算法设计与分析

Lecture 5: Divide-and-Conquer

卢杨
厦门大学信息学院计算机科学系
luyang@xmu.edu.cn

Divide-and-Conquer

- The divide-and-conquer (分治) algorithm divides an instance of a problem into two or more small instances.
  - The small instance belongs to the same problem as the original instance.
  - Assume that the small instance is easy to solve.
  - Combine solutions to the small instances to solve the original instance.
  - If the small instance is still difficult, divide again until it is easy.
- The divide-and-conquer is a top-down approach.
  - Recursion is usually adopted.

Divide-and-Conquer

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- Divide the problem instance into a number of small instances.
- Conquer the small instances by solving them recursively. If the sizes of small instances are small enough, just solve them without recursion.
- Combine (optional) the solutions to the small instances into the solution for the original instance.

Analyzing Divide-and-Conquer Algorithms

- When an algorithm contains a recursive call to itself, its running time can often be described by a recursion equation (递归方程).
- We can easily solve them by the methods we have learned in Lecture 4.

Analyzing Divide-and-Conquer Algorithms

- If the instance size is small enough, say $n \leq c$ for some constant c, we can simply assume that the straightforward solution takes constant time $\Theta(1)$.
- The running time of a divide-and-conquer algorithm is based on the three steps of the basic paradigm:

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

  - $D(n)$: the cost of dividing into small instances.
  - $aT(n/b)$: conquer a small instances with each size n/b.
  - $C(n)$: the cost of combining the solutions of small instances.
  - $D(n)$ and $C(n)$ are usually merged into a function $f(n)$ for analysis convenience.

MERGESORT

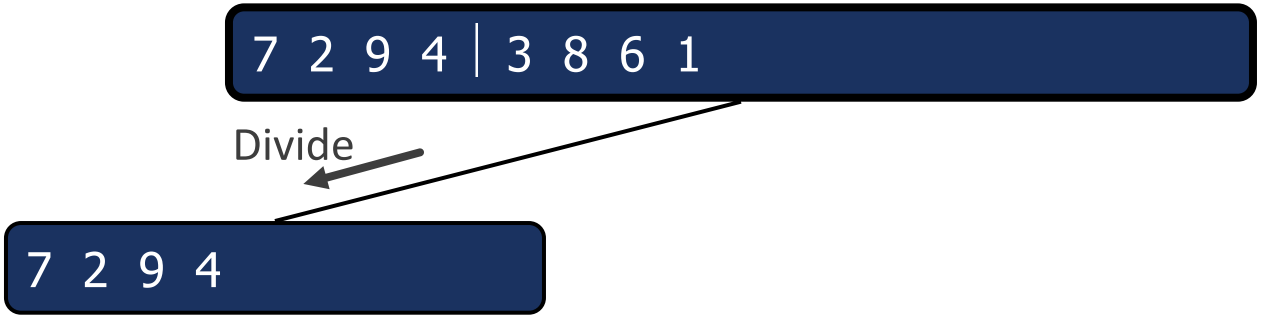
Mergesort

- Mergesort (合并排序) combines two sorted arrays into one sorted array.
- Given an array with n elements, Mergesort involves the following steps:
  1. Divide the array into two subarrays each with $n/2$ elements.
  2. Conquer each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
  3. Combine the solutions to the subarrays by merging them into a single sorted array.

Mergesort Example

7 2 9 4 3 8 6 1

Mergesort Example

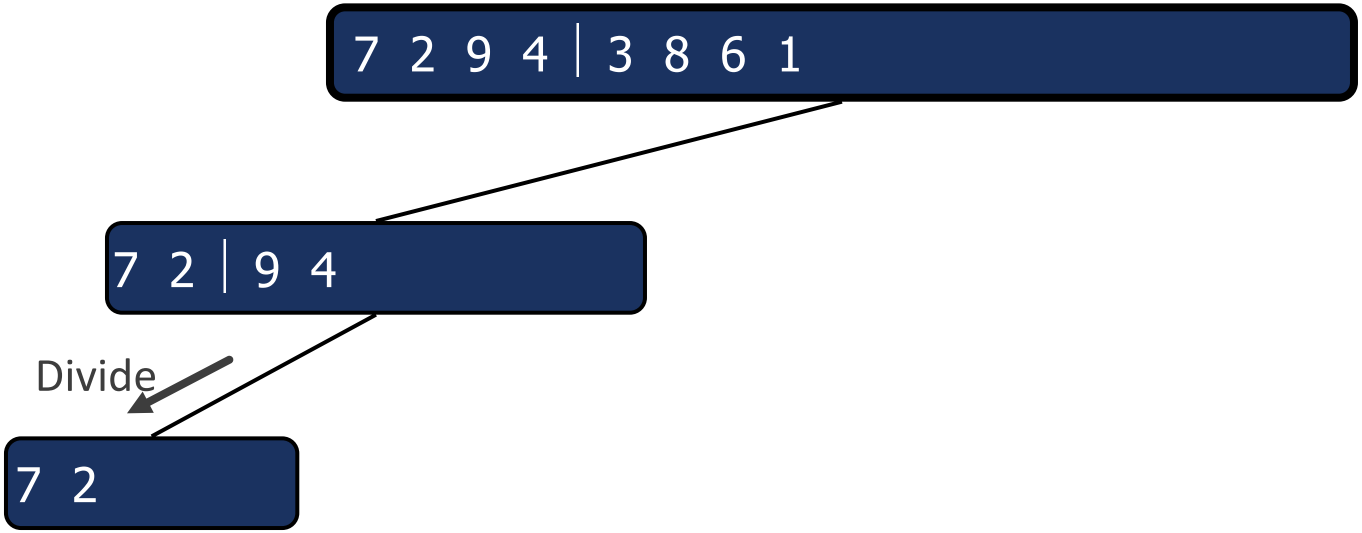


7 2 9 4 | 3 8 6 1

Divide

7 2 9 4

Mergesort Example



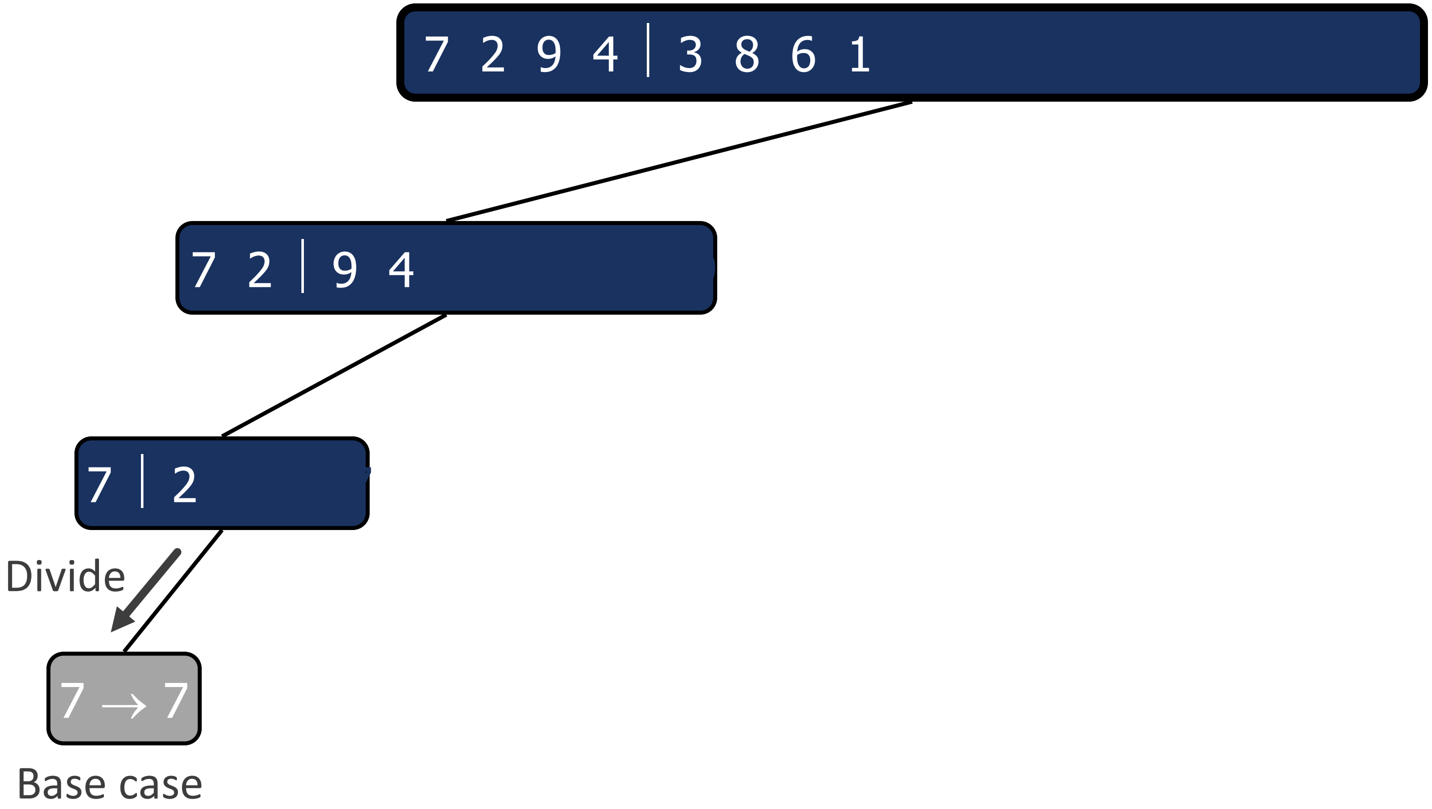
7 2 9 4 | 3 8 6 1

7 2 | 9 4

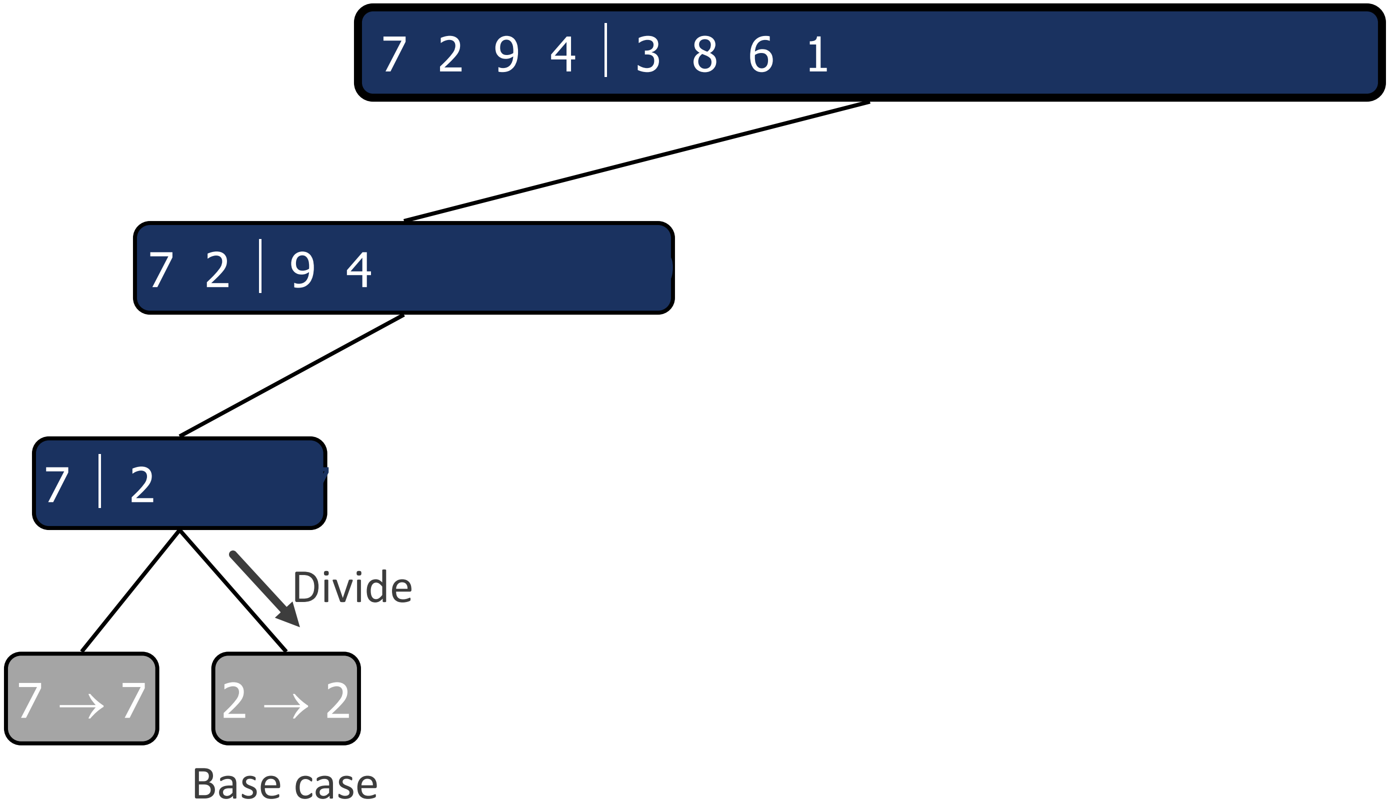
Divide

7 2

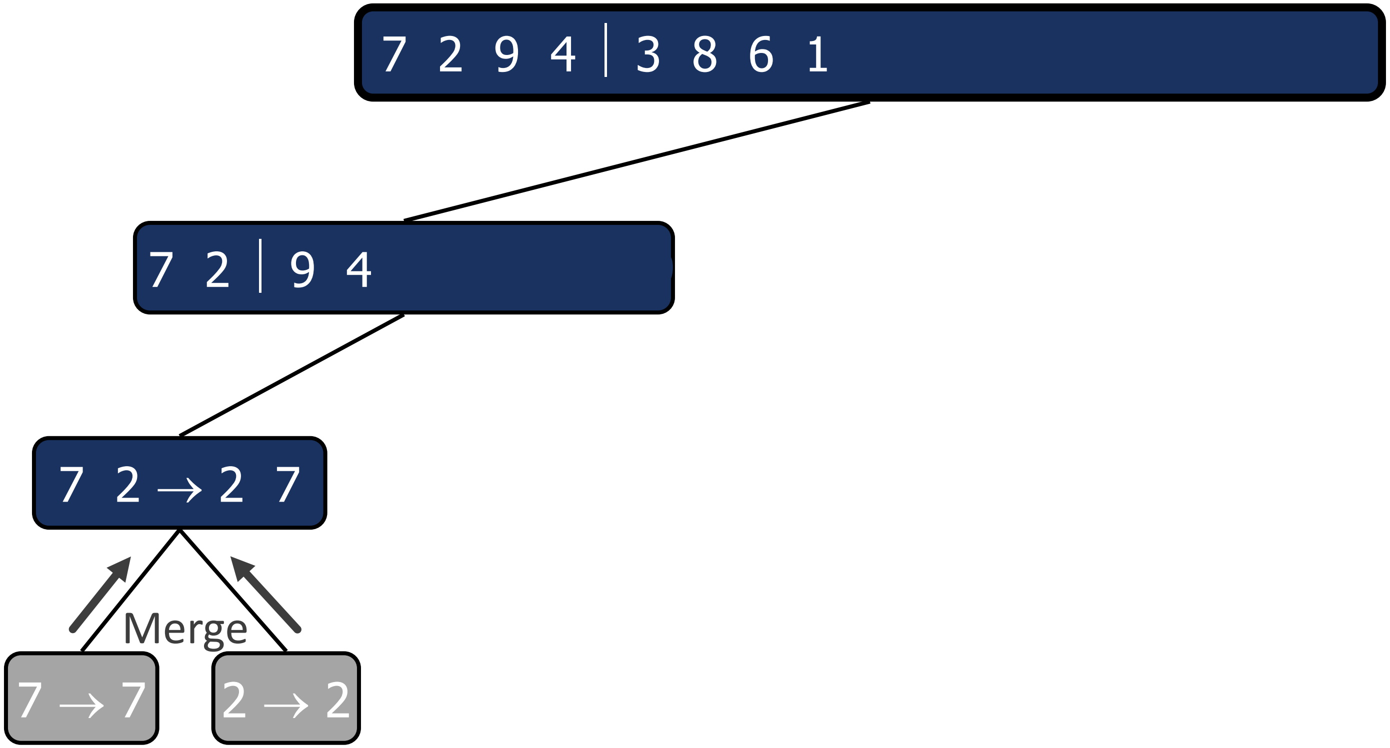
Mergesort Example



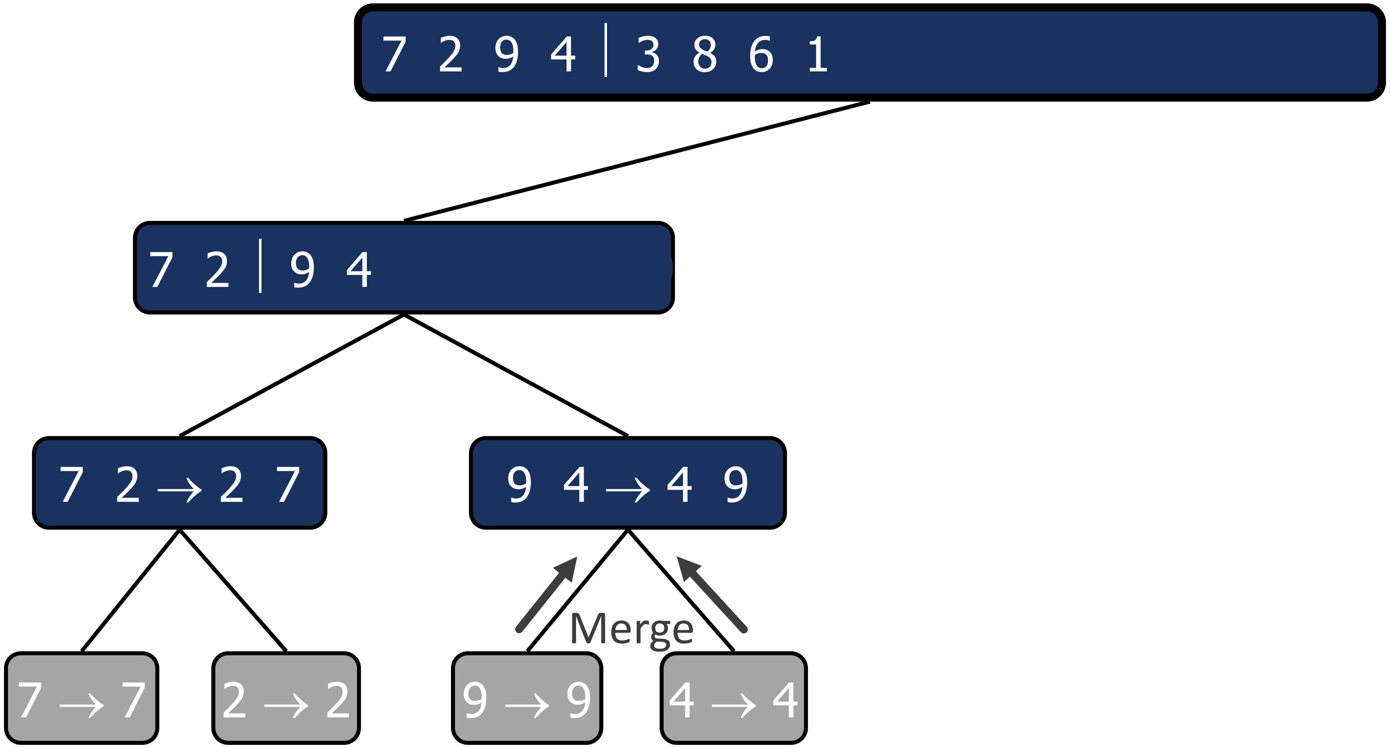
Mergesort Example



Mergesort Example



Mergesort Example



7 2 9 4 | 3 8 6 1

7 2 | 9 4

7 2 → 2 7

9 4 → 4 9

Merge

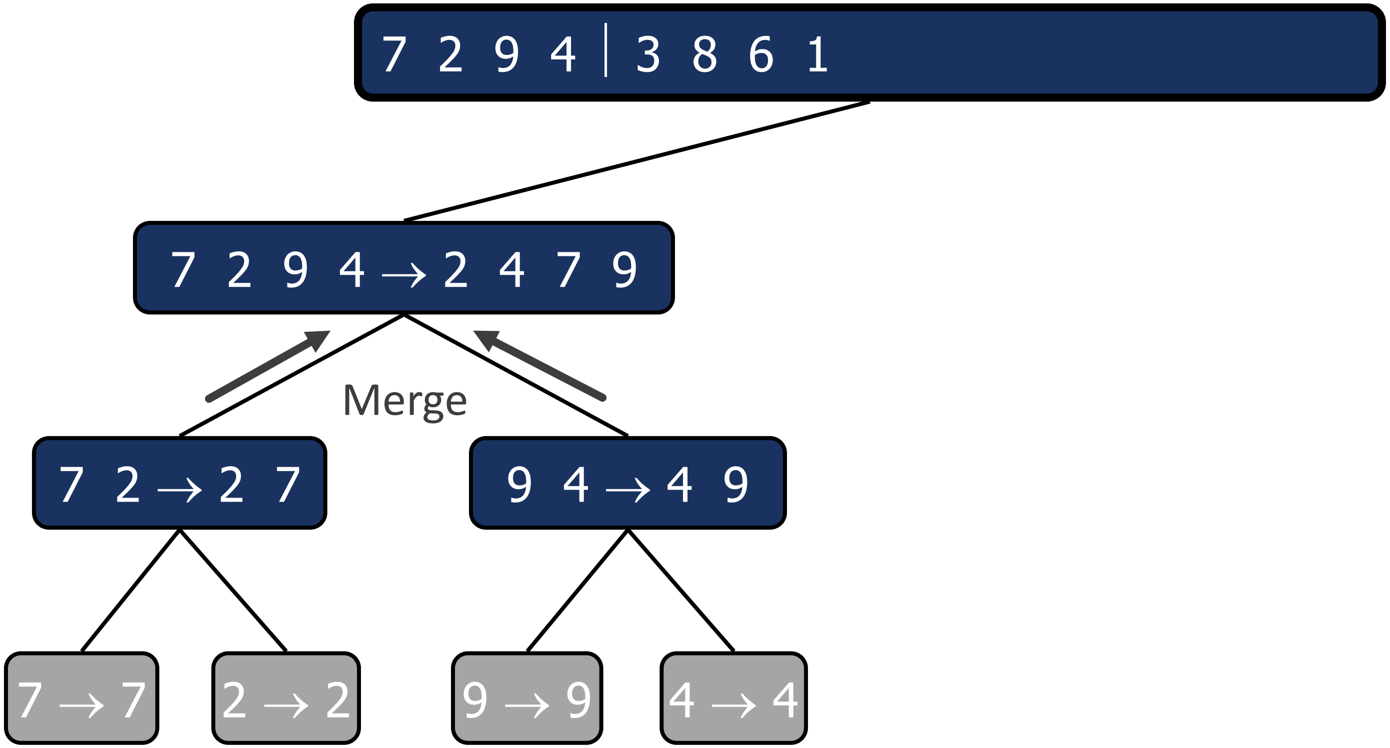
7 → 7

2 → 2

9 → 9

4 → 4

Mergesort Example



7 2 9 4 | 3 8 6 1

7 2 9 4 → 2 4 7 9

Merge

7 2 → 2 7

9 4 → 4 9

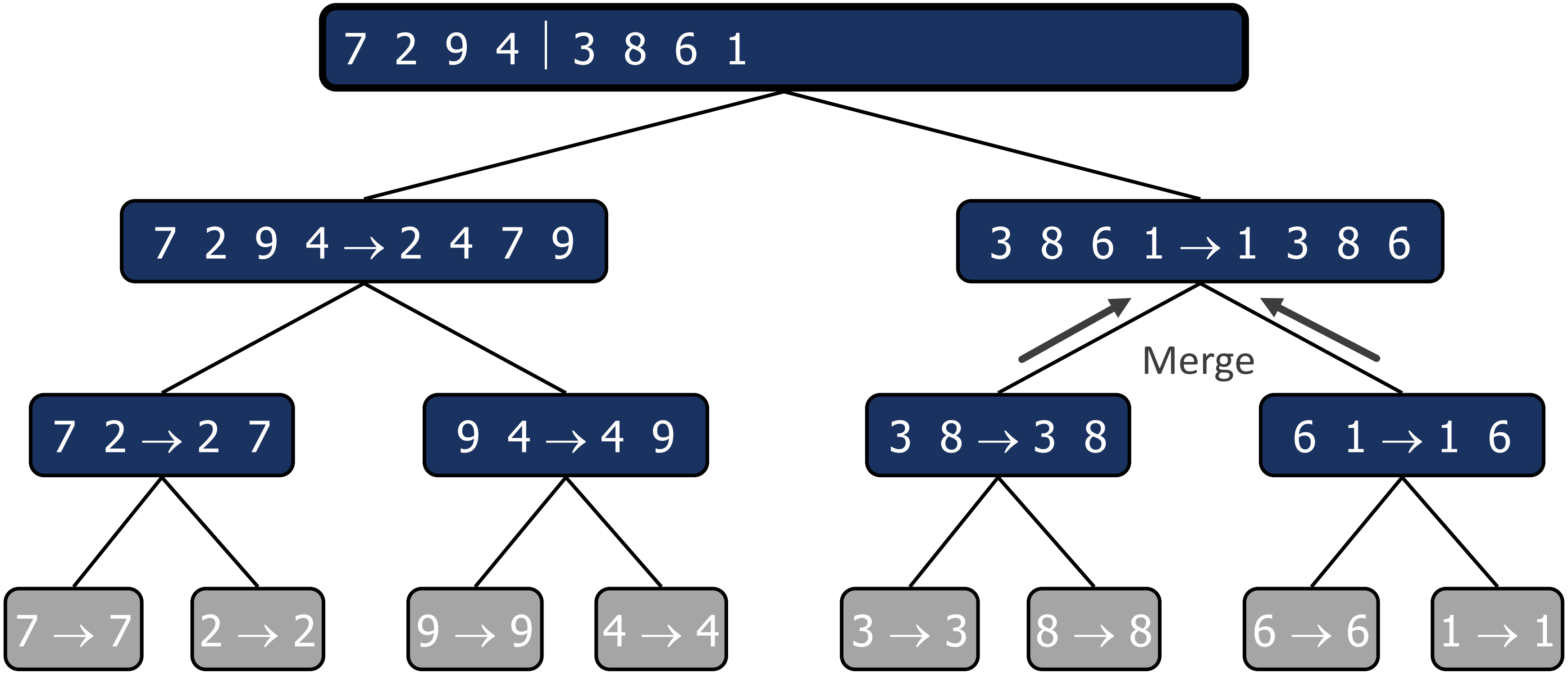
7 → 7

2 → 2

9 → 9

4 → 4

Mergesort Example



7 2 9 4 | 3 8 6 1

7 2 9 4 → 2 4 7 9

3 8 6 1 → 1 3 8 6

Merge

7 2 → 2 7

9 4 → 4 9

3 8 → 3 8

6 1 → 1 6

7 → 7

2 → 2

9 → 9

4 → 4

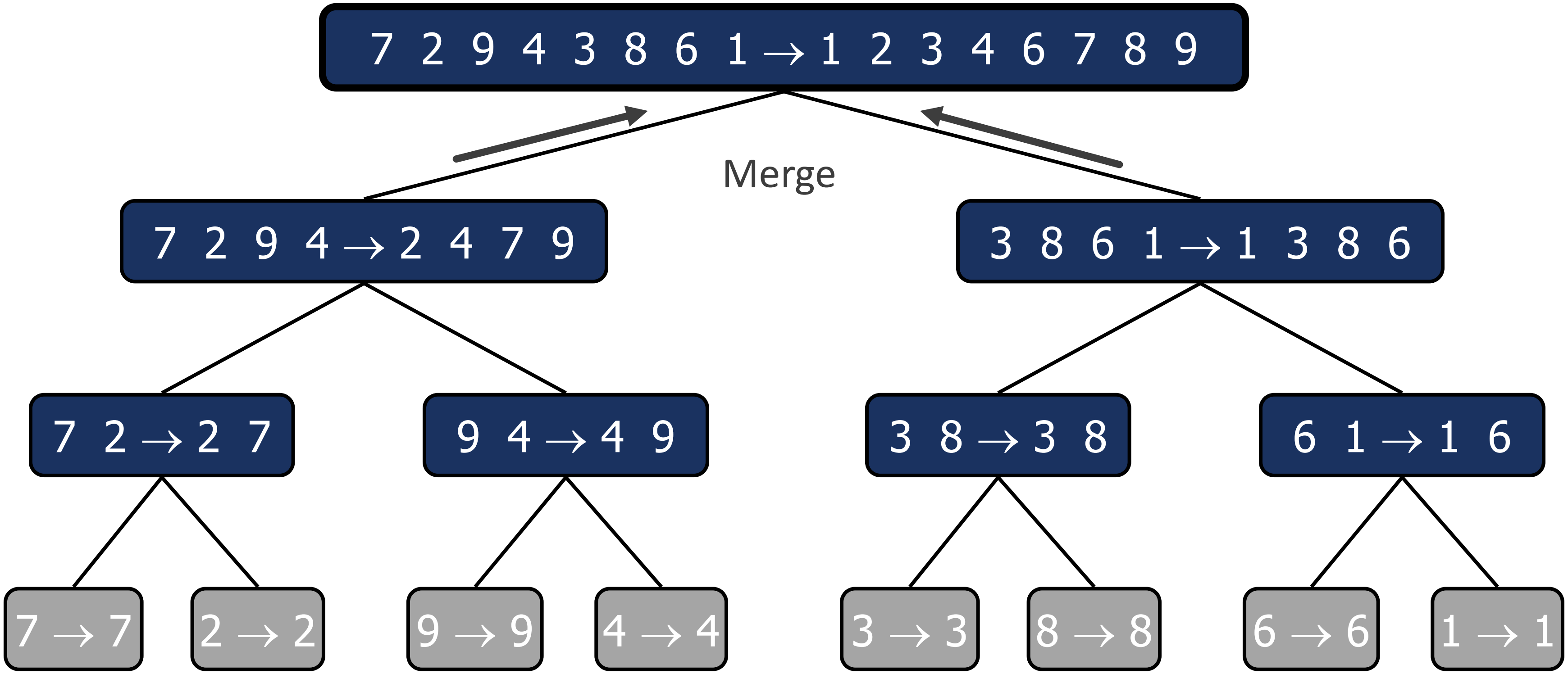
3 → 3

8 → 8

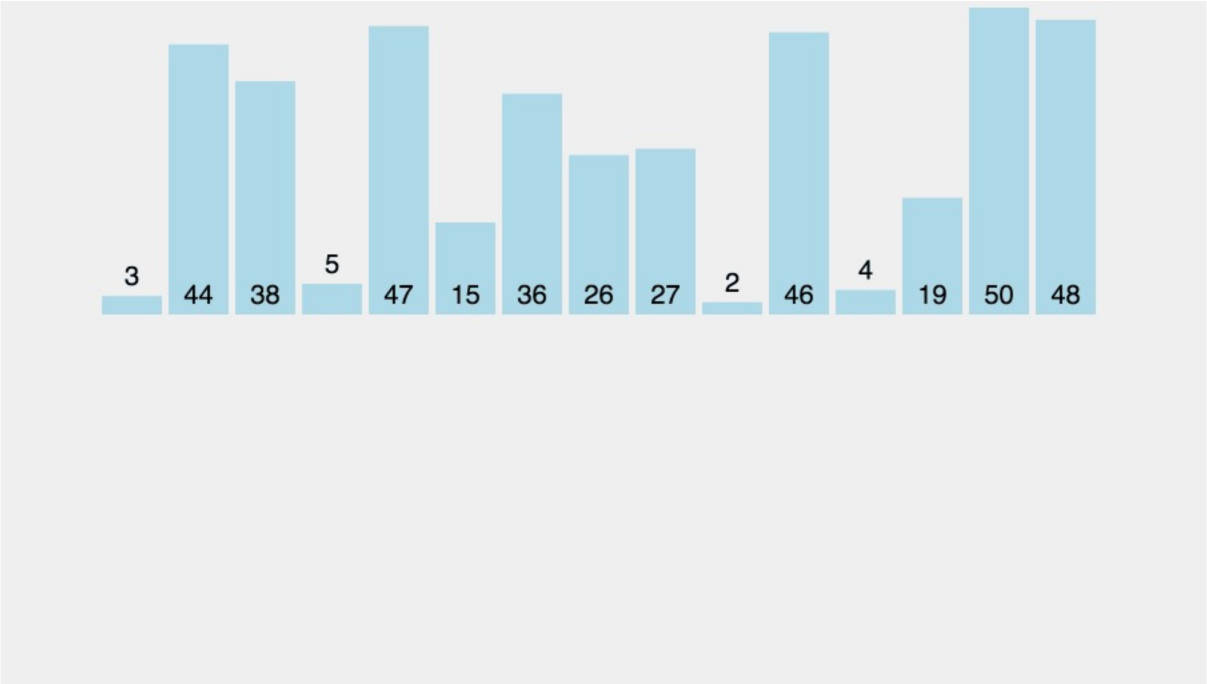
6 → 6

1 → 1

Mergesort Example



Mergesort Visualized Demo



厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Image source: https://thumbs.gfycat.com/ZealousAdolescentBellsnake-size_restricted.gif

Mergesort

- Call $\text{MergeSort}(A, 1, \text{len}[A])$ for the sorting problem.
- Recursive call with different array index:
    - p: starting index
    - q: middle index
    - r: end index
- Exit condition: $p = r$, there is only one element.

```
$\text{MergeSort}(A, p, r)$
if $p < r$ then
        $q \leftarrow \lfloor (p + r)/2 \rfloor$
        $\text{MergeSort}(A, p, q)$
        $\text{MergeSort}(A, q + 1, r)$
        $\text{Merge}(A, p, q, r)$
```

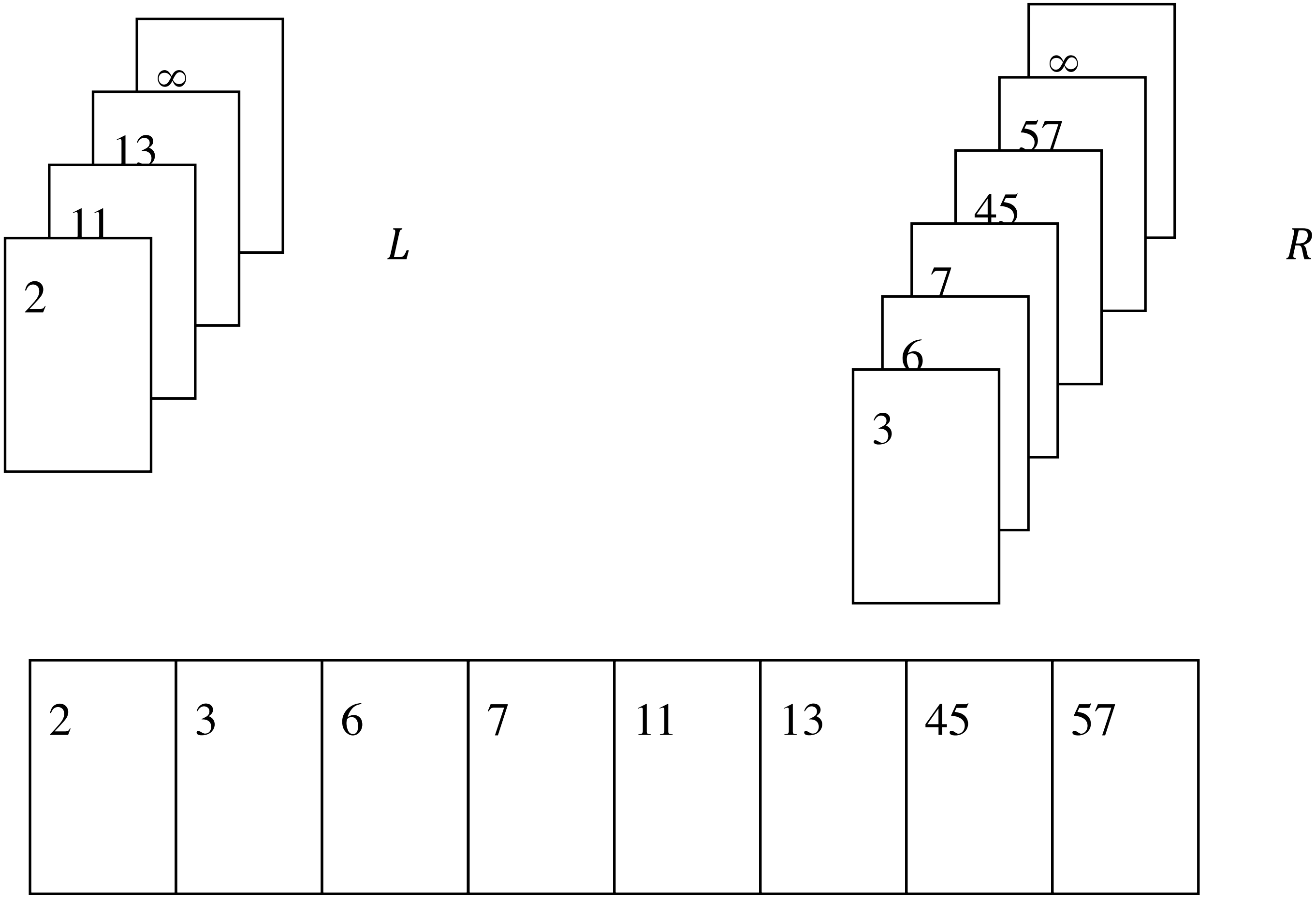
Mergesort

```
Merge($A, p, q, r$)
1   $n_1 \leftarrow q - p + 1$
2   $n_2 \leftarrow r - q$
3   for $i \leftarrow 1$ to $n_1$ do
4       $L[i] \leftarrow A[p + i - 1]$
5   for $j \leftarrow 1$ to $n_2$ do
6       $R[j] \leftarrow A[q + j]$
7   $L[n_1 + 1] \leftarrow \infty$
8   $R[n_2 + 1] \leftarrow \infty$
```

```
9   $i \leftarrow 1$
10  $j \leftarrow 1$
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \leq R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else  $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

- Line 1-6: L and R are used to store two sorted subarrays with size n_1 and n_2.
- Line 7-8: Assign infinity at the end of L and R for comparison convenience.
- Line 9-16: For each index from p to r, compare one by one and increase the index of the array with smaller element.

Mergesort



The Correctness of Merge

```
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \le R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else  $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

- Loop invariant:
  - At the start of each iteration of the for loop in Lines 12-17, the subarray $A[p \dots k-1]$ contains the $k-p$ smallest elements of L and R, in sorted order.
  - $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
- We show that this loop invariant holds
  - prior to the first iteration of the loop;
  - after the kth iteration of the loop;
  - when the loop terminates.

The Correctness of Merge

```
for $k \leftarrow p$ to $r$ do
    if $L[i] \le R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
```

Initialization

- Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p \dots p-1]$ is empty.
- Both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

The Correctness of Merge

```
11  for $k \leftarrow p$ to $r$ do
12      if $L[i] \le R[j]$ then
13          $A[k] \leftarrow L[i]$
14          $i \leftarrow i + 1$
15      else $A[k] \leftarrow R[j]$
16          $j \leftarrow j + 1$
```

Maintenance

- Hypothesis: Before the kth iteration, the loop invariant holds.
- Let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A.
- Because $A[p \dots k-1]$ contains the $k-p$ smallest elements, after Line 13 copies $L[i]$ into $A[k]$, the subarray $A[p \dots k]$ will contain the $k-p+1$ smallest elements.
- Before the next iteration, k and i are increased by 1.
    - $A[p \dots k]$ contain the $k-p+1$ smallest elements.
    - $L[i+1]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
- Therefore, before the $(k+1)$th iteration, the loop invariant holds.

The Correctness of Merge

```
for $k \leftarrow p$ to $r$ do
    if $L[i] \leq R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
```

Termination

- At termination, $k = r + 1$.
- By the loop invariant, the subarray $A[p \dots k-1]$, which is $A[p \dots r]$, contains the $k - p = r - p + 1$ smallest elements of L and R, in sorted order.
- Therefore, the loop invariant holds. Merge correctly merges two sorted arrays into one sorted array.

Time Complexity of Mergesort

- No matter how different the input is, Merge always does $r - p + 1 = n$ times of key comparison.
  - For sorting algorithms, we usually only count the number of key comparisons.
- So the recursion equation is:

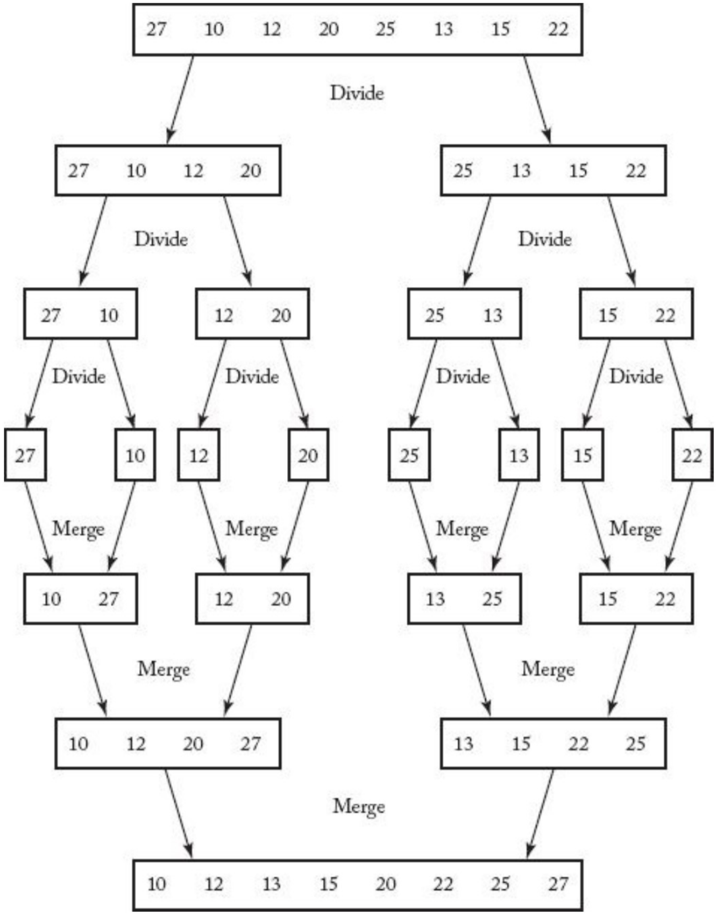
$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$$

- By the master method case 2, we have $f(n) = n = \Theta(n) = \Theta\left(n^{\log_2 2}\right)$.
- Therefore, $T(n) = \Theta(n \lg n)$ for best-, worst- and average-case.

Classroom Exercise

- Write each step of Mergesort to sort the following array:
$$\langle 27, 10, 12, 20, 25, 13, 15, 22 \rangle$$

Classroom Exercise



厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY 厦门大学计算机科学系 Computer Science Department of Xiamen University

Image source: Figure 2.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

QUICKSORT

Quicksort

- Mergesort splits the array first, and then combines them by merging.
- Can we roughly sort the array first, and then split it?
  - E.g. put small elements on the left, and large element on the right.
  - If we can do in this way, we don't need to merge.

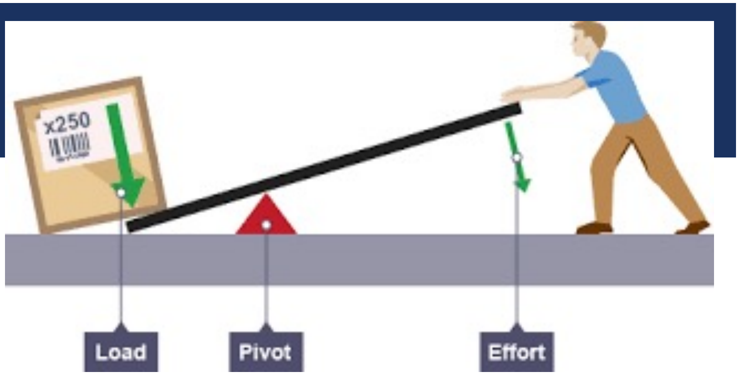
Quicksort

- Quicksort (快速排序) is developed by British computer scientist Charles Antony Richard Hoare (Tony Hoare) in 1962.
- You can know the main property of Quicksort by its name – quick!
- When implemented well, it can be about two or three times faster than Mergesort.

Tony Hoare in 2011

Image source: https://en.wikipedia.org/wiki/Tony_Hoare

Quicksort



Steps:

- Randomly select a pivot (支点) element.
  - Conventional use the first or last item.
- Put all the elements smaller than the pivot element on its left, and all the elements greater than the pivot element on its right.
- Recursively sort the left subarray and right subarray.
  - Each subarray is sorted after recursion call. Therefore, there’s no need to combine the results.

Image source: http://mrtremblaycambridge.weebly.com/p15-turning-on-a-pivot.html

Quicksort Visualized Demo

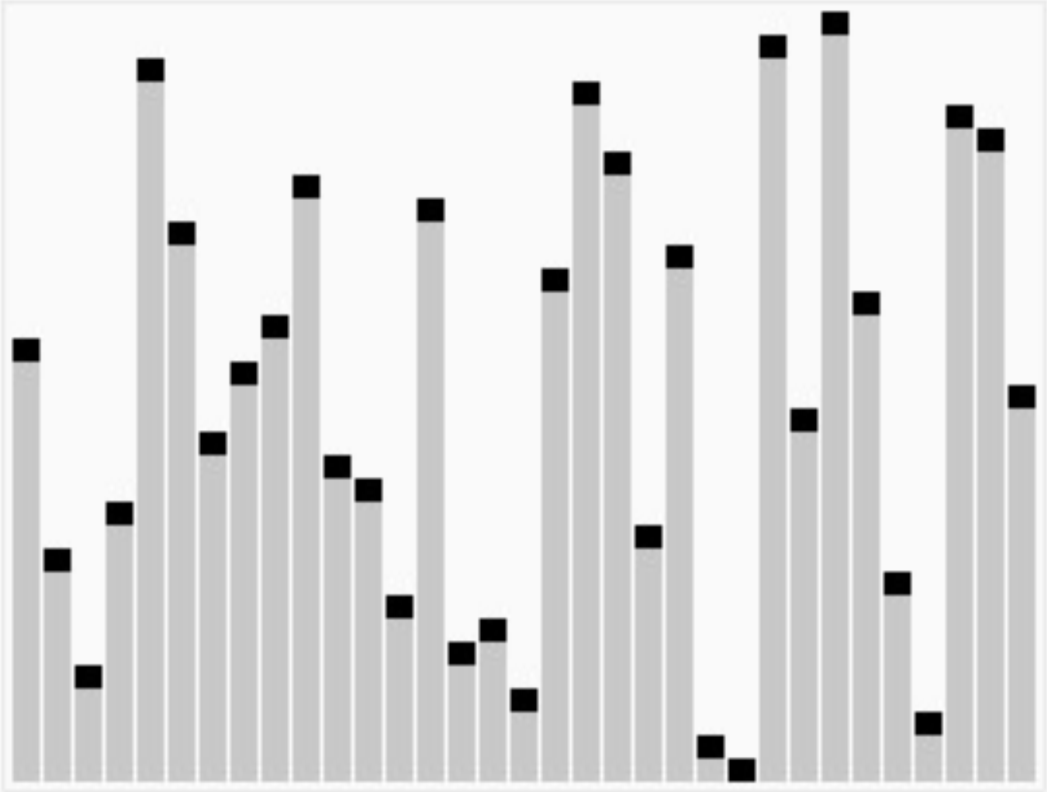


Image source: https://en.wikipedia.org/wiki/Quicksort

Quicksort

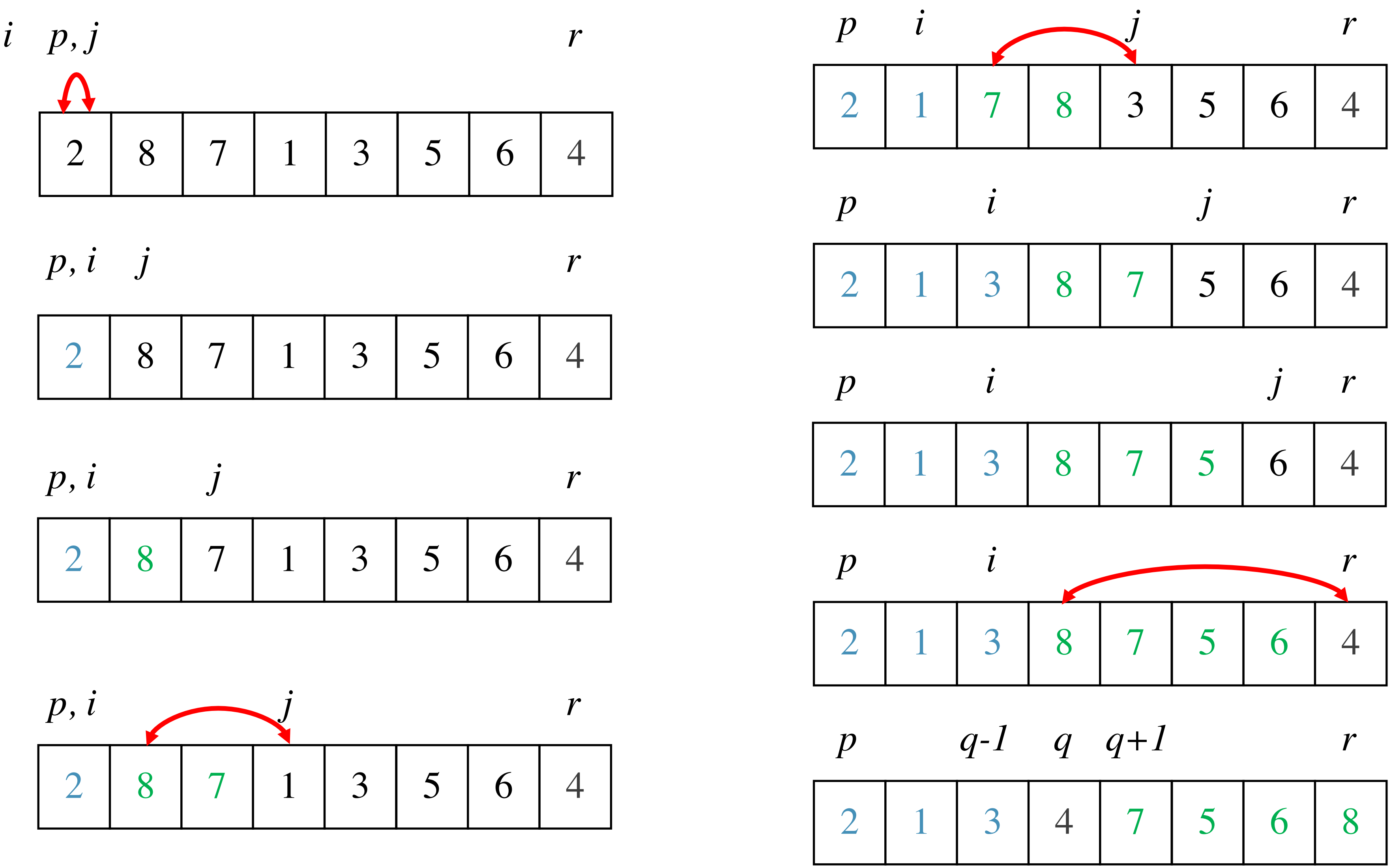
- Call $\text{QuickSort}(A, 1, \text{len}[A])$ for the sorting problem.
- Recursive call with different array index:
  - p: starting index
  - q: pivot index
  - r: end index
- Exit condition: $p = r$, there is only one element.

```
QuickSort($A, p, r$)
    **if** $p < r$ **then**
        $q \leftarrow \text{Partition}(A, p, r)$
        $\text{QuickSort}(A, p, q - 1)$
        $\text{QuickSort}(A, q + 1, r)$
```

Quicksort

- Line 1: Simply select the last element $A[r]$ as the pivot.
- Line 2: Use i to store the index for switching.
- Line 3-6: iterate over j to find elements smaller than x and switch them to the front.
- Line 7: put pivot at the proper position.

```
Partition($A, p, r$)
1  $pivot \leftarrow A[r]$
2  $i \leftarrow p - 1$
3  for $j \leftarrow p$ to $r - 1$ do
4      if $A[j] \leq pivot$ then
5          $i \leftarrow i + 1$
6          $A[i] \leftrightarrow A[j]$
7  $A[i + 1] \leftrightarrow A[r]$
8  return $i + 1$
```



i p, j r

2 8 7 1 3 5 6 4

p, i j r

2 8 7 1 3 5 6 4

p, i j r

2 8 7 1 3 5 6 4

p, i j r

2 8 7 1 3 5 6 4

p i j r

2 1 7 8 3 5 6 4

p i j r

2 1 3 8 7 5 6 4

p i j r

2 1 3 8 7 5 6 4

p i r

2 1 3 8 7 5 6 4

p q-1 q q+1 r

2 1 3 4 7 5 6 8

Time Complexity of Quicksort

- In Partition, each element in A is compared with the pivot except itself.
- Therefore, the number of comparisons in Partition is $n - 1$.

```
Partition($A, p, r$)
pivot $\leftarrow A[r]$
$i \leftarrow p - 1$
**for** $j \leftarrow p$ **to** $r - 1$ **do**
    **if** $A[j] \leq pivot$ **then**
        $i \leftarrow i + 1$
        $A[i] \leftrightarrow A[j]$
$A[i + 1] \leftrightarrow A[r]$
**return** $i + 1$
```

Worst-Case Time Complexity of Quicksort

- The worst-case occurs when he array is already sorted (in either nondecreasing or nonincreasing order).
- In each recursion step, the pivot element is always the smallest or largest item.
  - Thus, n elements are divided into $n-1$ and 0 elements during recursive call.
- The recursion equation is:

$$T(n) = T(n-1) + T(0) + n - 1$$

- Using recursion tree, we can easily get

$$T(n) = n(n-1)/2 = \Theta(n^2)$$

Worst-Case Time Complexity of Quicksort

- The closer the input array is to being sorted, the closer we are to the worst-case performance.
  - Because the pivot can't fairly separate two subarrays.
  - Recursion loses it power.
- How to wisely choose the pivot?
  - Random.
  - Median of $A[1]$, $A[\lfloor n/2 \rfloor]$, and $A[n]$. Safe to avoid the worst-case but more comparisons are needed.
- What will be the best case?

Best-Case Time Complexity of Quicksort

- The best-case occurs when the Partition almost evenly splits the array:
  - Array has odd number of elements: Both subarrays have $\lfloor n/2 \rfloor$ elements.
  - Array has even number of elements: One subarrays has $\lfloor n/2 \rfloor$ elements and another has $n/2$.
- In both case, the size of the subarray is no more than $n/2$.
- The recursion equation is:

$$T(n) = 2T(n/2) + n - 1 = O(n \lg n).$$

Classroom Exercise

What is the best case input for Quicksort when $n = 12$?

Classroom Exercise

Solution:

The best case occurs when each pivot evenly split the array:

$$1, 2, 4, 5, 3, 7, 8, 10, 12, 11, 9, 6$$

Think: how to write a best-case input generator for Quicksort?

Average-Case Time Complexity of Quicksort

- The worst-case of Quicksort is no faster than insertion sort (also $\Theta(n^2)$), and slower than Mergesort ($\Theta(n \log n)$).
- The best-case of Quicksort is slower than insertion sort ($\Theta(n)$).
- How dare it name itself “quick”?
  - The average-case behavior earns its name!

Average-Case Time Complexity of Quicksort

- To analyze the average-case time complexity, we can add randomization.
  - Randomly permutate the input array (uniform distributed input).
  - Randomly choose the pivot item.

Average-Case Time Complexity of Quicksort

- By randomization, now the probability of pivot being any item in the array is $1/n$.

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}[T(p-1) + T(n-p)] + n - 1$$

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

$$nT(n) = 2\sum_{p=1}^{n} T(p-1) + n(n-1) \text{ (multiply by } n\text{)}$$

$$(n-1)T(n-1) = 2\sum_{p=1}^{n-1} T(p-1) + (n-1)(n-2) \text{ (apply to } n-1\text{)}$$

Average-Case Time Complexity of Quicksort

$$nT(n) - (n-1)T(n-1)$$
$$= 2T(n-1) + 2(n-1) \text{ (subtraction)}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

- Let $a_n = \frac{T(n)}{n+1}$,

Harmonic series

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \approx 2 \sum_{i=1}^{n} \frac{1}{i} \approx 2 \ln n\,.$$

- Therefore, $T(n) \approx (n+1)2\ln n = (n+1)2\ln 2 \lg n \approx 1.38(n+1)\lg n = \Theta(n \lg n)$.

LARGE INTEGER MULTIPLICATION

Large Integer Multiplication

- Suppose that we need to do arithmetic operations on integers whose size is very large.
- In cryptography (密码学) and network security, encryption and decryption need to multiply very large numbers.
- How to do arithmetic for those large integers?

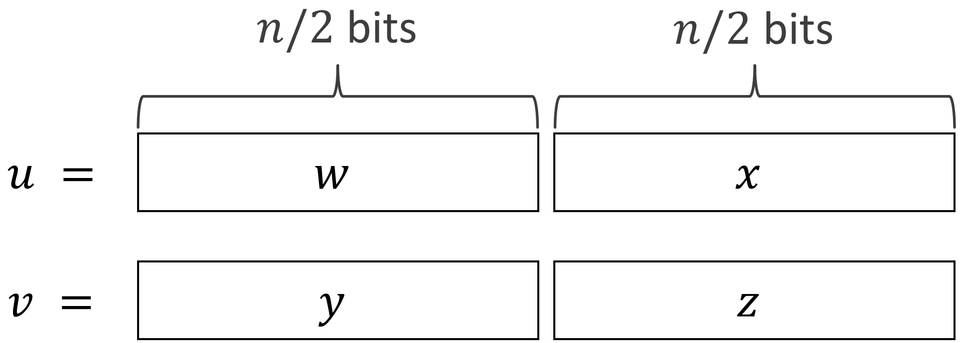
Large Integer Multiplication

- Tradition algorithm is nothing but what we have learned in primary school.
- It takes $\Theta(n^2)$ bit operations.

```
      ×    ************
           ************
-------------------------------
           ************
          ************
         ************
        ************
       ************
      ************
     ************
    ************
   ************
  ************
 ************
************
-------------------------------
************************
```

Large Integer Multiplication

- Use n-bit binary representation for u and v.
- We can use divide-and-conquer: Each integer is divided into two parts of $n/2$ bits each.



$$u = \overbrace{\boxed{\quad w \quad}}^{n/2 \text{ bits}}\ \overbrace{\boxed{\quad x \quad}}^{n/2 \text{ bits}}$$

$$v = \boxed{\quad y \quad}\ \boxed{\quad z \quad}$$

Example:
$(110011)_2$
$= (110)_2 \times 2^3 + (011)_2$

- Therefore, integers u and v can be represented as:

$$u = w2^{n/2} + x, \qquad v = y2^{n/2} + z.$$

- Then, we have:

$$uv = \left(w2^{n/2} + x\right)\left(y2^{n/2} + z\right) = wy2^{n} + (wz + xy)2^{n/2} + xz.$$

Large Integer Multiplication

- We need 4 recursive multiplications with size $n/2$ to calculate

$$wy2^n + (wz + xy)2^{n/2} + xz$$

  - Multiply with 2^n is to simply shift by n bits to the left with cost $\Theta(n)$.
  - 3 times of addition is also with cost $\Theta(n)$.
- The recursion equation is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 4T(n/2) + cn & n > 1 \end{cases}$$

- By the master method, we get $T(n) = \Theta(n^2)$.
- It is still quadratic. Why?

Large Integer Multiplication

- We decompose the instance of n into 4 small instances with size $n/2$.
- If we can decrease 4 to 3, by the master method we get $T(n) = \Theta(n^{\lg 3})$.
- As before, we need to calculate

$$wy, wz + xy, xz$$

- If instead we set

$$r = (w + x)(y + z) = wy + (wz + zy) + xz$$

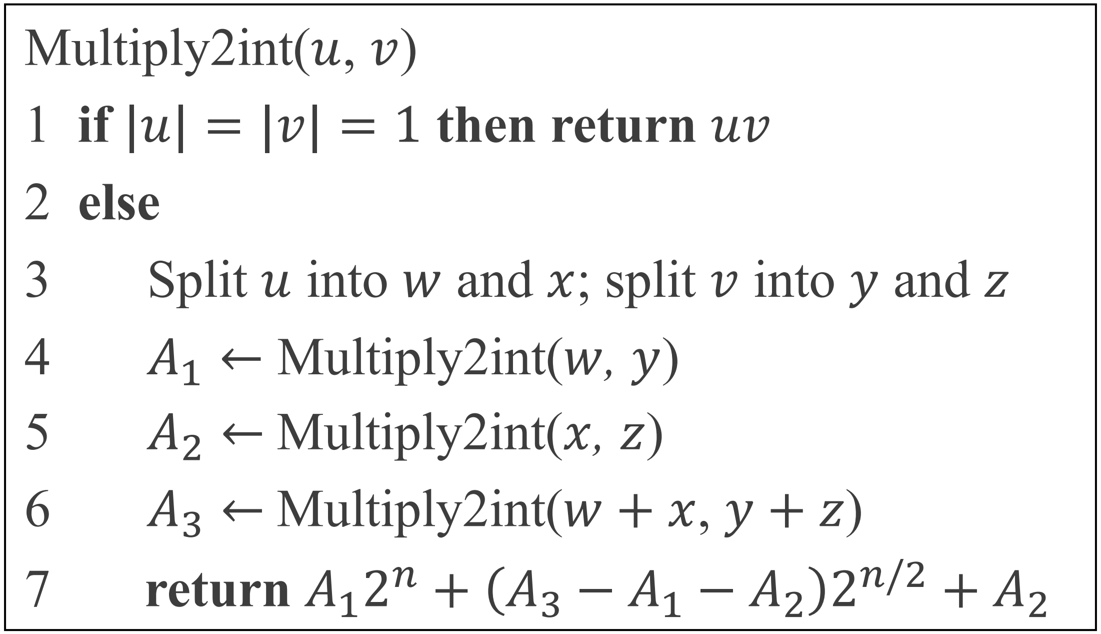
we have

$$wz + xy = r - wy - xz$$

- Then, we only need to calculate

$$r, wy, xz$$

Large Integer Multiplication



```
Multiply2int($u$, $v$)
if $|u| = |v| = 1$ then return $uv$
else
    Split $u$ into $w$ and $x$; split $v$ into $y$ and $z$
    $A_1 \leftarrow$ Multiply2int($w$, $y$)
    $A_2 \leftarrow$ Multiply2int($x$, $z$)
    $A_3 \leftarrow$ Multiply2int($w + x$, $y + z$)
    return $A_1 2^n + (A_3 - A_1 - A_2)2^{n/2} + A_2$
```

- The above method yields the following recursion equation:

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

- By the master method, we get $T(n) = \Theta(n^{\lg 3}) \approx \Theta(n^{1.59})$.

Classroom Exercise

Write the pseudocode of binary search algorithm:

- Given a sorted array A,
  - If x equals the middle item, quit.
  - Otherwise, compare x with the middle item.
    - If x is smaller, search the left subarray.
    - If x is greater, search the right subarray.
- What is the time complexity?

Classroom Exercise

Solution:

```
BinarySearch($A$, $x$)
$p \leftarrow 1$
$r \leftarrow n$
$k \leftarrow 0$
while $p <= r$ and $k = 0$ do
    $m \leftarrow \lfloor (p + r)/2 \rfloor$
    if $A[m] = x$ then return $m$
    else if $x < A[m]$ then $r \leftarrow m - 1$
        else $p \leftarrow m + 1$
return $0$
```

```
RecursiveBinarySearch($A$, $p$, $r$)
if $p > r$ then return $0$
else
    $m \leftarrow \lfloor (p + r)/2 \rfloor$
    if $A[m] = x$ then return $m$
    else if $x < A[m]$ then
        RecursiveBinarySearch($A$, $p$, $r - 1$)
    else
        RecursiveBinarySearch($A$, $p + 1$, $r$)
```

Divide-and-conquer algorithm is not necessarily implemented by recursion.

Classroom Exercise

- The recursion equation is:

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

- By using master method case 2, we have $T(n) = \Theta(\lg n)$.

MATRIX MULTIPLICATION

Matrix Multiplication

- Given matrices A and B with size $n \times n$, compute the matrix product $C = AB$.
- The formula we have learned in linear algebra for doing this is:

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j) .$$

- Calculating each $C(i,j)$ takes $O(n)$. Thus, calculating total $n \times n$ elements in C takes $O(n^3)$.

Matrix Multiplication

- Suppose we want to product C of two 2×2 matrices, A and B, That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

- The divide-and-conquer version consists of computing C as defined by the following equation:

$$C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

Matrix Multiplication

- Will it be better?
- The cost of multiplying two $n \times n$ matrices consists of:
  - 8 times the cost of multiplying two $n/2 \times n/2$ matrices;
  - 4 times the cost of adding two $n/2 \times n/2$ matrices.
- The recursion equation is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T(n/2) + 4(n/2)^2 & n > 1 \end{cases}$$

- Make use of the master method, $T(n) = \Theta(n^3)$. And thus this method is no faster than the ordinary one.
- What can we do?

Matrix Multiplication with Strassen's algorithm

- Strassen‘s algorithm (斯特拉森算法) reduces the number of multiplications from 8 to 7.
- Strassen determined that if we let

$$
\begin{aligned}
m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_2 &= (a_{21} + a_{22})b_{11} \\
m_3 &= a_{11}(b_{12} - b_{22}) \\
m_4 &= a_{22}(b_{21} - b_{11}) \\
m_5 &= (a_{11} + a_{12})b_{22} \\
m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
m_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}
$$

Less multiplication, at expense of more addition and subtraction.

the product C is given by

$$
C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}
$$

Matrix Multiplication with Strassen's algorithm

- To multiply two 2×2 matrices, Strassen’s method requires 7 multiplications and 18 additions/subtractions.
  - The standard method requires 8 multiplications and 4 additions/subtractions.
  - Use 14 more additions/subtractions to save 1 multiplication. It that worthy?
- Recursion equation:

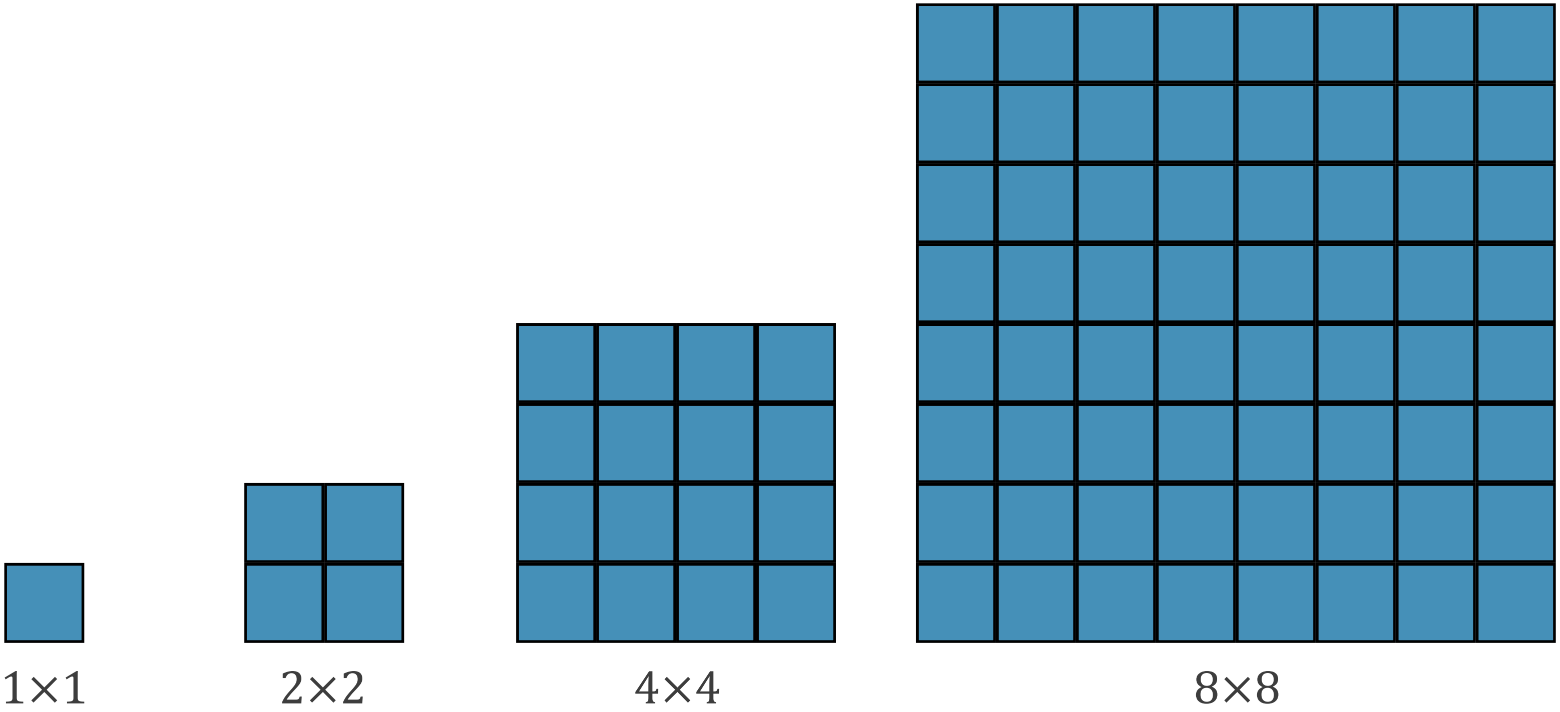
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 7T(n/2) + 18(n/2)^2 & n > 1 \end{cases}$$

- Use the master method case 1, $f(n) = \frac{18}{4}n^2 = O(n^{\log_2 7 - \epsilon}) \approx O(n^{2.81-\epsilon})$ for $\epsilon \approx 0.81$.
- Therefore, we have $T(n) = \Theta(n^{2.81})$.

DEFECTIVE CHESSBOARD

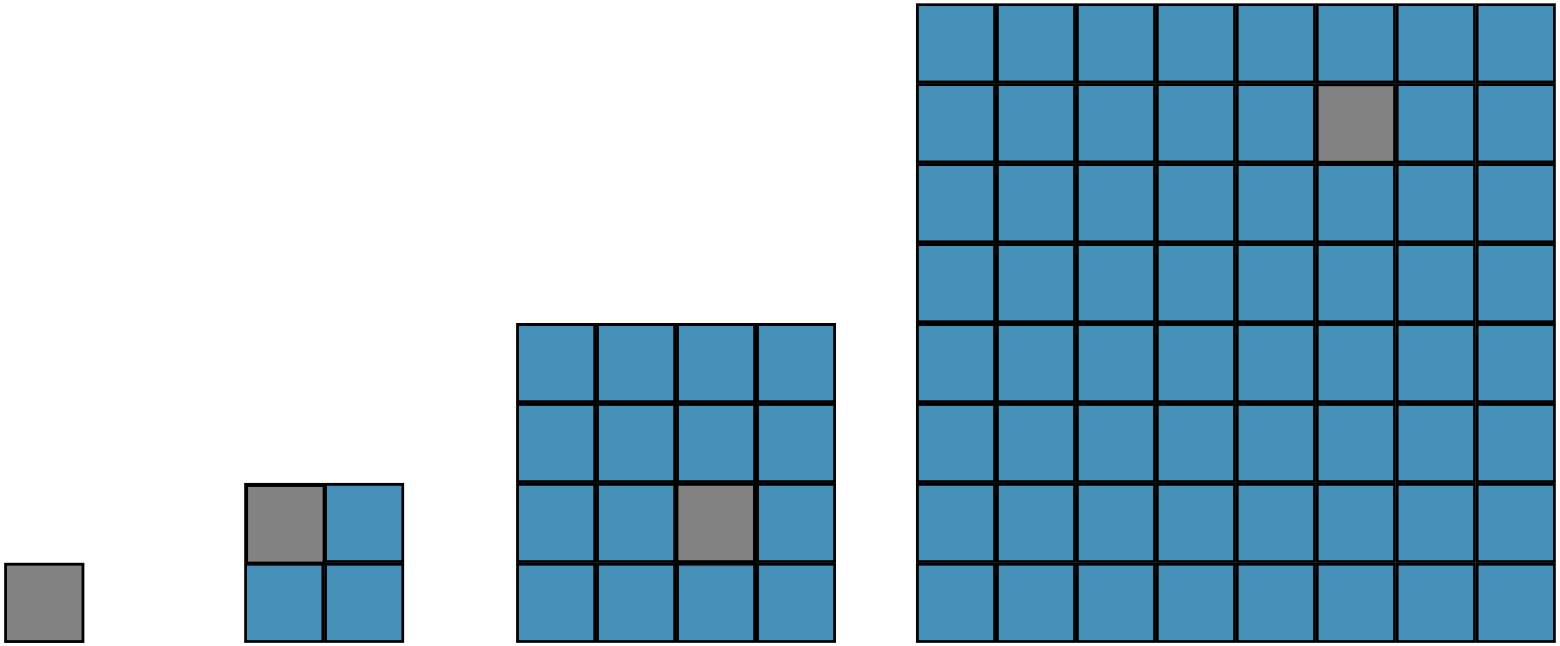
Defective Chessboard

- A chessboard is an $2^n \times 2^n$ grid, for $n \geq 0$:



Defective Chessboard

- A defective chessboard (残缺棋盘) is chessboard that has one unavailable (defective) position.



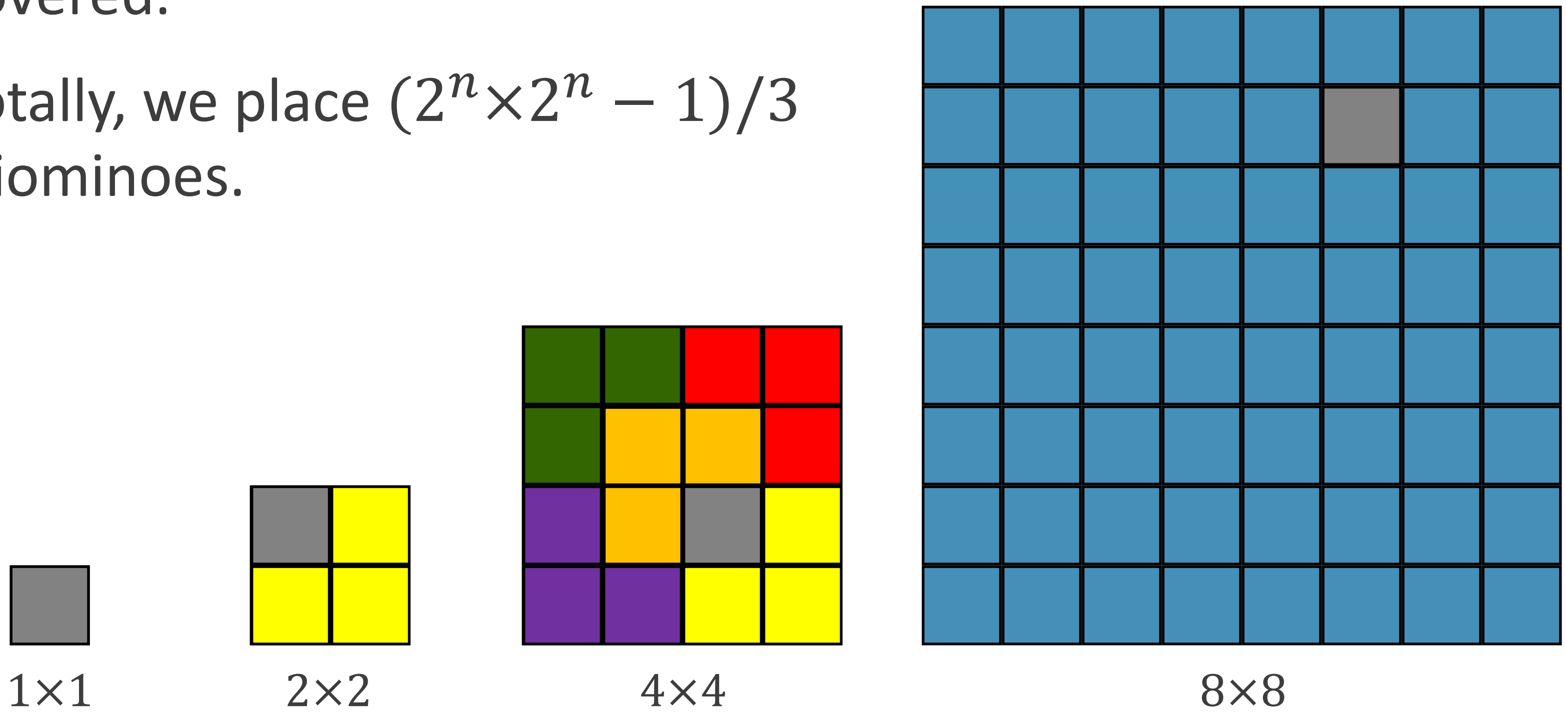
1×1 2×2 4×4 8×8

Defective Chessboard

- A triomino (三格板) is an L shaped object that can cover three squares of a chessboard.
- A triomino has four orientations.
- You can use infinite number of triominoes.

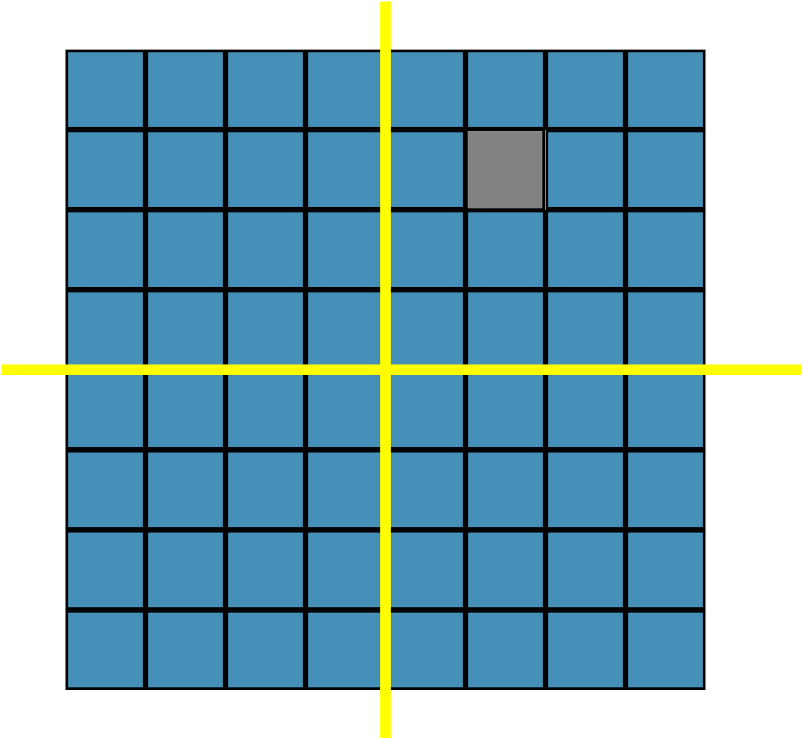
Defective Chessboard

- Task: Place triominoes on an $2^n \times 2^n$ $(n \geq 1)$ defective chessboard so that all $2^n \times 2^n - 1$ nondefective positions are covered.
- Totally, we place $(2^n \times 2^n - 1)/3$ triominoes.



Defective Chessboard

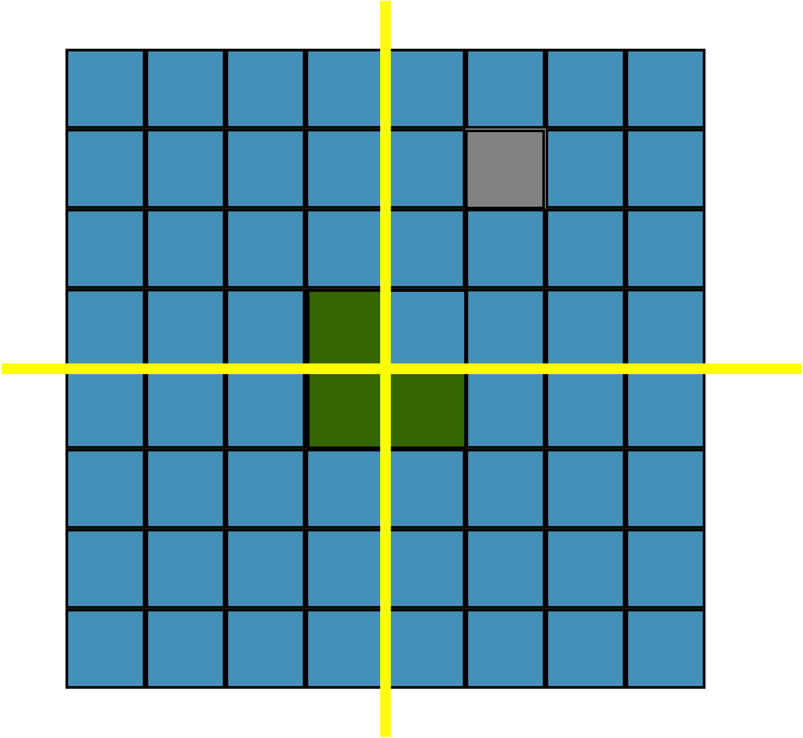
- How to use divide-and-conquer?



- If we divide it into 4 $2^{n-1} \times 2^{n-1}$ chessboard, only one is defective.

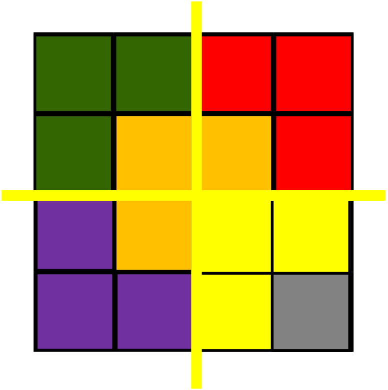
Defective Chessboard

- Put one triomino at their common corner, which makes all 4 small chessboards have a defective position.



Defective Chessboard

- Then, simply recursively solve this problem.



```
TileBoard($tr, tc, dr, dc, size$)
if $size = 1$ return ok
$tile \leftarrow tile + 1$; $t \leftarrow tile$
$s \leftarrow size/2$
```

Left top

```
if $dr < tr + s$ and $dc < tc + s$ then
    TileBoard($tr, tc, dr, dc, s$)
else $Board[tr + s - 1, tc + s - 1] \leftarrow t$
    TileBoard($tr, tc, tr + s - 1, tc + s - 1, s$)
```

Right top

```
if $dr < tr + s$ and $dc \geq tc + s$ then
    TileBoard($tr, tc + s, dr, dc, s$)
else $Board[tr + s - 1, tc + s] \leftarrow t$
    TileBoard($tr, tc + s, tr + s - 1, tc + s, s$)
```

Left bottom

```
if $dr \geq tr + s$ and $dc < tc + s$ then
    TileBoard($tr + s, tc, dr, dc, s$)
else $Board[tr + s, tc + s - 1] \leftarrow t$
    TileBoard($tr + s, tc, tr + s, tc + s - 1, s$)
```

Right bottom

```
if $dr \geq tr + s$ and $dc \geq tc + s$ then
    TileBoard($tr + s, tc + s, dr, dc, s$)
else $Board[tr + s, tc + s] \leftarrow t$
    TileBoard($tr + s, tc + s, tr + s, tc + s, s$)
```

When $size = 1$, the board is 1×1.

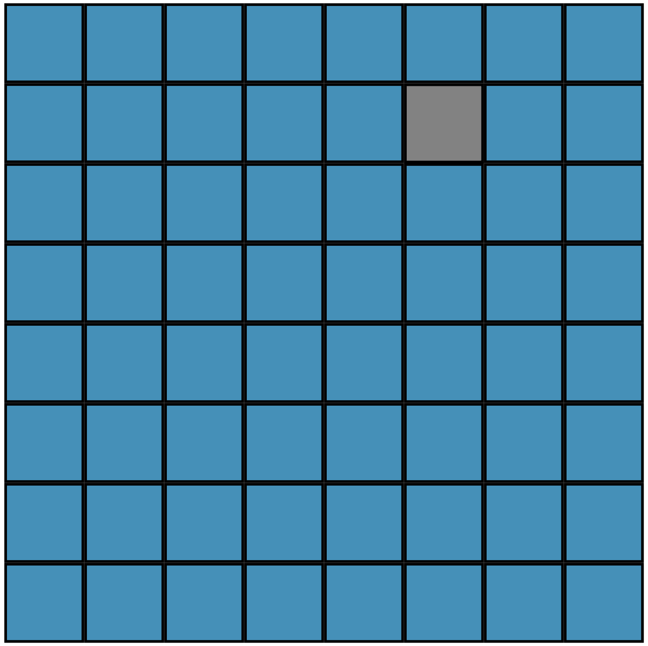
Check if defective position in this region.

Record triomino t on the board.

tr: row of left-upper square
tc: column of left-upper square
dr: row of defective square
dc: column of defective square
$tile$: accumulated triomino number
t: current triomino number

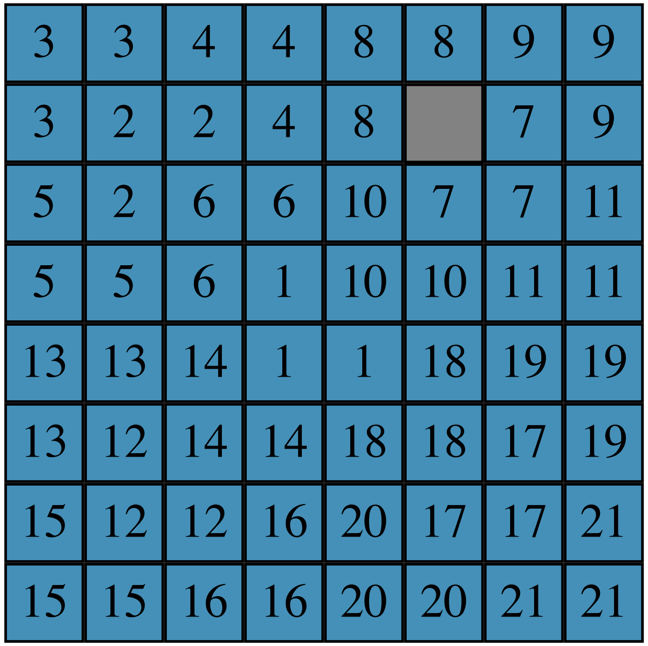
Classroom Exercise

Write down the triomino number in the following defective chessboard.



Classroom Exercise

Solution:



3	3	4	4	8	8	9	9
3	2	2	4	8	■	7	9
5	2	6	6	10	7	7	11
5	5	6	1	10	10	11	11
13	13	14	1	1	18	19	19
13	12	14	14	18	18	17	19
15	12	12	16	20	17	17	21
15	15	16	16	20	20	21	21

Time Complexity

- The recursion equation is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 4T(n-1) + c & n > 1 \end{cases}$$

where

$$\begin{aligned} & 4T(n-1) + c \\ = & 4[4T(n-2) + c] + c \\ = & 4^2T(n-2) + 4c + c \\ = & 4^3T(n-3) + 4^2c + 4c + c \\ = & 4^{n-1}T(1) + 4^{n-2}c + \cdots + 4c + c \\ = & \Theta(4^{n-1}) \end{aligned}$$

DETERMINING THRESHOLD

Determining Thresholds

- For matrix multiplication and large integer multiplication, when n is small, using standard algorithm will be even faster.
- For Mergesort, using recursive method on small array will also be slower than quadratic sorting algorithm like exchange sort.
- How to determine the threshold?

Determining Thresholds

- If we have the recursive equation of Mergesort measured by computational time:

$$T(n) = 32n \lg n \ \ \mu s$$

  and selection sort takes

$$T(n) = \frac{n(n-1)}{2} \mu s$$

- We can compare and get the threshold:

$$\frac{n(n-1)}{2} < 32n \lg n$$

$$n < 591.$$

When Not to Use Divide-and-Conquer

- An instance of size n is divided into two or more instances each almost of size n.
  - nth Fibonacci term: $T(n) = T(n-1) + T(n-2) + 1$.
  - Worst-case Quicksort is also not acceptable: $T(n) = T(n-1) + n - 1$.
- An instance of size n is divided into almost n instances of size n/c, where c is a constant.
  - E.g. $T(n) = T(n/2) + T(n/2) + \cdots + T(n/2)$.

Conclusion

After this lecture, you should know:

- What is the key idea of divide-and-conquer.
- How to divide a big problem instance into several small instances.
- How to use recursion to design a divide-and-conquer algorithm.
- How Mergesort and Quicksort work and what are their complexity.

Homework

- Page 63-65

  5.1

  5.3

  5.8

  5.10

  5.18

Experiment

- 5.19和5.20中选一题

谢谢

有问题欢迎随时跟我讨论