CE100 Algorithms and Programming II

Week-2 (Solving Recurrences / The Divide-and-Conquer)

Spring Semester, 2021-2022

Download DOC-PDF, DOC-DOCX, SLIDE, PPTX

Solving Recurrences

Outline (1)

- Solving Recurrences
  - Recursion Tree
  - Master Method
  - Back-Substitution

Outline (2)

- Divide-and-Conquer Analysis
  - Merge Sort
  - Binary Search
  - Merge Sort Analysis
  - Complexity

Outline (3)

- Recurrence Solution

Solving Recurrences (1)

- Reminder: Runtime $(T(n))$ of *MergeSort* was expressed as a recurrence

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

- Solving recurrences is like solving differential equations, integrals, etc.
  - Need to learn a few tricks

Solving Recurrences (2)

Recurrence: An equation or inequality that describes a function in terms of its value on smaller inputs.

Example :

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

Recurrence Example

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

- Simplification: Assume $n = 2^k$
- Claimed answer : $T(n) = lgn + 1$
- Substitute claimed answer in the recurrence:

$$lgn + 1 = \begin{cases} 1 & \text{if n=1} \\ lg(\lceil n/2 \rceil) + 2 & \text{if n>1} \end{cases}$$

- True when $n = 2^k$

Technicalities: Floor / Ceiling

Technically, should be careful about the floor and ceiling functions (as in the book).

e.g. For merge sort, the recurrence should in fact be:,

$$T(n) = \begin{cases} \Theta(1) & \text{if n=1} \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if n>1} \end{cases}$$

But, it's usually ok to:

- ignore floor/ceiling
- solve for the exact power of 2 (or another number)

Technicalities: Boundary Conditions

- Usually assume: $T(n) = \Theta(1)$ for sufficiently small n
  - Changes the exact solution, but usually the asymptotic solution is not affected (e.g. if polynomially bounded)
- For convenience, the boundary conditions generally implicitly stated in a recurrence
  - $T(n) = 2T(n/2) + \Theta(n)$ assuming that
  - $T(n) = \Theta(1)$ for sufficiently small n

Example: When Boundary Conditions Matter

Exponential function: $T(n) = (T(n/2))2$

Assume

$T(1) = c \text{ (where c is a positive constant)}$

$T(2) = (T(1))^2 = c^2$

$T(4) = (T(2))^2 = c^4$

$T(n) = \Theta(c^n)$

e.g.

$$\text{However } \Theta(2^n) \neq \Theta(3^n) \begin{cases} T(1) = 2 & \Rightarrow \ T(n) = \Theta(2^n) \\ T(1) = 3 & \Rightarrow \ T(n) = \Theta(3^n) \end{cases}$$

The difference in solution more dramatic when:

$$T(1) = 1 \Rightarrow T(n) = \Theta(1^n) = \Theta(1)$$

Solving Recurrences Methods

We will focus on 3 techniques

- Substitution method
- Recursion tree approach
- Master method

Substitution Method

The most general method:

- Guess
- Prove by induction
- Solve for constants

Substitution Method: Example (1)

Solve $T(n) = 4T(n/2) + n$ (assume $T(1) = \Theta(1)$)

1. Guess $T(n) = O(n^3)$ (need to prove O and Ω separately)
2. Prove by induction that $T(n) \leq cn^3$ for large n (i.e. $n \geq n_0$)
   - Inductive hypothesis: $T(k) \leq ck^3$ for any $k < n$
   - Assuming ind. hyp. holds, prove $T(n) \leq cn^3$

Substitution Method: Example (2)

Original recurrence: $T(n) = 4T(n/2) + n$

From inductive hypothesis: $T(n/2) \leq c(n/2)^3$

Substitute this into the original recurrence:

- $T(n) \leq 4c(n/2)^3 + n$
- $= (c/2)n^3 + n$
- $= cn^3 – ((c/2)n^3 – n) \Longleftarrow$ desired - residual
- $\leq cn^3$

  when $((c/2)n^3 – n) \geq 0$

Substitution Method: Example (3)

So far, we have shown:

$$T(n) \leq cn^3 \text{ when } ((c/2)n^3 - n) \geq 0$$

We can choose $c \geq 2$ and $n_0 \geq 1$

But, the proof is not complete yet.

Reminder: Proof by induction:

1.Prove the base cases $\Longleftarrow$ haven't proved the base cases yet

2.Inductive hypothesis for smaller sizes

3.Prove the general case

Substitution Method: Example (4)

- We need to prove the base cases
    - Base: $T(n) = \Theta(1)$ for small n (e.g. for $n = n_0$)
- We should show that:
    - $\Theta(1) \leq cn^3$ for $n = n_0$, This holds if we pick c big enough
- So, the proof of $T(n) = O(n^3)$ is complete
- But, is this a tight bound?

Example: A tighter upper bound? (1)

- Original recurrence: $T(n) = 4T(n/2) + n$
- Try to prove that $T(n) = O(n^2)$,
  - i.e. $T(n) \leq cn^2$ for all $n \geq n_0$
- **Ind. hyp:** Assume that $T(k) \leq ck^2$ for $k < n$
- **Prove the general case:** $T(n) \leq cn^2$

Example: A tighter upper bound? (2)

Original recurrence: $T(n) = 4T(n/2) + n$

Ind. hyp: Assume that $T(k) \leq ck^2$ for $k < n$

Prove the general case: $T(n) \leq cn^2$

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= cn^2 + n \\ &= O(n2) \Longleftarrow \text{ Wrong! We must prove exactly} \end{aligned}$$

Example: A tighter upper bound? (3)

Original recurrence: $T(n) = 4T(n/2) + n$

Ind. hyp: Assume that $T(k) \leq ck^2$ for $k < n$

Prove the general case: $T(n) \leq cn^2$

- So far, we have:
  $T(n) \leq cn^2 + n$
- No matter which positive c value we choose, this does not show that $T(n) \leq cn^2$
- Proof failed?

Example: A tighter upper bound? (4)

- What was the problem?
  - The inductive hypothesis was not strong enough
- **Idea:** Start with a stronger inductive hypothesis
  - Subtract a low-order term
- **Inductive hypothesis:** $T(k) \leq c_1k^2 – c_2k$ for $k < n$
- **Prove the general case:** $T(n) \leq c_1n^2 - c_2n$

Example: A tighter upper bound? (5)

Original recurrence: $T(n) = 4T(n/2) + n$

Ind. hyp: Assume that $T(k) \leq c_1 k^2 – c_2 k$ for $k < n$

Prove the general case: $T(n) \leq c_1 n^2 – c_2 n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4(c_1 (n/2)^2 – c_2 (n/2)) + n \\
&= c_1 n^2 – 2c_2 n + n \\
&= c_1 n^2 – c_2 n – (c_2 n – n) \\
&\leq c_1 n^2 – c_2 n \text{ for } n(c_2 – 1) \geq 0 \\
&\text{choose } c2 \geq 1
\end{aligned}
$$

Example: A tighter upper bound? (6)

We now need to prove

$$T(n) \leq c_1 n^2 - c_2 n$$

for the base cases.

$T(n) = \Theta(1)$ for $1 \leq n \leq n_0$ (implicit assumption)

$\Theta(1) \leq c_1 n^2 - c_2 n$ for n small enough (e.g. $n = n_0$)

- We can choose c1 large enough to make this hold

We have proved that $T(n) = O(n^2)$

Substitution Method: Example 2 (1)

For the recurrence $T(n) = 4T(n/2) + n$,

prove that $T(n) = \Omega(n^2)$

i.e. $T(n) \geq cn^2$ for any $n \geq n_0$

Ind. hyp: $T(k) \geq ck^2$ for any $k < n$

Prove general case: $T(n) \geq cn^2$

$T(n) = 4T(n/2) + n$
$\geq 4c(n/2)^2 + n$
$= cn^2 + n$
$\geq cn^2$ since $n > 0$

Proof succeeded – no need to strengthen the ind. hyp as in the last example

Substitution Method: Example 2 (2)

We now need to prove that

$T(n) \geq cn^2$

for the base cases

$T(n) = \Theta(1)$ for $1 \leq n \leq n_0$ (implicit assumption)

$\Theta(1) \geq cn^2$ for $n = n_0$

n_0 is sufficiently small (i.e. constant)

We can choose c small enough for this to hold

We have proved that $T(n) = \Omega(n^2)$

Substitution Method - Summary

- Guess the asymptotic complexity
- Prove your guess using induction
  - Assume inductive hypothesis holds for $k < n$
  - Try to prove the general case for n
    - Note: $MUST$ prove the $EXACT$ inequality $CANNOT$ ignore lower order terms, If the proof fails, strengthen the ind. hyp. and try again
- Prove the base cases (usually straightforward)

Recursion Tree Method

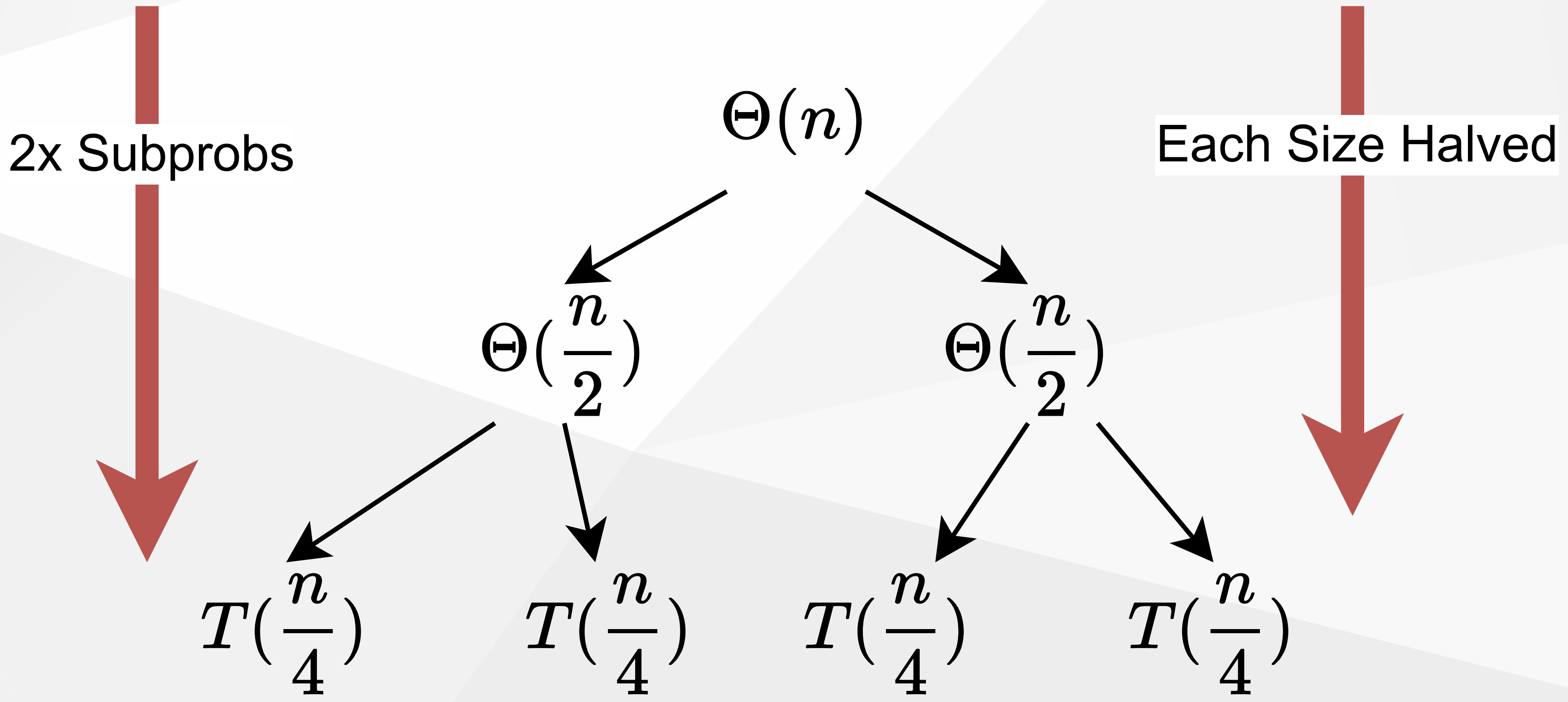
- A recursion tree models the runtime costs of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable.
  - Not suitable for formal proofs
- The recursion-tree method promotes intuition, however.

Solve Recurrence (1) : $T(n) = 2T(n/2) + \Theta(n)$

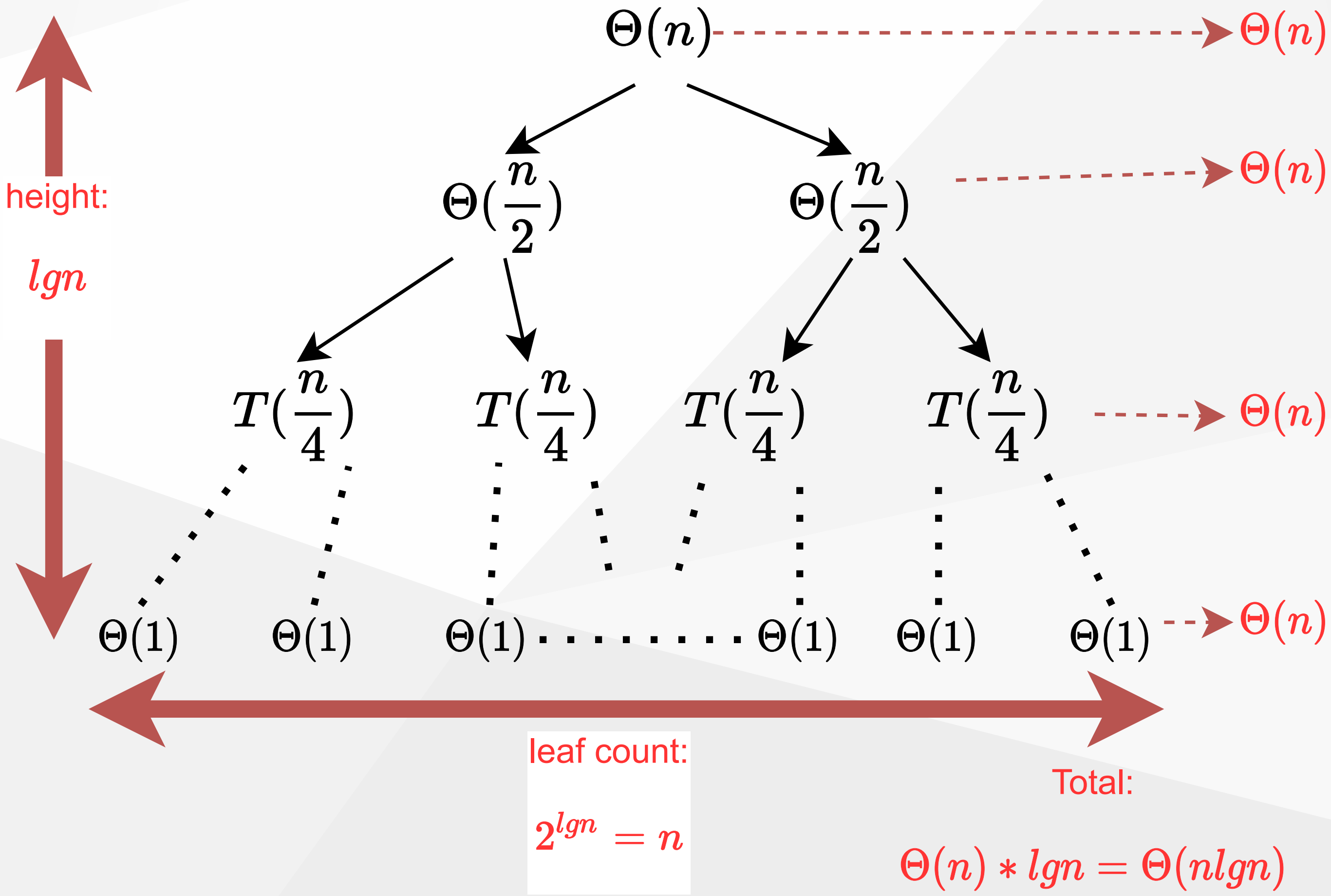
$$\Theta(n)$$

$$T(\frac{n}{2}) \qquad T(\frac{n}{2})$$

Solve Recurrence (2) : $T(n) = 2T(n/2) + \Theta(n)$



Solve Recurrence (3) : $T(n) = 2T(n/2) + \Theta(n)$



Example of Recursion Tree (1)

Solve $T(n) = T(n/4) + T(n/2) + n^2$

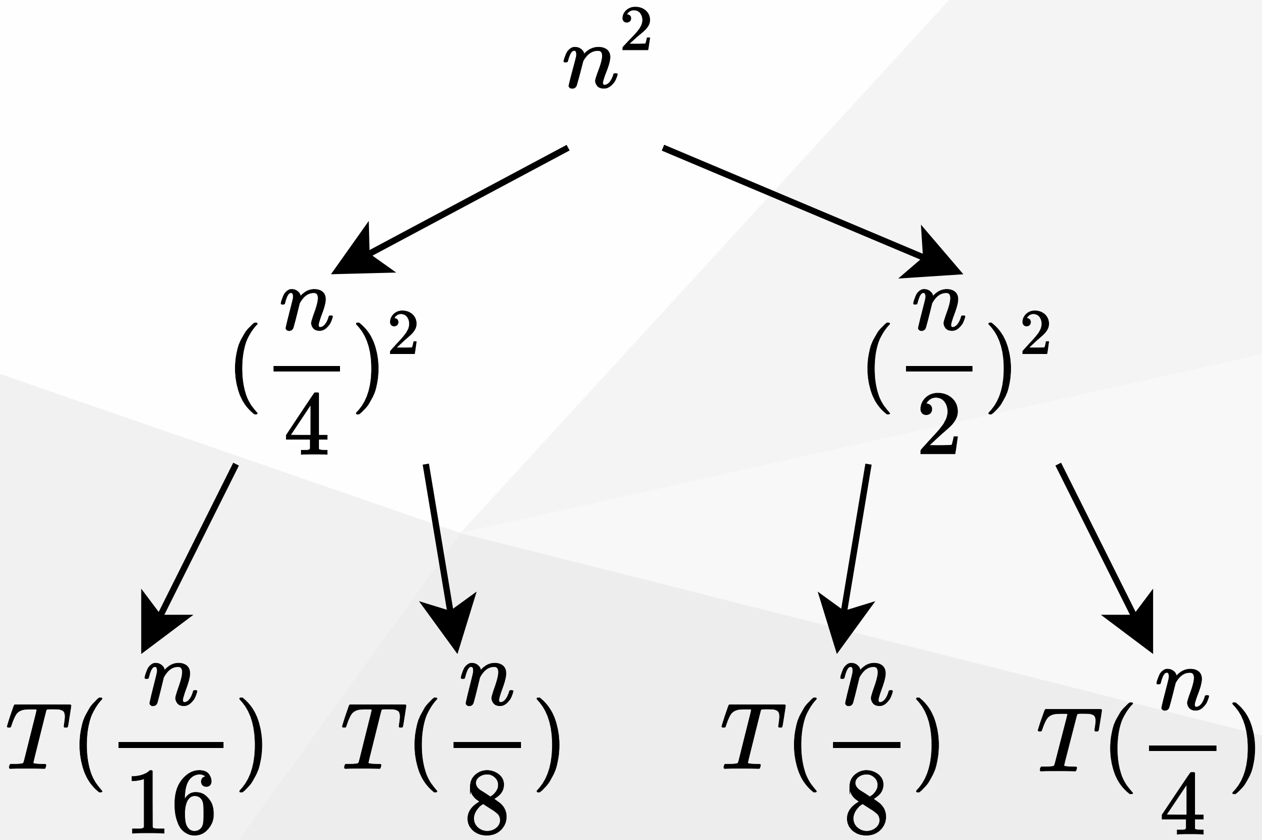
$$\begin{tikzcd}[column sep=small] & T(n) = n^2 \arrow[dl] \arrow[dr] & \\ T(\frac{n}{4}) & & T(\frac{n}{2}) \end{tikzcd}$$

$$\Downarrow$$

$$\begin{tikzcd}[column sep=small] & n^2 \arrow[dl] \arrow[dr] & \\ (\frac{n}{4})^2 & & (\frac{n}{2})^2 \end{tikzcd}$$

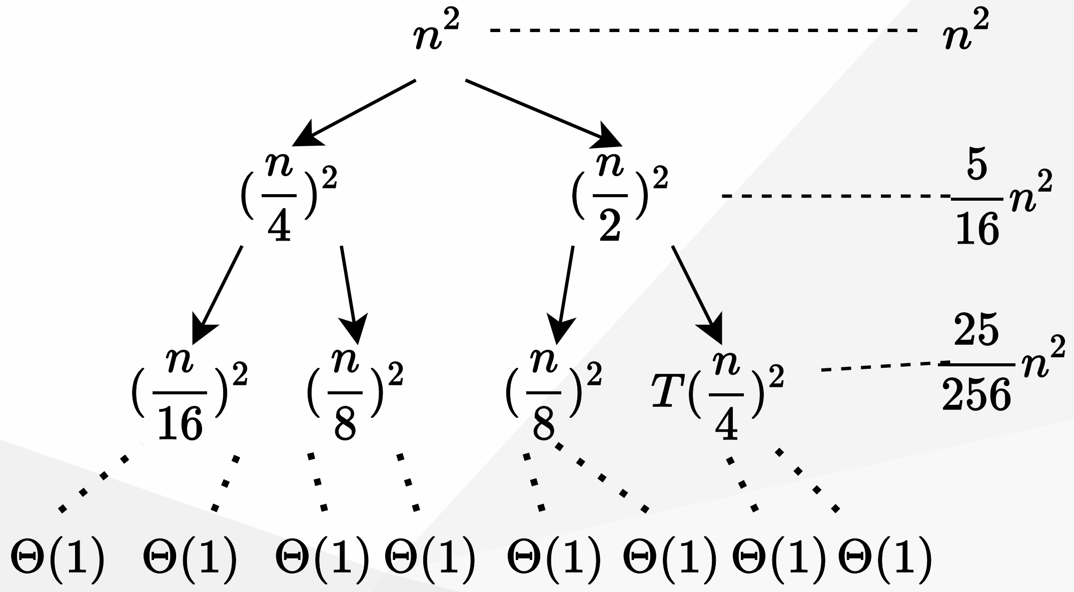
Example of Recursion Tree (2)

Solve $T(n) = T(n/4) + T(n/2) + n^2$



Example of Recursion Tree (3)

Solve $T(n) = T(n/4) + T(n/2) + n^2$



$$Total = n^2(1 + \frac{5}{16} + (\frac{5}{16})^2 + (\frac{5}{16})^3 + \ldots)$$

Geometric Series

$$Total = n^2((\frac{5}{16})^0 + (\frac{5}{16})^1 + (\frac{5}{16})^2 + (\frac{5}{16})^3 + \ldots)$$

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

The Master Method

- A powerful black-box method to solve recurrences.
- The master method applies to recurrences of the form
  - $T(n) = aT(n/b) + f(n)$
- where $a \geq 1, b > 1$, and f is **asymptotically positive**.

The Master Method: 3 Cases

(TODO : Add Notes)

- Recurrence: $T(n) = aT(n/b) + f(n)$
- Compare $f(n)$ with $n^{log_b^a}$
- Intuitively:
  - **Case 1:** $f(n)$ grows polynomially slower than $n^{log_b^a}$
  - **Case 2:** $f(n)$ grows at the same rate as $n^{log_b^a}$
  - **Case 3:** $f(n)$ grows polynomially faster than $n^{log_b^a}$

The Master Method: Case 1 (Bigger)

- Recurrence: $T(n) = aT(n/b) + f(n)$
- *Case 1:* $\frac{n^{log_b^a}}{f(n)} = \Omega(n^{\varepsilon})$ for some constant $\varepsilon > 0$
- i.e., $f(n)$ grows polynomialy slower than $n^{log_b^a}$ (by an n^{ε} factor)
- **Solution:** $T(n) = \Theta(n^{log_b^a})$

The Master Method: Case 2 (Simple Version) (Equal)

- Recurrence: $T(n) = aT(n/b) + f(n)$
- *Case 2:* $\frac{f(n)}{n^{log_b^a}} = \Theta(1)$
- i.e., $f(n)$ and $n^{log_b^a}$ grow at similar rates
- **Solution:** $T(n) = \Theta(n^{log_b^a} lgn)$

The Master Method: Case 3 (Smaller)

- *Case 3:* $\frac{f(n)}{n^{log_b^a}} = \Omega(n^{\varepsilon})$ for some constant $\varepsilon > 0$
- i.e., $f(n)$ grows polynomialy faster than $n^{log_b^a}$ (by an n^{ε} factor)
- and the following regularity condition holds:
  - $af(n/b) \leq cf(n)$ for some constant $c < 1$
- Solution: $T(n) = \Theta(f(n))$

The Master Method Example (case-1) : $T(n) = 4T(n/2) + n$

- $a = 4$
- $b = 2$
- $f(n) = n$
- $n^{log_b^a} = n^{log_2^4} = n^{log_2^{2^2}} = n^{2log_2^2} = n^2$
- $f(n) = n$ grows polynomially slower than $n^{log_b^a} = n^2$
    - $\frac{n^{log_b^a}}{f(n)} = \frac{n^2}{n} = n = \Omega(n^{\varepsilon})$
- CASE-1:
    - $T(n) = \Theta(n^{log_b^a}) = \Theta(n^{log_2^4}) = \Theta(n^2)$

The Master Method Example (case-2) : $T(n) = 4T(n/2) + n^2$

- $a = 4$
- $b = 2$
- $f(n) = n^2$
- $n^{log_b^a} = n^{log_2^4} = n^{log_2^{2^2}} = n^{2log_2^2} = n^2$
- $f(n) = n^2$ grows at similar rate as $n^{log_b^a} = n^2$
  - $f(n) = \Theta(n^{log_b^a}) = n^2$
- CASE-2:
  - $T(n) = \Theta(n^{log_b^a} lgn) = \Theta(n^{log_2^4} lgn) = \Theta(n^2 lgn)$

The Master Method Example (case-3) (1) : $T(n) = 4T(n/2) + n^3$

- $a = 4$
- $b = 2$
- $f(n) = n^3$
- $n^{log_b^a} = n^{log_2^4} = n^{log_2^{2^2}} = n^{2log_2^2} = n^2$
- $f(n) = n^3$ grows polynomially faster than $n^{log_b^a} = n^2$
  - $\frac{f(n)}{n^{log_b^a}} = \frac{n^3}{n^2} = n = \Omega(n^{\varepsilon})$

The Master Method Example (case-3) (2) : $T(n) = 4T(n/2) + n^3$ (con't)

- Seems like CASE 3, but need to check the regularity condition
- Regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$
- $4(n/2)^3 \leq cn^3$ for $c = 1/2$
- CASE-3:
  - $T(n) = \Theta(f(n)) \Longrightarrow T(n) = \Theta(n^3)$

The Master Method Example (N/A case) : $T(n) = 4T(n/2) + n^2 lgn$

- $a = 4$
- $b = 2$
- $f(n) = n^2 lgn$
- $n^{log_b^a} = n^{log_2^4} = n^{log_2^{2^2}} = n^{2log_2^2} = n^2$
- $f(n) = n^2 lgn$ grows slower than $n^{log_b^a} = n^2$
  - but is it polynomially slower?
  - $\frac{n^{log_b^a} f(n)}{=} \frac{n^2}{\frac{n^2}{lgn}} = lgn \neq \Omega(n^\varepsilon)$ for any $\varepsilon > 0$
    - is not CASE-1
    - Master Method does not apply!

The Master Method : Case 2 (General Version)

- Recurrence : $T(n) = aT(n/b) + f(n)$
- Case 2: $\frac{f(n)}{n^{log_b^a}} = \Theta(lg^k n)$ for some constant $k \geq 0$
- Solution : $T(n) = \Theta(n^{log_b^a} lg^{k+1} n)$

General Method (Akra-Bazzi)

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n)$$

Let p be the unique solution to

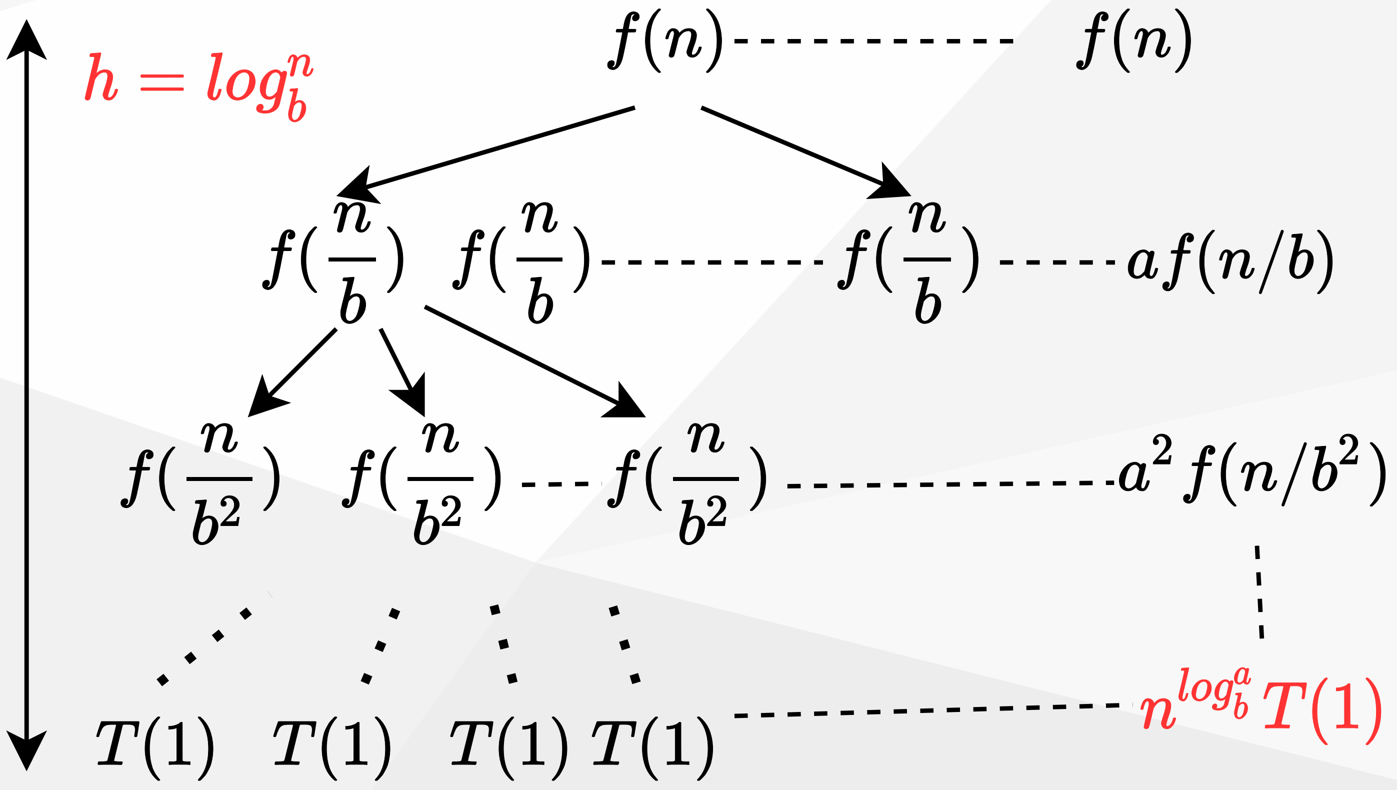
$$\sum_{i=1}^{k} (a_i/b_i^p) = 1$$

Then, the answers are the same as for the master method, but with n^p instead of $n^{log_b^a}$

(Akra and Bazzi also prove an even more general result.)

Idea of Master Theorem (1)

Recursion Tree:



$$\text{leaves count} = a^h = a^{log_b^n} = n^{log_b^a}$$

Idea of Master Theorem (2)

CASE 1 : The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

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

Idea of Master Theorem (3)

CASE 2 : $(k = 0)$ The weight is approximately the same on each of the $log_b n$ levels.

$$n^{log_b^a} T(1) = \Theta(n^{log_b^a} lgn)$$

Idea of Master Theorem (4)

CASE 3 : The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

$$n^{log_b^a}T(1) = \Theta(f(n))$$

Proof of Master Theorem: Case 1 and Case 2

- Recall from the recursion tree (note $h = lg_b n = \text{tree height}$)

$$\text{Leaf Cost} = \Theta(n^{log_b^a})$$

$$\text{Non-leaf Cost} = g(n) = \sum_{i=0}^{h-1} a^i f(n/b^i)$$

$$T(n) = \text{Leaf Cost} + \text{Non-leaf Cost}$$

$$T(n) = \Theta(n^{log_b^a}) + \sum_{i=0}^{h-1} a^i f(n/b^i)$$

Proof of Master Theorem Case 1 (1)

- $\frac{n^{log_b^a}}{f(n)} = \Omega(n^{\varepsilon})$ for some $\varepsilon > 0$
- $\frac{n^{log_b^a}}{f(n)} = \Omega(n^{\varepsilon}) \implies O(n^{-\varepsilon}) \implies f(n) = O(n^{log_b^{a-\varepsilon}})$
- $g(n) = \sum_{i=0}^{h-1} a^i O((n/b^i)^{log_b^{a-\varepsilon}}) = O(\sum_{i=0}^{h-1} a^i (n/b^i)^{log_b^{a-\varepsilon}})$
- $O(n^{log_b^{a-\varepsilon}} \sum_{i=0}^{h-1} a^i b^{i\varepsilon} / b^{ilog_b^{a-\varepsilon}})$

Proof of Master Theorem Case 1 (2)

- $\sum_{i=0}^{h-1} \frac{a^i b^{i\varepsilon}}{b^{ilog_b^a}} = \sum_{i=0}^{h-1} a^i \frac{(b^\varepsilon)^i}{(b^{log_b^a})^i} = \sum a^i \frac{b^{i\varepsilon}}{a^i} = \sum_{i=0}^{h-1} (b^\varepsilon)^i$

= An increasing geometric series since $b > 1$

$\frac{b^{h\varepsilon}-1}{b^\varepsilon-1} = \frac{(b^h)^\varepsilon-1}{b^\varepsilon-1} = \frac{(b^{log_b^n})^\varepsilon-1}{b^\varepsilon-1} = \frac{n^\varepsilon-1}{b^\varepsilon-1} = O(n^\varepsilon)$

Proof of Master Theorem Case 1 (3)

- $g(n) = O(n^{log_b a - \varepsilon} O(n^{\varepsilon})) = O(\frac{n^{log_b^a}}{n^{\varepsilon}} O(n^{\varepsilon})) = O(n^{log_b^a})$
- $T(n) = \Theta(n^{log_b^a}) + g(n) = \Theta(n^{log_b^a}) + O(n^{log_b^a}) = \Theta(n^{log_b^a})$

Q.E.D.

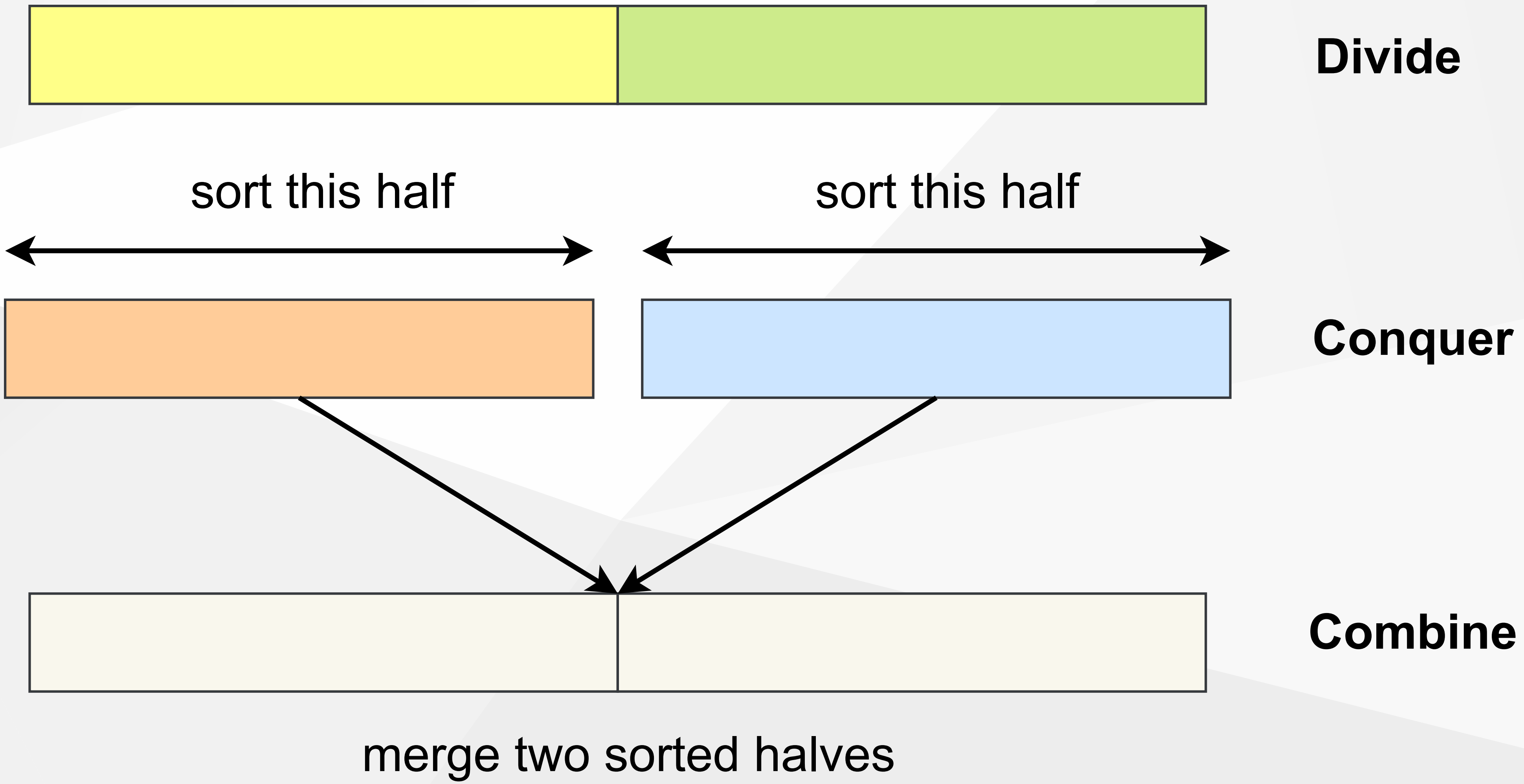
(Quod Erat Demonstrandum)

Proof of Master Theorem Case 2 (limited to k=0)

- $\frac{f(n)}{n^log_b^a} = \Theta(lg^0 n) = \Theta(1) \implies f(n) = \Theta(n^{log_b^a}) \implies f(n/b^i) = \Theta((n/b^i)^{log_b^a})$
- $g(n) = \sum_{i=0}^{h-1} a^i \Theta((n/b^i)^{log_b^a})$
- $= \Theta(\sum_{i=0}^{h-1} a^i \frac{n^{log_b^a}}{b^{ilog_b^a}})$
- $= \Theta(n^{log_b^a} \sum_{i=0}^{h-1} a^i \frac{1}{(b^{log_b^a})^i})$
- $= \Theta(n^{log_b^a} \sum_{i=0}^{h-1} a^i \frac{1}{a^i})$
- $= \Theta(n^{log_b^a} \sum_{i=0}^{log_b^{n-1}} 1) = \Theta(n^{log_b^a} log_b n) = \Theta(n^{log_b^a} lgn)$
- $T(n) = n^{log_b^a} + \Theta(n^{log_b^a} lgn)$
- $= \Theta(n^{log_b^a} lgn)$

Q.E.D.

The Divide-and-Conquer Design Paradigm (1)



The Divide-and-Conquer Design Paradigm (2)

1. **Divide** we divide the problem into a number of subproblems.
2. **Conquer** we solve the subproblems recursively.
3. **BaseCase** solve by Brute-Force
4. **Combine** subproblem solutions to the original problem.

The Divide-and-Conquer Design Paradigm (3)

- $a = \text{subproblem}$
- $1/b = \text{each size of the problem}$

$$T(n) = \begin{cases} \Theta(1) & \text{if} \quad n \leq c \quad (basecase) \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Merge-Sort

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

$T(n) = \Theta(nlgn)$

Selection Sort Algorithm

```
SELECTION-SORT(A)
    n = A.length;
    for j=1 to n-1
        smallest=j;
        for i= j+1 to n
            if A[i]<A[smallest]
                smallest=i;
        endfor
        exchange A[j] with A[smallest]
    endfor
```

Selection Sort Algorithm

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n-1) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- Sequential Series

$$cost = n(n+1)/2 = 1/2n^2 + 1/2n$$

- Drop low-order terms
- Ignore the constant coefficient in the leading term

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

Merge Sort Algorithm (initial setup)

Merge Sort is a recursive sorting algorithm, for initial case we need to call `Merge-Sort(A,1,n)` for sorting $A[1..n]$

initial case

```
A : Array
p : 1 (offset)
r : n (length)
Merge-Sort(A,1,n)
```

Merge Sort Algorithm (internal iterations)

internal iterations

$$p = start - point$$

$$q = mid - point$$

$$r = end - point$$

```
A : Array
p : offset
r : length
Merge-Sort(A,p,r)
    if p=r then                        (CHECK FOR BASE-CASE)
        return
    else
        q = floor((p+r)/2)    (DIVIDE)
        Merge-Sort(A,p,q)     (CONQUER)
        Merge-Sort(A,q+1,r)   (CONQUER)
        Merge(A,p,q,r)        (COMBINE)
    endif
```

Merge Sort Combine Algorithm (1)

```
Merge(A,p,q,r)
    n1 = q-p+1
    n2 = r-q

    //allocate left and right arrays
    //increment will be from left to right
    //left part will be bigger than right part

    L[1...n1+1] //left array
    R[1...n2+1] //right array

    //copy left part of array
    for i=1 to n1
        L[i]=A[p+i-1]

    //copy right part of array
    for j=1 to n2
        R[j]=A[q+j]

    //put end items maximum values for termination
    L[n1+1]=inf
    R[n2+1]=inf

    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
```

Example : Merge Sort

1. **Divide:** Trivial.
2. **Conquer:** Recursively sort 2 subarrays.
3. **Combine:** Linear- time merge.

- $T(n) = 2T(n/2) + \Theta(n)$
  - Subproblems $\Longrightarrow 2$
  - Subproblemsize $\Longrightarrow n/2$
  - Work dividing and combining $\Longrightarrow \Theta(n)$

Master Theorem: Reminder

- $T(n) = aT(n/b) + f(n)$
  - Case 1: $\frac{n^{log_b^a}}{f(n)} = \Omega(n^{\varepsilon}) \Longrightarrow T(n) = \Theta(n^{log_b^a})$
  - Case 2: $\frac{f(n)}{n^{log_b^a}} = \Theta(lg^k n) \Longrightarrow T(n) = \Theta(n^{log_b^a} lg^{k+1} n)$
  - Case 3: $\frac{n^{log_b^a}}{f(n)} = \Omega(n^{\varepsilon}) \Longrightarrow T(n) = \Theta(f(n))$ and $af(n/b) \leq cf(n)$ for $c < 1$

Merge Sort: Solving the Recurrence

$T(n) = 2T(n/2) + \Theta(n)$
$a = 2, b = 2, f(n) = \Theta(n), n^{log_b^a} = n$

Case-2: $\frac{f(n)}{n^{log_b^a}} = \Theta(lg^k n) \implies T(n) = \Theta(n^{log_b^a} lg^{k+1} n)$ holds for $k = 0$

$T(n) = \Theta(nlgn)$

Binary Search (1)

Find an element in a sorted array:

1. **Divide:** Check middle element.
2. **Conquer:** Recursively search 1 subarray.
3. **Combine:** Trivial.

Binary Search (2)

$$\text{PARENT} = \lfloor i/2 \rfloor$$

$$\text{LEFT-CHILD} = 2i,\ \text{2i>n}$$

$$\text{RIGHT-CHILD} = 2i+1,\ \text{2i>n}$$

Binary Search (3) : Iterative

```
ITERATIVE-BINARY-SEARCH(A,V,low,high)
    while low<=high
        mid=floor((low+high)/2);
        if v == A[mid]
            return mid;
        elseif v > A[mid]
            low = mid + 1;
        else
            high = mid - 1;
    endwhile
    return NIL
```

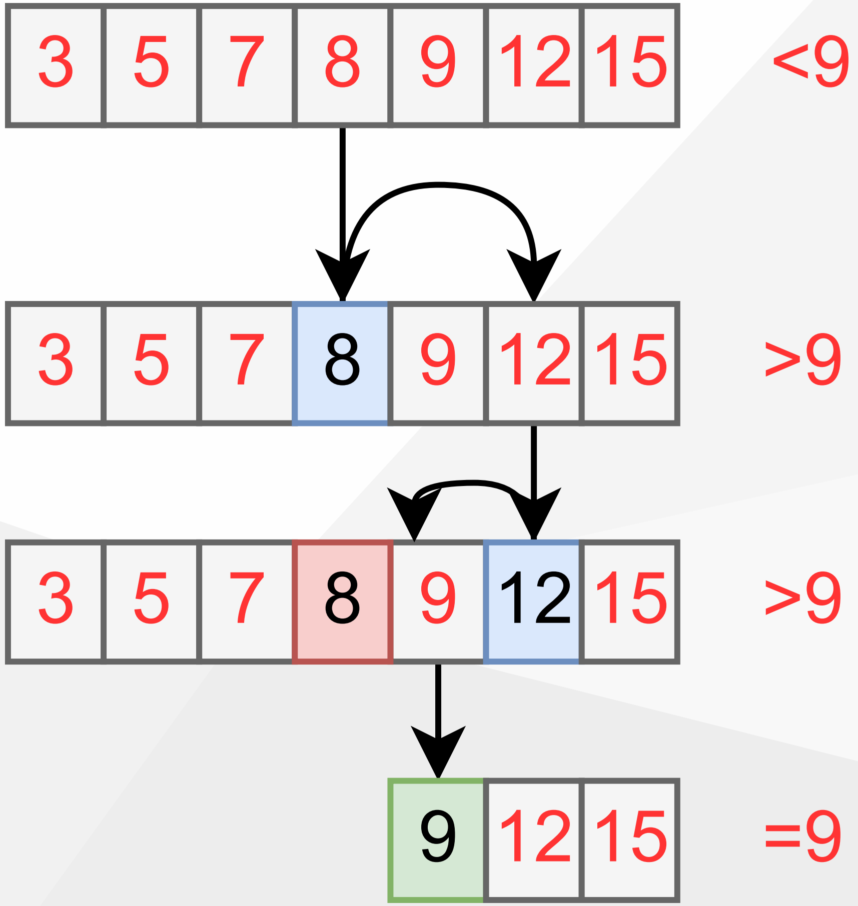
Binary Search (4): Recursive

```
RECURSIVE-BINARY-SEARCH(A,V,low,high)
    if low>high
        return NIL;
    endif

    mid = floor((low+high)/2);

    if v == A[mid]
        return mid;
    elseif v > A[mid]
        return RECURSIVE-BINARY-SEARCH(A,V,mid+1,high);
    else
        return RECURSIVE-BINARY-SEARCH(A,V,low,mid-1);
    endif
```

Binary Search (5): Recursive

$$T(n) = T(n/2) + \Theta(1) \Longrightarrow T(n) = \Theta(lgn)$$

Binary Search (6): Example (Find 9)



Recurrence for Binary Search (7)

$$T(n) = 1T(n/2) + \Theta(1)$$

- Subproblems $\Longrightarrow 1$
- Subproblemsize $\Longrightarrow n/2$
- Work dividing and combining $\Longrightarrow \Theta(1)$

Binary Search: Solving the Recurrence (8)

- $T(n) = T(n/2) + \Theta(1)$
- $a = 1, b = 2, f(n) = \Theta(1) \Longrightarrow n^{log_b^a} = n^0 = 1$
- **Case 2:** $\frac{f(n)}{n^{log_b^a}} = \Theta(lg^k n) \Longrightarrow T(n) = \Theta(n^{log_b^a} lg^{k+1} n)$ holds for $k = 0$
- $T(n) = \Theta(lgn)$

Powering a Number: Divide & Conquer (1)

Problem: Compute an, where n is a natural number

```
NAIVE-POWER(a, n)
    powerVal = 1;
    for i = 1 to n
        powerVal = powerVal * a;
    endfor
return powerVal;
```

- What is the complexity? $\Longrightarrow T(n) = \Theta(n)$

Powering a Number: Divide & Conquer (2)

- Basic Idea:

$$a^n = \begin{cases} a^{n/2} * a^{n/2} & \text{if n is even} \\ a^{(n-1)/2} * a^{(n-1)/2} * a & \text{if n is odd} \end{cases}$$

Powering a Number: Divide & Conquer (3)

```
POWER(a, n)
    if n = 0 then
        return 1;
    else if n is even then
        val = POWER(a, n/2);
        return val * val;
    else if n is odd then
        val = POWER(a,(n-1)/2)
        return val * val * a;
    endif
```

Powering a Number: Solving the Recurrence (4)

- $T(n) = T(n/2) + \Theta(1)$
- $a = 1, b = 2, f(n) = \Theta(1) \implies n^{log_b^a} = n^0 = 1$
- **Case 2:** $\frac{f(n)}{n^{log_b^a}} = \Theta(lg^k n) \implies T(n) = \Theta(n^{log_b^a} lg^{k+1} n)$ holds for $k = 0$
- $T(n) = \Theta(lgn)$

Correctness Proofs for Divide and Conquer Algorithms

- **Proof by induction** commonly used for Divide and Conquer Algorithms
- **Base case:** Show that the algorithm is correct when the recursion bottoms out (i.e., for sufficiently small n)
- **Inductive hypothesis:** Assume the alg. is correct for any recursive call on any smaller subproblem of size k, $(k < n)$
- **General case:** Based on the inductive hypothesis, prove that the alg. is correct for any input of size n

Example Correctness Proof: Powering a Number

- **Base Case:** $POWER(a, 0)$ is correct, because it returns 1
- **Ind. Hyp:** Assume $POWER(a, k)$ is correct for any $k < n$
- **General Case:**
  - In $POWER(a, n)$ function:
    - If n is $even$:
      - $val = a^{n/2}$ (due to ind. hyp.)
      - it returns $val * val = a^n$
    - If n is odd:
      - $val = a^{(n-1)/2}$ (due to ind. hyp.)
      - it returns $val * val * a = a^n$
- The correctness proof is complete

References

- Introduction to Algorithms, Third Edition | The MIT Press
- Bilkent CS473 Course Notes (new)
- Bilkent CS473 Course Notes (old)
- Insertion Sort - GeeksforGeeks
- NIST Dictionary of Algorithms and Data Structures
- NIST - Dictionary of Algorithms and Data Structures
- NIST - big-O notation
- NIST - big-Omega notation

$$-End - Of - Week - 2 - Course - Module-$$