CME 2001
Data Structures and Algorithms

Zerrin Işık
zerrin@cs.deu.edu.tr

Merge Sort

Designing Algorithms

- Many ways to design algorithms.

- Insertion sort is ***incremental*** : having sorted *A[1…j-1]*, place *A[j]* correctly, so that *A[1…j]* is sorted.

- Another common approach is **Divide and Conquer**.

Divide and Conquer Algorithms

- **Divide** problem into sub-problems.

- **Conquer** by solving sub-problems recursively. If the sub-problems are small enough, solve them in brute force fashion.

- **Combine** the solutions of sub-problems into a solution of the original problem.

Merge Sort

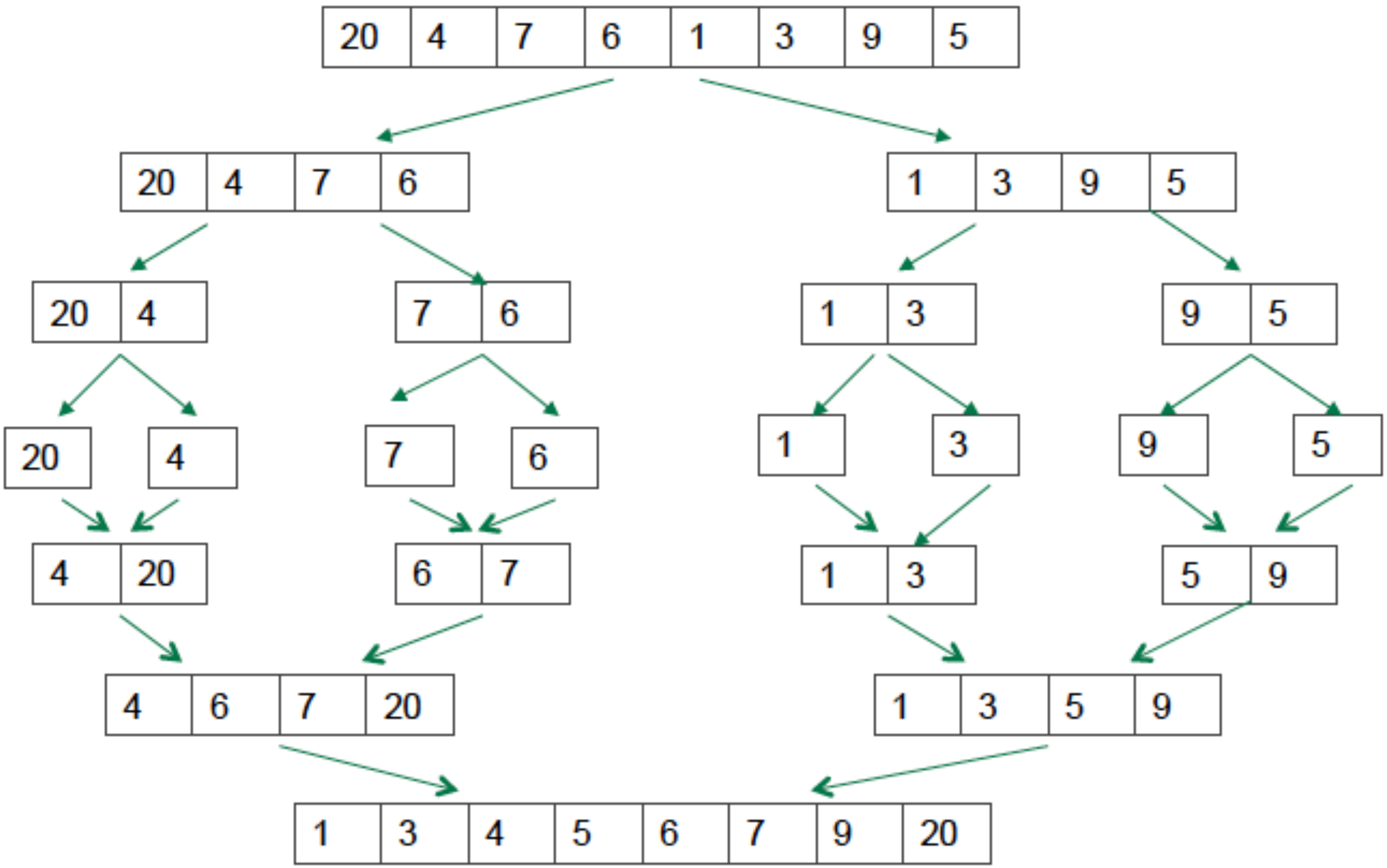
Define each sub-problem as sorting a sub-array $A[p \ldots r]$.

Initially: $p=1$, $r=n$ (these values change as we recurse through sub-problems)

To sort $A[p \ldots r]$:

- **Divide** by splitting into two sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$, where q is the halfway point of $A[p \ldots q]$.
- **Conquer** by recursively sorting two sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$.
- **Combine** by merging two sorted sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ to create a single sorted sub-array $A[p \ldots r]$. To perform this task define a *MERGE(A,p,q,r)* subroutine.

Merge Sort Example



```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
  n1 = q − p + 1
  n2 = r − q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i − 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

Note: The recursion (MERGE-SORT call) will end when the sub-array has just 1 element, which is already sorted.

MERGE-SORT(A, p, r)

if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

MERGE(A, p, q, r)

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
    $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
    if $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$

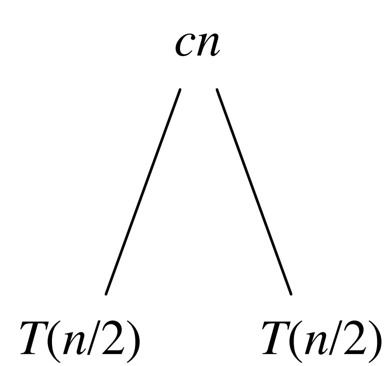
MERGE-SORT running time :

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

Recursion Tree for Recurrence

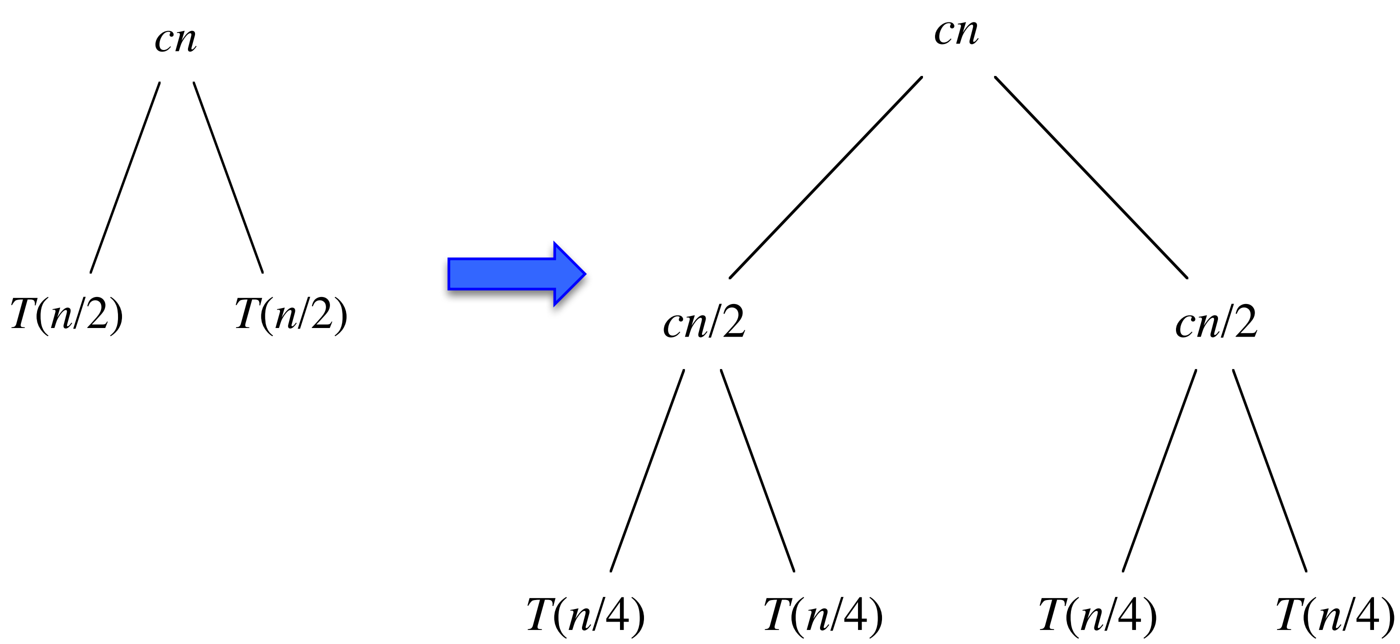
$$T(n) = \begin{cases} c & \text{if } n = 1 \,, \\ 2T(n/2) + cn & \text{if } n > 1 \,. \end{cases}$$

- Draw a **recursion tree** that shows successive expansions of the recurrence.
- We have a cost of ***cn*** and the two sub-problems, each one has a cost of ***T(n/2)***



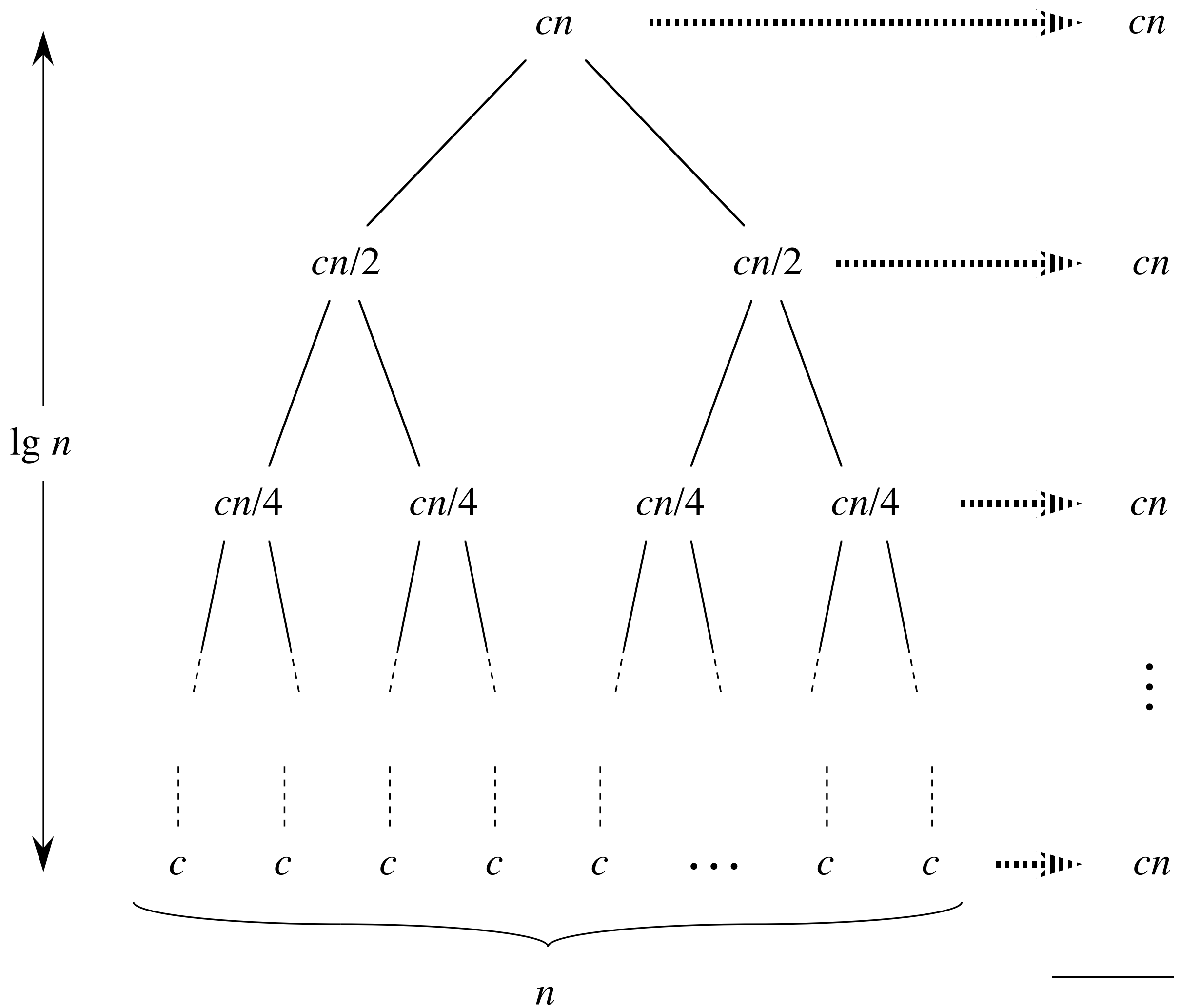
Recursion Tree for Recurrence

- For each of the ***size-n/2*** sub-problems, we have a cost of ***cn/2*** and the two sub-problems, each one has a cost of ***T(n/4)***



- Continue the expansion until the problem size becomes 1

Recursion Tree for Recurrence



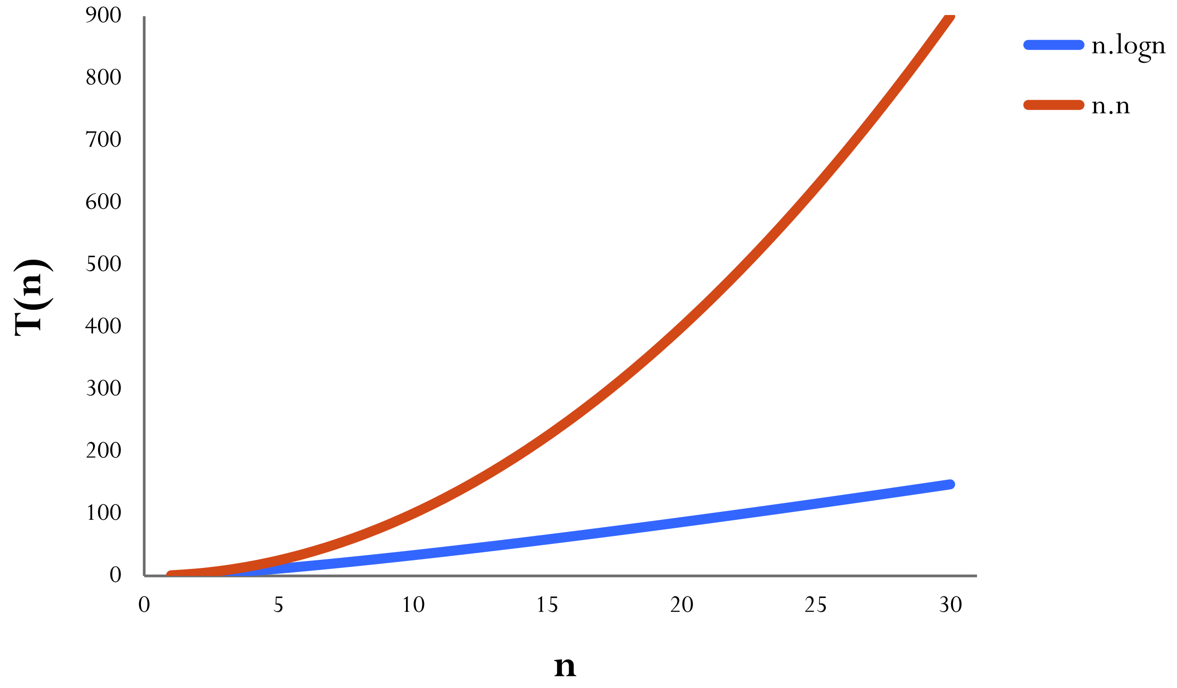
Height = logn

Levels = logn +1

Total = level * level_cost =>

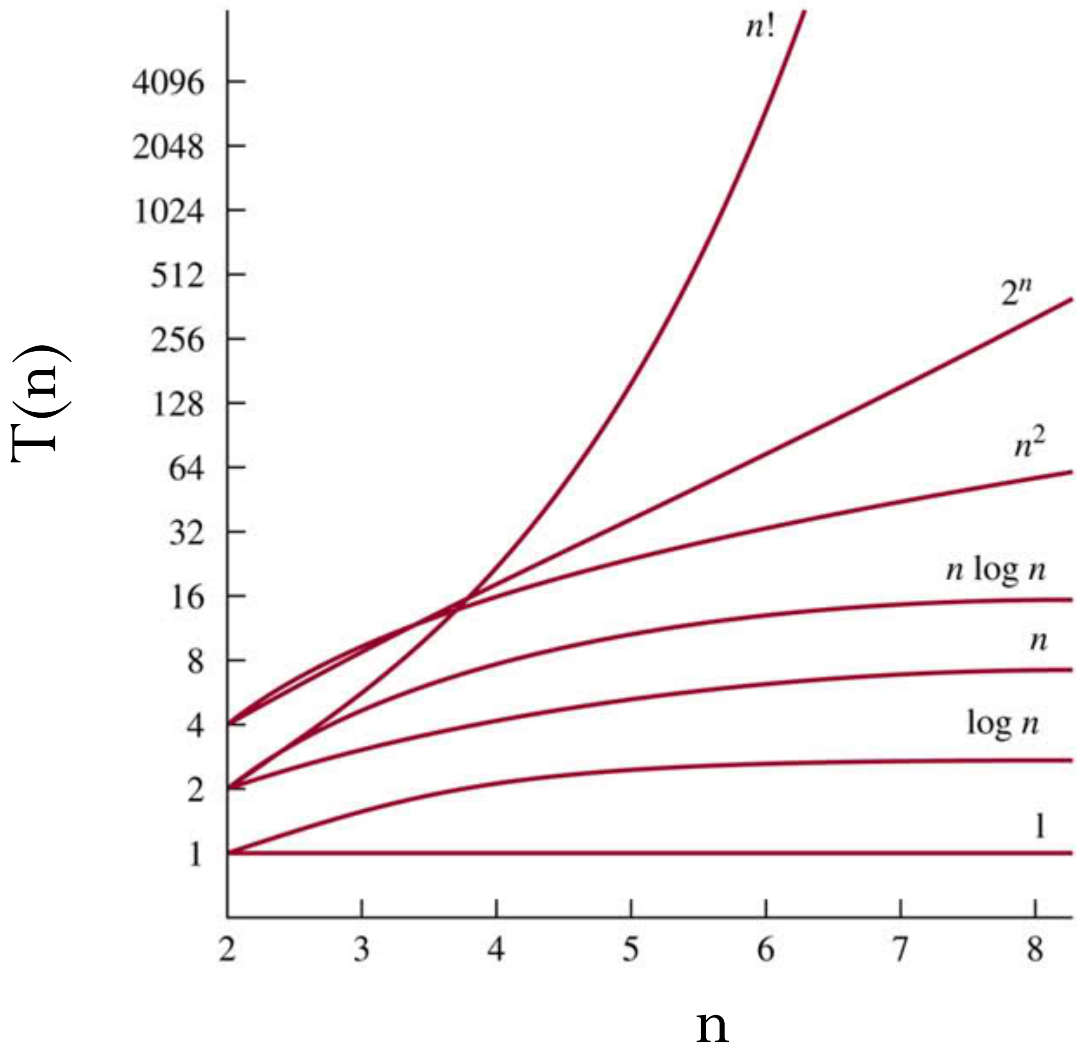
Comparison of Two Algorithms

- Merge Sort asymptotically beats Insertion Sort in the worst-case
- Because $\Theta(n.\log n)$ grows slowly than $\Theta(n^2)$



Growth of Functions

Comparison of Growth-Rate Functions



Asymptotic Notation

- If an algorithm ***A*** requires time proportional to ***f(n)***, it is order ***f(n)***, and it is denoted as ***O(f(n))***

- ***f(n)*** is called the **growth-rate function** of the algorithm ***A***.

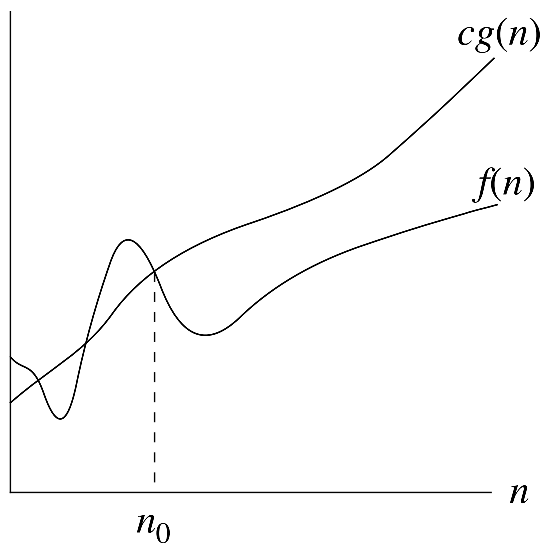
Big-O Notation

Given two growth-rate functions f(n) and g(n):

$$\mathbf{f(n) = O(g(n))}$$

if there exist positive constants c and n_0 such that $f(n) \leq c.g(n)$ for all $n \geq n_0$.

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



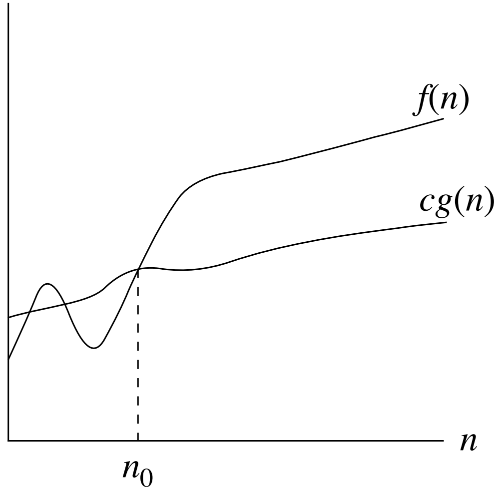
Ω (Omega) Notation

Given two growth-rate functions f(n) and g(n):

$$\mathbf{f(n) = \Omega\ (g(n))}$$

if there exist positive constants c and n_0 such that $c.g(n) \leq f(n)$ for all $n \geq n_0$.

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



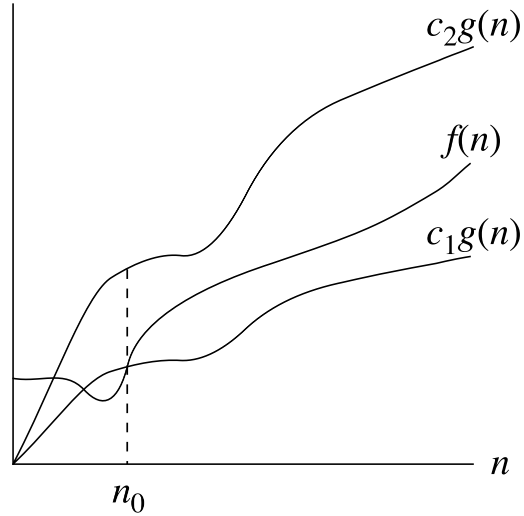
Θ-Notation

Given two growth-rate functions f(n) and g(n):

$$\mathbf{f(n) = \Theta(g(n))}$$

if there exist positive constants c_1, c_2 and n_0 such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_0$.

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



Substitution Method to Solve Recurrence

Recurrence relations represent the running times of divide-and-conquer algorithms.

To solve recurrence relations :

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show this solution works.

Substitution Method …

E.g. $$T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ 2T(n/2) + n & \text{if } n > 1\,. \end{cases}$$

1. Guess: $T(n) = n \lg n + n$
2. Induction:

Basis: $n=1 \Rightarrow n \lg n + n = 1 = T(n)$ ✓

Inductive Step: Our hypothesis is that

$$T(k) = k \lg k + k \qquad \text{for all } k < n$$

We will use this inductive hypothesis for $T(n/2)$

➔

Substitution Method …

Assume T(k) = k lgk + k, then

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\
&= n\lg\frac{n}{2} + n + n \\
&= n(\lg n - \lg 2) + n + n \\
&= n\lg n - n + n + n \\
&= n\lg n + n\ . \qquad \blacksquare
\end{aligned}
$$

Master Method to Solve Recurrence

Useful to solve recurrences of the form:

$$T(n) = aT(n/b) + f(n) ,$$

where $a \geq 1, b > 1$, and $f(n) > 0$.

Compare $n^{\log_b a}$ vs. $f(n)$:

# of leaves

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

  * $f(n)$ is polynomially smaller than $n^{\log_b a}$

Solution: $T(n) = \Theta(n^{\log_b a})$.

(* cost is dominated by leaves.)

Master Method to Solve Recurrence

- **<u>Case 2:</u>** $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.

  $*\ f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.

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

  (* cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)

- **<u>Case 3:</u>** $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $a\,f(n/b) \leq c\,f(n)$ for some constant $c < 1$.

  $*\ f(n)$ is polynomially greater than $n^{\log_b a}$.

  Solution: $T(n) = \Theta(f(n))$.

  (* cost is dominated by root.)