Imperial College London

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 $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

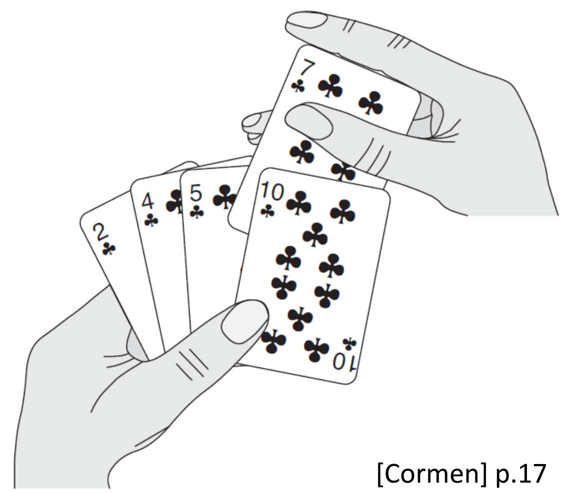
1	2	3	4	5	6	7	8
1	2	2	3	4	5	6	7

Reminder: Insertion Sort

Incremental Approach

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

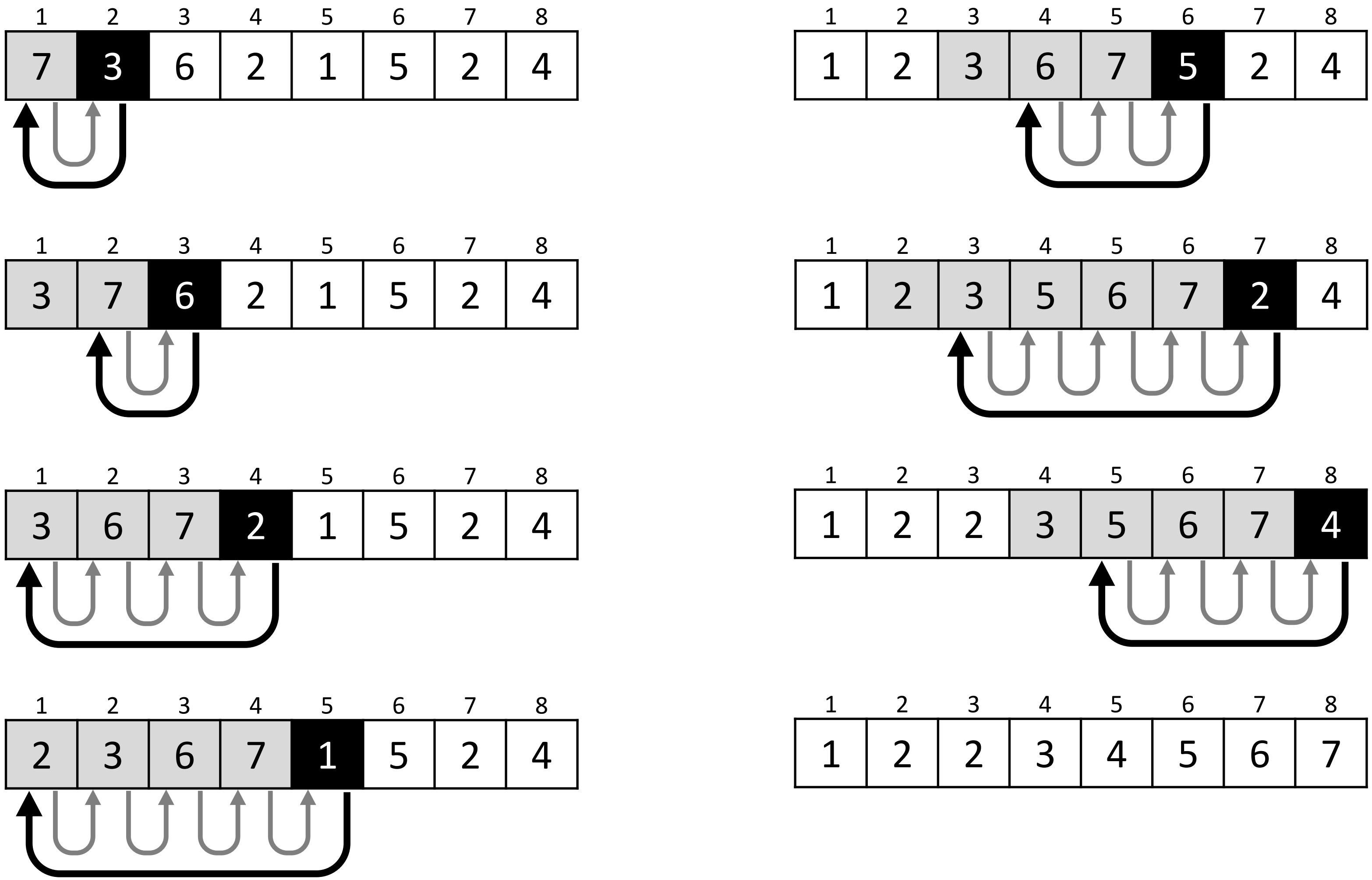
```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    # Insert A[j] into the sorted sequence A[1..j-1].
    i = j-1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
```



[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
for j = 2 to A.length
    key = A[j]
    # Insert A[j]...
    i = j-1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
```

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1)$$

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

Running Time of Insertion Sort

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

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

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

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

Reminder

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

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

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

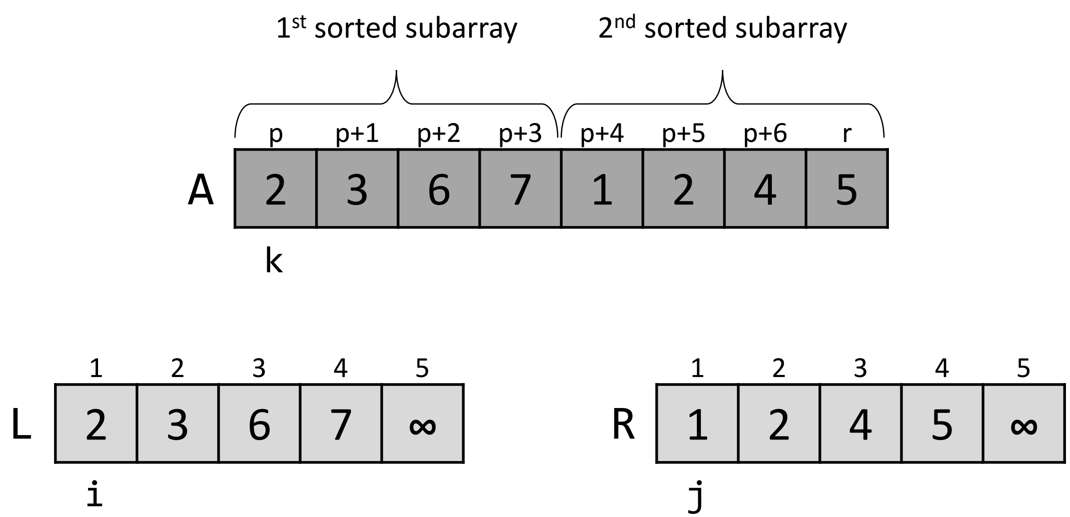
2. **Conquer:** Sort the two subsequences recursively using merge sort

3. **Combine:** Merge the two sorted subsequences to produce the sorted answer

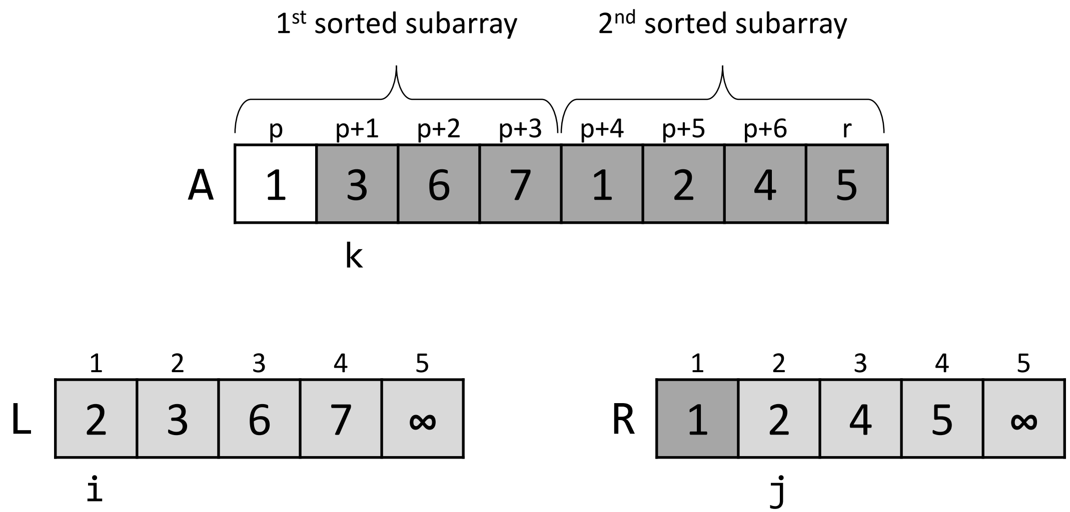
Combine: Merge

```
MERGE(A,p,q,r)
n1 = q-p+1                                        # length of 1st subarray
n2 = r-q                                          # length of 2nd subarray
let L[1..n1+1] and R[1..n2+1] be new arrays
for i = 1 to n1
    L[i] = A[p+i-1]                               # copy values to 1st array
for j = 1 to n2
    R[j] = A[q+j]                                 # copy values to 2nd array
L[n1+1] = ∞                                       # set sentinel
R[n2+1] = ∞                                       # set sentinel
i = 1
j = 1
for k = p to r                                    # merge subarrays
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i+1
    else
        A[k] = R[j]
        j = j+1
```

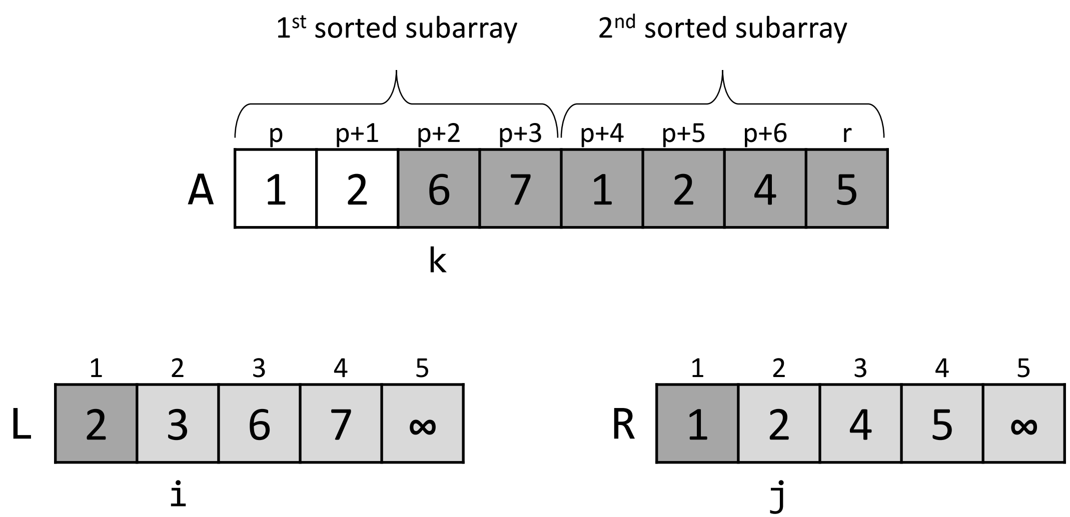
Combine: Merge (cont'd)



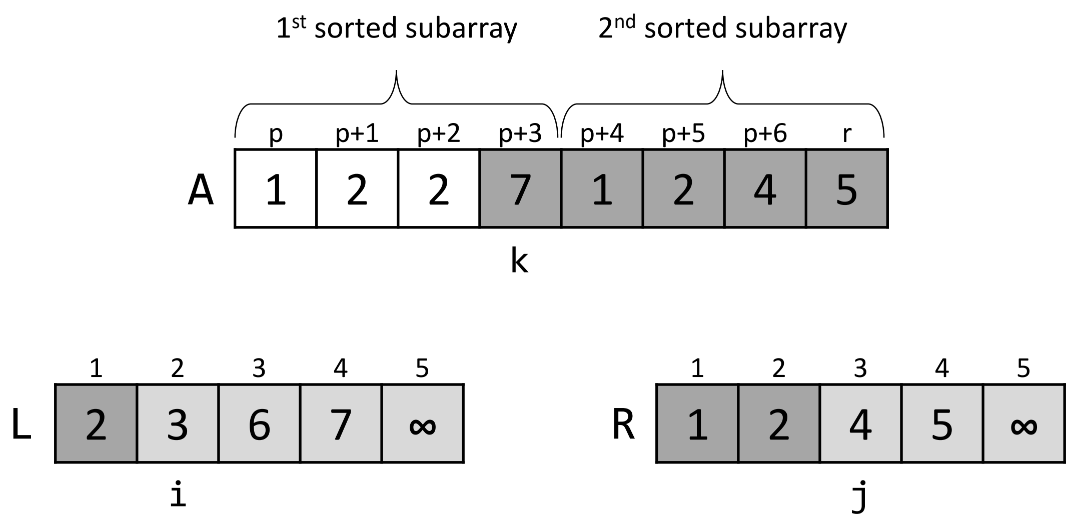
Combine: Merge (cont'd)



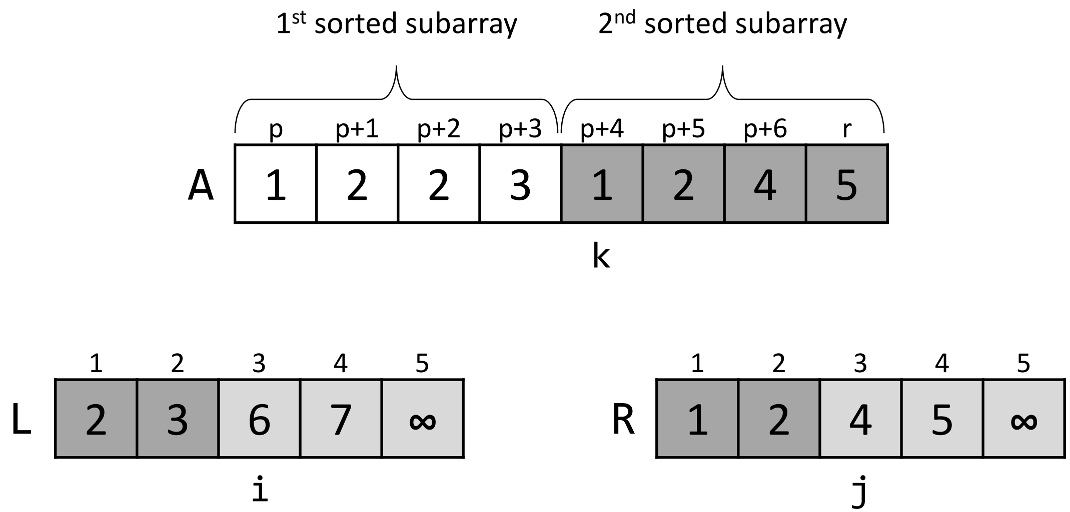
Combine: Merge (cont'd)



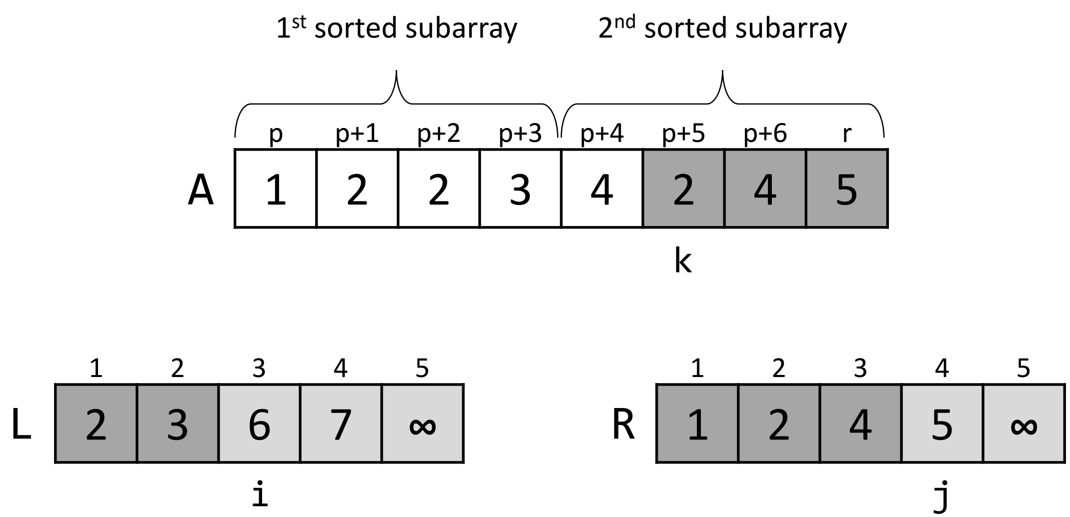
Combine: Merge (cont'd)



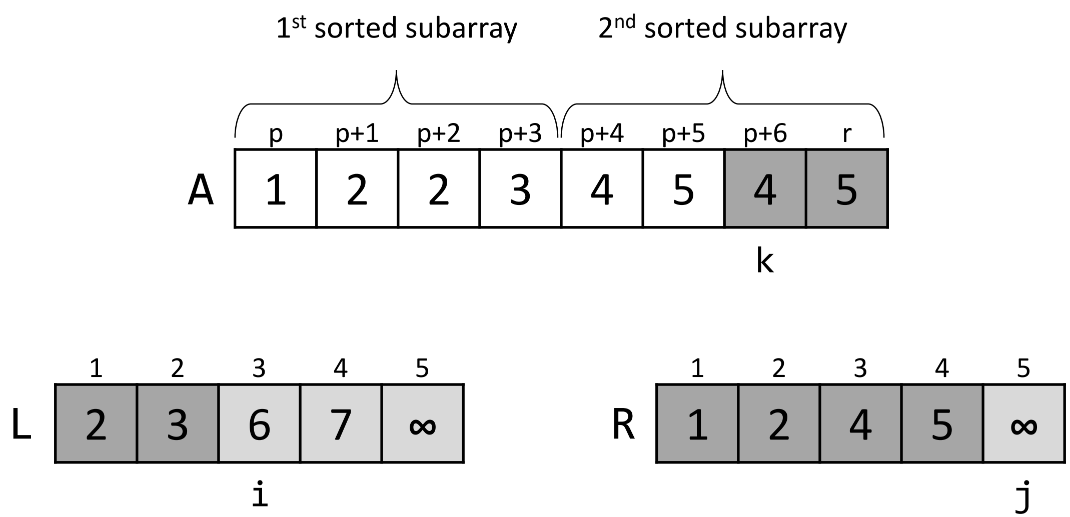
Combine: Merge (cont'd)



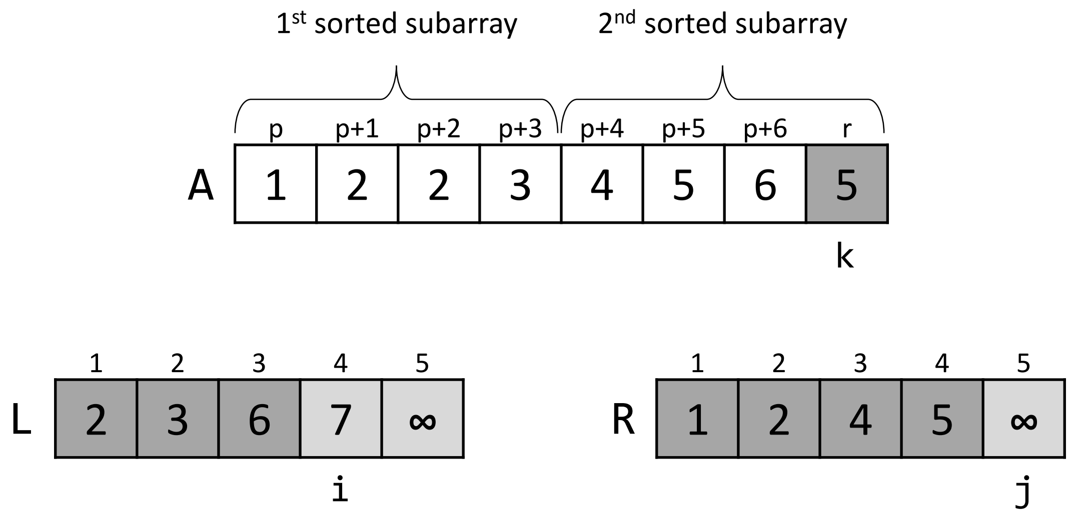
Combine: Merge (cont’d)



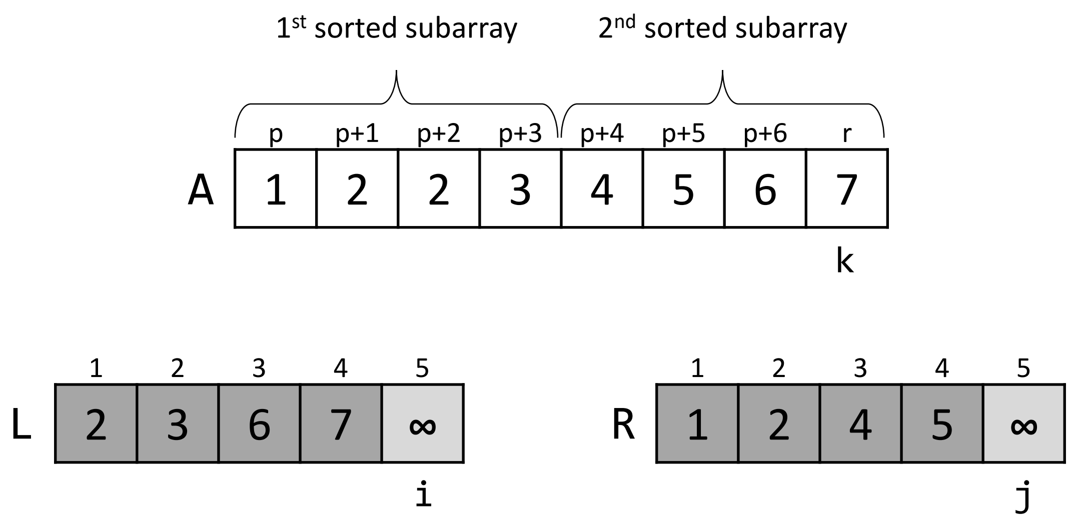
Combine: Merge (cont'd)



Combine: Merge (cont’d)



Combine: Merge (cont'd)

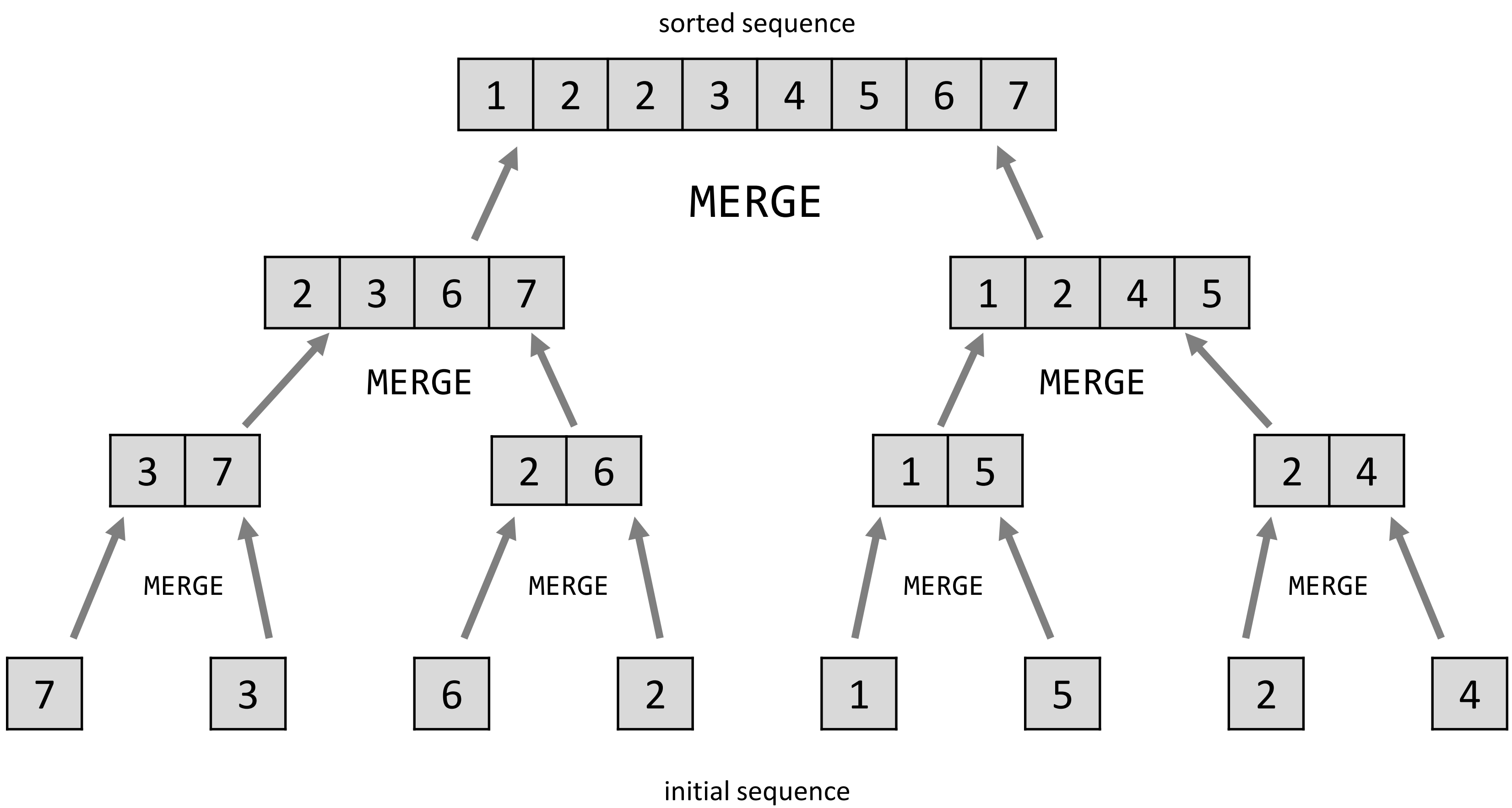


Merge Sort (cont’d)

Divide and Conquer Approach

```
MERGE-SORT(A,p,r)
if p < r                          # more than 1 item?
     q = floor((p+r)/2)           # divide array
     MERGE-SORT(A,p,q)            # conquer 1st subarray
     MERGE-SORT(A,q+1,r)          # conquer 2nd subarray
     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
if p < r	c_1	?
q = floor((p+r)/2)	c_2	?
MERGE-SORT(A,p,q)	c_3	?
MERGE-SORT(A,q+1,r)	c_4	?
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 $1/b$
- **Divide** takes time $D(n)$ and **combine** takes $C(n)$

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

Merge Sort Recurrence

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

- **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 $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T\left(n/_2\right) + \underbrace{\Theta(1) + \Theta(n)}_{\Theta(n)}, & n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T\left(n/_2\right) + \Theta(n), & n > 1 \end{cases}$$

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:

$$
\begin{aligned}
T(n) &\leq 2\left(c(^n/_2 \lg {}^n/_2)\right) + n \\
&= cn \lg {}^n/_2 + n \\
&= cn \lg n - cn \lg 2 + n \\
&= cn \lg n - cn + n \\
&\leq cn \lg n \qquad \text{holds for } c \geq 1
\end{aligned}
$$

What about the base case? $T(1) = 1 \nleq c1 \lg 1 = 0$

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

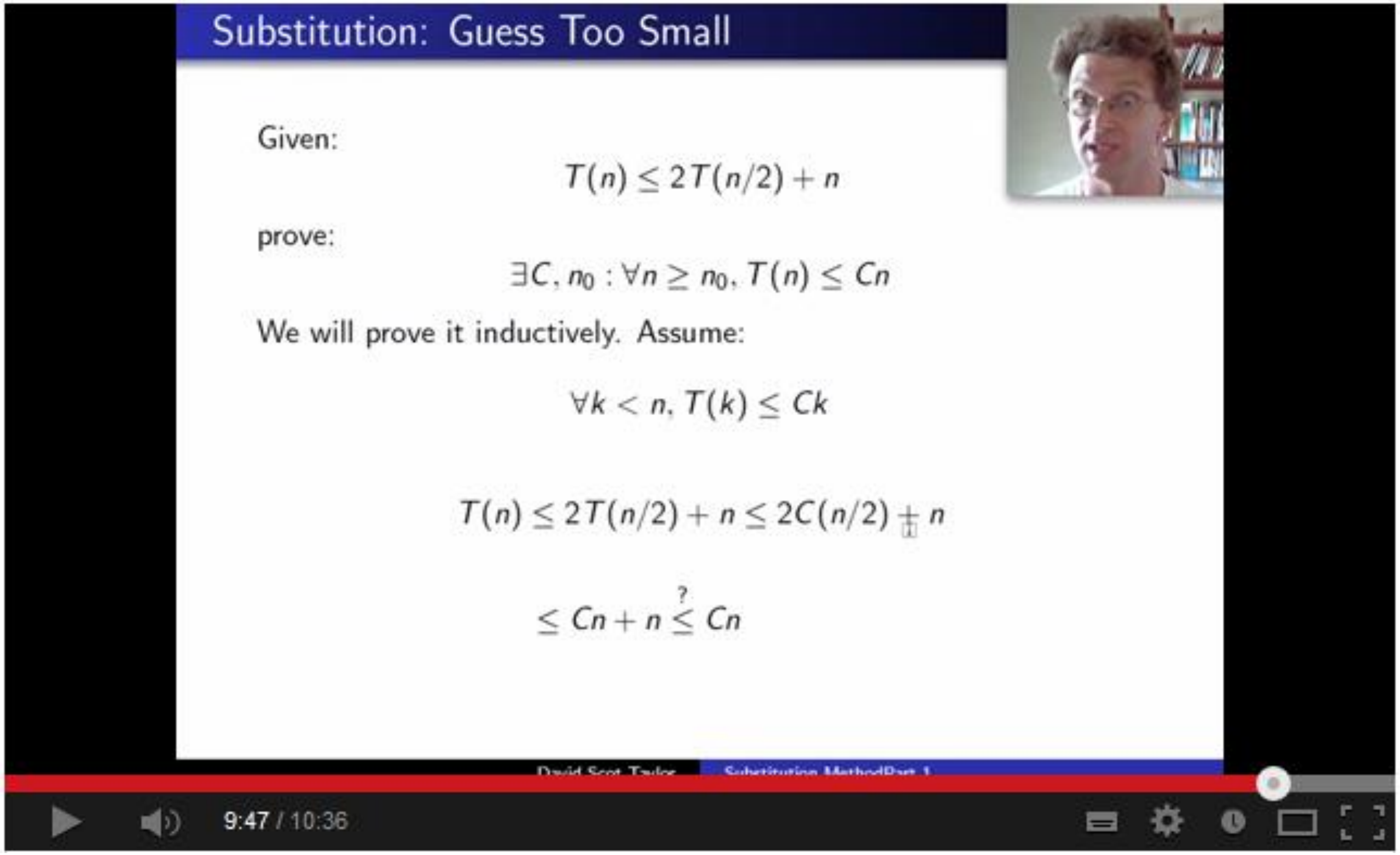
■

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

Substitution Method (cont'd)

What happens if guess is too small?

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



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:

$$
\begin{aligned}
T(n) &\leq 4\left(c\left(\frac{n}{2}\right)^2\right) + n \\
&= 4\left(c\,\frac{n^2}{4}\right) + n \\
&= cn^2 + n \\
&\nleq cn^2
\end{aligned}
$$

…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 $T(n) \le cn^2 - bn$

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

By substitution:

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

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

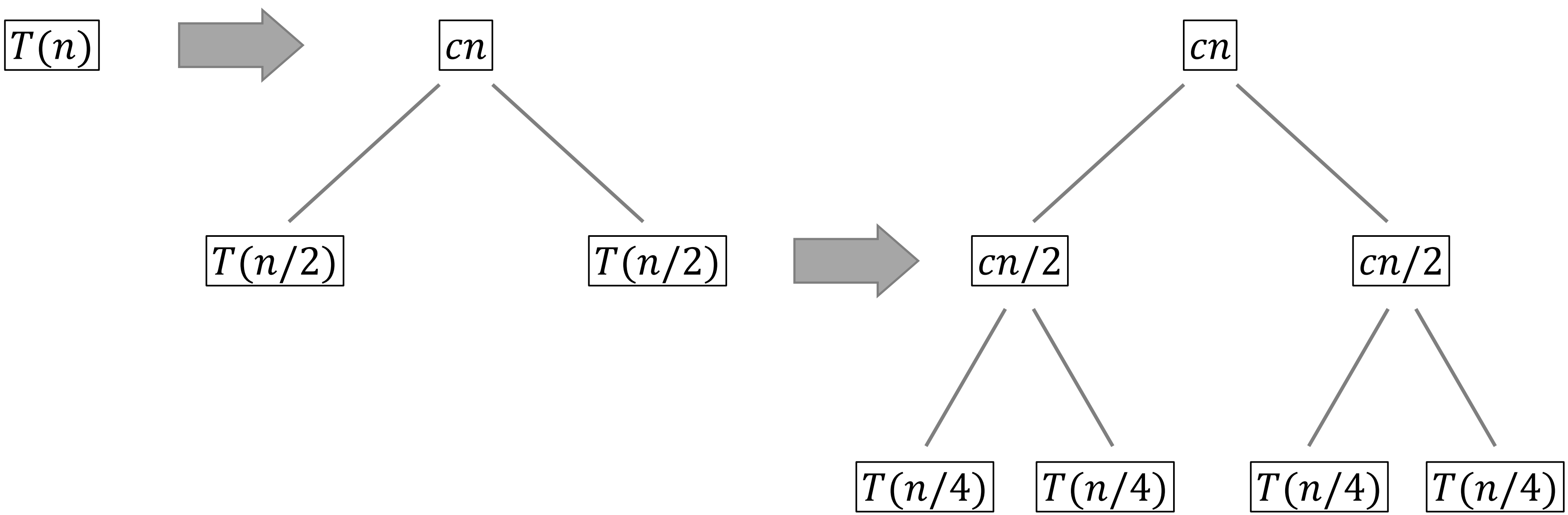
■

Skipping the base case here.

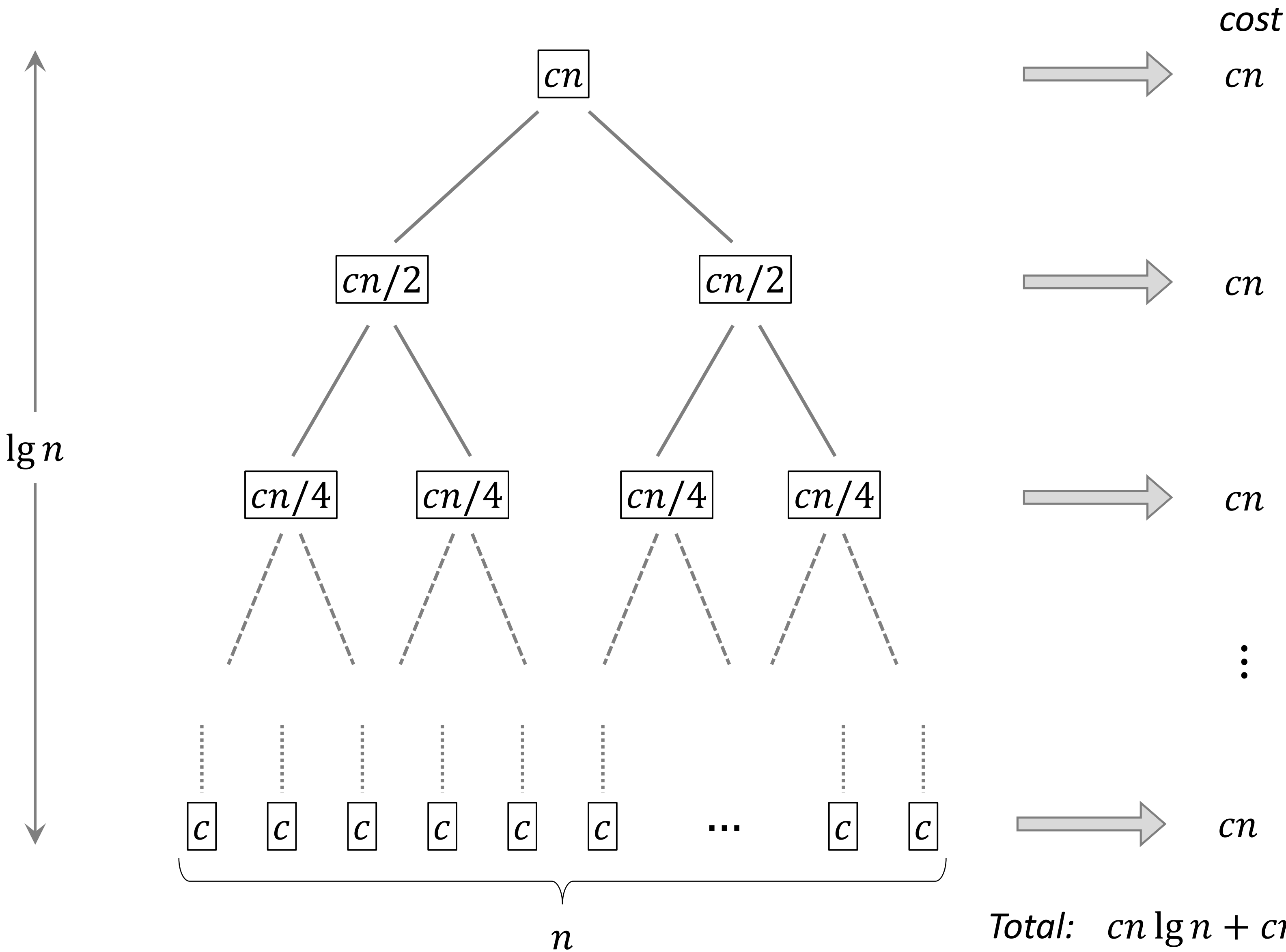
Recursion Tree: Merge Sort

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T\left(n/_2\right) + \Theta(n), & n > 1 \end{cases} \quad \Longrightarrow \quad T(n) = \begin{cases} c, & n = 1 \\ 2T\left(n/_2\right) + cn, & n > 1 \end{cases}$$

Replace constant operations by c



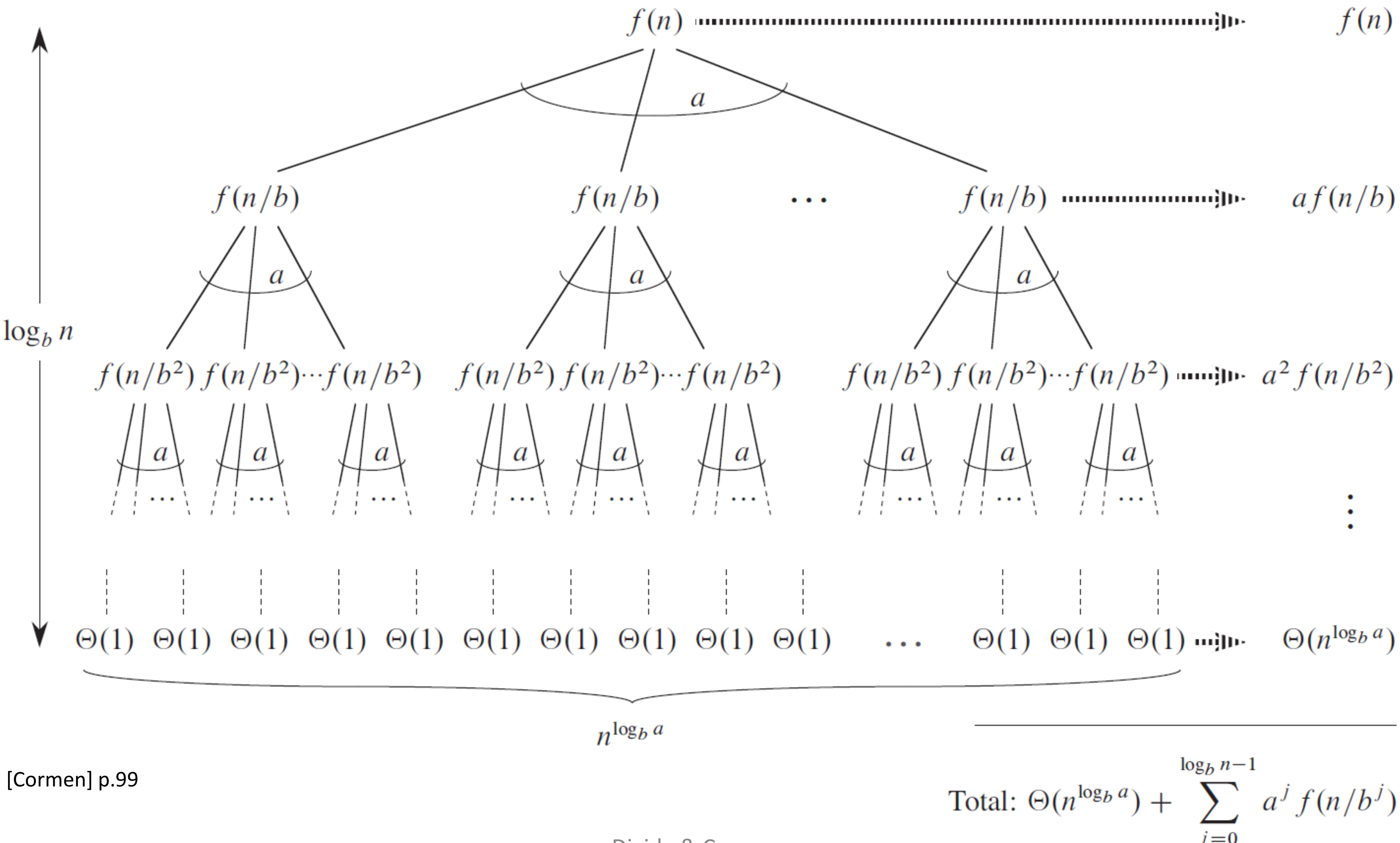
Recursion Tree: Merge Sort (cont'd)



Total: $cn \lg n + cn = \Theta(n \lg n)$

The D&C Recursion Tree

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



[Cormen] p.99

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)
- 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

$$T(n) = \underbrace{a}_{\text{\#subproblems}}\underbrace{T(n/b)}_{\text{time per subproblem}} + \underbrace{f(n)}_{\text{time for divide and combine}}$$

Master Theorem (v1)

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

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.

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

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 $b^k > b^{\log_\mathrm{b} a} = a$, and so $a/b^k < 1$. Let $c = a/b^k$.

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

■

Using the Master Theorem (v1)

Examples

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

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

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

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

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

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

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

#subproblems

time per subproblem

time for divide and combine

Master Theorem (v2)

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

Using the Master Theorem (v2)

Examples

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

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

- $T(n) = T(^{2n}/_3) + 1$ $\qquad \log_{3/2} 1 = 0$

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

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

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

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: $x = 5678 \quad y = 1234$

$$
\begin{array}{r}
5678 \\
\times\, 1234 \\
\hline
22712 \\
17034 \\
11356 \\
5678 \\
\hline
7006652
\end{array}
$$

n rows

$\leq 2n$ 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: $x = \underbrace{56}_{a}\underbrace{78}_{b}$ $\quad y = \underbrace{12}_{c}\underbrace{34}_{d}$

$$x = 10^2 \cdot a + b$$
$$y = 10^2 \cdot c + d$$

$$x \cdot y = (10^2 \cdot a + b) \cdot (10^2 \cdot c + d)$$
$$= 10^4 ac + 10^2 (ad + bc) + bd$$

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: $x = \underbrace{x_1 x_2 \ldots}_{a} \underbrace{x_{n-1} x_n}_{b}$ $\quad y = \underbrace{y_1 y_2 \ldots}_{c} \underbrace{y_{n-1} y_n}_{d}$

Split numbers at digit $\left\lfloor \frac{n}{2} \right\rfloor$

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

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

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

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

D&C Examples (cont'd)

Karatsuba Multiplication

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

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

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

Only **three** subproblems!

$$T(n) = \mathbf{3}T(n/2) + n$$

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)
if n == 0
    return 0
if n == 1
    return 1
return FIBONACCI(n-1) + 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