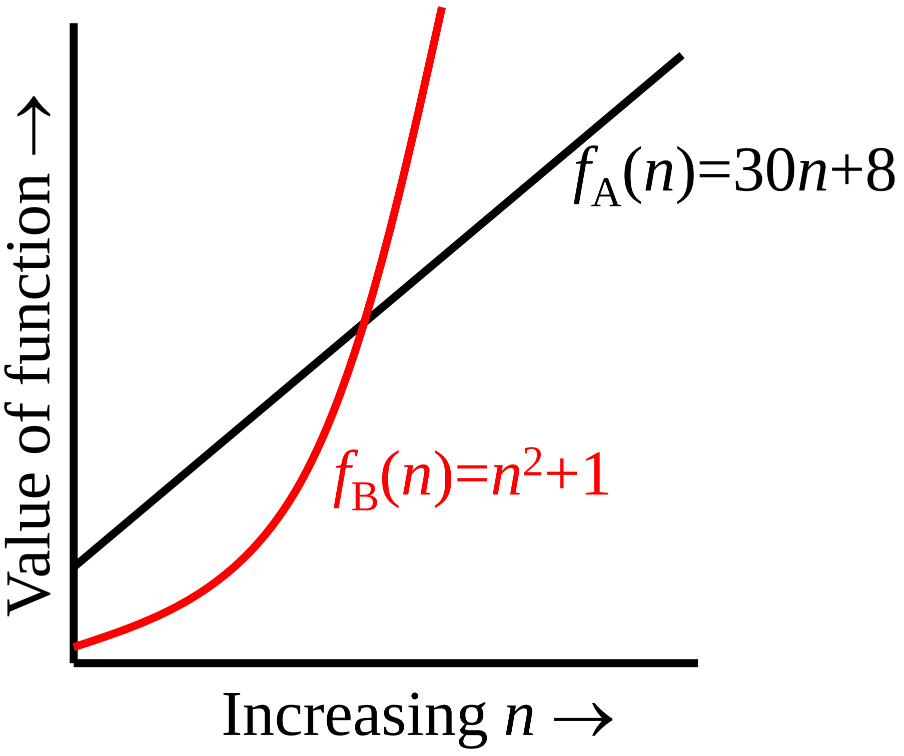
- A way to describe behavior of functions in the limit
  - To indicate running times of algorithms; i.e., to describe the running time of an algorithm as n grows to ∞
- O notation: asymptotic "less than or equal to":
  - $f(n)=O(g(n))$ implies: $f(n)$ "$\leq$" $g(n)$
- Ω notation: asymptotic "greater than or equal to":
  - $f(n)= \Omega\ (g(n))$ implies: $f(n)$ "$\geq$" $g(n)$
- Θ notation: asymptotic "equality":
  - $f(n)= \Theta\ (g(n))$ implies: $f(n)$ "$=$" $g(n)$

Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or $O(n)$ It is, at most, roughly *proportional* to n.
- $f_B(n)=n^2+1$ is *order* n^2, or $O(n^2)$. It is, at most, roughly proportional to n^2.
- In general, any $O(n^2)$ function is faster-growing than any $O(n)$ function.

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



More Examples …

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- constants
  - 10 is $O(1)$
  - 1273 is $O(1)$

Example

- *Algorithm 3* Cost | Frequency

	Cost	Frequency
sum ← 0;	c_1	1
for(i ← 0; i<n; i++)	c_2	n+1
for(j ← 0; j<n; j++)	c_2	n(n+1)
sum ← sum + arr[i][j];	c_3	n x n

$$T(n) = c_1 + c_2(n+1) + c_2 n(n+1) + c_3 n^2$$

$$= O(n^2)$$

Space complexity of an algorithm

- *A posteriori approach: making observations*
- *A priori approach: count the number of variables and weight them by the number of bytes according to their cost* (use asymptotic measures for worst case space needs.)
  - Typical memory requirement for primitive type:
    Boolean=1 byte, Char=2 bytes, int=4 bytes=float,
    long=8 bytes=double, object=16 bytes, references=8 bytes

Note: Take care of passing by pointer and passing by reference.

Note: Suppose a procedure A uses 'a' units of its own space (for its local variables and parameters) and it calls a procedure B which uses 'b' units of space, then

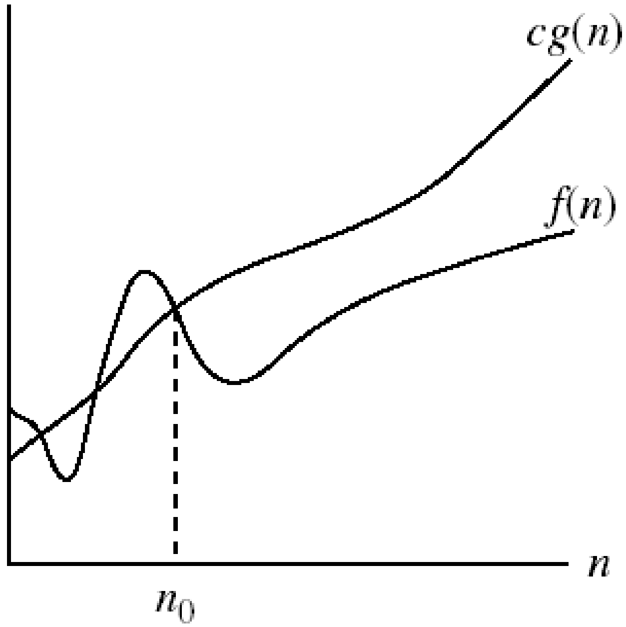
- A overall requires a+b units of space
- If B is called k times then too A requires a+b units of space because when one run of B is completed, its space is freed
- If A calls itself (recursively) then more space is required because freeing-up is limited.

Addendum / Appendix

Asymptotic notations

- *O-notation*

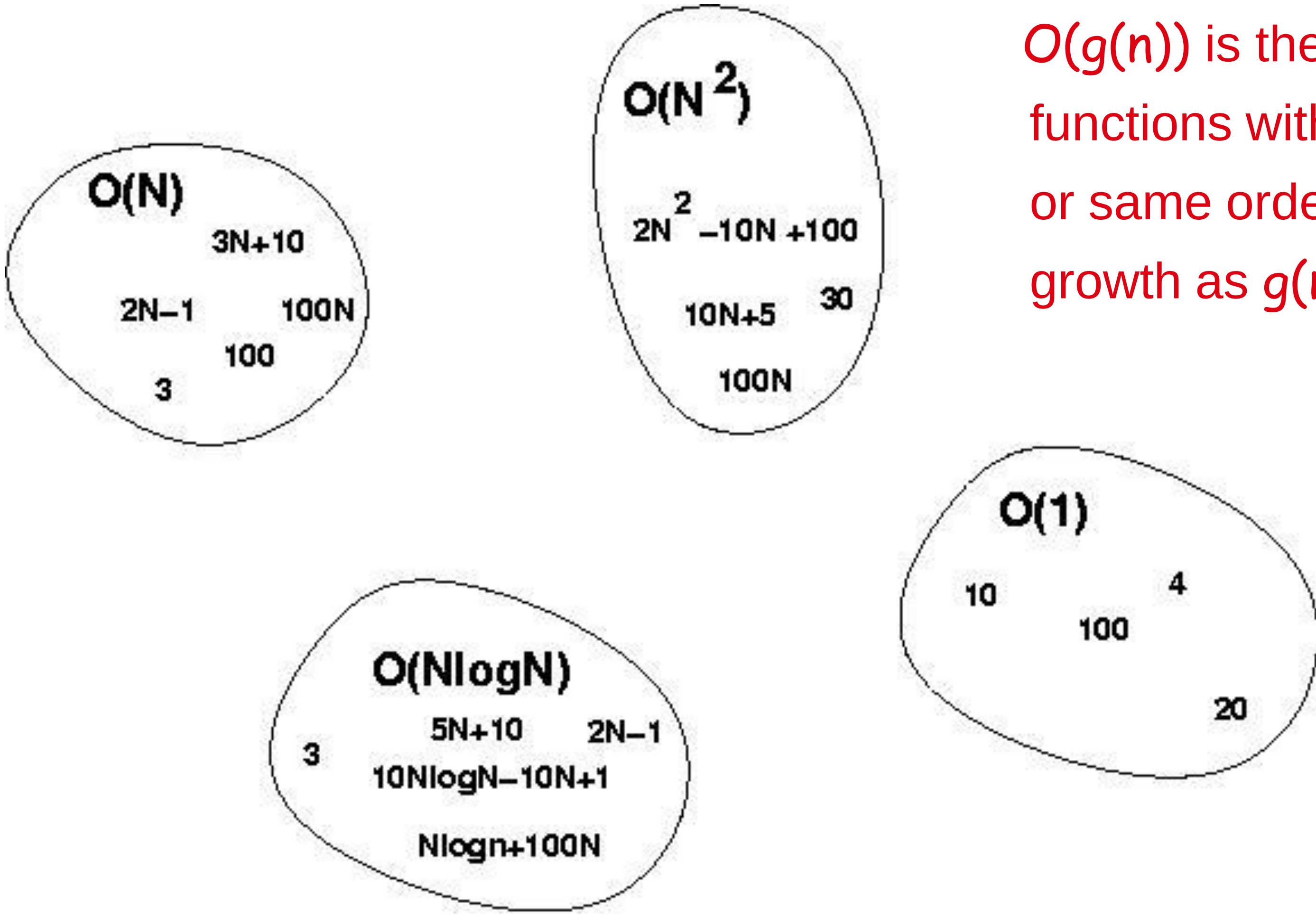
$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



- Intuitively: O(g(n)) = the set of functions with a smaller or same order of growth as g(n)

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O Visualization



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

Examples

- $2n^2 = O(n^3)$: $2n^2 \le cn^3 \Rightarrow 2 \le cn \Rightarrow c = 1$ and $n_0 = 2$

- $n^2 = O(n^2)$: $n^2 \le cn^2 \Rightarrow c \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

- $1000n^2+1000n = O(n^2)$:

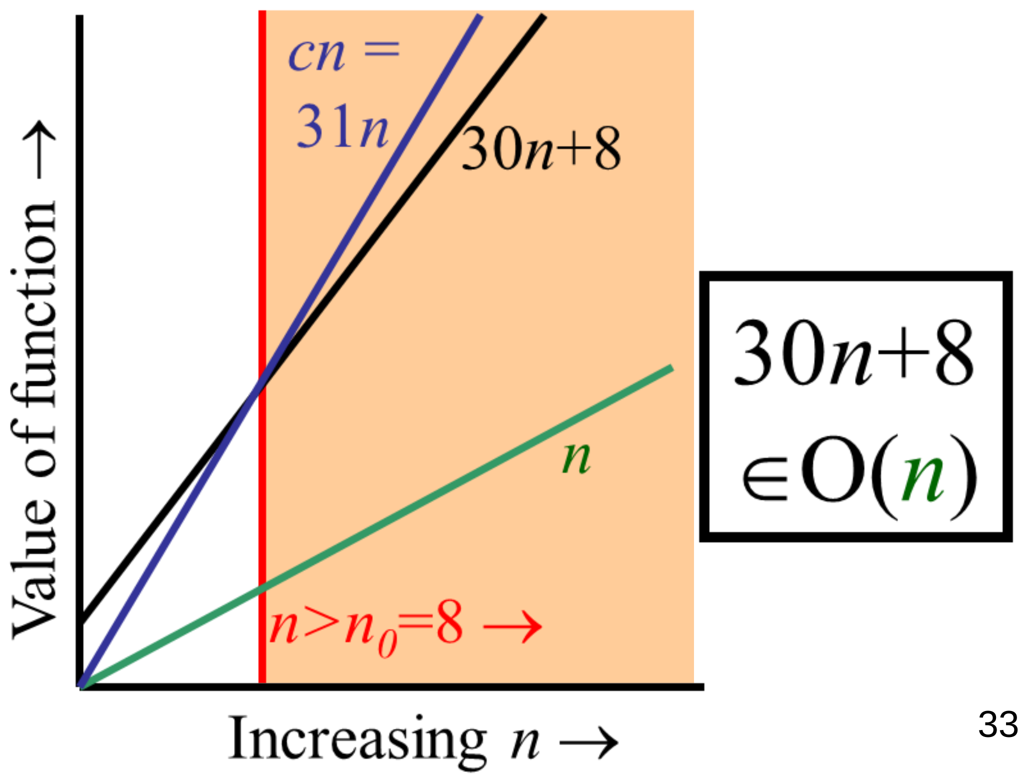
$1000n^2+1000n \le 1000n^2+ n^2 = 1001n^2 \Rightarrow c=1001$ and $n_0 = 1000$

- $n = O(n^2)$: $n \le cn^2 \Rightarrow cn \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

More Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c,n_0$: $30n+8 \leq cn$, $\forall n>n_0$.
    - Let $c=31$, $n_0=8$. Then
      $cn = 31n = 30n + n > 30n+8$ for $n \geq 8$, so $30n+8 < cn$ for $n \geq 8$

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ everywhere to the right of $n=8$.



No Uniqueness

- There is no unique set of values for n_0 and c in proving the asymptotic bounds
- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \le 100n + n = 101n \le 101n^2$

    for all $n \ge 5$

    $n_0 = 5$ and $c = 101$ is a witness
  - $100n + 5 \le 100n + 5n = 105n \le 105n^2$

    for all $n \ge 1$

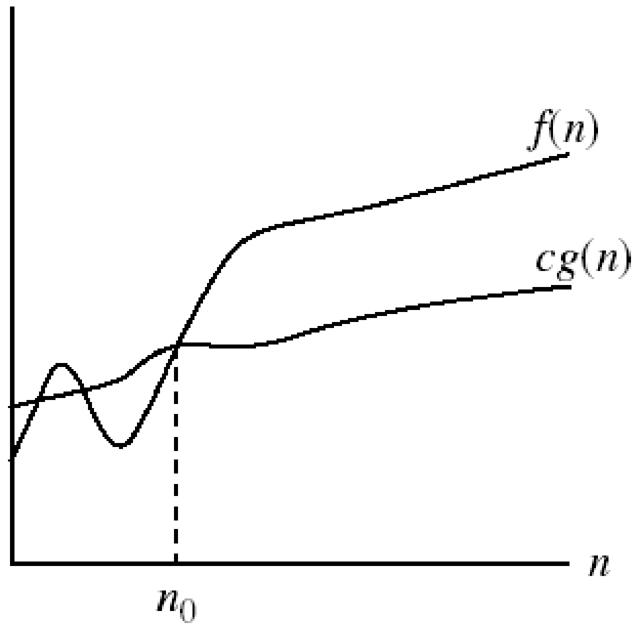
    $n_0 = 1$ and $c = 105$ is also a witness

Must find **SOME** constants c and n_0 that satisfy the asymptotic notation relation

Asymptotic notations (cont.)

- Ω - *notation*

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



- Intuitively: **Ω**(g(n)) = the set of functions with a larger or same order of growth as g(n)

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Examples

- $5n^2 = \Omega(n)$

  $\exists\ c, n_0$ such that: $0 \le cn \le 5n^2 \Rightarrow cn \le 5n^2 \Rightarrow c = 1$ and $n_0 = 1$

- $100n + 5 \ne \Omega(n^2)$

  $\exists\ c, n_0$ such that: $0 \le cn^2 \le 100n + 5$

  $100n + 5 \le 100n + 5n\ (\forall\ n \ge 1) = 105n$

  $cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

  Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

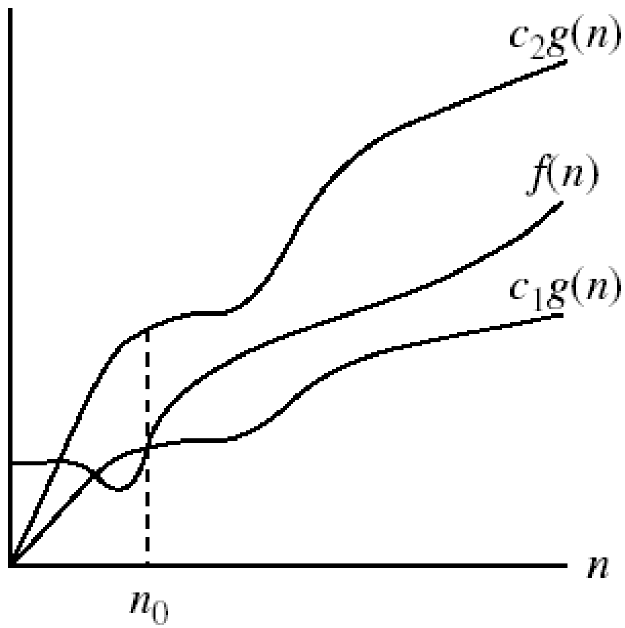
  $\Rightarrow$ contradiction: n cannot be smaller than a constant

- $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$

Asymptotic notations (cont.)

- *Θ-notation*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



- Intuitively Θ(g(n)) = the set of functions with the same order of growth as g(n)
- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Examples

- $n^2/2 - n/2 = \Theta(n^2)$

  - $\frac{1}{2} n^2 - \frac{1}{2} n \le \frac{1}{2} n^2 \ \forall n \ge 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$

  - $\frac{1}{2} n^2 - \frac{1}{2} n \ge \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \ (\forall n \ge 2) = \frac{1}{4} n^2$

    $\Rightarrow \quad c_1 = \frac{1}{4}$

- $n \ne \Theta(n^2)$: $c_1 n^2 \le n \le c_2 n^2$

  $\Rightarrow$ only holds for: $n \le 1/c_1$

Examples

- $6n^3 \neq \Theta(n^2)$: $c_1 n^2 \leq 6n^3 \leq c_2 n^2$

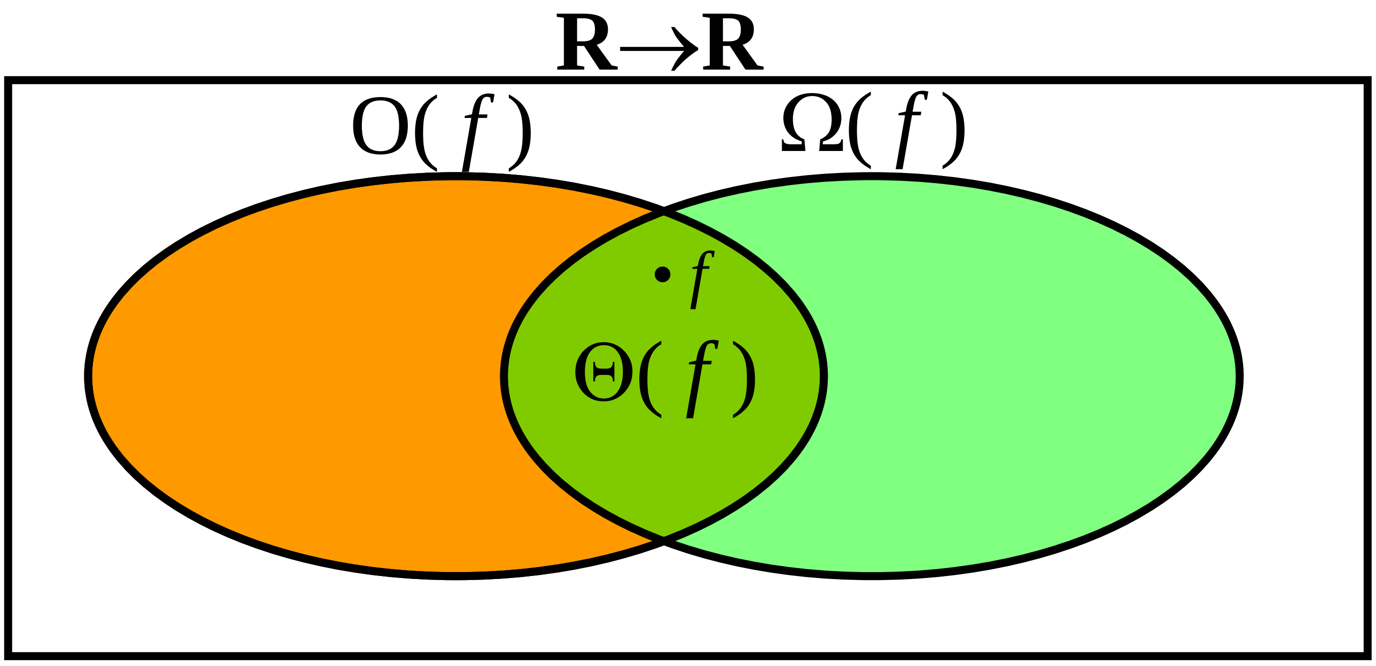
  $\Rightarrow$ only holds for: $n \leq c_2 /6$

- $n \neq \Theta(\log n)$: $c_1 \log n \leq n \leq c_2 \log n$

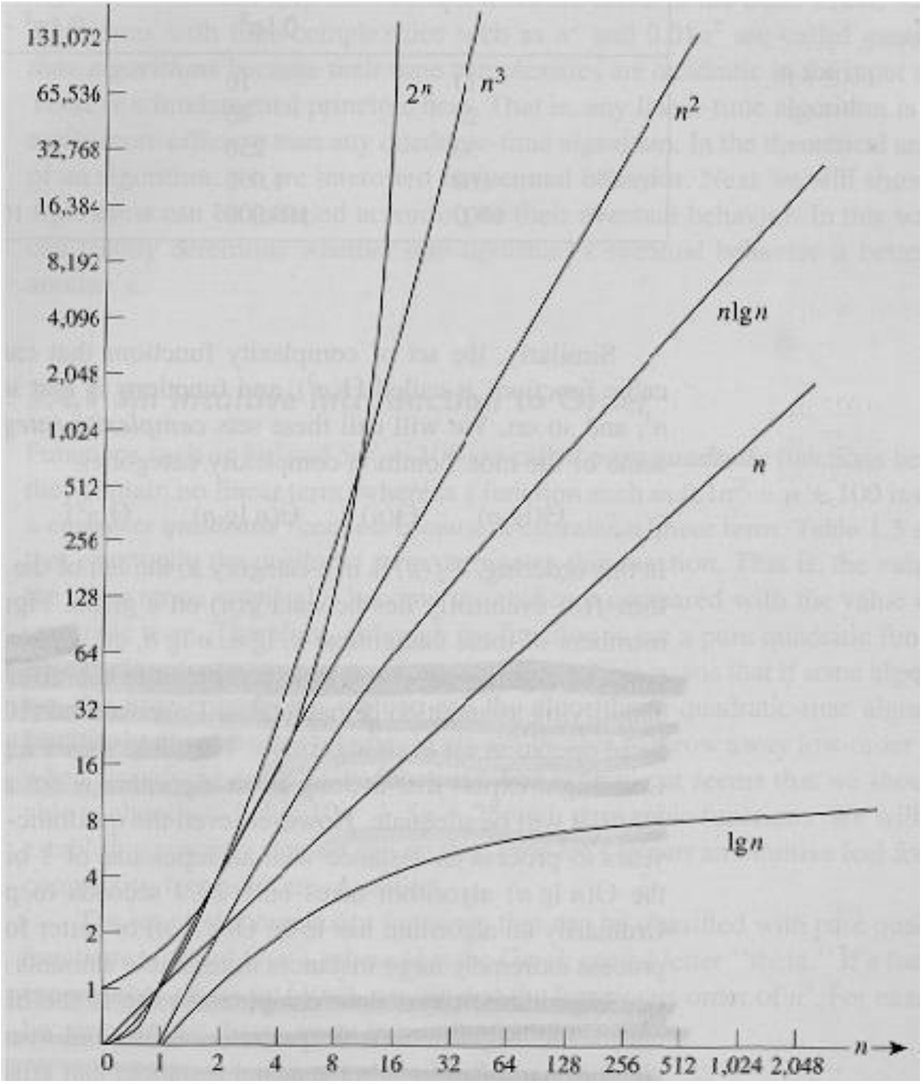
  $\Rightarrow c_2 \geq n/\log n, \forall\, n \geq n_0$ – impossible

Relations Between Different Sets

- Subset relations between order-of-growth sets.



Common orders of magnitude



Common orders of magnitude

Table 1.4 Execution times for algorithms with the given time complexities

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 μs = 10^{-6} second.
†1 ms = 10^{-3} second.

Properties

- *Theorem:*

  $$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Analysing recursive algorithms – Powering a number

- Powering a number
  - Finding a^n for $n \geq 0$

Algorithm Ipower (a, n)

$p \leftarrow 1$

for $i \leftarrow 1$ to n

$p \leftarrow p \times a$

$T(n)=C_1+C_2n+C_3n = \Theta(n)$

Algorithm Rpower (a,n)

If n=0 then Return (1)

Else

$p \leftarrow$ Rpower (a, n-1)

Return (p x a)

$$T(n) = \begin{cases} C_5, & if\ n = 0 \\ T(n-1) + C_5, & otherwise \end{cases}$$

$= \Theta(n)$

Search Algorithms

- Lsearch(x, A)
  - Finding x in a list A of size n

    Algorithm Lsearch (x, A)

    for i $\leftarrow$ 1 to n

    if A_i = x then declare found (and STOP)

    $T(n)=C_2 n + C_4 n = \Theta(n)$

Algorithm Lsearch(x,A)
c$\leftarrow$1
while c $\leq$ n
  if A_c= x then STOP // found
  else c $\leftarrow$ c+1
Declare x not found

  - Finding x in a sorted list A of size n
    - Using linear search, it takes Θ(n) time; not exploiting the information that the list is sorted
    - Comparing x with A_c helps us identify possible location for presence of x in A (Good choice: c = n/2, bad choice: c=1)

Search Algorithms (contd.)

- Binarysearch(x, A): Finding x in a sorted (non-decreasing) list A of size n

Algorithm Binarysearch (x, A[left … right])

if left > right then STOP // not found

else

mid ← left + floor[(right – left)/2]

if A_{mid} = x then STOP // found

elseif A_{mid} < x then

right ← mid-1

Return(Binarysearch(x,A[left … right])

elseif A_{mid} > x then

left ← mid+1

Return(Binarysearch(x,A[left … right])

$$T(n) = \begin{cases} C_6, & if\ n = 1 \\ T\left(\frac{n}{2}\right) + C_7, & otherwise \end{cases} = \Theta(\log_2 n)$$

Recursive Algorithms

Algorithm Recursive(input of size n)

If n < some constant k then

- solve the problem directly // base case

Else

- create subproblems, each of size n/b
- call the algorithm recursively on each of the subproblem // recursive case(s)
- combine the results from the subproblem

$$T(n) = \begin{cases} \boldsymbol{T(base\ case),} & \boldsymbol{if\ n = size\ of\ base\ case} \\ \boldsymbol{number\ of\ subproblems * T\left(\frac{n}{b}\right) + f(n),} & \boldsymbol{otherwise} \end{cases}$$

recurrence relation

Time taken for creating the subproblems and combining the results of the subproblems

Solving recurrence relations for analysis of recursive algorithms

- Substitution method
  - Backward substitution
  - Forward substitution
  - Guess, verify (by induction) and solve for the constants
- Recursion tree method
- The master method
  - Three common cases

Substitution method

- Powering a number

$T(n) = T(n-1) + c_5$

$= (T(n-2) + c_5) + c_5 = T(n-2) + 2c_5$

$= (T(n-3) + c_5) + 2c_5 = T(n-3) + 3c_5$

.

.

.

$= T(0) + nc_5 = c_5 + nc_5 = \Theta(n)$

Aliter:

$T(0) = c_5$

$T(1) = T(0) + c_5 = 2c_5$

$T(2) = T(1) + c_5 = 3c_5$

.

.

.

$T(n) = (n+1)c_5 = \Theta(n)$

Recursion tree method

- $T(n) = T(n-1) + c_5$

$\cancel{T(n)}\ c_5$ ---------------------------------- c_5

$\downarrow$

$\cancel{T(n-1)}\ c_5$ ---------------------------------- c_5

$\downarrow$

$\cancel{T(n-2)}\ c_5$ ---------------------------------- c_5

$\downarrow$

$\cancel{T(1)}\ c_5$ ---------------------------------- c_5

$\downarrow$

$T(0)=c_5$ ---------------------------------- c_5

$(n+1)c_5 = \Theta(n)$

Recursion tree method (contd.)

- $T(n) = T(n/2) + c_7$

$\cancel{T(n)}$ c_7 ---------------------------------- c_7

$\cancel{T(n/2)}$ c_7---------------------------------- c_7

$\cancel{T(n/4)}$ c_7---------------------------------- c_7

$\cancel{T(2)}$ c_7 ---------------------------------- c_7

$T(1)=c_6$---------------------------------- c_6

$T(n) = c_7 + c_7 + \ldots + c_7 + c_6 = c_7 \times \text{ceil}(\log_2 n) + c_6 = \Theta(\log_2 n)$

Recursion tree method (contd.)

- T(n) = T(n/2) + T(n/4) + n

~~T(n)~~ n ------------------------------------ n

~~T(n/2)~~ n/2 ~~T(n/4)~~ n/4------------------------ 3n/4

~~T(n/4)~~ n/4 ~~T(n/8)~~ n/8 ~~T(n/8)~~ n/8 ~~T(n/16)~~ n/16 ------ 9n/16

. .

. .

. .

~~T(base~~) Θ(1)

T(n) = $n\left(1+\frac{3}{4}+\left(\frac{3}{4}\right)^2+\cdots\right)$ = Θ(n)

The master method

- Applicable for recurrences of the form **$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$**
  where $a \geq 1$, $b > 1$, and f is asymptotically positive.
- Common cases on the basis of growth of f(n) being slower than or similar to or faster than the growth of $n^{\log_b a}$ (by an n^{ϵ} factor)

Case 1: **$f(n) = O(n^{\log_b a - \epsilon})$** for some constant $\epsilon > 0$. **$T(n) = \Theta(n^{\log_b a})$**

e.g., **$T(n) = 4T(n/2)+n$**; $a=4$, $b=2$, $f(n)=n$;

So, $n^{\log_b a} = n^2$, $O(n^{\log_b a}) = O(n^{2-\epsilon}) = f(n)=n \leftrightarrow \epsilon=1$ **$T(n) = \Theta(n^2)$**

Case 2: **$f(n) = \Theta(n^{\log_b a} \lg^k n)$** for some constant $k \geq 0$.

$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

e.g., **$T(n) = 4T(n/2)+n^2$**; $a=4$, $b=2$, $f(n)=n^2$; So, $n^{\log_b a} = n^2$,

$\Theta(n^{\log_b a} \lg^k n) = \Theta(n^2 \lg^k n) = f(n)=n^2 \leftrightarrow k=0$ **$T(n) = \Theta(n^2 \lg n)$**

Case 3: **$f(n) = \Omega(n^{\log_b a + \epsilon})$** for some constant $\epsilon > 0$. **$T(n) = \Theta(f(n))$**

e.g., **$T(n) = 4T(n/2)+n^3$**; $a=4$, $b=2$, $f(n)=n^3$; So, $n^{\log_b a} = n^2$,

$\Omega(n^{\log_b a + \epsilon}) = \Omega(n^{2+\epsilon}) = f(n)=n^3 \leftrightarrow \epsilon=1$ **$T(n) = \Theta(n^3)$**

The master method (non applicable case)

- Applicable for recurrences of the form $\mathbf{T(n) = a\ T\left(\frac{n}{b}\right) + f(n)}$
  where $a \geq 1$, $b > 1$, and f is asymptotically positive.
- Common cases on the basis of growth of f(n) being slower than or similar to or faster than the growth of $n^{\log_b a}$ (by an n^{ϵ} factor)
- When the growth of f(n) cannot be compared with the growth of $n^{\log_b a}$, The master method cannot be applied.

e.g., $T(n) = 4T(n/2) + n^2/\lg n$

Since for every constant $\epsilon>0$, $n^{\epsilon} = \omega(\lg n)$, The master method does not apply.

Some frequently appearing asymptotic growth of algorithms

order of growth	name	typical steps in the algorithm	description	example	T(2n)/T(n)
$\Theta(1)$	constant	$a \leftarrow b+c$	statement	add two numbers	1
$\Theta(\log n)$	logarithmic	while n>1 $n \leftarrow n/2$	divide in half	binary search	~1
$\Theta(n)$	linear	for $i \leftarrow 1$ to n $a \leftarrow b+c$	single loop	find the maximum	2

$\Theta(n \log n)$	linearithmic	for $i \leftarrow 1$ to n ↑doubling
		  for $j \leftarrow 1$ to i
		    $a \leftarrow b+c$
			divide and conquer	mergesort	~2

$\Theta(n^2)$	quadratic	for $i \leftarrow 1$ to n
		  for $j \leftarrow 1$ to n
		    $a \leftarrow b+c$
			double loop	check all pairs	4

$\Theta(n^3)$	cubic	for $i \leftarrow 1$ to n
		  for $j \leftarrow 1$ to n
		    for $k \leftarrow 1$ to n
		      $a \leftarrow b+c$
			triple loop	check all triples	8

$\Theta(2^n)$	exponential	Usually recursive (non-iterative)	exhaustive search (combinations)	check all subsets	2^n
$\Theta(n!)$	factorial	Usually recursive	exhaustive search (permutations)	check all permutations	$\sim n^n$

Asymptotic Notations in Equations

- $\Theta(g(n))$ stands for some anonymous function in $\Theta(g(n))$
- On the right-hand side

  $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means:

  There exists a function $f(n) \in \Theta(n)$ such that

  $$2n^2 + 3n + 1 = 2n^2 + f(n)$$
- On the left-hand side

  $2n^2 + \Theta(n) = \Theta(n^2)$

  No matter how the anonymous function is chosen on the left-hand side, there is a way to choose the anonymous function on the right-hand side to make the equation valid.

Logarithms and properties

- In algorithm analysis we often use the notation "log n" without specifying the base

Binary logarithm $\lg n = \log_2 n$

Natural logarithm $\ln n = \log_e n$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

More Examples

- For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct.
  - $f(n) = \log n^2$; $g(n) = \log n + 5$ $\quad f(n) = \Theta(g(n))$
  - $f(n) = n$; $g(n) = \log n^2$ $\quad f(n) = \Omega(g(n))$
  - $f(n) = \log \log n$; $g(n) = \log n$ $\quad f(n) = O(g(n))$
  - $f(n) = n$; $g(n) = \log^2 n$ $\quad f(n) = \Omega(g(n))$
  - $f(n) = n \log n + n$; $g(n) = \log n$ $\quad f(n) = \Omega(g(n))$
  - $f(n) = 10$; $g(n) = \log 10$ $\quad f(n) = \Theta(g(n))$
  - $f(n) = 2^n$; $g(n) = 10n^2$ $\quad f(n) = \Omega(g(n))$
  - $f(n) = 2^n$; $g(n) = 3^n$ $\quad f(n) = O(g(n))$

Common Summations

- Arithmetic series: $$\sum_{k=1}^{n} k = 1 + 2 + ... + n = \frac{n(n+1)}{2}$$
- Geometric series: $$\sum_{k=0}^{n} x^k = 1 + x + x^2 + ... + x^n = \frac{x^{n+1} - 1}{x - 1} \left(x \neq 1\right)$$
  – Special case: $|x| < 1$: $$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$
- Harmonic series: $$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + ... + \frac{1}{n} \approx \ln n$$
- Other important formulas: $$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + ... + n^p \approx \frac{1}{p+1} n^{p+1}$$