UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Divide and Conquer (1)

Recurrences and The Solution

DAA Term 2 2023/2024

Divide and Conquer

- Insertion sort uses **incremental approach** to sort an array in place
- Another approach: divide and conquer (it uses **recursive** structure)
- Algorithms with divide and conquer paradigm:
  - Break the problem into several sub problems that are similar to the original problem but smaller in size [divide]
  - Solve the sub problems recursively [conquer]
  - Combine the solutions to create a solution to the original problem [combine]
- Example: Merge sort

Merge Sort

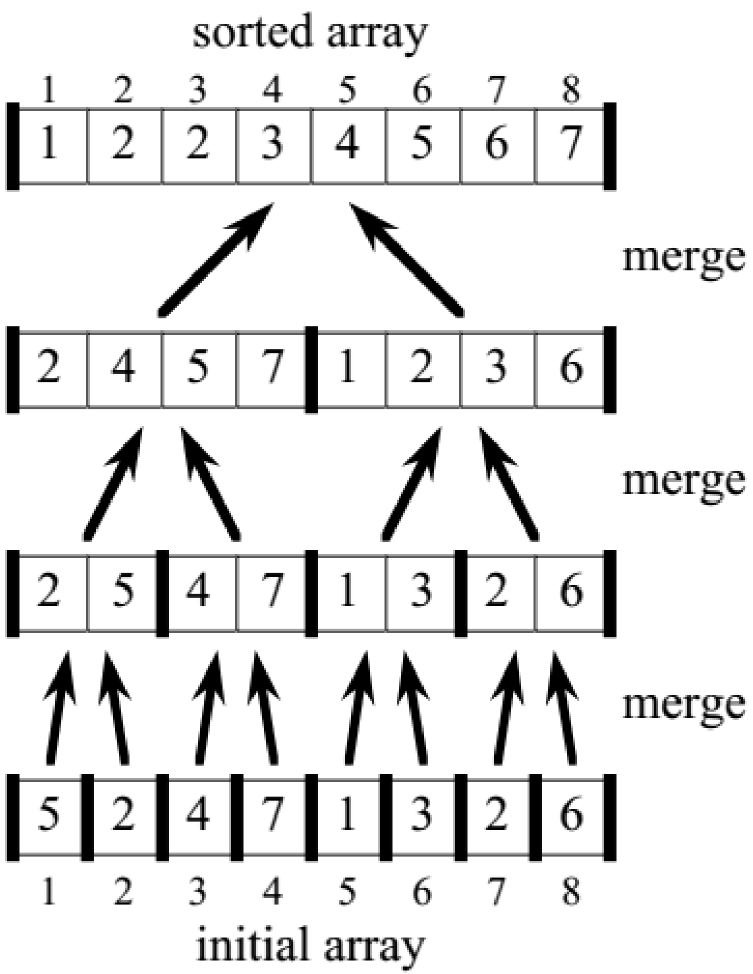
- `Merge-Sort(A, p, r)`

```
1. if p<r                         // check for base case
2.     q = ⌊(p + r)/2⌋            // divide
3.     Merge-Sort(A, p, q)        // conquer
4.     Merge-Sort(A, q+1, r)      // conquer
5.     Merge(A, p, q, r)          // combine
```

- Merge sort divides the array of n-element ($A[p \dots r]$) into two subarrays of $n/2$ elements each ($A[p \dots q]$ and $A[q+1 \dots r]$) [divide]
- The two subarrays ($A[p \dots q]$ and $A[q+1 \dots r]$) are sorted recursively using merge sort [conquer]
- The solution of each subarray (sorted version of $A[p \dots q]$ and $A[q+1 \dots r]$) is merged to produce the sorted array $A[p \dots r]$ [combine]

Merge Sort

The "Merge" Procedure



- Merge(A, p, q, r)

```
n1 = q-p+1
n2 = r-q
L ← [1…n1+1] and R ← [1…n2+1]
for i = 1 to n1
   L[i] = A[p+i-1]
for j = 1 to n2
   R[j] = A[q+j]
L[n1+1] = ∞
L[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
```

$\Theta(n_1)$

$\Theta(n_2)$

Sentinel, to avoid checking whether the subarray is empty

$\Theta(n)$

It takes $\Theta(n)$ in total to combine

Correctness of Merge Sort?

- Show that the **Merge** procedure correctly merges the subarray A[1…q] and A[q+1…r].
  - Loop invariant for **Merge** procedure:

At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p \mathinner{.\,.} k-1]$ contains the $k-p$ smallest elements of $L[1 \mathinner{.\,.} n_1+1]$ and $R[1 \mathinner{.\,.} n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- Show that **Merge-Sort** procedure correctly sorts the whole input array A[1…A.length].

Correctness of Merge Sort? (2)

Correctness of Other Recursive Problems

- Compute x^n, (*x is a nonzero real number and n is a positive integer*)

```
power (x, n):
        if (n = 0) then return 1
        else return x * power (x, n-1)
```

- Compute GCD (x, y), (*x and y are positive integers and x < y*)

```
GCD (x, y):
        if (x = 0) then return y
        else return GCD (y mod x, x)
```

Complexity

- Let $T(n)$ denotes the **running time of a recursive algorithm** with input size n. Divide the problems into a smaller problems, size $\frac{n}{b}$ each.
  - It takes $D(n)$ to **divide** the original problem, $T\left(\frac{n}{b}\right)$ to **solve (conquer)** each sub problems, and $C(n)$ to **combine** the solutions.

$$T(n) = \begin{cases} \Theta(1), if\ n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), otherwise \end{cases}$$

    - When the input size is small enough $(n \leq c)$ for some constant c, the running time is constant

Complexity of Merge Sort?

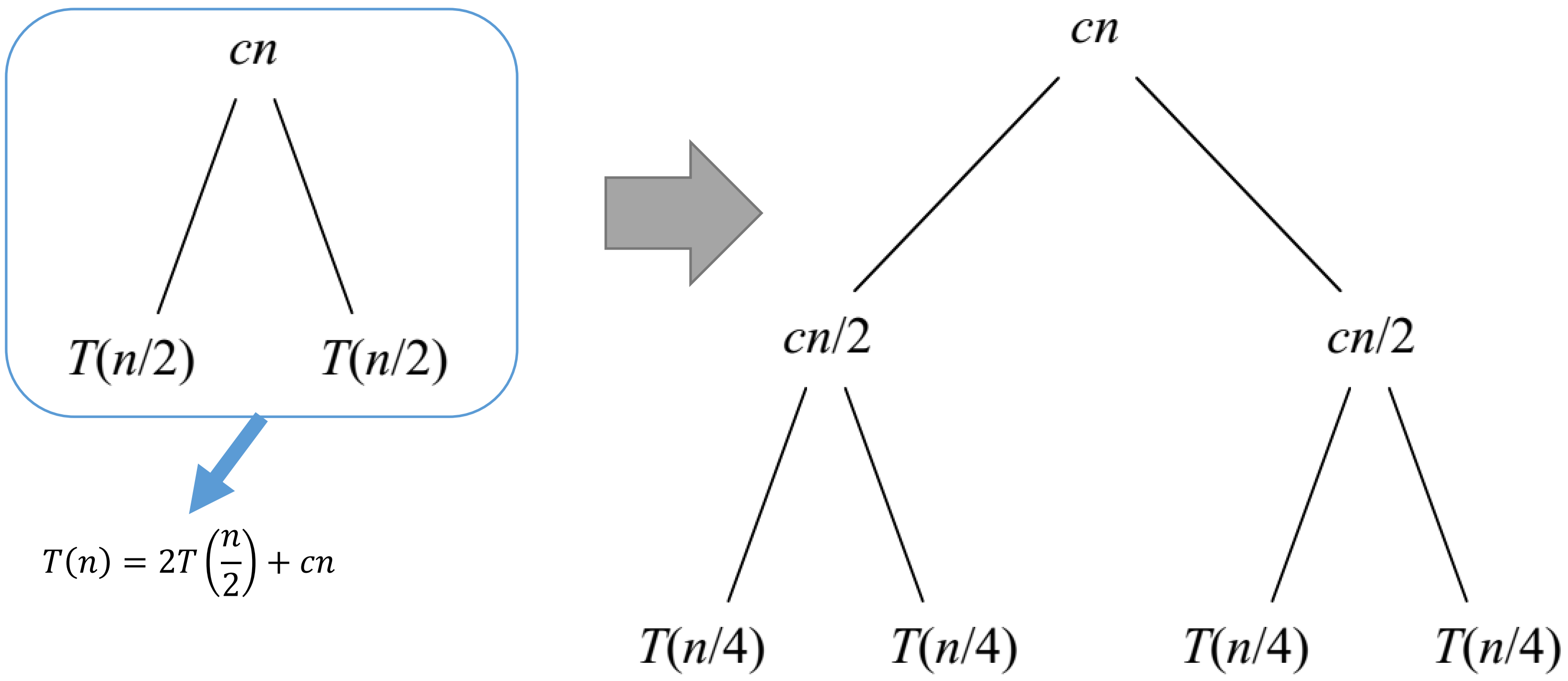
- Assume that n is a power of 2 → each divide step yields two sub problems, both of size exactly $\frac{n}{2}$
- Let the running time of Merge Sort for n-element is $T(n)$
  - **Divide**: Compute the average of p and r → takes constant time ($\Theta(1)$)
  - **Conquer**: Solve 2 sub problems recursively, each takes $T\left(\frac{n}{2}\right)$ → $2T\left(\frac{n}{2}\right)$
  - **Combine**: Merge n-element sub array → $\Theta(n)$

- Running time of Merge Sort in a recurrence function:

$$T(n) = \begin{cases} \Theta(1), if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), if\ n > 1 \end{cases}$$

Complexity of Merge Sort?

- The recursion tree illustration:



Solving Recurrences

Recurrences

- Following recurrence function denotes the running time of Merge Sort.

$$T(n) = \begin{cases} \Theta(1), if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), if\ n > 1 \end{cases}$$

- We need to know the **explicit form** of this recurrence function.
  - Example: Merge sort runs in $\boldsymbol{\Theta(n \lg n)}$

Recurrences

- A recurrence is
  - a **recursive description** of a function, or
  - a description of a function **in terms of itself**
  - that consists of
    - one or more **base cases**
    - one or more **recursive cases**

- A solution of a recurrence is:
  - A **non-recursive description** of a function that satisfies the recurrence.
  - It is also known as the **closed form** or **explicit form.**
  - It can be defined as
    - An **exact, tight solution**, or
    - A solution written in **asymptotic notation**

Example

- Tower of Hanoi

$$T(n) = \begin{cases} 0 & if\ n = 0 \\ 2T(n-1)+1 & otherwise \end{cases}$$

Solution: $\mathbf{2^n - 1}$

- Merge Sort

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

Solution: $\boldsymbol{\Theta(n \lg n)}$

Solving a Recurrence

- **Iterative method**
  - Iteratively expand the function until we reach the boundary condition
- **Substitution method (Guess and Prove)**
  - Guess the answer (sometimes we use the **iterative method** or **recursion tree**), then prove it by using mathematical induction (explicitly).
- **Recursion tree**
  - Like the **iterative method**, but it is visualized in a tree structure.
  - It can be used to generate a good guess for Substitution Method
- **Master method**
  - Existed theorem to understand the running time of an algorithm from its recurrence function $T(n) = aT(n/b) + f(n), a \geq 1, b > 1$

Iterative Method

- Tower of Hanoi

$$T(n) = \begin{cases} 0 & if\ n = 0 \\ 2T(n-1) + 1 & otherwise \end{cases}$$

$$\begin{aligned} 2T(n-1) + 1 &= 2(2T(n-2) + 1) + 1 \\ &= 2^2T(n-2) + 3 \\ &= 2^3T(n-3) + 7 \\ &= \cdots \\ &= 2^iT(n-i) + \left(2^i - 1\right) \\ &= \cdots \\ &= 2^nT(0) + (2^n - 1) \end{aligned}$$

The boundary condition is reached when $i = n$, hence $T(n) = 2^n - 1$

Iterative Method

- Merge Sort

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

$$\begin{aligned} 2T\left(\frac{n}{2}\right) + cn &= 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\ &= \cdots \end{aligned}$$

Substitution / Guess and Prove

- Example 1: Tower of Hanoi

$$T(n) = \begin{cases} 0 & if\ n = 0 \\ 2T(n-1) + 1 & otherwise \end{cases}$$

Solution: $2^n - 1$

Show that the closed form of recurrence above is $2^n - 1$

- Base Case: when $n = 0$ then $2^0 - 1 = 0 = T(0)$ ■
- Inductive Case:
  - Assume $\underline{T(k) = 2^k - 1 \text{ is true for all } k < n}$, so our Inductive Hypotheses is $T(n-1) = 2^{n-1} - 1$
  - Show that $T(n) = 2^n - 1$ is also true

Substitution / Guess and Prove

- Example 1 (cont’d)

Substitution / Guess and Prove

- Example 2: A recurrence function is defined below.

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

Show that the solution is $n \lg n + n$

Substitution / Guess and Prove

- Example 2 (cont’d)

Substitution / Guess and Prove

- Example 3: Merge Sort

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

Show that the solution is $\Theta(n \lg n)$

Substitution / Guess and Prove

- Example 3 (cont'd)

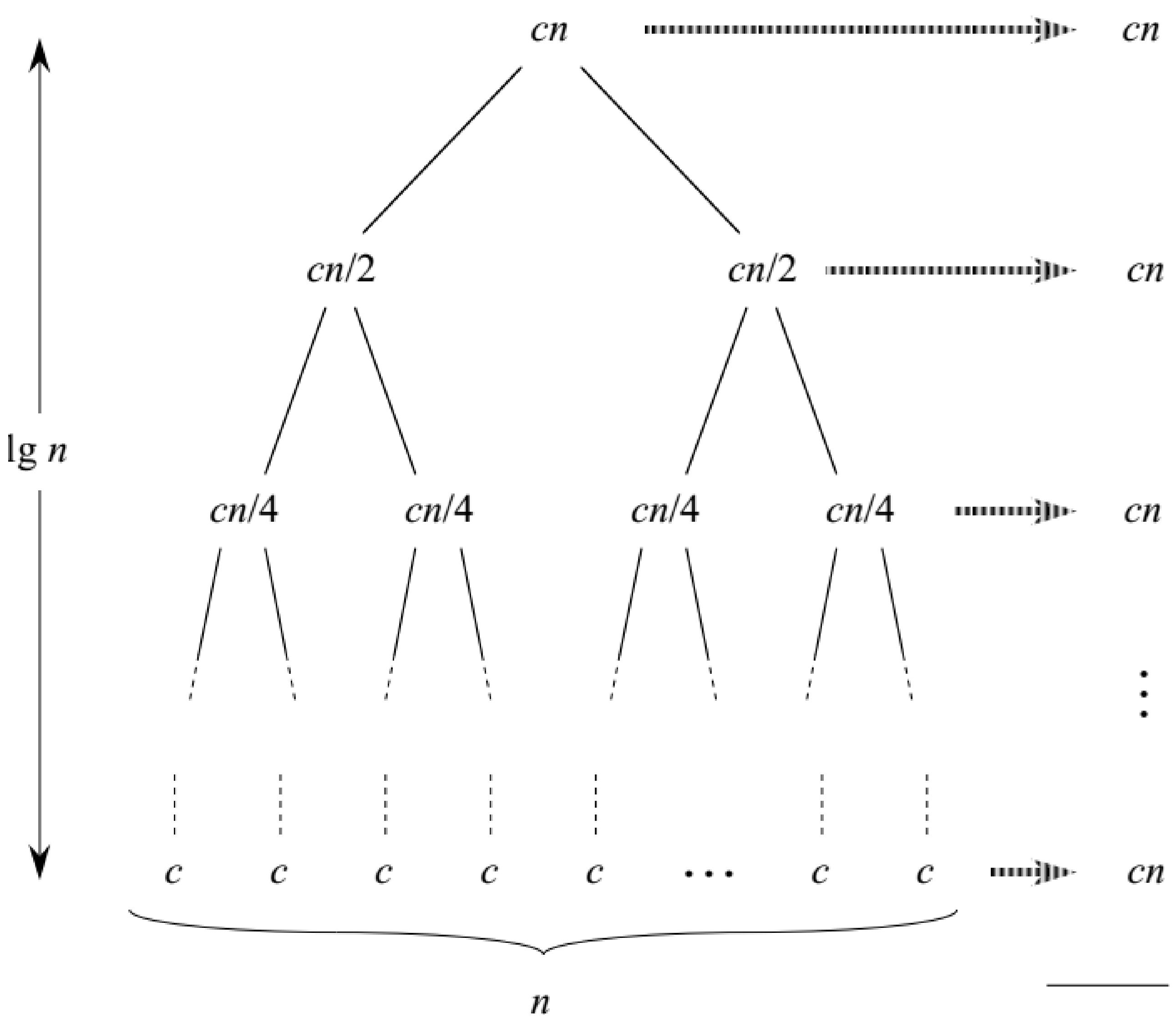
More Examples

- Show that $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$ is $O(n^3)$.
  - Show an explicit proof
    - For example by lowering the upper bound
      - Assume that we want to show that $T(n) \leq cn^3 - dn^2$
      - Our IH: $T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^3 - d\left(\frac{n}{2}\right)^2$

More Examples

- Find the solution of $T(n) = 2T(\sqrt{n}) + \lg n$.
  - By changing variable: Bring this form into a "standard" recursive function, i.e. $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.
    - For example by substituting n with 2^m, so $\lg n = m$

Recursion Tree



Merge Sort

- $\lg n + 1$ levels
- Total cost = $cn(\lg n + 1)$

  = $cn \lg n + cn$
- $T(n) = \Theta(n \lg n)$

Recursion Tree

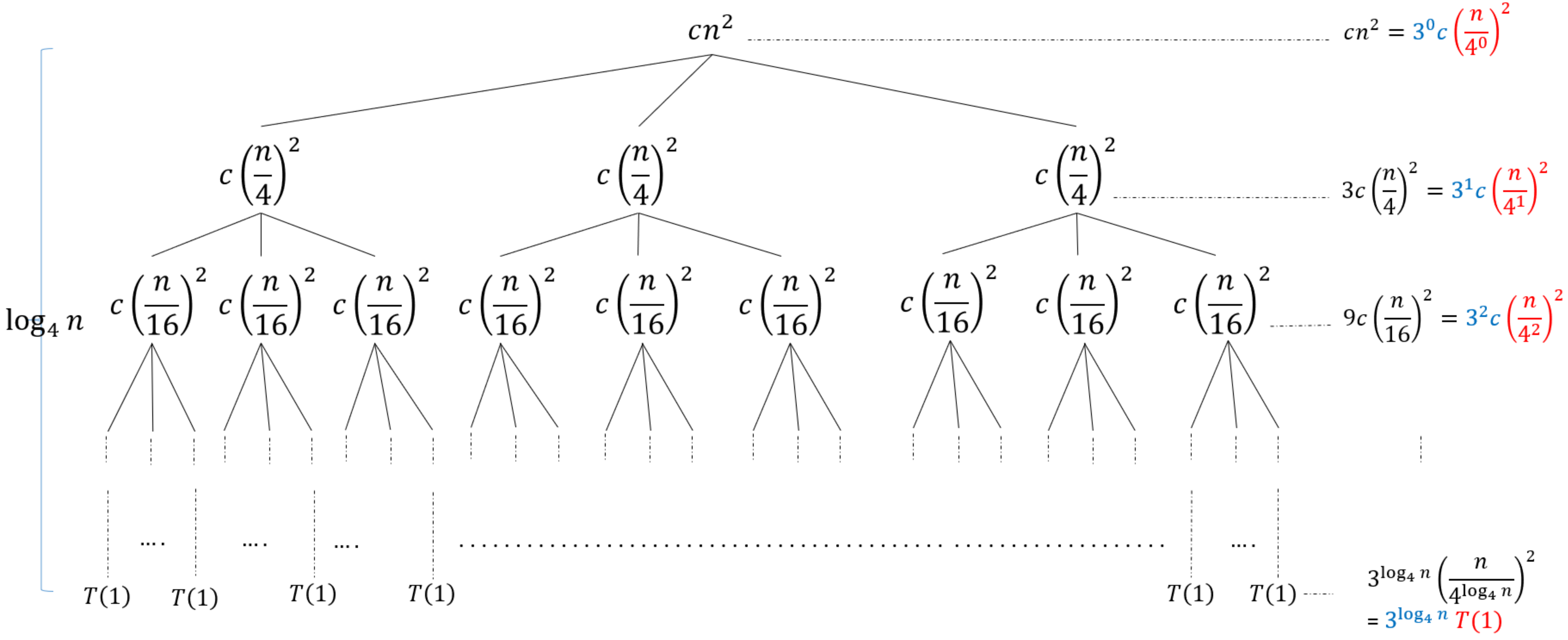
- Tower of Hanoi
  - Cost at level-i = $1 \cdot 2^i$
  - Height of the tree = n (there are $n + 1$ levels)
  - Total cost = $1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$
  - $T(n) = 2^n - 1$

Recursion Tree

- Example: $T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \Theta(n^2)$
  - Assumption: ignore the floor function, assume n is power of 4
  - Build recursion tree for $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$
  - At level-i:
    - Size of each sub problem → $\frac{n}{4^i}$
    - Number of node → 3^i
  - At leaf (when the size of sub problem =1):
    - $\frac{n}{4^i} = 1 \Leftrightarrow n = 4^i \Leftrightarrow i = \log_4 n$
  - Height of the tree = $\log_4 n$, number of level = $\log_4 n + 1$
  - Number of node in level $\log_4 n = 3^{\log_4 n}$ (leaf)

Recursion Tree

- $T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \Theta(n^2)$



Recursion Tree

Summary for the previous illustration:

- Total cost at each level = number of node * cost for each node
  - Cost at level $i = 3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$, for $i = 0,1,2,\ldots,\log_4 n - 1$
  - Cost at leaf or level $\log_4 n = 3^{\log_4 n}\left(\frac{n}{4^{\log_4 n}}\right)^2 = \Theta\left(n^{\log_4 3}\right)$
    - $3^{\log_4 n} = n^{\log_4 3}$
- Total cost for the entire tree:
  - $\mathrm{T}(n) = \sum_{i=0}^{(\log_4 n)-1}\left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \mathrm{O}(n^2)$

Recursion Tree

- The detailed calculation to obtain $O(n^2)$:

$$
\begin{aligned}
T(n) &= cn^2 + \tfrac{3}{16}cn^2 + \left(\tfrac{3}{16}\right)^2 cn^2 + \cdots + \left(\tfrac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\tfrac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\tfrac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \tfrac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2).
\end{aligned}
$$

> Next, we can use the substitution method to verify that T(n) = O(n^2) is an upper bound for the given recurrence. We have to show that T(n) ≤ dn^2 for some constant d > 0.

$$
\begin{aligned}
T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\
&\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\
&\le 3d(n/4)^2 + cn^2 \\
&= \tfrac{3}{16} dn^2 + cn^2 \\
&\le dn^2, \qquad \text{provided that } d \ge (16/13)c.
\end{aligned}
$$

- *From lecturer slide by Bpk LYS*

Exercise

- Use recursion tree to find/guess the solution of

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

  - Assume the running time is constant when $n = 1$

Master Theorem

- Let $a \geq 1$ and $b > 1$ be constant. Let $f(n)$ be a function > 0 and $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where we interpret $\frac{n}{b}$ to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$.

Then $T(n)$ has the following asymptotic bounds:

1. If $\boldsymbol{f(n) = O\left(n^{\log_b a - \epsilon}\right)}$ for some constant $\epsilon > 0$, then $\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right)}$
2. If $\boldsymbol{f(n) = \Theta\left(n^{\log_b a}\right)}$, then $\boldsymbol{T(n) = \Theta\left(n^{\log_b a} \lg n\right)}$

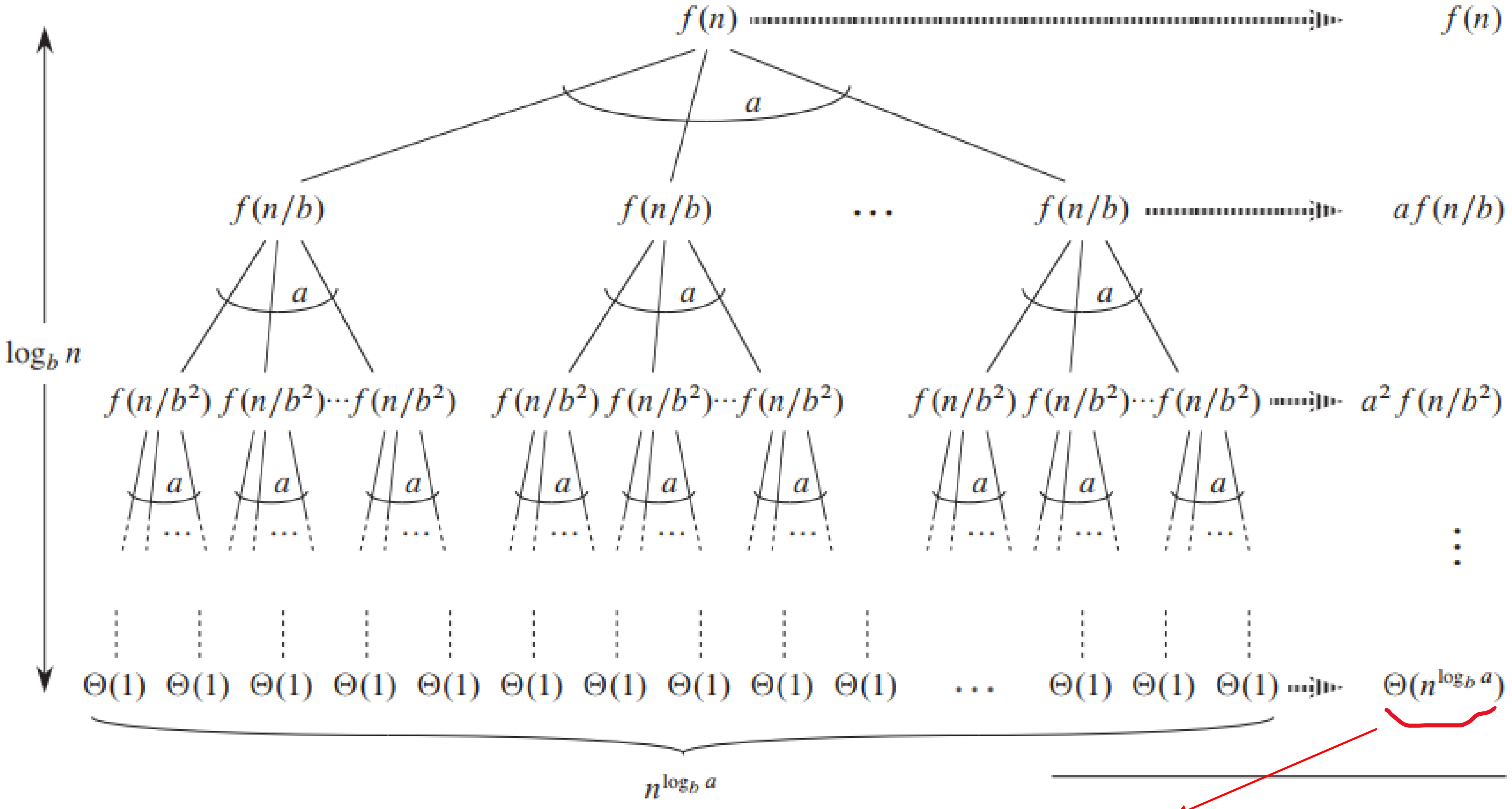
   In general: If $\boldsymbol{f(n) = \Theta\left(n^{\log_b a} \lg^k n\right)}$, then $\boldsymbol{T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)}$ for a constant $k \geq 0$
3. If $\boldsymbol{f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)}$ for some constant $\epsilon > 0$, and if $\boldsymbol{af\left(\frac{n}{b}\right) \leq cf(n)}$ for some constant c < 1 and all sufficiently large n, then $\boldsymbol{T(n) = \Theta(f(n))}$ -> regularity condition

Master Theorem

- What does it mean?
  - Suppose we have a recursion tree for function $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
  - By using master theorem, we compare the cost in the root of the recursion tree and in the remaining subtree.
  - The solution to the recurrence function: the most dominant cost.

Master Theorem

It compares $\boldsymbol{f(n)}$ as the driving function to $\boldsymbol{n^{log_b a}}$ (cost on all leaves)



Cost on all internal nodes.

Cost on all leaves

Which one is dominating the total cost?

$$\text{Total: } \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j\, f(n/b^j)$$

Master Theorem

Intuitively, we compare the function $f(n)$ (driving function) with the function $n^{\log_b a}$ (watershed function). The larger *(polinomially larger)* of the two functions determines the solution of the recurrence.

- Case 1: $f(n)$ is polynomially smaller than $n^{\log_b a}$
  - $T(n) = \Theta\left(n^{\log_b a}\right)$ → the leaves dominate the total cost
- Case 2: $f(n)$ is (nearly) equal to $n^{\log_b a}$
  - $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$ → the cost is distributed evenly among the levels of the tree
- Case 3: $f(n)$ is polynomially larger than $n^{\log_b a}$ and $f(n)$ satisfy the **regularity condition** $af\left(\frac{n}{b}\right) \leq cf(n)$
  - $T(n) = \Theta\left(f(n)\right)$ → the root dominates the total cost

Master Theorem

Example

- Find the solution of $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$.
  - $a = 2, b = 2, f(n) = \Theta(n), n^{\log_b a} = n$
  - Therefore, $f(n) = \Theta(n)$ (case 2 applied)
  - The solution is $T(n) = \Theta\left(n^{\log_b a} lg\ n\right) = \Theta(n\ lg\ n)$

- Can we use master method to solve

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

Exercise

- Find the solution of following recurrences using master method (if applicable):
  - $T(n) = 9T\left(\frac{n}{3}\right) + n$
  - $T(n) = T\left(\frac{2n}{3}\right) + 1$
  - $T(n) = 2T(n/2) + n \lg n$
  - $T(n) = 2T\left(\frac{n}{2}\right) + 1$
  - $T(n) = 4T(n/2) + n^3$
  - $T(n) = 5T(n/3) + \Theta(n^3)$
  - $T(n) = 27T(n/3) + \Theta(n^3/\lg n)$

Conclusion

- Solving Recurrences
  - Recursion tree
    - Recursion tree and iterative method are similar
    - With extra care in the development of a recursion tree, it can be used as direct proof for a solution to a recurrence.
    - When some tolerable sloppiness are applied, the substitution method is necessary to complete the proof.
      - Recursion tree can be used to generate a “good guess” for the substitution method
  - Substitution
    - Based on the concept of “mathematical induction”
    - Proof the solution explicitly.
  - Master method
    - A “cook book” for solving a recurrence function
    - Not applicable for all recursive functions

References

- Lecturer Slides by Bapak L. Yohanes Stefanus

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.