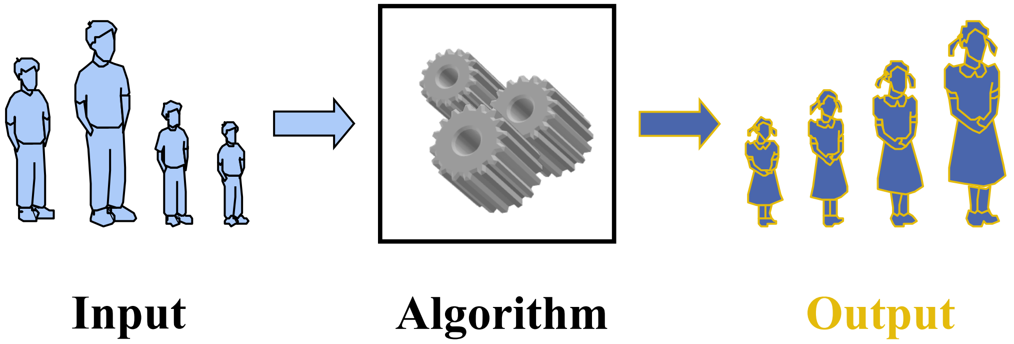
Fundamentals of Algorithms

Unit - I

Algorithm

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



Why to analysis algorithm?

- Writing a working program is not good enough
- The program may be inefficient!
- If the program is run on a large data set, then the running time becomes an issue

Example: Selection Problem

- Given a list of N numbers, determine the *k*th largest, where $k \leq N$.
- Algorithm 1:
  (1) Read N numbers into an array
  (2) Sort the array in decreasing order by some simple algorithm
  (3) Return the element in position k

Example: Selection Problem…

- Algorithm 2:
  (1) Read the first k elements into an array and sort them in decreasing order
  (2) Each remaining element is read one by one
    - If smaller than the kth element, then it is ignored
    - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.

  (3) The element in the kth position is returned as the answer.

Example: Selection Problem…

- Which algorithm is better when
  - N =100 and k = 100?
  - N =100 and k = 1?
- What happens when N = 1,000,000 and k = 500,000?
- There exist better algorithms

Algorithm Analysis

- We only analyze *correct* algorithms
- An algorithm is correct
  - If, for every input instance, it halts with the correct output
- Incorrect algorithms
  - Might not halt at all on some input instances
  - Might halt with other than the desired answer
- Analyzing an algorithm
  - Predicting the resources that the algorithm requires
  - Resources include
    - Memory
    - Communication bandwidth
    - Computational time (usually most important)

Algorithm Analysis…

- Factors affecting the running time
  - computer
  - compiler
  - algorithm used
  - input to the algorithm
    - The content of the input affects the running time
    - typically, the *input size* (number of items in the input) is the main consideration
      - E.g. sorting problem $\Rightarrow$ the number of items to be sorted
      - E.g. multiply two matrices together $\Rightarrow$ the total number of elements in the two matrices
- Machine model assumed
  - Instructions are executed one after another, with no concurrent operations $\Rightarrow$ Not parallel computers

Example

- Calculate

```
int sum(int n)
{
        int partialSum;

1       partialSum=0;                  1
2       for (int i=1;i<=n;i++)         2N+2
3           partialSum += i*i*i;       4N
4       return partialSum;             1
}
```

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time four units
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 6N + 4 $\Rightarrow$ O(N)

Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion?
  3) How much of computer memory does it use?
- Software engineering is primarily concerned with the first criteria
- In this course we are interested in the second and third criteria

Comparing Algorithms

- Time complexity
  - The amount of time that an algorithm needs to run to completion
- Space complexity
  - The amount of memory an algorithm needs to run
- We will occasionally look at space complexity, but we are mostly interested in time complexity in this course
- Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)

How to Calculate Running time

- Most algorithms transform input objects into output objects

- The running time of an algorithm typically grows with the input size
  - idea: analyze running time as a function of input size

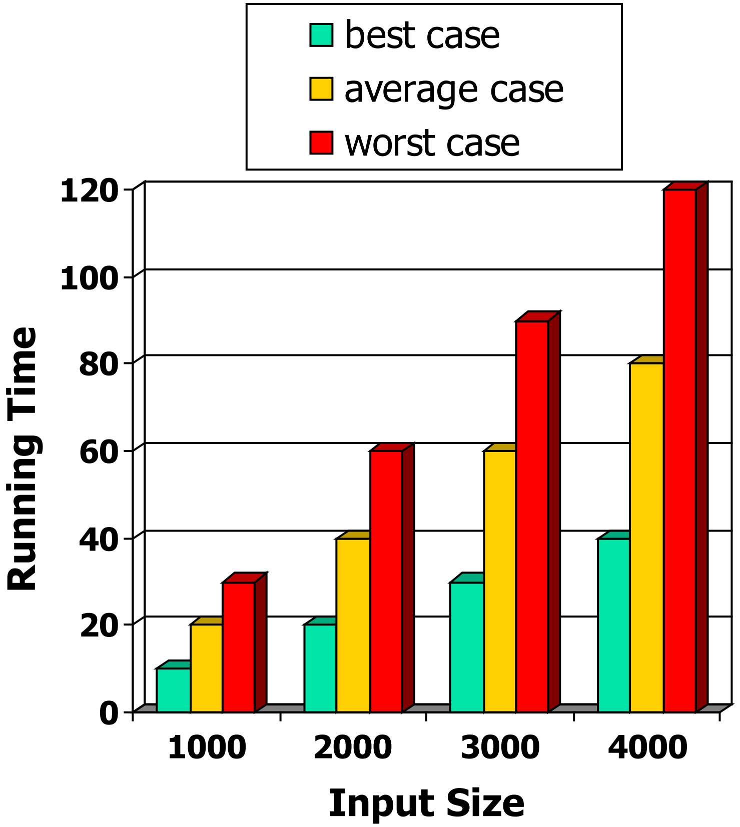
How to Calculate Running Time

- Even on inputs of the same size, running time can be very different
  - Example: algorithm that finds the first prime number in an array by scanning it left to right

- Idea: analyze running time in the
  - best case
  - worst case
  - average case

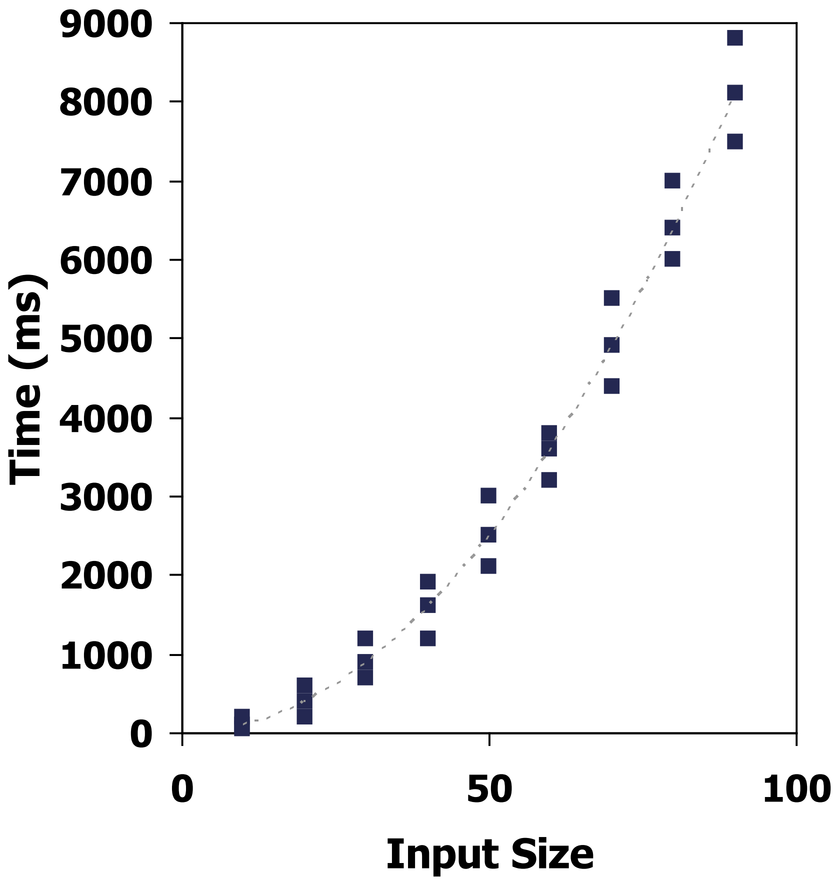
Worst- / average- / best-case

- Worst-case running time of an algorithm
  - The longest running time for **any** input of size n
  - An upper bound on the running time for any input
    $\Rightarrow$ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
  - The worst case can occur fairly often
    - E.g. in searching a database for a particular piece of information
- Best-case running time
  - sort a set of numbers in increasing order; and the data is already in increasing order
- Average-case running time
  - May be difficult to define what "average" means



Experimental Evaluation of Running Time

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



Limitations of Experiments

- Experimental evaluation of running time is very useful but

  - 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

Running-time of algorithms

- Bounds are for the algorithms, rather than programs
  - programs are just implementations of an algorithm, and almost always the details of the program do not affect the bounds

- Bounds are for algorithms, rather than problems
  - A problem can be solved with several algorithms, some are more efficient than others

Theoretical Analysis of Running Time

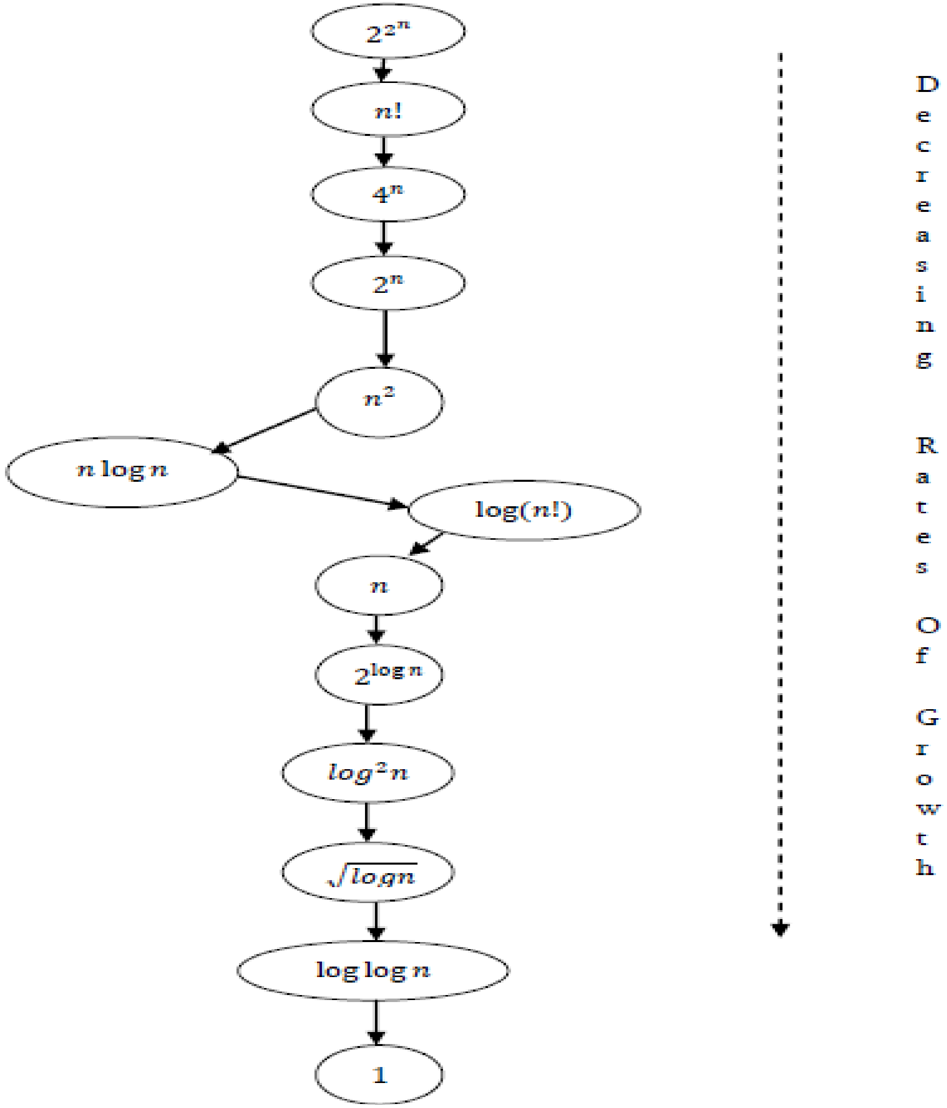
- Uses a pseudo-code description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Rate of Growth

- The rate at which the running time increases as a function of input is called rate of growth.
- Changing the hardware/software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- Ignore the low order terms that are relatively insignificant

$$n^4+2n^2+100n+500=n^4$$

Commonly Used Rates of Growth



Important Functions

- Often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n

Important Functions Growth Rates

n	log(n)	n	nlog(n)	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4.3×10^9
64	6	64	384	4096	262144	1.8×10^{19}
128	7	128	896	16384	2097152	3.4×10^{38}
256	8	256	2048	65536	16777218	1.2×10^{77}

Growth Rates Illustration

Running Time in ms (10^{-3} of sec)	Maximum Problem Size (n) 1000 ms (1 second)	60000 ms (1 minute)	$36*10^5$ m (1 hour)
n	1000	60,000	3,600,000
n^2	32	245	1,897
2^n	10	16	22

Types of Analysis

- Worst-case running time of an algorithm
  - The longest running time for **any** input of size n
  - An upper bound on the running time for any input

    $\Rightarrow$ guarantee that the algorithm will never take longer
  - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
  - The worst case can occur fairly often
    - E.g. in searching a database for a particular piece of information
- Best-case running time
  - sort a set of numbers in increasing order; and the data is already in increasing order
- Average-case running time
  - May be difficult to define what “average” means

Asymptotic notation: Big-Oh

- $f(N) = O(g(N))$
- There are positive constants c and n_0 such that

$$f(N) \leq c\, g(N) \text{ when } N \geq n_0$$

- The growth rate of f(N) is *less than or equal to* the growth rate of g(N)
- g(N) is an upper bound on f(N)

Big-Oh: example

- Let $f(N) = 2N^2$. Then
  - $f(N) = O(N^4)$
  - $f(N) = O(N^3)$
  - $f(N) = O(N^2)$ (best answer, asymptotically tight)

- $O(N^2)$: reads “order N-squared” or “Big-Oh N-squared”

Big Oh: more examples

- $N^2 / 2 - 3N = O(N^2)$
- $1 + 4N = O(N)$
- $7N^2 + 10N + 3 = O(N^2) = O(N^3)$
- $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
- $\sin N = O(1)$; $10 = O(1)$, $10^{10} = O(1)$

$$\sum_{i=1}^{N} i \le N \cdot N = O(N^2)$$

$$\sum_{i=1}^{N} i^2 \le N \cdot N^2 = O(N^3)$$

- $\log N + N = O(N)$
- $\log^k N = O(N)$ for any constant k
- $N = O(2^N)$, but 2^N is not $O(N)$

Math Review: logarithmic functions

$$x^a = b \quad iff \quad \log_x b = a$$

$$\log ab = \log a + \log b$$

$$\log_a b = \frac{\log_m b}{\log_m a}$$

$$\log a^b = b \log a$$

$$a^{\log n} = n^{\log a}$$

$$\log^b a = (\log a)^b \neq \log a^b$$

$$\frac{d \log_e x}{dx} = \frac{1}{x}$$

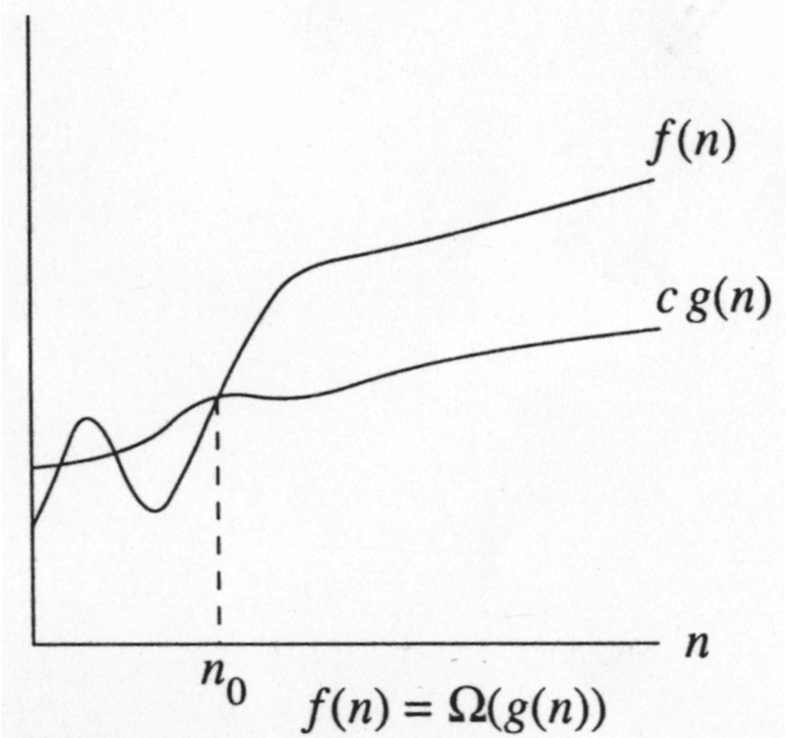
Some rules

When considering the growth rate of a function using Big-Oh

- Ignore the lower order terms and the coefficients of the highest-order term
- No need to specify the base of logarithm
  - Changing the base from one constant to another changes the value of the logarithm by only a constant factor

- If $T_1(N) = O(f(N)$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = max(O(f(N)),\ O(g(N)))$,
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$

Big-Omega



- $\exists$ c , n_0 > 0 such that f(N) $\geq$ c g(N) when N $\geq$ n_0
- f(N) grows no slower than g(N) for “large” N

Big-Omega

- $f(N) = \Omega(g(N))$
- There are positive constants c and n_0 such that

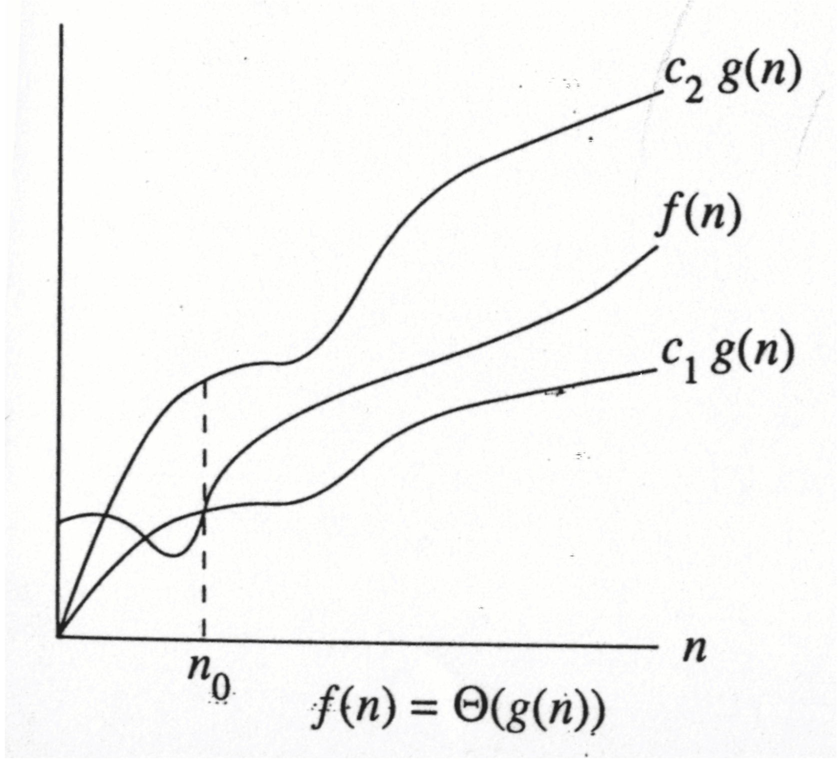
$$f(N) \geq c\, g(N) \text{ when } N \geq n_0$$

- The growth rate of $f(N)$ is *greater than or equal to* the growth rate of $g(N)$.

Big-Omega: examples

- Let $f(N) = 2N^2$. Then
  - $f(N) = \Omega(N)$
  - $f(N) = \Omega(N^2)$ (best answer)

f(N) = Θ(g(N))



- the growth rate of f(N) *is the same as* the growth rate of g(N)

Big-Theta

- $f(N) = \Theta(g(N))$ iff

  $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$
- The growth rate of $f(N)$ *equals* the growth rate of $g(N)$
- Example: Let $f(N)=N^2$, $g(N)=2N^2$
  - Since $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$,

    thus $f(N) = \Theta(g(N))$.
- Big-Theta means the bound is the tightest possible.

Some rules

- If T(N) is a polynomial of degree k, then

  $T(N) = \Theta(N^k)$.

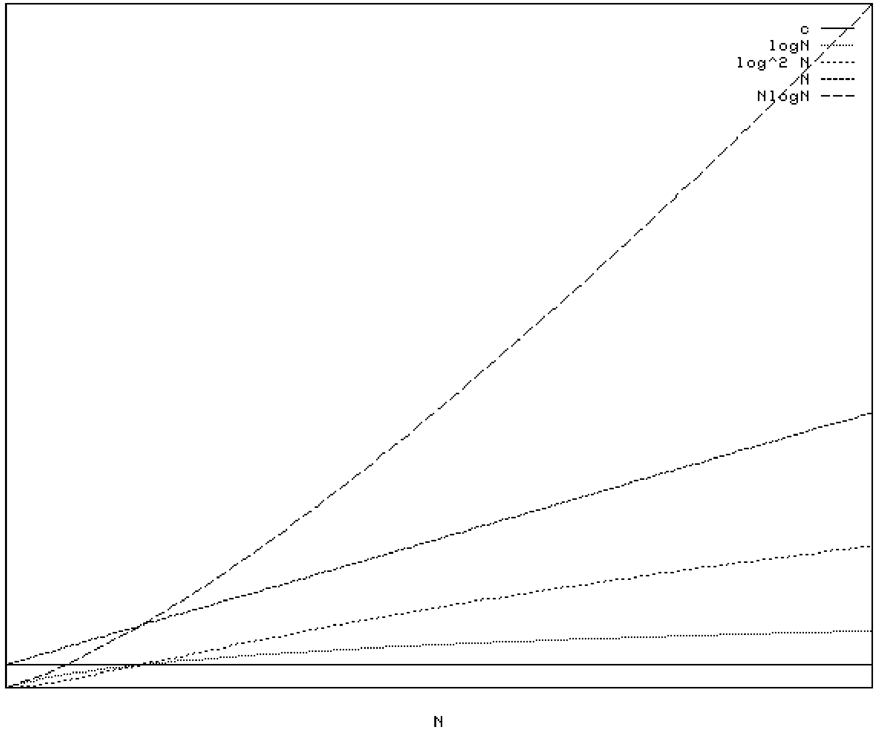
- For logarithmic functions,

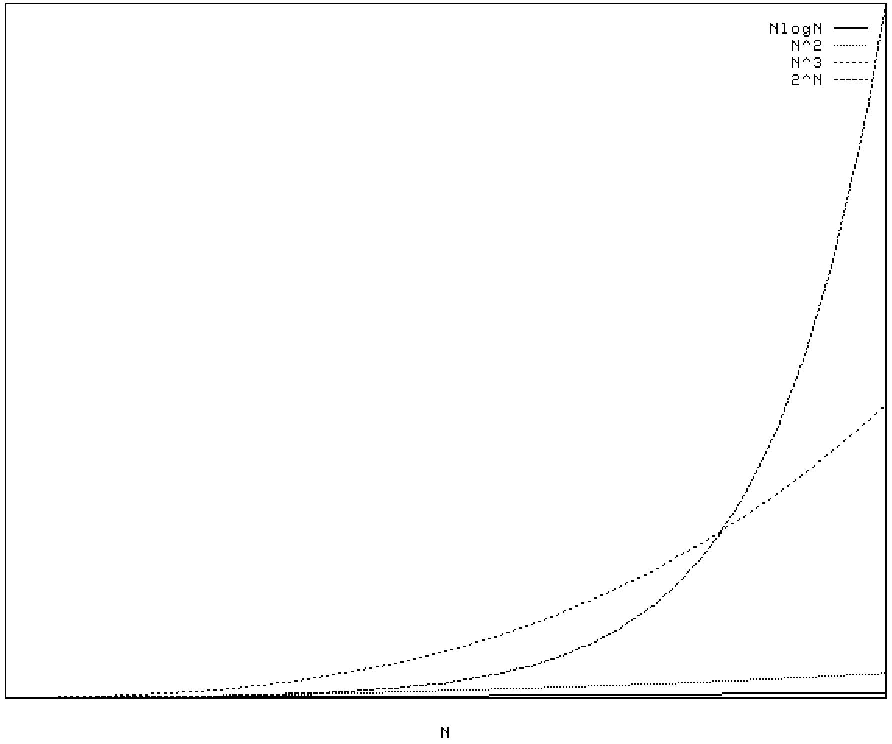
  $T(\log_m N) = \Theta(\log N)$.

Typical Growth Rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates





Growth rates …

- Doubling the input size
  - $f(N) = c \Rightarrow f(2N) = f(N) = c$
  - $f(N) = \log N \Rightarrow f(2N) = f(N) + \log 2$
  - $f(N) = N \Rightarrow f(2N) = 2 f(N)$
  - $f(N) = N^2 \Rightarrow f(2N) = 4 f(N)$
  - $f(N) = N^3 \Rightarrow f(2N) = 8 f(N)$
  - $f(N) = 2^N \Rightarrow f(2N) = f^2(N)$
- Advantages of algorithm analysis
  - To eliminate bad algorithms early
  - pinpoints the bottlenecks, which are worth coding carefully

General Rules

- For loops
  - at most the running time of the statements inside the for-loop (including tests) times the number of iterations.
- Nested for loops

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        k++;
```

  - the running time of the statement multiplied by the product of the sizes of all the for-loops.
  - $O(N^2)$

General rules (cont'd)

- Consecutive statements

```
for (i=0;i<n;i++)
      a[i]=0;
for (i=0;i<n;i++)
      for (j=0;j<n;j++)
            a[i] += a[j]+i+j;
```

  - These just add
  - $O(N) + O(N^2) = O(N^2)$
- `If S1`

  `Else S2`

  - never more than the running time of the test plus the larger of the running times of S1 and S2.

Divide-and-Conquer

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

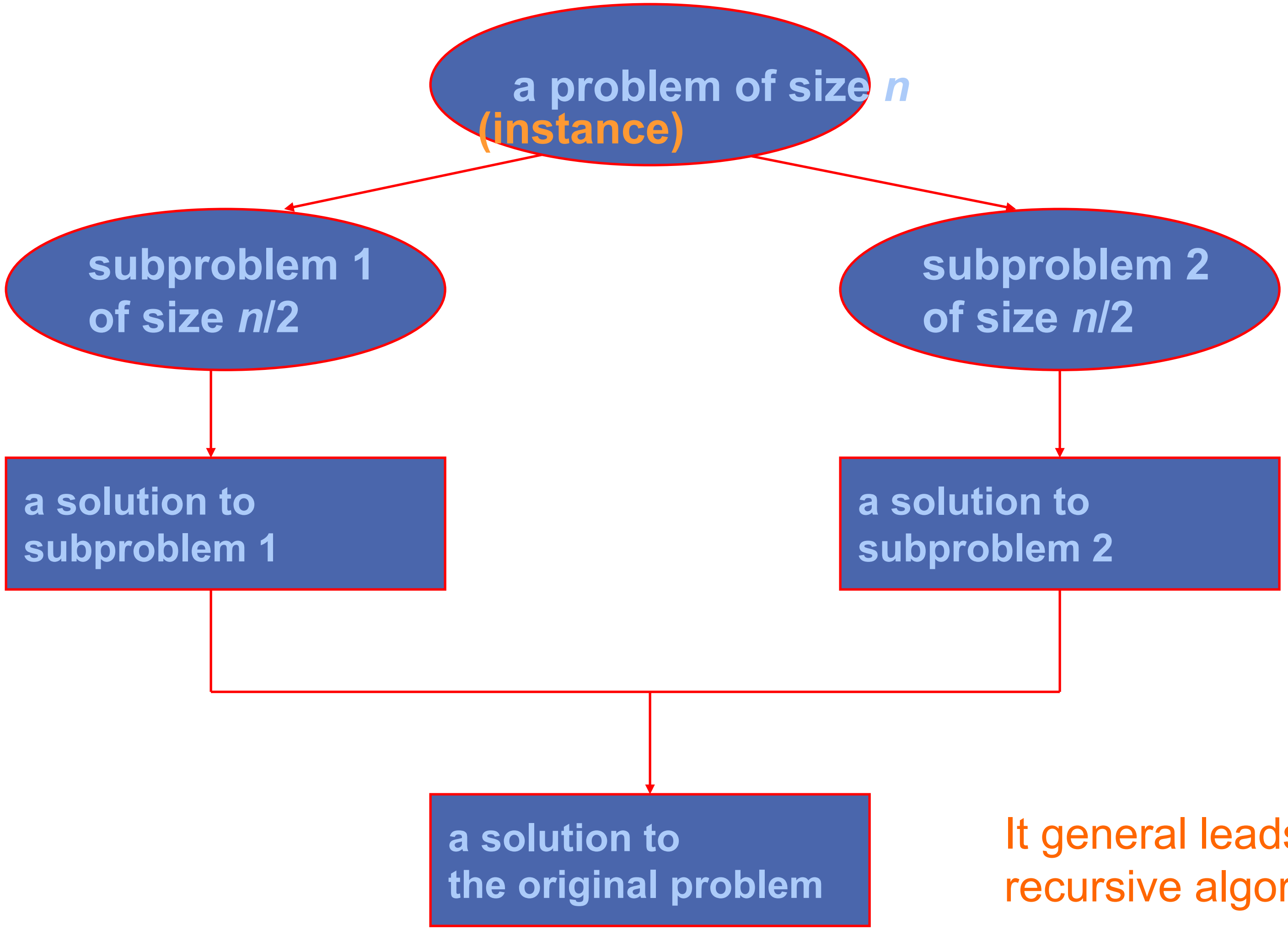
2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms

Divide-and-Conquer Technique (cont.)

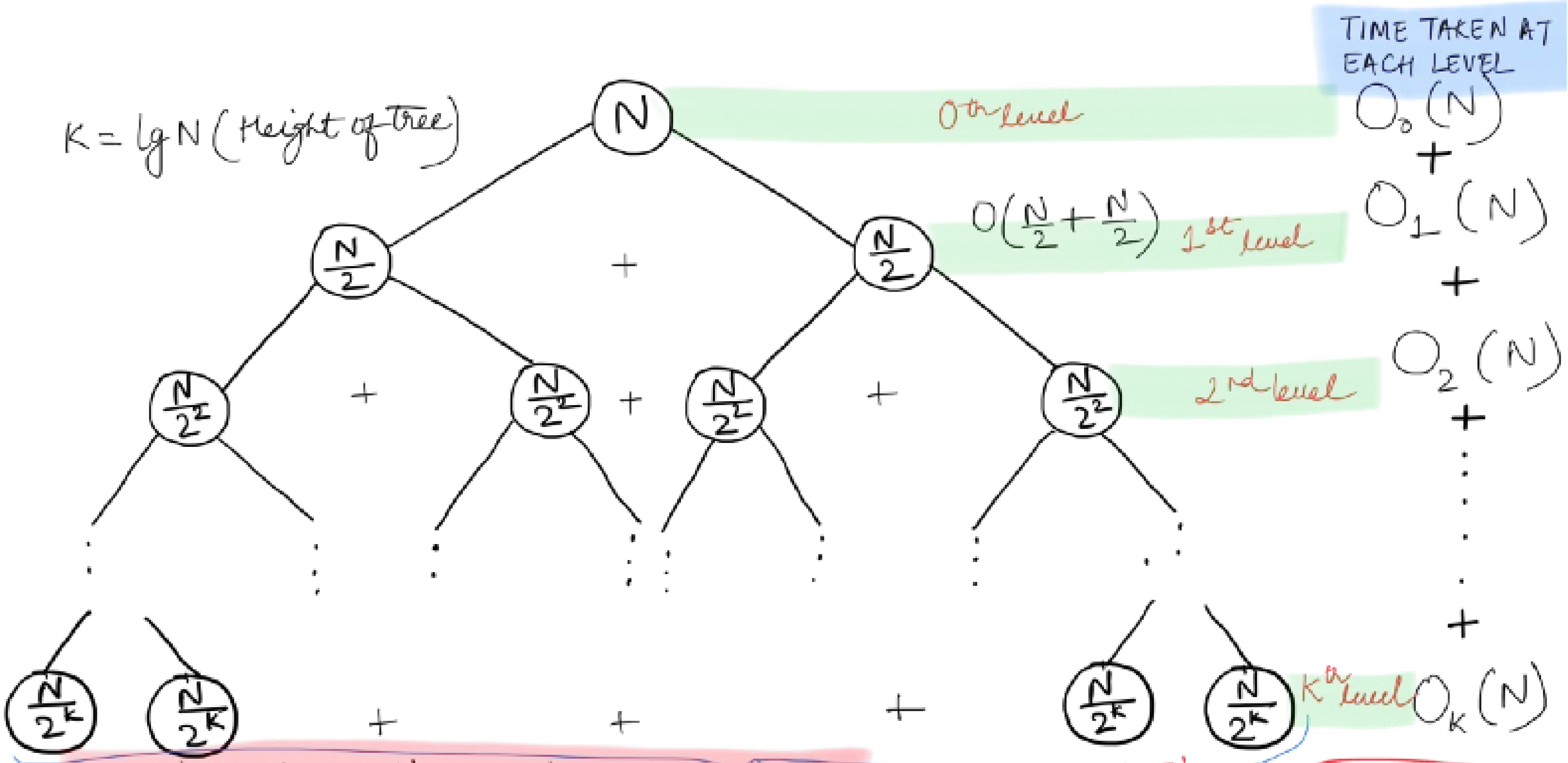


It general leads to a recursive algorithm!

- Recurrence equation

$$T(1) = 1$$
$$T(N) = 2T(\frac{N}{2}) + N$$

  - 2 T(N/2): two subproblems, each of size N/2
  - N: for “patching” two solutions to find solution to whole problem



Total time = Summation of time at each level.

Total Time

$$O\left(\sum_{i=0}^{K} 2^i \cdot \frac{N}{2^i}\right) \Rightarrow O\left(\sum_{i=0}^{K} N\right) \Longrightarrow O(K \cdot N) \Rightarrow O(N \cdot \log N)$$

- Solving the recurrence:

$$
\begin{aligned}
T(N) &= 2T(\frac{N}{2}) + N \\
&= 4T(\frac{N}{4}) + 2N \\
&= 8T(\frac{N}{8}) + 3N \\
&= \cdots \\
&= 2^k T(\frac{N}{2^k}) + kN
\end{aligned}
$$

- With k=log N (i.e. 2^k = N), we have

$$
\begin{aligned}
T(N) &= N\,T(1) + N \log N \\
&= N \log N + N
\end{aligned}
$$

- Thus, the running time is O(N log N)
  - ➢ faster than Algorithm 1 for large data sets

Master theorem for divide and conquer

This theorem is an advance version of master theorem that can be used to determine running time of divide and conquer algorithms if the recurrence is of the following form :-

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem

a = number of subproblems in the recursion and a >= 1

n/b = size of each subproblem

b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Master theorem for divide and conquer

- **Example-1:** $T(n) = 3T(n/2) + n^2$

  $a = 3, b = 2, k = 2, p = 0$

  $b^k = 4$. So, $a < b^k$ and $p = 0$

  [Case 3.(a)]

  $T(n) = \theta(n^k \log^p n)$

  $T(n) = \theta(n^2)$

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Master theorem for divide and conquer

- **Example-2:** $T(n) = 4T(n/2) + n^2$

  $a = 4$, $b = 2$, $k = 2$, $p = 0$

  $b^k = 4$. So, $a = b^k$ and $p = 0$

  [Case 2.(a)]

  $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

  $T(n) = \theta(n^2 \log n)$

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Master theorem for divide and conquer

1. $T(n)=16T\left(\frac{n}{4}\right)+n$ $T(n)=\Theta(n^2)$

2. $T(n)=7T\left(\frac{n}{3}\right)+n^2$ $T(n)=\Theta(n^2)$

3. $T(n)=2T\left(\frac{n}{2}\right)+n/logn$ T(n)=Θ(nloglogn)

4. $T(n)=2T\left(\frac{n}{2}\right)+nlogn$ T(n)=Θ($nlog^2n$)

5. $T(n)=6T\left(\frac{n}{3}\right)+n^2logn$ T(n)=Θ(n^2logn)

6. $T(n)=2T\left(\frac{n}{4}\right)+n^{0.51}$ T(n)=Θ($n^{0.51}$)

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Master theorem for divide and conquer

1. $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$ — *Doesn't apply*

2. $T(n) = 0.5T\left(\frac{n}{2}\right) + 1/n$ — *Doesn't apply*

3. $T(n) = 3T\left(\frac{n}{2}\right) + n$ — $T(n) = \Theta(n^{log3})$

4. $T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + logn$ — $T(n) = \Theta(\sqrt{n})$

5. $T(n) = 64T\left(\frac{n}{8}\right) - n^2 logn$ — *Doesn't apply*

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Master theorem for subtract and conquer

Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases},$$

for some constants $c, a > 0, b > 0, d \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^d)$, then

$$T(n) \text{ is in } \begin{cases} O(n^d), & \text{if } a < 1, \\ O(n^{d+1}), & \text{if } a = 1, \\ O(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

$\square$

Master theorem for subtract and conquer

- $T(n)=2T(n-1)+1$

$a=2$, $b=1$ and $d=0$.

$a>1$ so, applying case 2 : $O(n^0 2^{n/1})$

➡ $O(2^n)$