

CO202 – Software Engineering – Algorithms
Divide and Conquer

Ben Glocker
Huxley Building, Room 377
b.glocker@imperial.ac.uk

Algorithm Design

How to design an (efficient) algorithm?

Structure the problem!*

Make use of **algorithmic schemes/design paradigms**

- Incremental Approach
 - **Divide and Conquer** (this week)
 - Dynamic Programming
 - Greedy Algorithms
- } upcoming weeks

*Take some time and think!

Divide and Conquer

Solve a problem recursively, apply three steps:

1. **Divide** the problem into a number of **subproblems** that are smaller instances of the same problem
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are **small enough**, just solve them in a **straightforward manner**
3. **Combine** the solutions to the subproblems into the solution for the original problem

D&C Principle

In Divide and Conquer, we solve a problem **recursively**

- When the subproblems are large enough to solve recursively, we call that the **recursive case**
- Once the subproblems become small enough, we say the recursion “bottoms out” and we have gotten down to the **base case**

Example: Sorting Problem

- **Input:** Sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 7 | 3 | 6 | 2 | 1 | 5 | 2 | 4 |

- **Output:** Permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 |

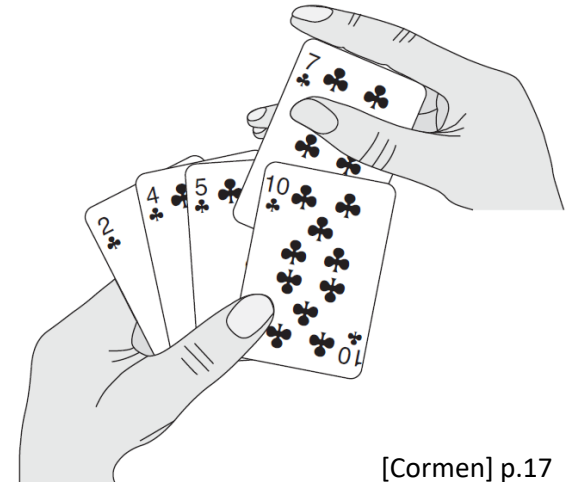
Reminder: Insertion Sort

Incremental Approach

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 7 | 3 | 6 | 2 | 1 | 5 | 2 | 4 |

INSERTION-SORT(A)

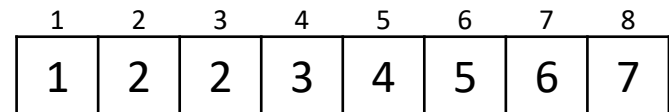
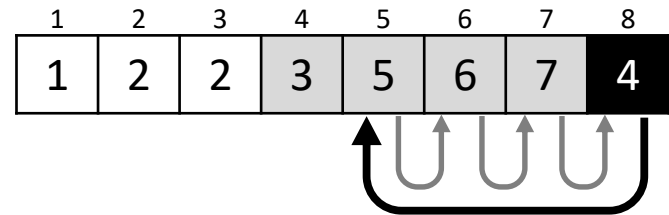
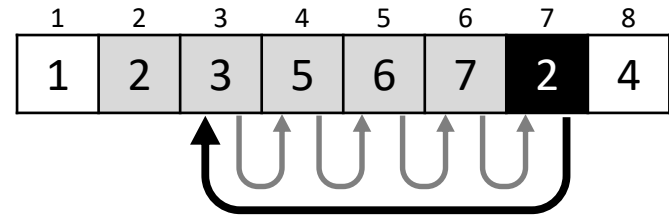
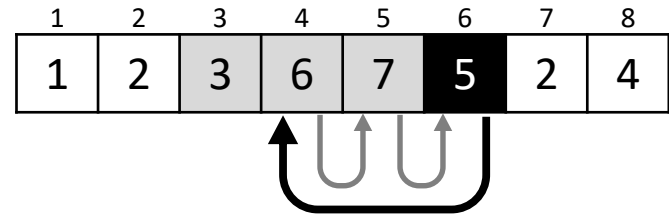
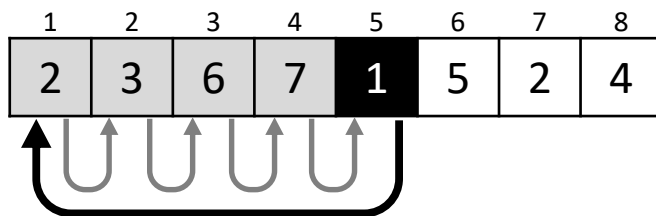
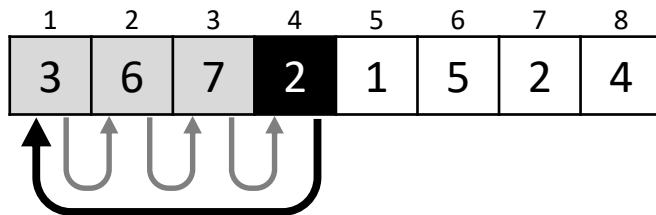
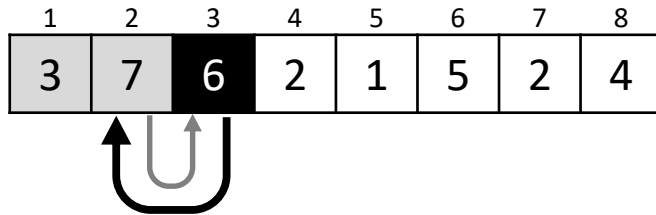
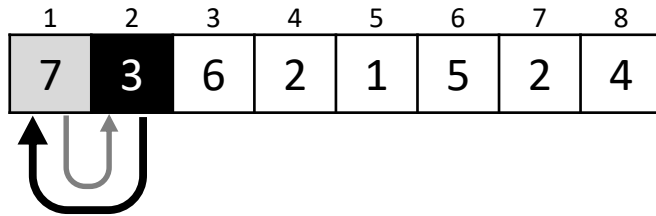
```
1: for j = 2 to A.length
2:     key = A[j]
3:     # Insert A[j] into the sorted sequence A[1..j-1].
4:     i = j-1
5:     while i > 0 and A[i] > key
6:         A[i+1] = A[i]
7:         i = i-1
8:     A[i+1] = key
```



[Cormen] p.17

Reminder: Insertion Sort (cont'd)

key line 2 A[i] line 5



Running Time of Insertion Sort

Depends on the length of the sequence $n = A.length$

INSERTION-SORT(A)

cost

times

```
1: for j = 2 to A.length
2:   key = A[j]
3:   # Insert A[j]...
4:   i = j-1
5:   while i > 0 and A[i] > key
6:     A[i+1] = A[i]
7:     i = i-1
8:   A[i+1] = key
```

$$T(n) = c_1n + c_2(n - 1) + c_4(n - 1) \\ + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1)$$

Running Time of Insertion Sort

Depends on the length of the sequence $n = A.length$

| INSERTION-SORT(A) | cost | times |
|--|-------|--------------------------|
| 1: for $j = 2$ to $A.length$ | c_1 | n |
| 2: $key = A[j]$ | c_2 | $n-1$ |
| 3: # Insert $A[j]...$ | 0 | $n-1$ |
| 4: $i = j-1$ | c_4 | $n-1$ |
| 5: while $i > 0$ and $A[i] > key$ | c_5 | $\sum_{j=2}^n t_j$ |
| 6: $A[i+1] = A[i]$ | c_6 | $\sum_{j=2}^n (t_j - 1)$ |
| 7: $i = i-1$ | c_7 | $\sum_{j=2}^n (t_j - 1)$ |
| 8: $A[i+1] = key$ | c_8 | $n-1$ |

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1)$$

Running Time of Insertion Sort (cont'd)

Depends also on the nature of the input

- **Best-case** (already sorted): $t_j = 1$

$$T_{\text{best}}(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1)$$

- **Worst-case** (reversely sorted): $t_j = j$

$$T_{\text{worst}}(n) = c_1 n + c_2(n - 1) + c_4(n - 1)$$

Reminder

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$
$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

$$+ c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n - 1)$$

Running Time of Insertion Sort (cont'd)

Depends also on the nature of the input

- **Best-case** (already sorted): $t_j = 1$

$$T_{\text{best}}(n) = \text{linear}$$

- **Worst-case** (reversely sorted): $t_j = j$

$$T_{\text{worst}}(n) = \text{quadratic}$$

Reminder

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$
$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

Worst-case Running Time

We usually focus on the worst-case running time:

- It provides an **upper bound** on the running time **for any input**. It is guaranteed that the algorithm will never take any longer.
- Worst-case can occur fairly often, e.g. in searching a database for a piece of information that is not present.
- “Average-case” is often roughly as bad as the worst-case

Merge Sort

Divide & Conquer Approach

1. **Divide** the n -element sequence to be sorted into two subsequences of $n/2$ elements each
2. **Conquer:** Sort the two subsequences recursively using merge sort
3. **Combine:** Merge the two sorted subsequences to produce the sorted answer

Combine: Merge

MERGE(A, p, q, r)

1: $n_1 = q - p + 1$

2: $n_2 = r - q$

3: let $L[1..n_1+1]$ and $R[1..n_2+1]$ be new arrays

4: **for** $i = 1$ **to** n_1

5: $L[i] = A[p+i-1]$

6: **for** $j = 1$ **to** n_2

7: $R[j] = A[q+j]$

8: $L[n_1+1] = \infty$

9: $R[n_2+1] = \infty$

10: $i = 1$

11: $j = 1$

12: **for** $k = p$ **to** r

13: **if** $L[i] \leq R[j]$

14: $A[k] = L[i]$

15: $i = i + 1$

16: **else**

17: $A[k] = R[j]$

18: $j = j + 1$

length of 1st subarray

length of 2nd subarray

copy values to 1st array

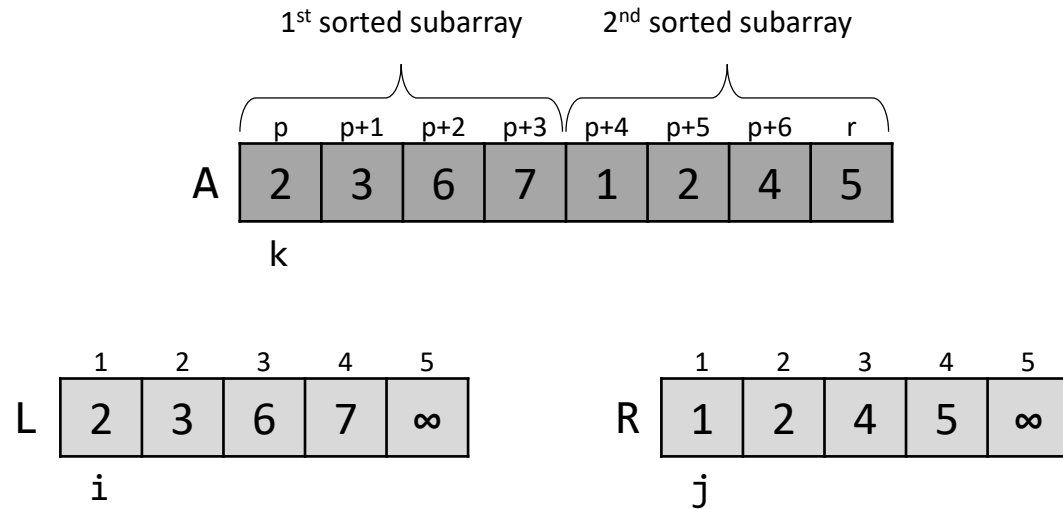
copy values to 2nd array

set sentinel

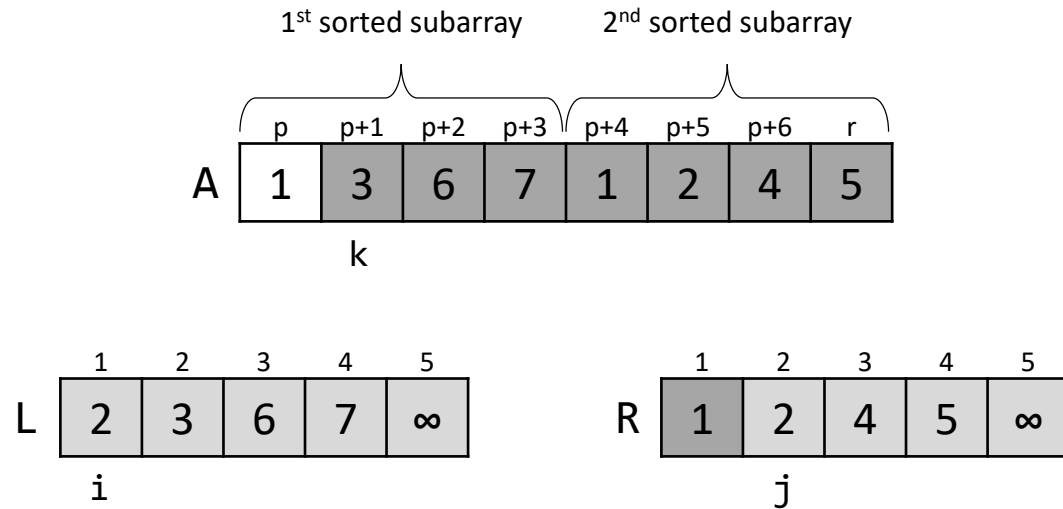
set sentinel

merge subarrays

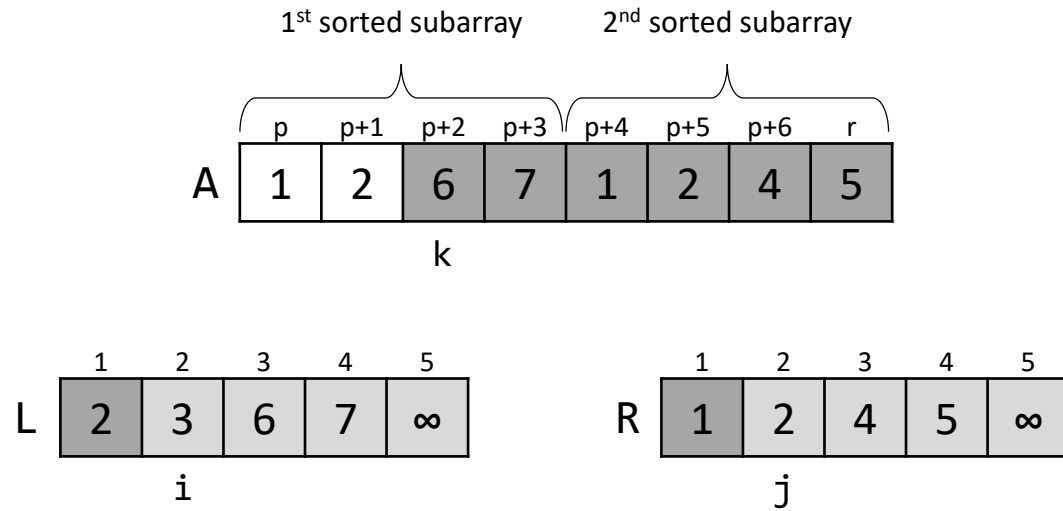
Combine: Merge (cont'd)



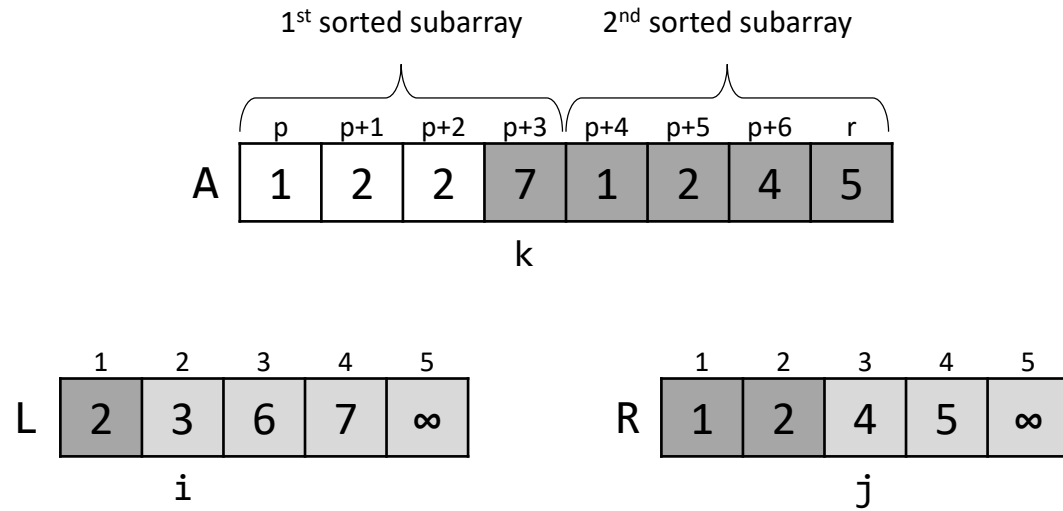
Combine: Merge (cont'd)



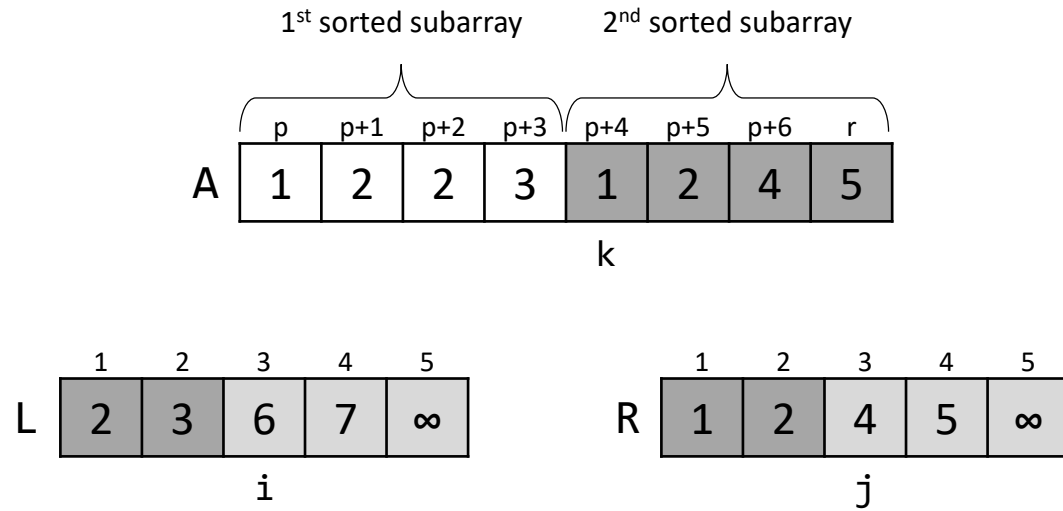
Combine: Merge (cont'd)



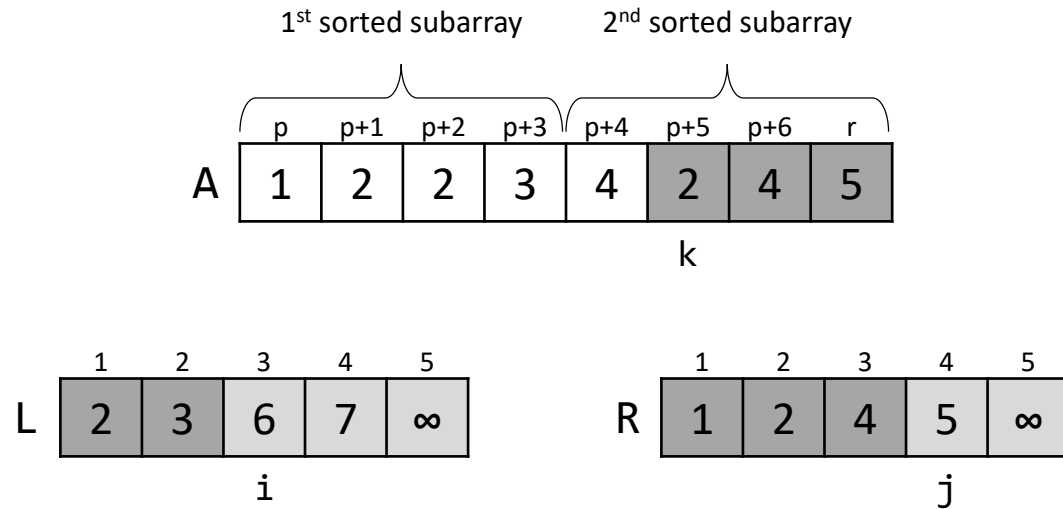
Combine: Merge (cont'd)



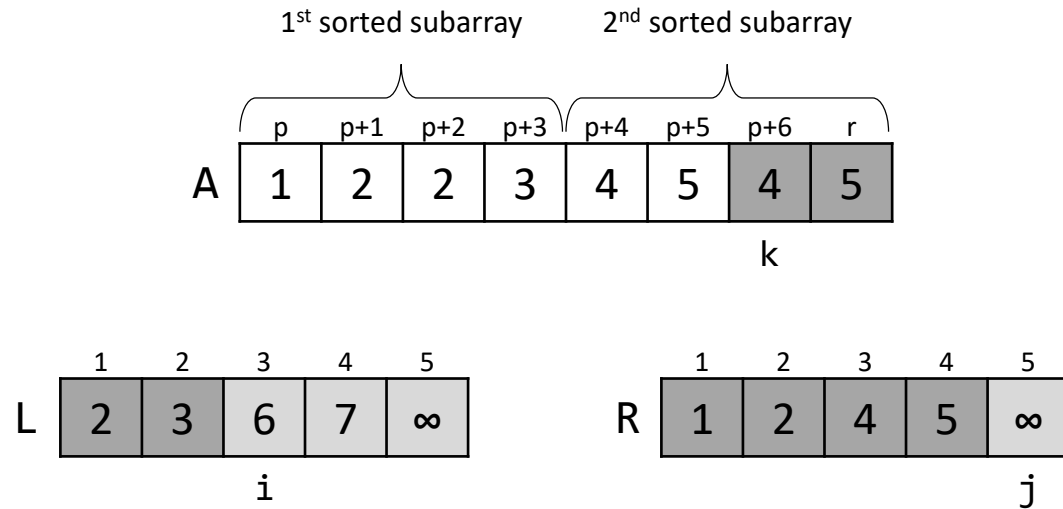
Combine: Merge (cont'd)



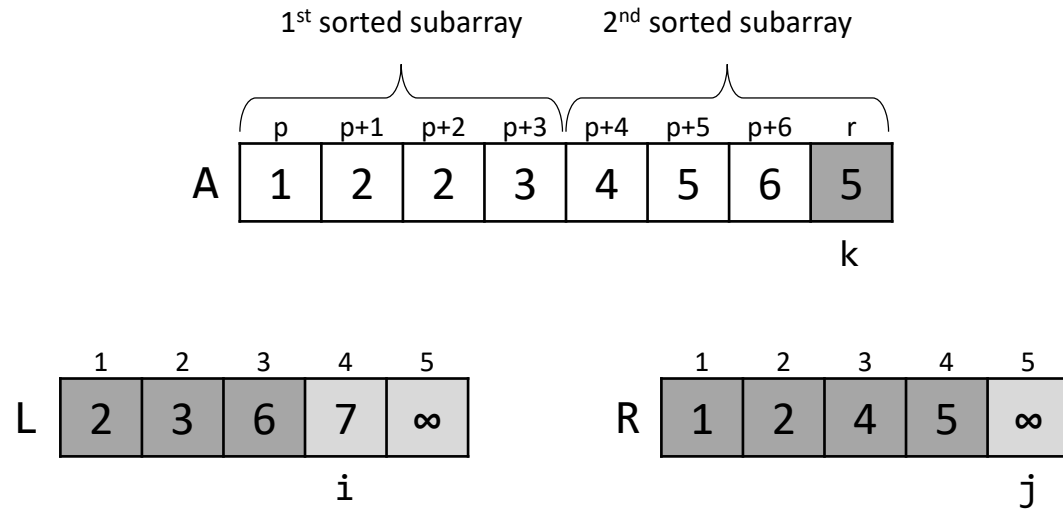
Combine: Merge (cont'd)



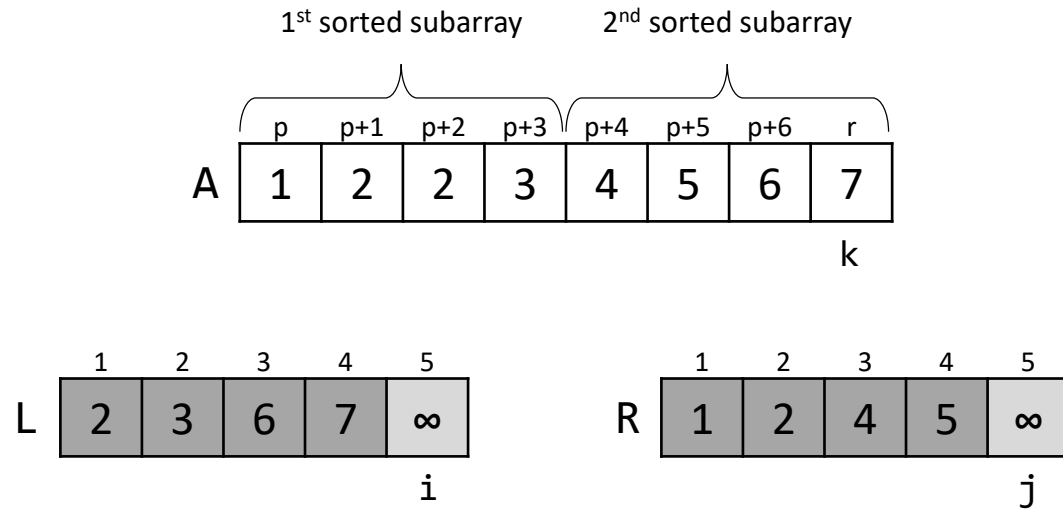
Combine: Merge (cont'd)



Combine: Merge (cont'd)



Combine: Merge (cont'd)



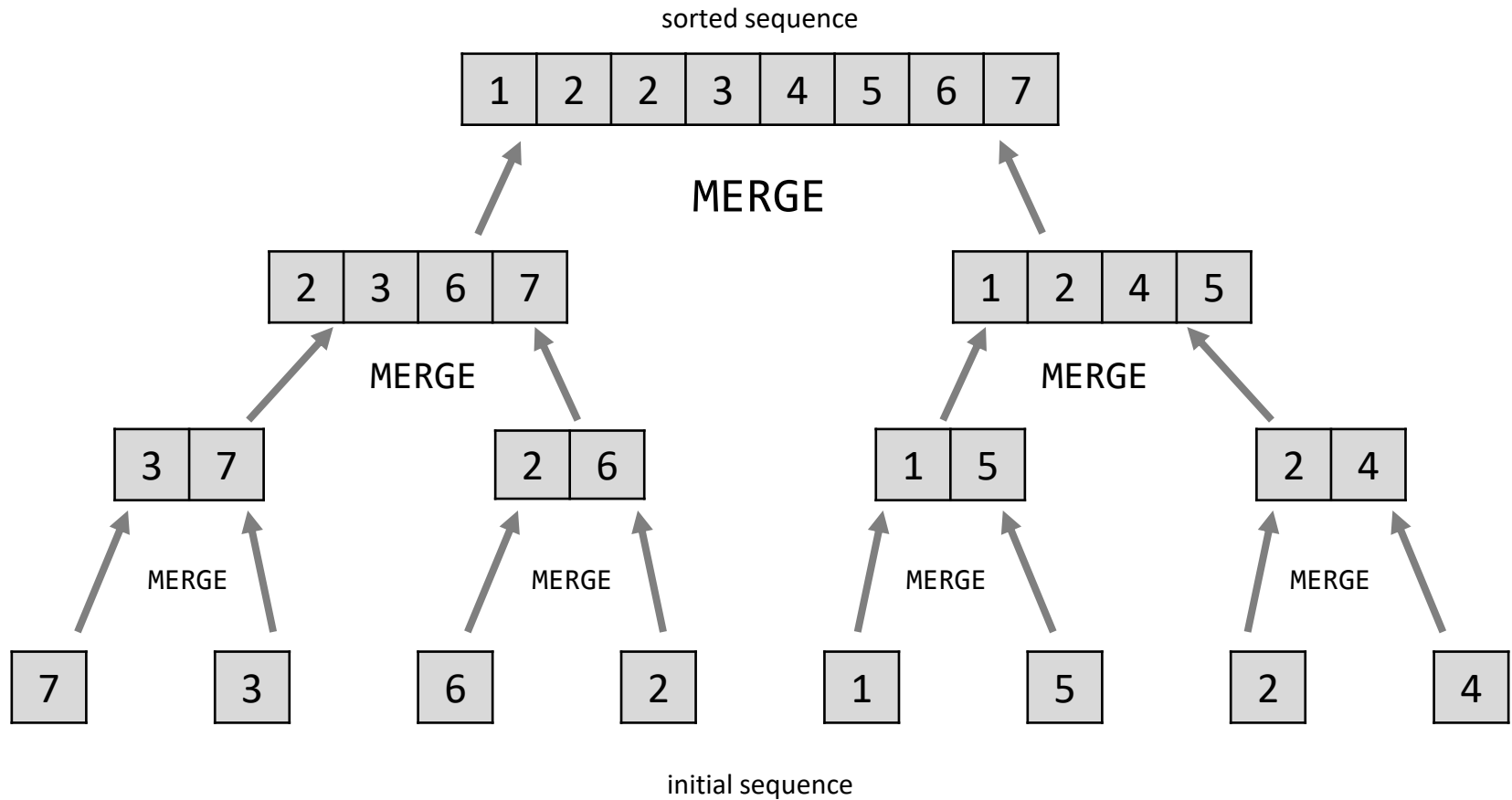
Merge Sort (cont'd)

Divide and Conquer Approach

MERGE-SORT(A, p, r)

| | |
|----------------------------------|------------------------------------|
| 1: if $p < r$ | # more than 1 item? |
| 2: $q = \text{floor}((p+r)/2)$ | # divide array |
| 3: MERGE-SORT(A, p, q) | # conquer 1 st subarray |
| 4: MERGE-SORT($A, q+1, r$) | # conquer 2 nd subarray |
| 5: MERGE(A, p, q, r) | # combine subarrays |

Merge Sort (cont'd)



Running Time of Merge Sort

How to deal with recursion in running time analysis?

| MERGE-SORT(A,p,r) | cost | times |
|--------------------------------|-------|-------|
| 1: if $p < r$ | c_1 | ? |
| 2: $q = \text{floor}((p+r)/2)$ | c_2 | ? |
| 3: MERGE-SORT(A,p,q) | c_3 | ? |
| 4: MERGE-SORT(A,q+1,r) | c_4 | ? |
| 5: MERGE(A,p,q,r) | c_5 | ? |

Recurrences

A recurrence is an equation (or inequality) that describes a function in terms of its value on smaller inputs

Recurrence for Divide & Conquer

- Trivial problems $n \leq c$ can be solved in constant time
- Suppose division yields a subproblems each of size n/b
- **Divide** takes time $D(n)$ and **combine** takes $C(n)$

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

Merge Sort Recurrence

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

- **Base case:** Merge Sort on one element takes constant time: $c = 1$
- **Divide:** Computes the middle of the subarray in constant time: $D(n) = \Theta(1)$
- **Conquer:** Recursively solve two subproblems of size $n/2$, thus: $a = 2, b = 2$
- **Combine:** The MERGE procedure on an n -element subarray takes linear time, so $C(n) = \Theta(n)$

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

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

Technicalities in Recurrences

Calling MERGE-SORT on n elements when n is odd yields

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

We neglect certain technical details, because in many cases it doesn't make a difference

- Omit the **boundary conditions** (constant for small n)
- Omit **floors and ceilings**, assume n is power of b (which is 2 above)

Solving Recurrences

The three main methods for solving recurrences, and obtaining asymptotic bounds on the running time

- **Substitution Method:** Guess a bound and use induction to prove it is correct
- **Recursion Tree Method:** Convert recurrence into a tree whose nodes represent the costs at different levels
- **Master Method:** Look-up bounds for recurrences of

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

Substitution Method

Two steps:

1. Guess the form of the solution
 2. Use mathematical induction to show it's correct
- Making a good guess can be difficult and requires experience (and sometimes creativity)
 - Inductive proof can be tricky and sometimes needs a bit of algebraic mastery

Substitution Method (cont'd)

Example

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

Guess: $O(n \lg n)$

Prove that $T(n) \leq c(n \lg n)^*$

Assume this holds for all positive $m < n$ (strong induction),
in particular for $m \leq n/2$, yielding $T(n/2) \leq c(n/2 \lg n/2)$

By substitution:

$$T(n) \leq 2(c(n/2 \lg n/2)) + n$$

$$= cn \lg n/2 + n$$

$$= cn \lg n - cn \lg 2 + n$$

$$= cn \lg n - cn + n$$

$$\leq cn \lg n$$

holds for $c \geq 1$

What about the base case? $T(1) = 1 \not\leq c1 \lg 1 = 0$

Need to choose a suitable n_0 , such that $\forall n \geq n_0$. $T(2) = 4 \leq c2 \lg 2 = c2$ holds for $c \geq 2$

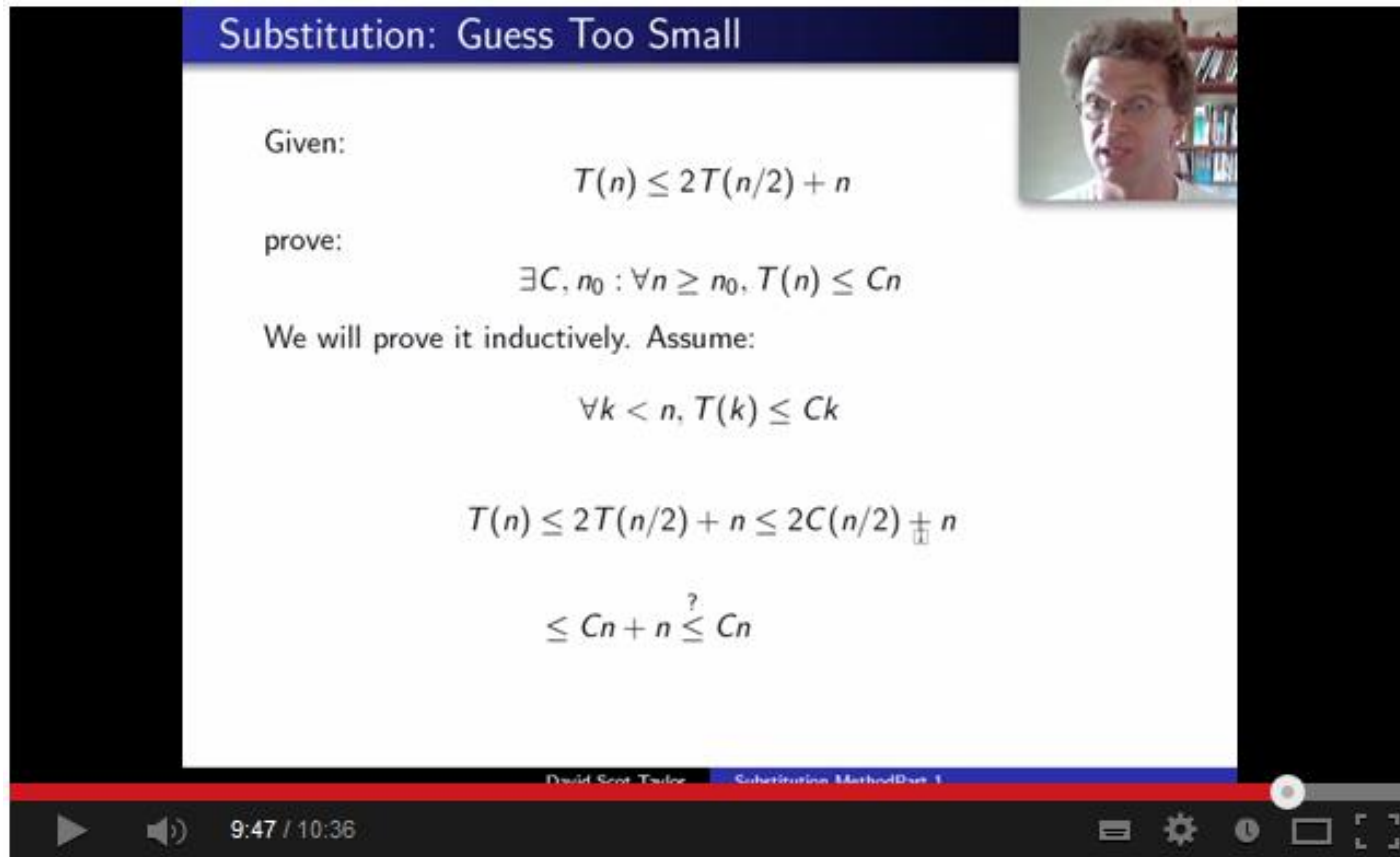
■

$$*O(g(n)) = \{f(n): \exists c, n_0 > 0 \text{ such that } \forall n \geq n_0 \ 0 \leq f(n) \leq cg(n)\}$$

Substitution Method (cont'd)

What happens if guess is too small?

- <http://youtu.be/ad5zs6Uin3U?t=9m11s>



The video player shows a lecture titled "Substitution: Guess Too Small". The content on the screen is as follows:

Given:

$$T(n) \leq 2T(n/2) + n$$

prove:

$$\exists C, n_0 : \forall n \geq n_0, T(n) \leq Cn$$

We will prove it inductively. Assume:

$$\forall k < n, T(k) \leq Ck$$
$$T(n) \leq 2T(n/2) + n \leq 2C(n/2) + n$$
$$\leq Cn + n \stackrel{?}{\leq} Cn$$

The video player interface includes a play button, a volume icon, a progress bar showing 9:47 / 10:36, and icons for settings, full screen, and a share button. A small inset video of the lecturer is visible in the top right corner.

by David Scot Taylor, SJSU

Substitution Method Subtleties

Sometimes the guess is correct...

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

Guess: $O(n^2)$

Prove that $T(n) \leq cn^2$

Assume $T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^2$

By substitution:

$$\begin{aligned} T(n) &\leq 4\left(c\left(\frac{n}{2}\right)^2\right) + n \\ &= 4\left(c\frac{n^2}{4}\right) + n \\ &= cn^2 + n \\ &\not\leq cn^2 \end{aligned}$$

...but the math fails to work.

Substitution Method Subtleties (cont'd)

Subtracting lower-order terms can help

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

Guess: $O(n^2)$

Prove that $T(n) \leq cn^2 - bn$

Assume $T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^2 - b\frac{n}{2}$

By substitution:

$$\begin{aligned} T(n) &\leq 4\left(c\left(\frac{n}{2}\right)^2 - b\frac{n}{2}\right) + n \\ &= cn^2 - 2bn + n \\ &= cn^2 - n(2b - 1) \\ &\leq cn^2 - bn \end{aligned}$$

holds for $c \geq 1$ and $b = 1$

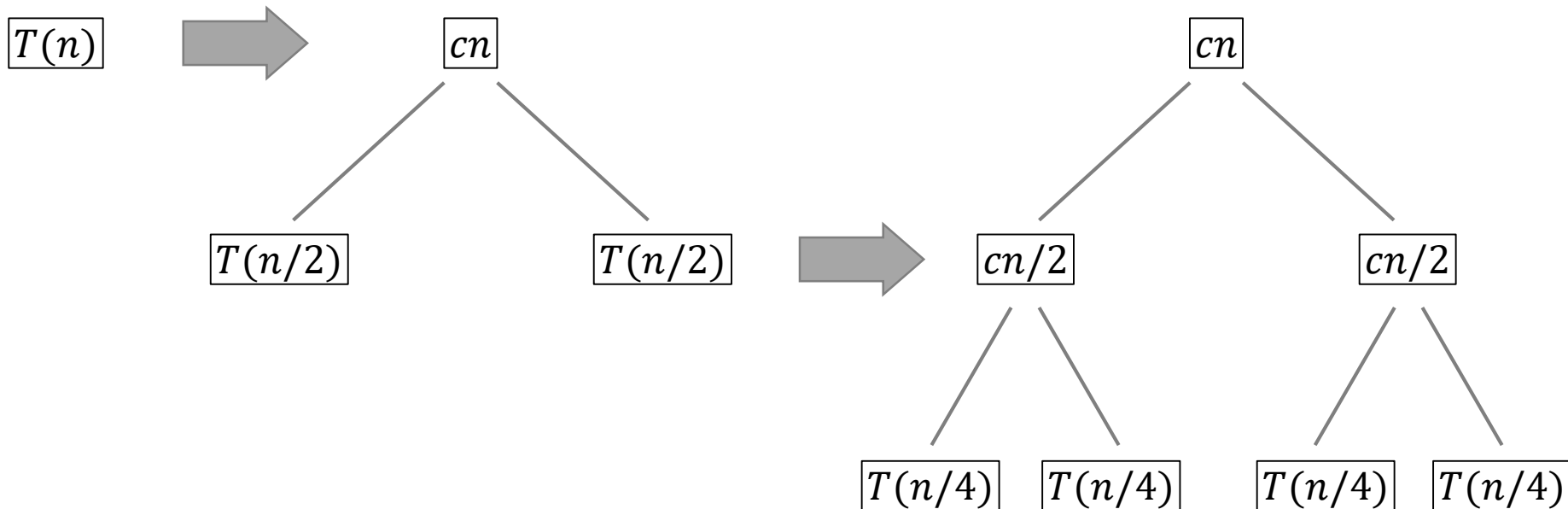


Skipping the base case here.

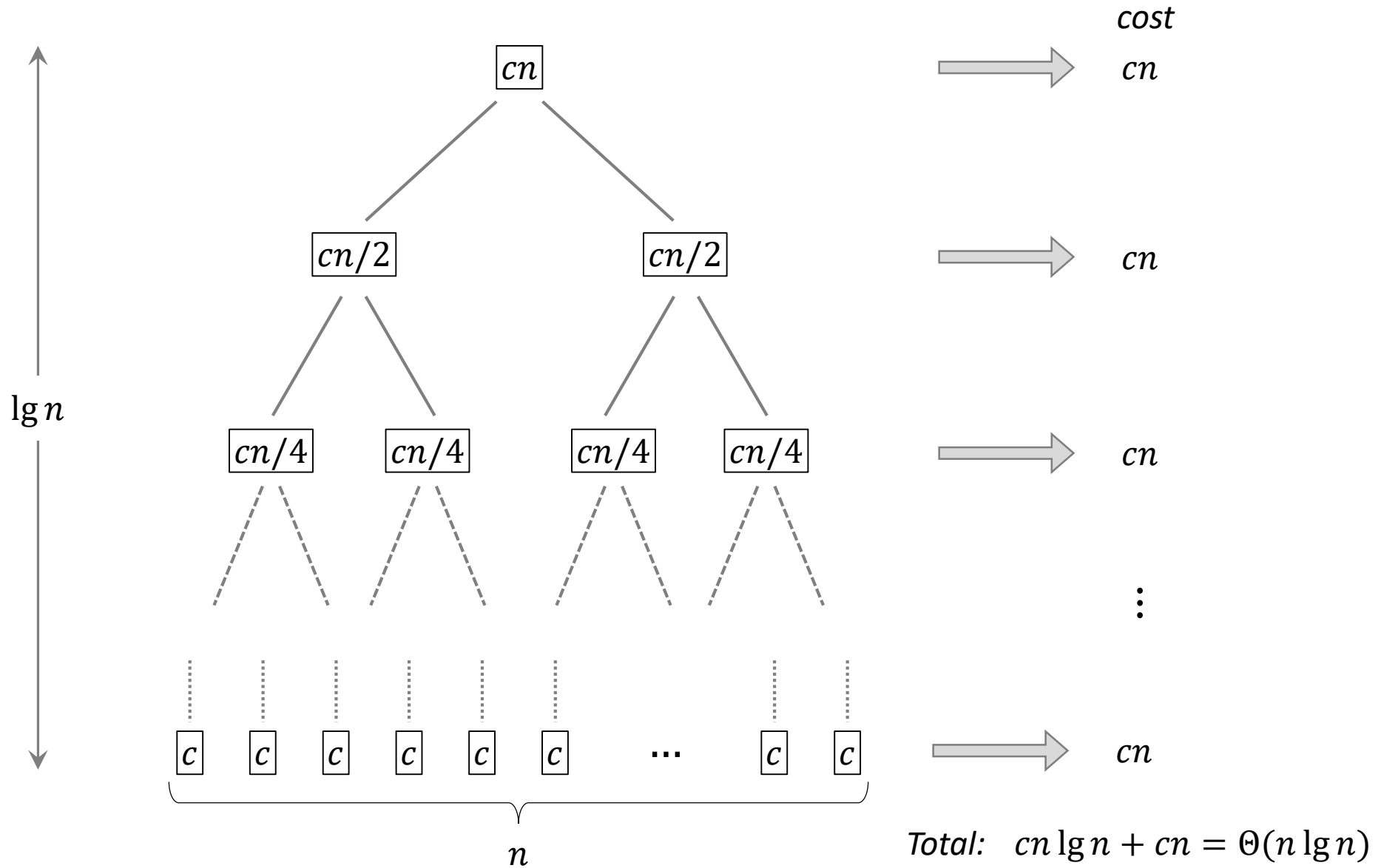
Recursion Tree: Merge Sort

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(n/2) + \Theta(n), & n > 1 \end{cases} \quad \longrightarrow \quad T(n) = \begin{cases} c, & n = 1 \\ 2T(n/2) + cn, & n > 1 \end{cases}$$

Replace constant operations by c

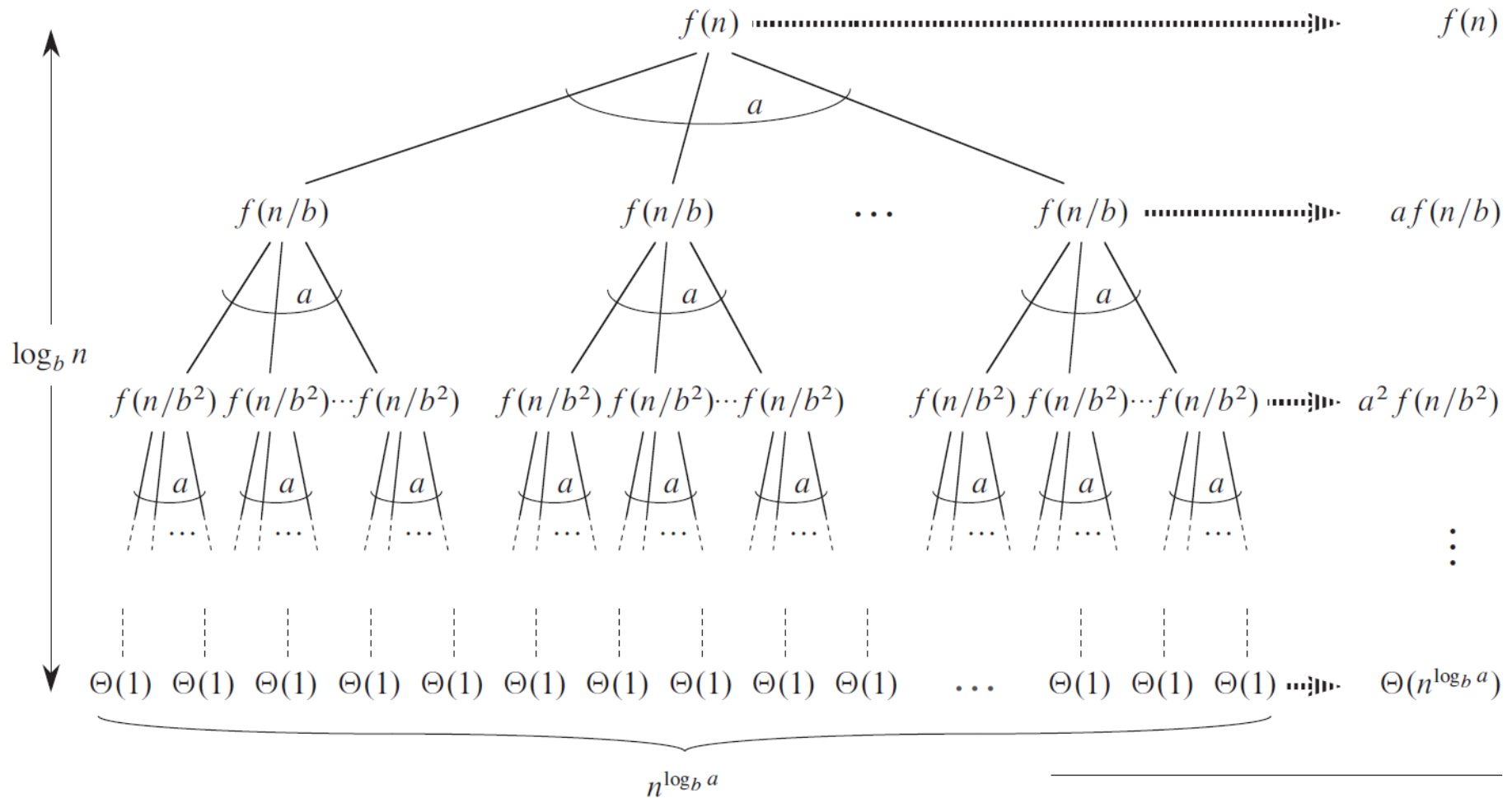


Recursion Tree: Merge Sort (cont'd)



The D&C Recursion Tree

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



[Cormen] p.99

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

The D&C Recursion Tree - Remarks

- The **height** of a tree is the number of levels – 1
- The height of a recursion tree is given by $\log_b n$
- We have $\log_b n$ levels of recursion (including the root, but without the bottom level of base cases)
- There are a^j nodes at depth j , so $a^{\log_b n} = n^{\log_b a}$ leaves
- The cost of divide and combine is given by summing the operations for each recursion level: $\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$

Example: MERGE-SORT on $n = 64$

- The recursion tree has 7 levels
- $\lg 64 = 6$ levels of recursion + 1 bottom level of base cases

Master Method

A “cookbook” for solving recurrences

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

#subproblems

time per subproblem

time for divide and combine

Master Theorem (v1)

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$.

regularity condition

Intuition: The larger of $f(n)$ and $n^{\log_b a}$ determines the solution of the recurrence

Case 3: Regularity Condition

$$af(n/b) \leq cf(n)$$

for some constant $c < 1$

- Generally not a problem
- It always holds whenever $f(n) = n^k$, so no need to check when $f(n)$ is a polynomial

Proof:

Since $f(n) = n^k$ and $f(n) = \Omega(n^{\log_b a + \epsilon})$, we have $k > \log_b a$. Using a base of b and treating both sides as exponents, we have $b^k > b^{\log_b a} = a$, and so $a/b^k < 1$. Let $c = a/b^k$.

$$af(n/b) = a(n/b)^k = (a/b^k)n^k = cf(n)$$

■

Using the Master Theorem (v1)

Examples

▪ $T(n) = 9T(n/3) + n$ $n^{\log_3 9} = n^2 = \Theta(n^2)$

Case 1: $f(n) = O(n^{2-\epsilon})$ where $\epsilon = 1$ so $T(n) = \Theta(n^2)$.

▪ $T(n) = T(n/3) + 1$ $n^{\log_{3/2} 1} = n^0 = \Theta(1)$

Case 2: $f(n) = \Theta(1)$ so $T(n) = \Theta(\lg n)$.

▪ $T(n) = 2T(n/2) + n^3$ $n^{\log_2 2} = n^1 = \Theta(n)$

Case 3: $f(n) = \Omega(n^{1+\epsilon})$ where $\epsilon = 2$ so $T(n) = \Theta(n^3)$.

“Simpler” Master Method

An even simpler “cookbook” for solving recurrences

$$T(n) = aT(n/b) + \Theta(n^d)$$

#subproblems

time per subproblem

time for divide and combine

Master Theorem (v2)

1. If $d < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $d = \log_b a$, then $T(n) = \Theta(n^d \lg n)$.
3. If $d > \log_b a$, then $T(n) = \Theta(n^d)$.

Using the Master Theorem (v2)

Examples

- $T(n) = 9T(n/3) + n$ $\log_3 9 = 2$

Case 1: $d = 1 < 2$ so $T(n) = \Theta(n^2)$.

- $T(n) = T(n/3) + 1$ $\log_{3/2} 1 = 0$

Case 2: $d = 0 = 0$ so $T(n) = \Theta(\lg n)$.

- $T(n) = 2T(n/2) + n^3$ $\log_2 2 = 1$

Case 3: $d = 3 > 1$ so $T(n) = \Theta(n^3)$.

D&C Examples

Binary Search: The “ultimate” D&C algorithm

Find a key k in a large file containing many keys in sorted order.

Example: Find 4

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 |

D&C Examples

Binary Search: The “ultimate” D&C algorithm

Find a key k in a large file containing many keys in sorted order.

Example: Find 4

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 |

D&C Examples

Binary Search: The “ultimate” D&C algorithm

Find a key k in a large file containing many keys in sorted order.

Example: Find 4

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 |

D&C Examples

Binary Search: The “ultimate” D&C algorithm

Find a key k in a large file containing many keys in sorted order.

Example: Find 4

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 |

D&C Examples

Binary Search: The “ultimate” D&C algorithm

Find a key k in a large file containing many keys in sorted order.

Example: Find 4

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 |

D&C Examples

Binary Search: The “ultimate” D&C algorithm

Find a key k in a large file containing many keys in sorted order.

Example: Find 4

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 2 | 2 | 3 | 4 | 5 | 6 |

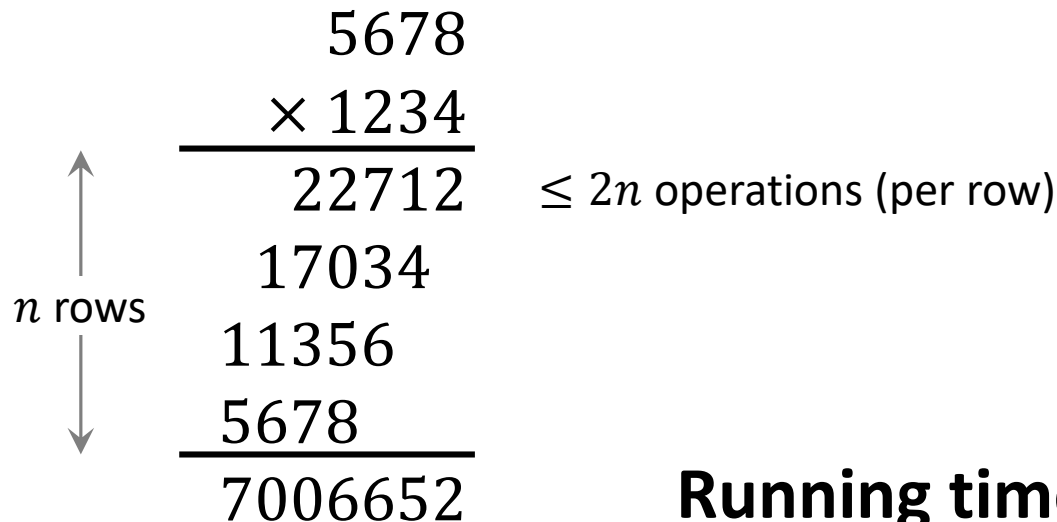
$$T(n) = T(n/2) + 1 \text{ with } T(n) = O(\lg n)$$

D&C Examples (cont'd)

Integer Multiplication

- **Input:** two n -digit numbers x and y
- **Output:** the product $x \cdot y$

Example: $x = 5678$ $y = 1234$



The diagram illustrates the standard multiplication of two 4-digit numbers, 5678 and 1234. It shows four rows of partial products, each shifted one position to the left. A vertical double-headed arrow on the left, labeled "n rows", indicates the number of rows. To the right of the partial products, the text " $\leq 2n$ operations (per row)" is shown. The final product, 7006652, is shown at the bottom.

$$\begin{array}{r} 5678 \\ \times 1234 \\ \hline 22712 \\ 17034 \\ 11356 \\ 5678 \\ \hline 7006652 \end{array} \leq 2n \text{ operations (per row)}$$

Running time: $\Theta(n^2)$

D&C Examples (cont'd)

Integer Multiplication

- **Input:** two n -digit numbers x and y
- **Output:** the product $x \cdot y$

Example: $x = \underbrace{56}_{a}\underbrace{78}_{b}$ $y = \underbrace{12}_{c}\underbrace{34}_{d}$

$$x = 10^2 \cdot a + b$$

$$y = 10^2 \cdot c + d$$

$$\begin{aligned} x \cdot y &= (10^2 \cdot a + b) \cdot (10^2 \cdot c + d) \\ &= 10^4 \underbrace{ac} + 10^2 \underbrace{(ad + bc)} + \underbrace{bd} \end{aligned}$$

Idea: Recursively compute ac , ad , bc , bd

D&C Examples (cont'd)

Integer Multiplication

- **Input:** two n -digit numbers x and y
- **Output:** the product $x \cdot y$

Example: $x = \boxed{x_1 x_2 \dots x_{n-1} x_n}_a \boxed{y_1 y_2 \dots y_{n-1} y_n}_d$ $y = \boxed{y_1 y_2 \dots y_{n-1} y_n}_c$ Split numbers at digit $\lfloor \frac{n}{2} \rfloor$

$$x = 10^{\lfloor \frac{n}{2} \rfloor} a + b \qquad y = 10^{\lfloor \frac{n}{2} \rfloor} c + d$$

$$x \cdot y = (10^{\lfloor \frac{n}{2} \rfloor} a + b) \times (10^{\lfloor \frac{n}{2} \rfloor} c + d)$$

$$x \cdot y = 10^{2\lfloor \frac{n}{2} \rfloor} ac + 10^{\lfloor \frac{n}{2} \rfloor} (ad + bc) + bd$$

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

D&C Examples (cont'd)

Karatsuba Multiplication

- **Input:** two n -digit numbers x and y
- **Output:** the product $x \cdot y$

$$x \cdot y = 10^{2\lfloor \frac{n}{2} \rfloor} ac + 10^{\lfloor \frac{n}{2} \rfloor} (ad + bc) + bd$$

$$\text{Rewrite: } (ad + bc) = (a + b)(c + d) - ac - bd$$

Only **three** subproblems!

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

D&C Examples (cont'd)

Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

A Fibonacci number $F(n)$ is the sum of its two predecessors.

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

NAÏVE-FIBONACCI(n)

1: **if** $n == 0$

2: **return** 0

3: **if** $n == 1$

4: **return** 1

5: **return** NAÏVE-FIBONACCI($n-1$) + NAÏVE-FIBONACCI($n-2$)

$$T(n) = O(2^{0.694n})$$

Bad, bad...really bad!

Conclusions

Divide & Conquer...

...is a powerful algorithm design technique

...algorithms can be analysed using recurrences

...often, but not always, leads to efficient algorithms

References

Books

- **[Cormen] Introduction to Algorithms**
T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. MIT Press. 2009 (3rd Edition)
- **[Sedgewick] Algorithms**
R. Sedgewick, K. Wayne. Addison-Wesley. 2011 (4th Edition)
- **[Dasgupta] Algorithms**
S. Dasgupta, C. Papadimitriou, U. Vazirani. McGraw-Hill Higher Education. 2006

Online

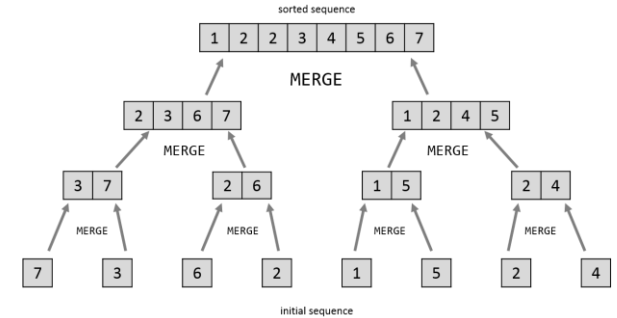
- <http://algs4.cs.princeton.edu/lectures/>
- <https://www.coursera.org/courses?query=algorithms>

CO202 – Software Engineering – Algorithms

Divide and Conquer - Exercises

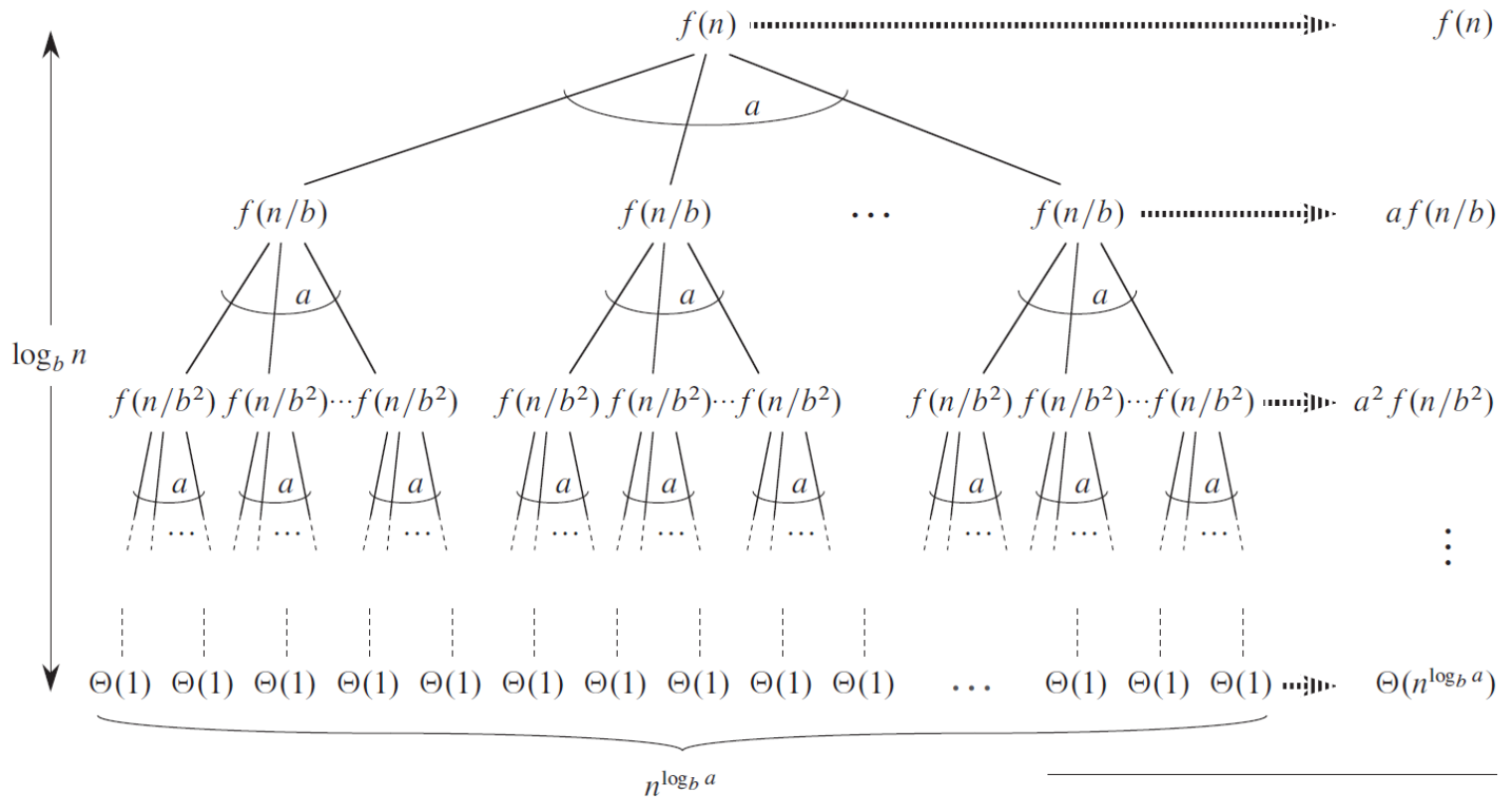
Exercise 1: Illustrate the Operations of Merge Sort

$A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$



Exercise 2: Draw a Recursion Tree, Guess, and Verify

$$T(n) = 3T(n/4) + cn^2$$



b.socratic.com - room ALGO202

A: $O(\lg n)$ **B:** $O(n)$ **C:** $O(n \lg n)$ **D:** $O(n^2)$ **E:** $O(2^n)$

Exercise 3: Master Method

- $T(n) = 3T(n/4) + cn^2$

- $T(n) = 2T(n/4) + \sqrt{n}$

- $T(n) = 8T(n/2) + n^2$

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

1. If $d < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $d = \log_b a$, then $T(n) = \Theta(n^d \lg n)$.
3. If $d > \log_b a$, then $T(n) = \Theta(n^d)$.

Exercise 4: Substitution Method

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

- 1) Obtain the running time using the Master Method
- 2) Confirm with the Substitution Method

Exercise 5: Divide and Conquer

1) Write in pseudo-code a recursive function $f(x, n) = x^n$ for powering a number using divide-and-conquer.

Hint:

- $x^n = x^{n/2} \cdot x^{n/2}$ for even n
- $x^n = x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x$ for odd n

2) Show that the running time complexity is $O(\lg n)$.