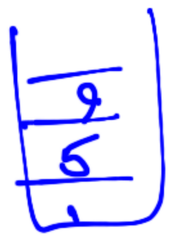
COL106
Data Structures and Algorithms

Subodh Sharma and Rahul Garg

A Note on Expression Evaluation

- Simple Expressions
  - int a, x, y, z
  - a = (x + y - z)
  - Can be evaluated in two ways
    - a = (x + y) - z
    - a = x + (y - z)
  - Both the ways are equivalent
- What if expressions have side effects?
  - a = (expr1) - (expr2)
  - expr1 and expr2 may have side effects (eg, stack pop operation)
  - Result will depend on the order of the evaluation



a = pop() - pop()

First

Case 1 : 9 - 5 ⇒ 4

First

Case 2 : 5 - 9 = -4

x_1 = pop()

x_2 = pop()

a = $x_1 - x_2$

Other Type of Evaluation

- Poor programming practices
  - Expressions with side effects
  - Assumptions about order of evaluation
  - Assumptions about evaluation of expressions

Example: Conditional expressions

if ((a) && (b) && (c)) { .... }

- If expressions a, b, c have side effects then the results may be unpredictable
- If a is false, compiler may not evaluate b and c
  - This is called short circuit evaluation
- a, b, c may be evaluated in any order

Short Circuit Evaluation with Side Effects

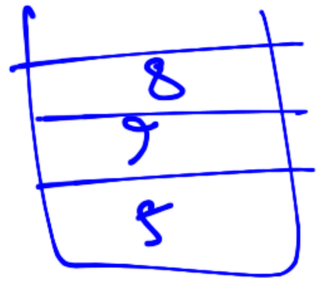
```
if ((pop() == 1) && (pop() == 2) && (pop() == 3))
        cout << "1, 2, 3 found in the stack\n"
```

How many times pop() will be called?

Which condition will be checked in first call to pop()?

A better version

```
a = pop(); b = pop(); c = pop();
if ((a == 1) && (b == 2) && (c == 3))
        cout << "1, 2, 3 found in the stack\n"
```



2.12 Precedence and Order of Evaluation

Table 2.1 summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence, which is higher than that of binary `+` and `-`. The ``operator'' `()` refers to function call. The operators `->` and `.` are used to access members of structures; they will be covered in Chapter 6, along with `sizeof` (size of an object). Chapter 5 discusses `*` (indirection through a pointer) and `&` (address of an object), and Chapter 3 discusses the comma operator.

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - *` (*type*) `sizeof`	right to left
`* / %`	left to right
`+ -`	left to right

Kernighan, B. W., & Ritchie, D. M. (2002). The C programming language.

C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are `&&`, `||`, `?:`, and `,`.) For example, in a statement like

```
x = f() + g();
```

`f` may be evaluated before `g` or vice versa; thus if either `f` or `g` alters a variable on which the other depends, `x` can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

Similarly, the order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n));    /* WRONG */
```

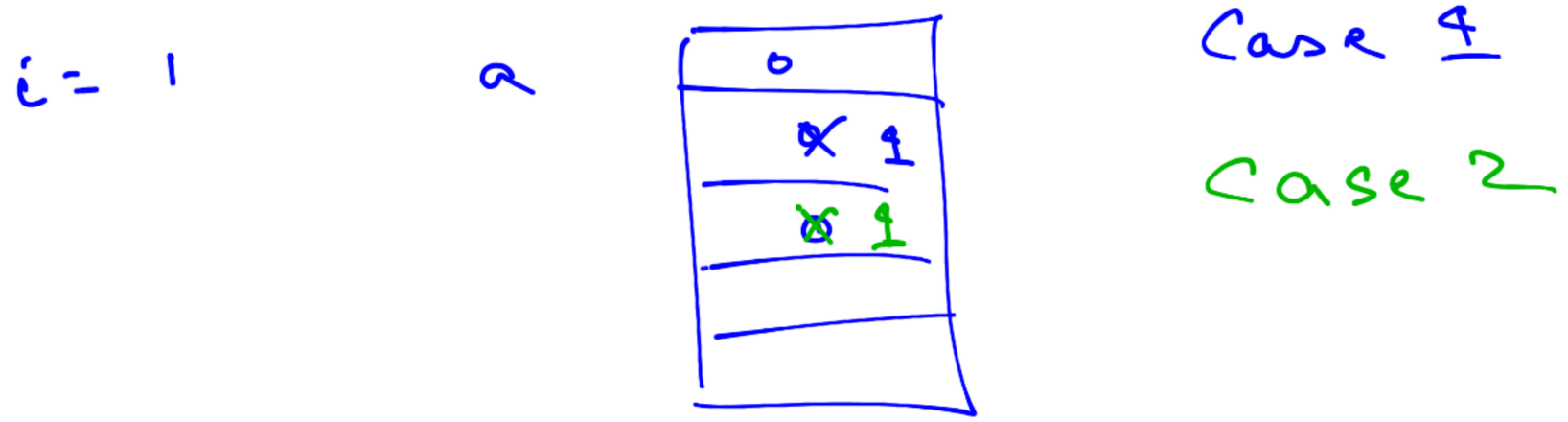
can produce different results with different compilers, depending on whether `n` is incremented before `power` is called. The solution, of course, is to write

```
++n;
printf("%d %d\n", n, power(2, n));
```

Function calls, nested assignment statements, and increment and decrement operators cause ``side effects" - some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated. One unhappy situation is typified by the statement

```
a[i] = i++;
```

The question is whether the subscript is the old value of `i` or the new. Compilers can interpret this in different ways, and generate different answers depending on their interpretation. The



A Note on Plagiarism Policy

COL106 Plagiarism Policy and Honor Code

- Copying or collaborating or use of any other unethical means in exams, quizzes is not permitted
- Every assignment will have its own honor code
  - Indicating what is permitted and what is not
  - Read the honor code carefully, ask in case of doubts
  - Violation of Honor code will lead to
    - D grade
    - Disciplinary committee (DISCO)
    - Any other penalty deemed fit by the instructors

COL106 A1 Plagiarism Issues

COL106 A1 Plagiarism Issues

- Some students have ignored the honour code and plagiarism policy

COL106 A1 Plagiarism Issues

- Some students have ignored the honour code and plagiarism policy
- They will get an email over the weekend

COL106 A1 Plagiarism Issues

- Some students have ignored the honour code and plagiarism policy
- They will get an email over the weekend
- Current penalty: -10 / 100 in the course total

COL106 A1 Plagiarism Issues

- Some students have ignored the honour code and plagiarism policy
- They will get an email over the weekend
- Current penalty: -10 / 100 in the course total
- Implies one or two grade down

COL106 A1 Plagiarism Issues

- Some students have ignored the honour code and plagiarism policy
- They will get an email over the weekend
- Current penalty: -10 / 100 in the course total
- Implies one or two grade down
- Penalty has been decided to be soft considering first offence
  - Future offences will carry stricter panelty

COL106 Plagiarism Policy and Honor Code

- Copying or collaborating or use of any other unethical means in exams, quizzes is not permitted
- Every assignment will have its own honor code
  - Indicating what is permitted and what is not
  - Read the honor code carefully, ask in case of doubts
  - Violation of Honor code will lead to
    - D grade
    - Disciplinary committee (DISCO)
    - Any other penalty deemed fit by the instructors

Sorting

Problem Specification

- Input: A sequence of n numbers $a_1, a_2, \ldots a_n$
- Output: A permutation of the input $b_1, b_2, \ldots b_n$ such that $b_i \leq b_{i+1}$ for all $i \in [1 \,..\, (n-1)]$

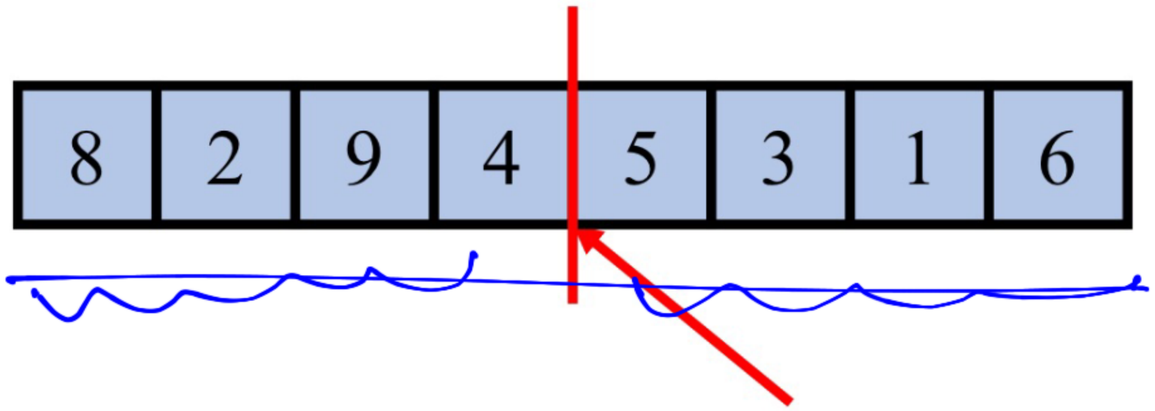
Divide and Conquer

- Very important strategy to solve complex problems
- Divide a complex problem into small sub-problems
- Solve the small sub-problems independently
- Combine the solutions of sub-problems to get the solution to the original problem

Merge Sort: Basic Idea

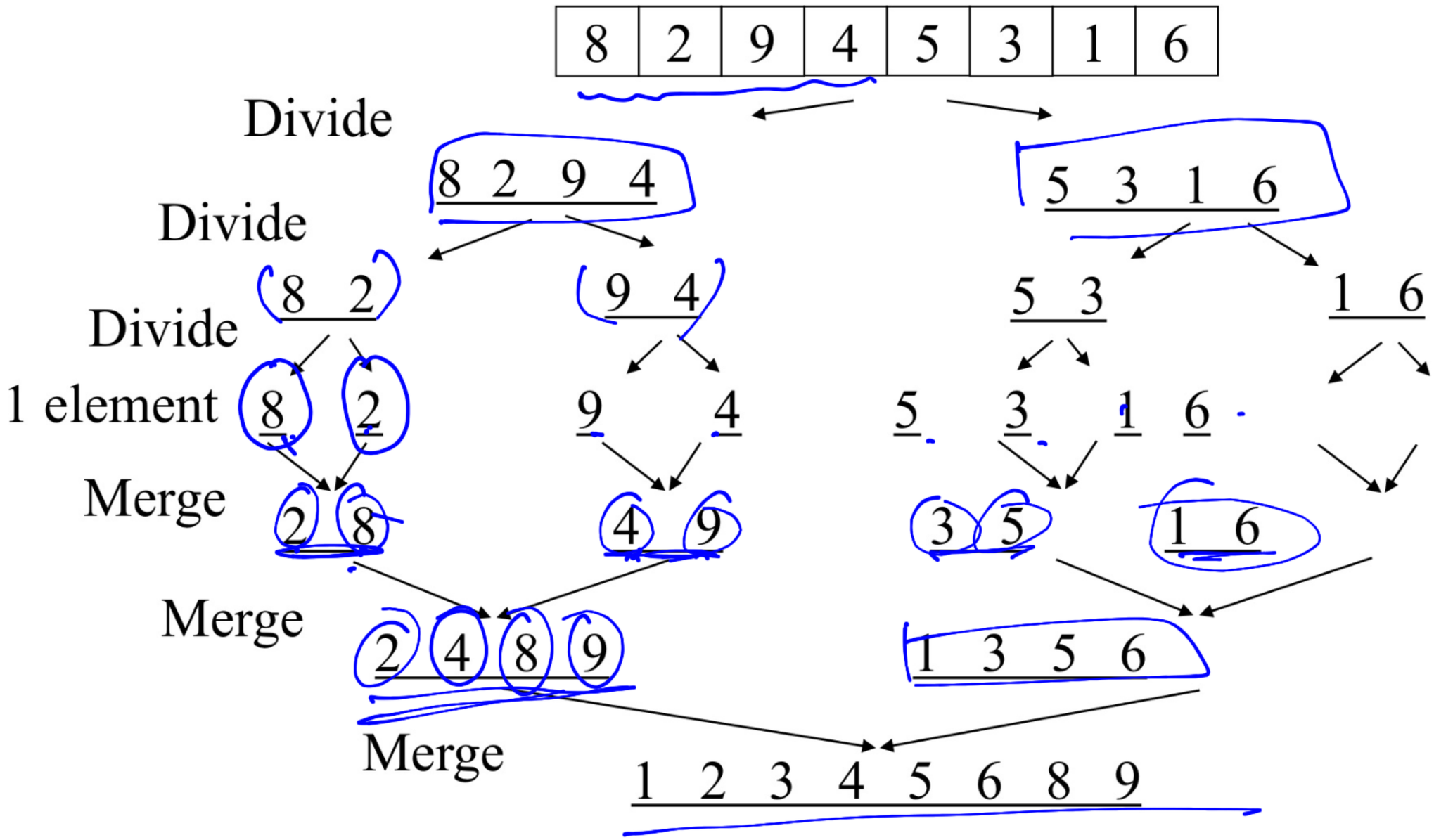
- Divide the input into two (approx.) equal halves
- Independently sort each of the two halves
- Merge the two sorted halves to get one large sorted list of numbers

Merge Sort



- Split at the center
- Recursively sort left and right sub-arrays
- Merge the two subarrays

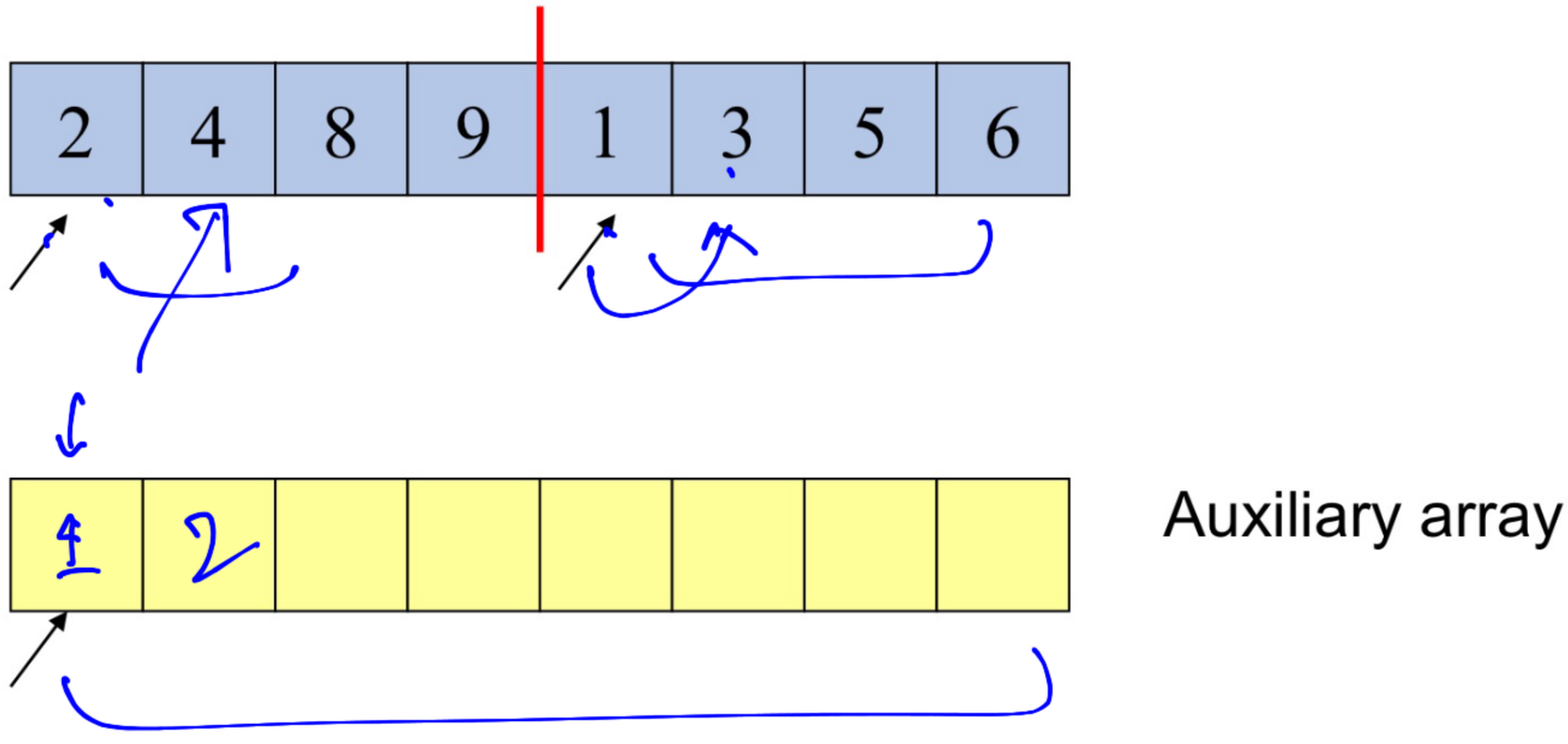
Merge Sort Example



Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

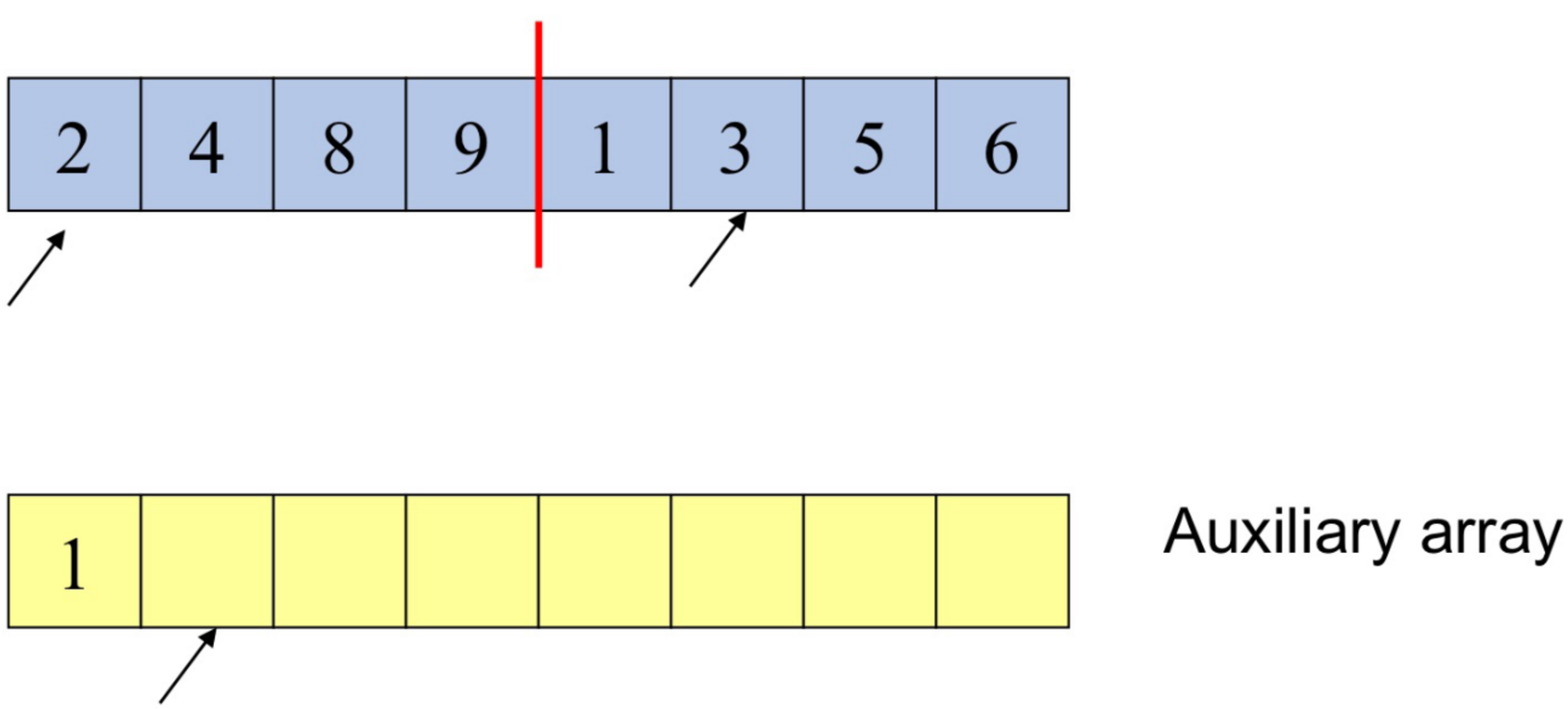
How to Merge?

- Maintain three points
- Need an auxiliary array to merge and copy



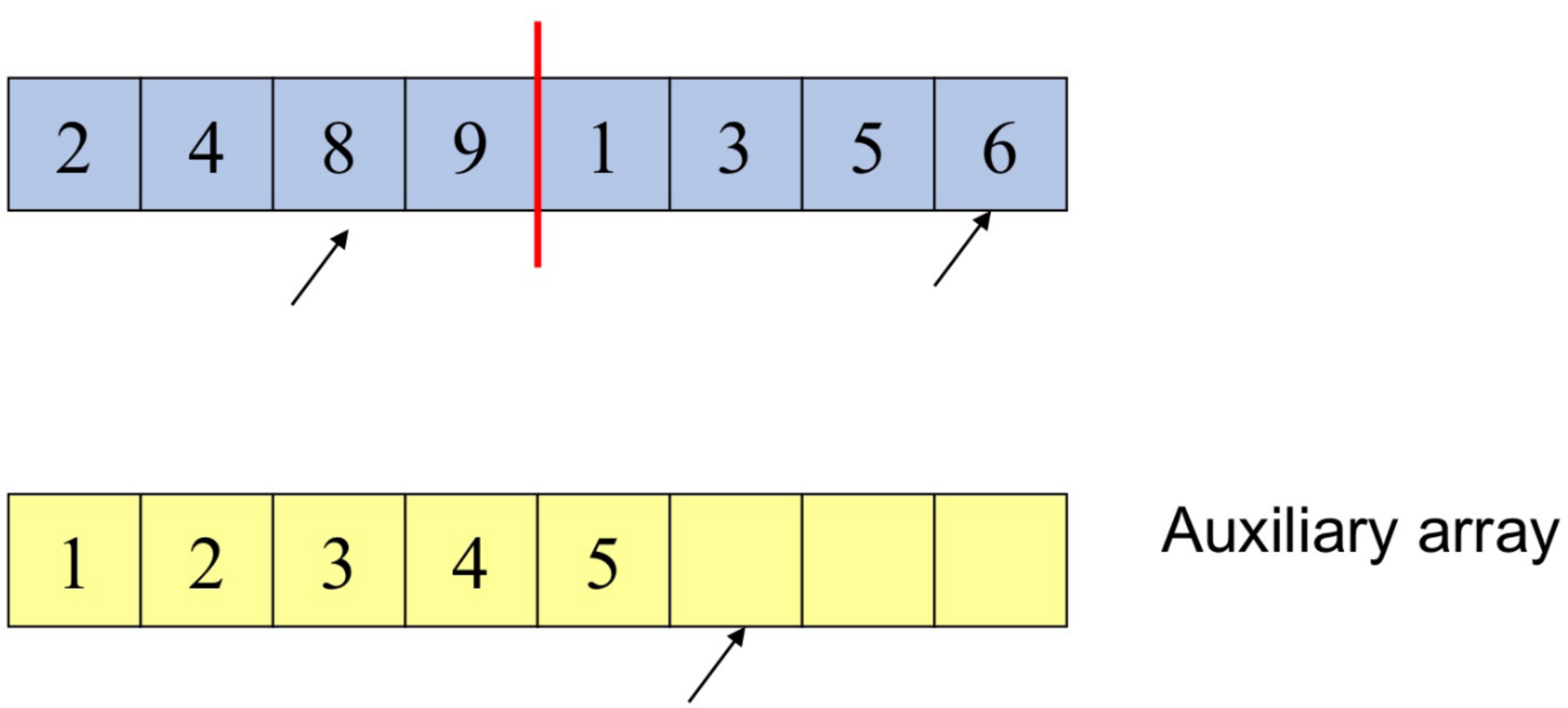
Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Merging



Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

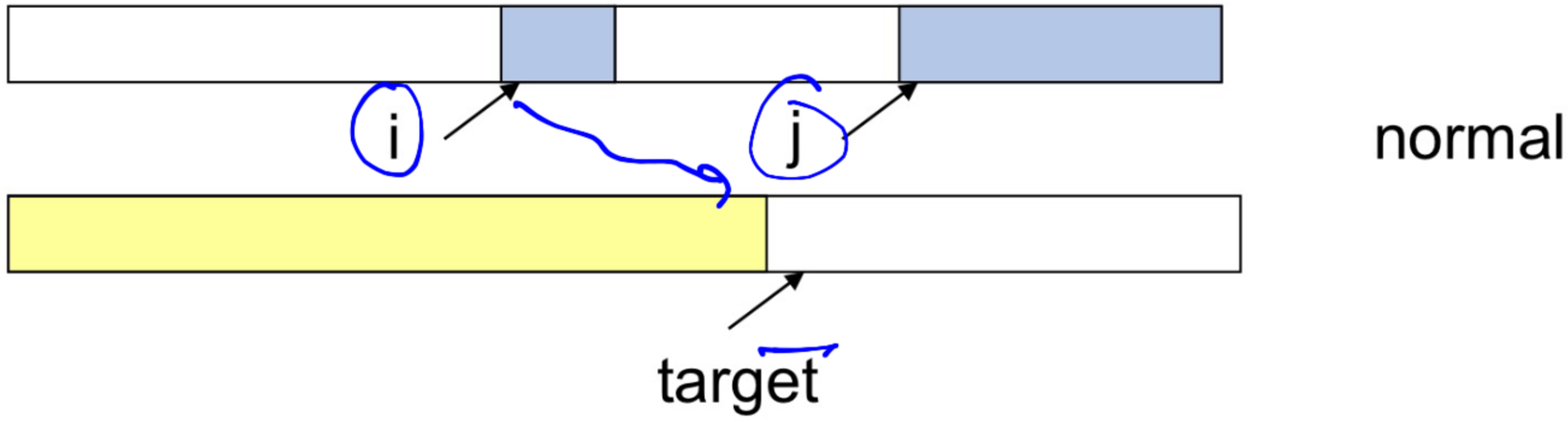
Merging



Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

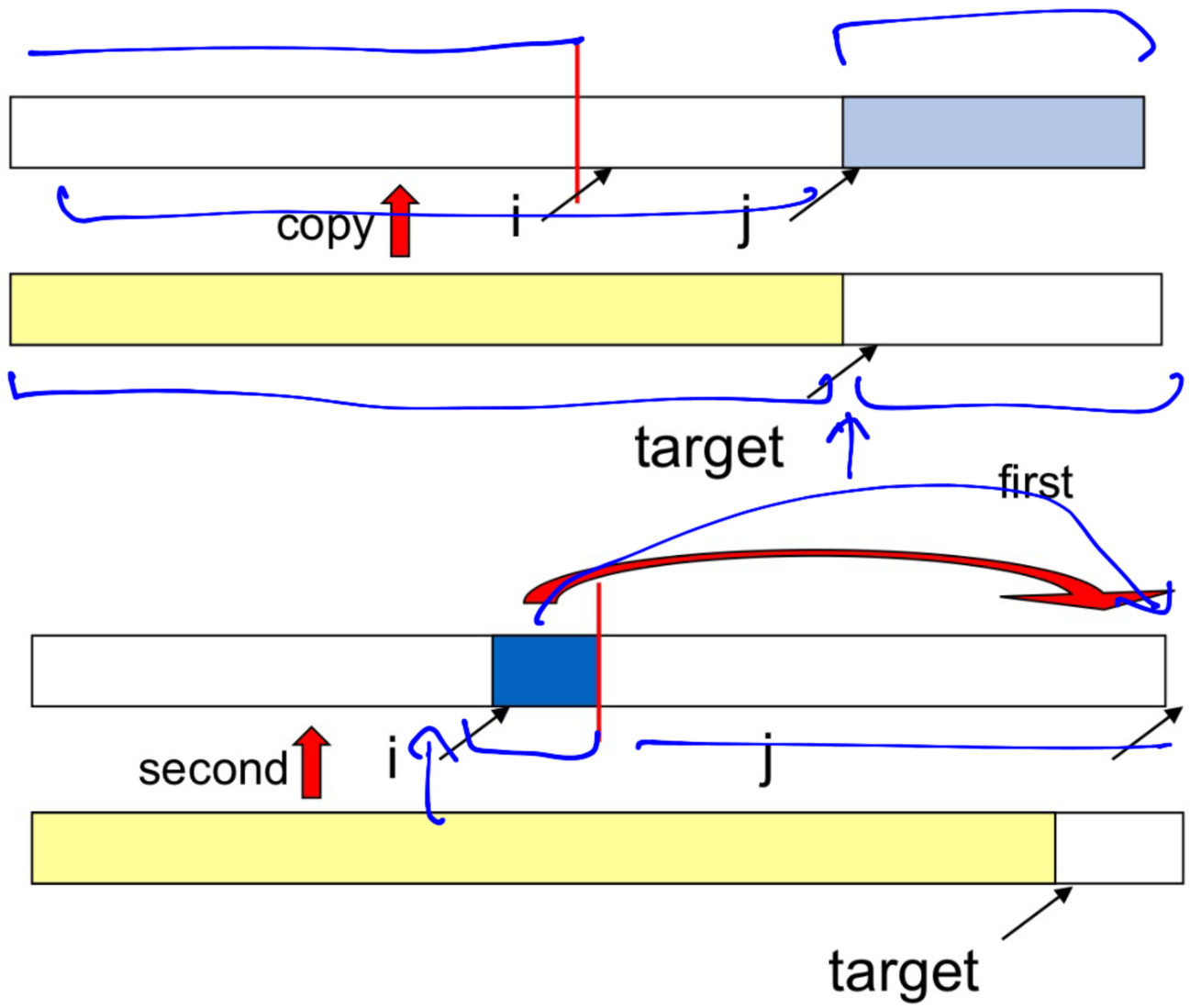
Merging

- Maintaining three points



Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Merging



Case 1: Left completed first

Case 2: Right completed first

Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Merging

```
merge(int *A, int left, int mid, int right, int *T) {
    // Merges the sorted arrays A[left..mid-1] and
    // A[mid..right-1] using the into the auxiliary
    // array T. It is assumed that T has capacity
    // to store right-left elements.
    int i = left, j = mid, d = 0;

    while (i < mid && j < right) {
        // Current elements are A[i], B[j]
        // Target is T[d]
        if (A[i] < A[j])
            T[d++] = A[i++];
        else
            T[d++] = A[j++];
    }
```

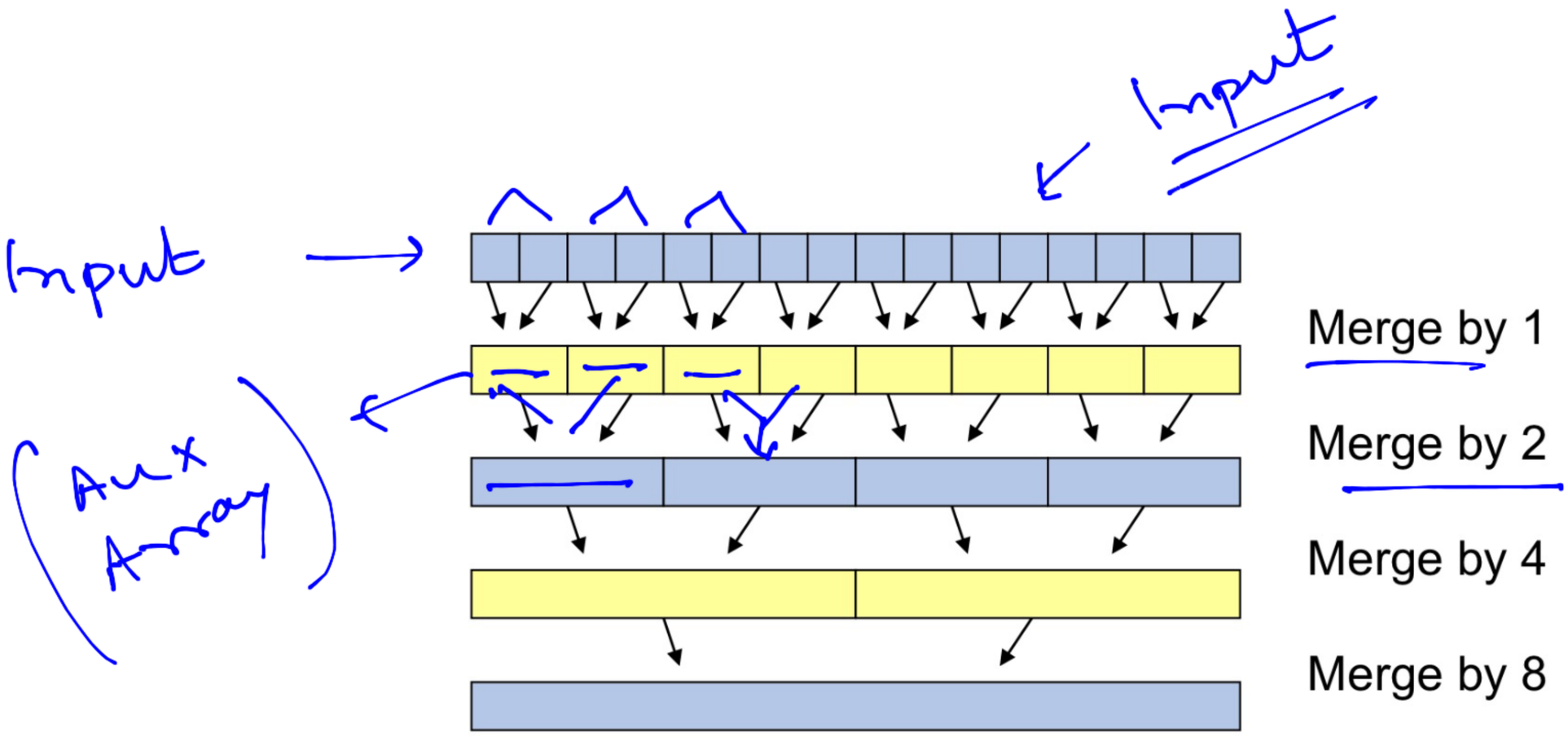
Merging

```
while (i < mid && j < right) {
    // Current elements are A[i], A[j]
    // Target is T[d]
    if (A[i] < A[j])
        T[d++] = A[i++];
    else
        T[d++] = A[j++];
}
if (i == mid) { // Left array completed first
    for (i = left; i < j; i++) A[i] = T[i-left];
} else { // Right array completed first
    for (k = mid-1, l = right-1; k >= i;  k--, l--)
        A[l] = A[k];
    for (i = left; i <= l; i++)
        A[i] = T[i - left];
}
```

Merge Sort

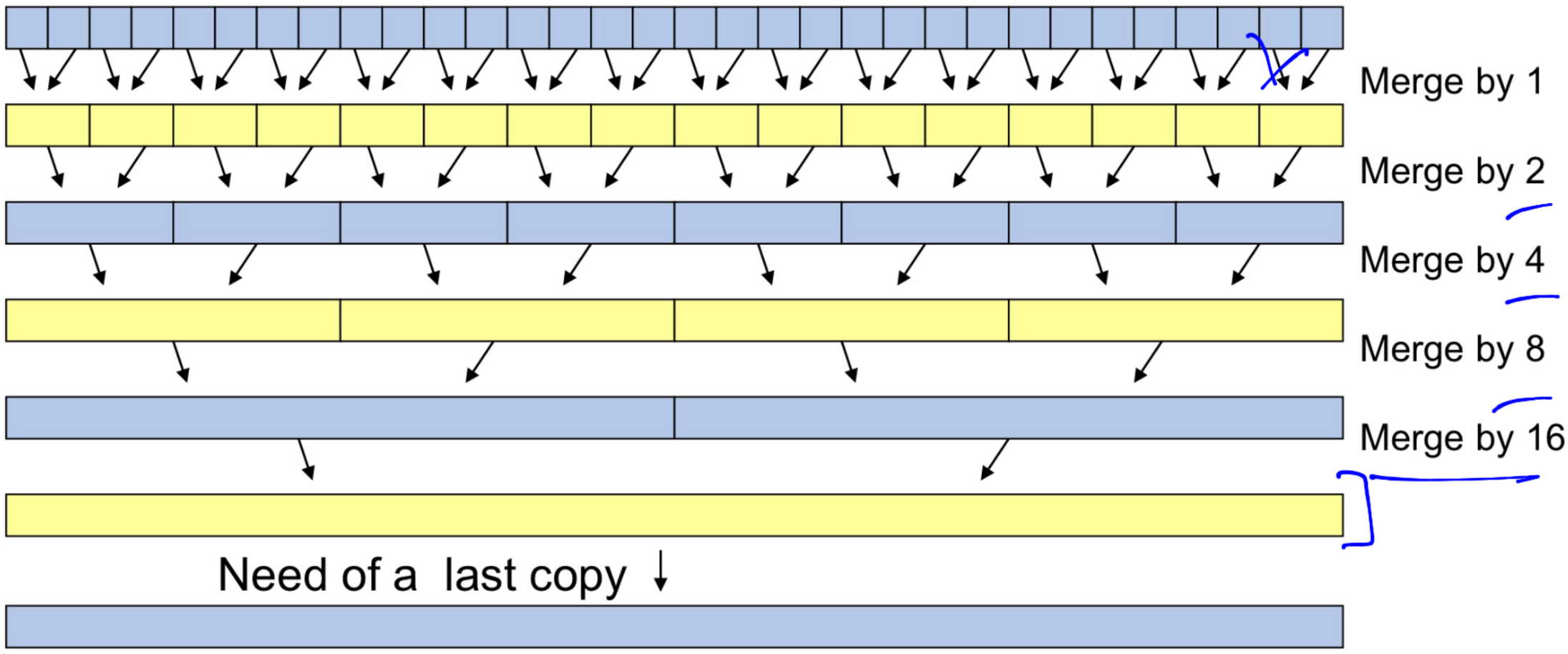
```
mergeSort(int *A, int sa, int *T) {
    // Sorts the array A of size sa using auxiliary
    // array T. Assumes T can store sa elements
    int mid = sa/2;
    if (sa == 1) return;

    mergeSort(A, mid, T); // Sort A[0] to A[mid-1]
    mergeSort(&A[mid], sa - mid, T); // Sort A[mid..sa-mid-1]
    merge(A, 0, mid, sa, T);
}
```

Iterative Merge Sort



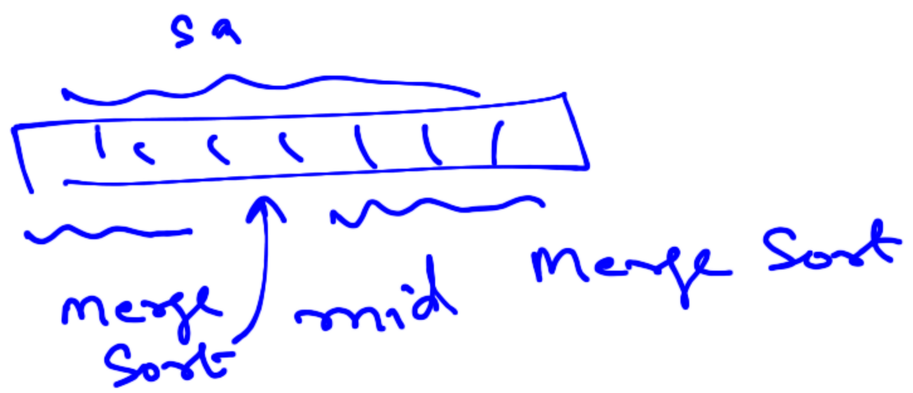
Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Iterative Merge Sort



Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Merge Sort Complexity Analysis



T(N)

```
mergeSort(int *A, int sa, int *T) {
    // Sorts the array A of size sa using auxiliary
    // array T. Assumes T can store sa elements
    int mid = sa/2;
    if (sa == 1) return;

    mergeSort(A, mid, T); // Sort A[0] to A[mid-1]
    mergeSort(&A[mid], sa - mid, T); // Sort A[mid..sa-mid-1]
    merge(A, 0, mid, sa, T);
}
```

const

T(N/2)

T(N/2)

c N

Merge Sort Complexity Analysis

$T(\lfloor N/2 \rfloor) + T(N - \lfloor N/2 \rfloor)$

$T(N) = T(N/2) + T(N/2) + cN$

$T(1) = b$

$T(N) = 2\,T(N/2) + cN$

$= 4\,T(N/4) + cN + cN$

$= N\,T(1) + cN + cN + \ldots + cN$

$\log_2(N)$ times

$= O(N \log(N))$

Merge Sort Properties

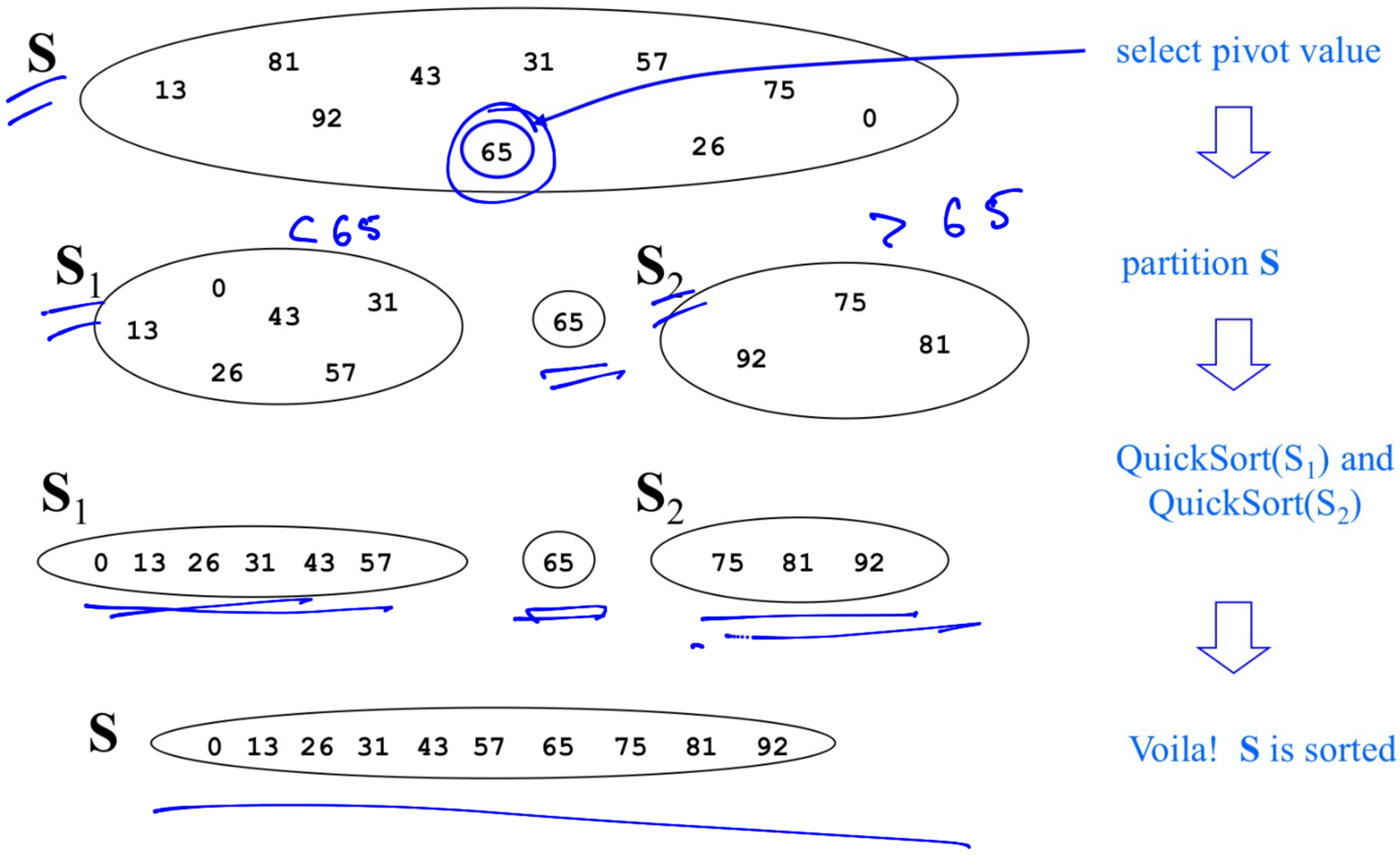
- Need extra array to store copies
- Too much copying
- Iterative merge sort reduces copying

Quick Sort

Quick Sort

- Divide and conquer
- Partition the array into two sub-arrays using a pivot
  - Left sub-array contains smaller elements
  - Right sub-array contains larger elements
- Recursively sort the two sub-arrays
- No need to merge the two sub-arrays
  - Just concatenate them one after the other
- No extra memory or extra copying needed

Quick Sort Steps



Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Quick Sort Algorithm

Input: An array S of numbers

Output: An array containing S in ascending order

Algorithm:

1. If the number of elements in **S** is 0 or 1, then return S. The array is sorted.
2. Pick a pivot element *v* in **S**.
3. Partition **S**-{*v*} into two disjoint subsets
   - $\mathbf{S}_1$ = {all values $x < v$}, and
   - $\mathbf{S}_2$ = {all values $x > v$}.
4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Quick Sort Analysis

T(N)

const

Algorithm Quicksort(S)

1. If the number of elements in **S** is 0 or 1, then return S. The array is sorted.
2. Pick a pivot element *v* in **S**. — const
3. Partition **S**-{*v*} into two disjoint subsets — c N
   - $\mathbf{S}_1$ = {all values $x \leq v$}, and
   - $\mathbf{S}_2$ = {all values $x \geq v$}.
4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

K

$T(|S_1|)$ $T(|S_2|)$

$T(N) = cN + T(K) + T(N-K-1)$

Slide adapted from: Jean-Loup Baer, Department of Computer Science, University of Wisconsin

Quick Sort Analysis

$N_1 = |S_1|$

$T(N) = cN + T(N_1) + T(N-N_1-1)$

$T(1) = b$

What if we can pick a pivot such that $|S_1| = N/2$?

$T(N) = cN + T(N/2) + T(N/2-1)$

$T(N) \leq 2T(N/2) + cN$

$T(N) \leq O(N \log(N))$

Quick Sort Analysis

$T(N) = cN + T(N_1) + T(N-N_1-1)$

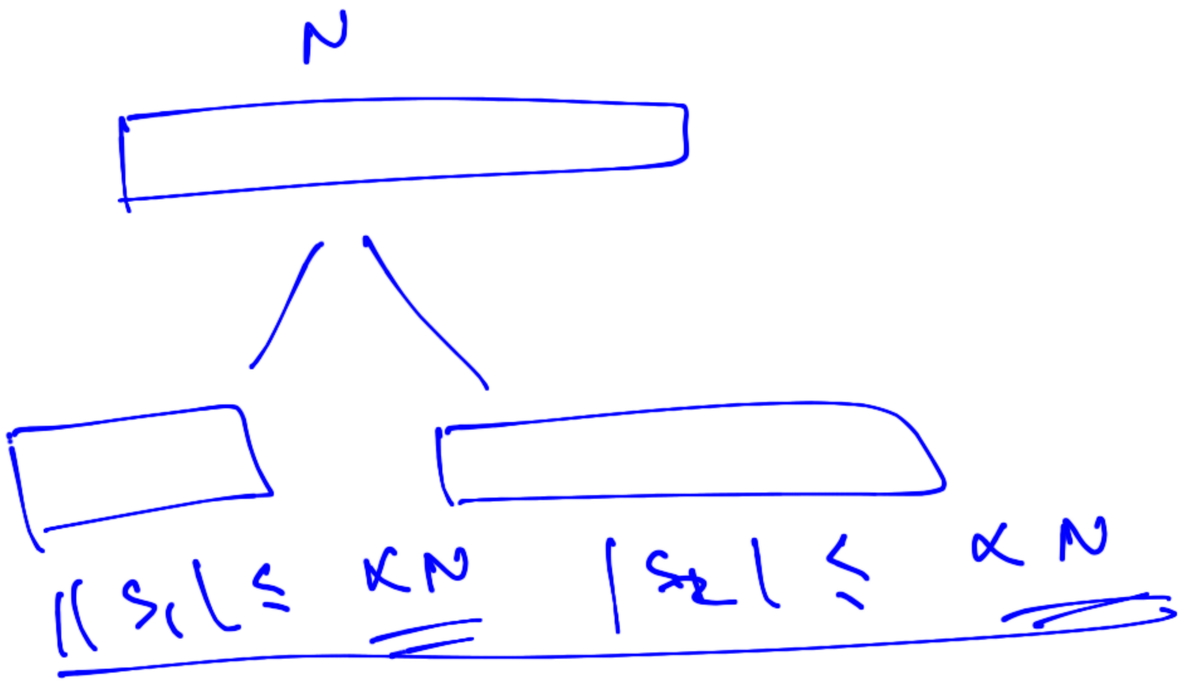
$T(1) = b$

More general case

Quick Sort Analysis

$T(N) = cN + T(N_1) + T(N-N_1-1); T(1) = b$

What if we can always pick a pivot such that $|S_1| \leq \alpha N$ and $|S_2| \leq \alpha N$ for some α in (0.5, 1)?



$(0.5, 1)$

open interval

0.5, 1 are excluded

Quick Sort Analysis

$T(N) = cN + T(N_1) + T(N-N_1-1);\ T(1) = b$

What if we can always pick a pivot such that $|S_1| \leq \alpha N$ and $|S_2| \leq \alpha N$ for some α in (0.5, 1)?

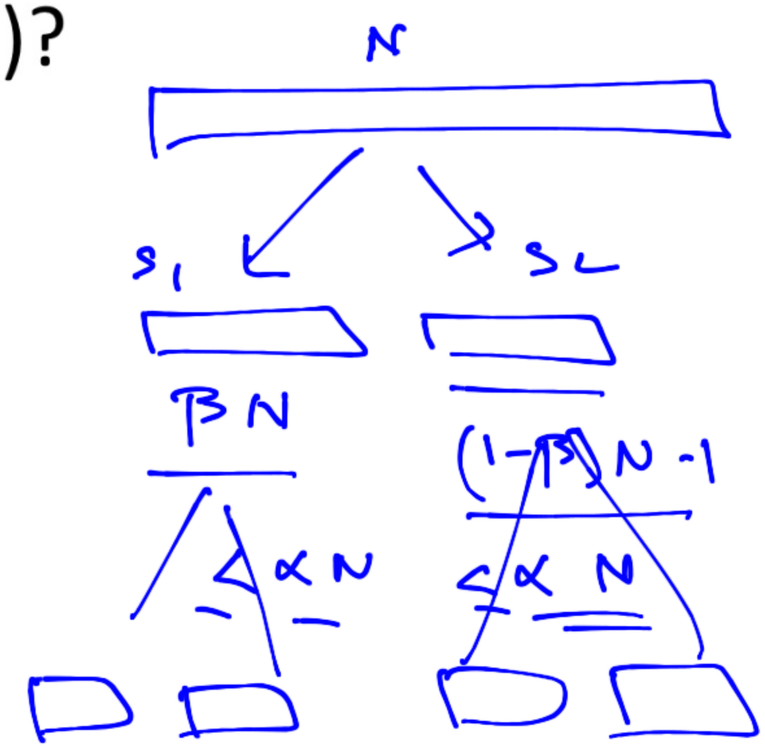
$T(N) = cN + T(|S_1|) + T(N - |S_1| - 1)$

Let $\beta = |S_1|/N$ $\quad \beta < \alpha$

$T(N) \leq cN + T(\beta N) + T((1-\beta)N)$

$T(N) \leq cN +$

$c\beta N + T(\beta' \beta N) + T((1-\beta')\beta N)$



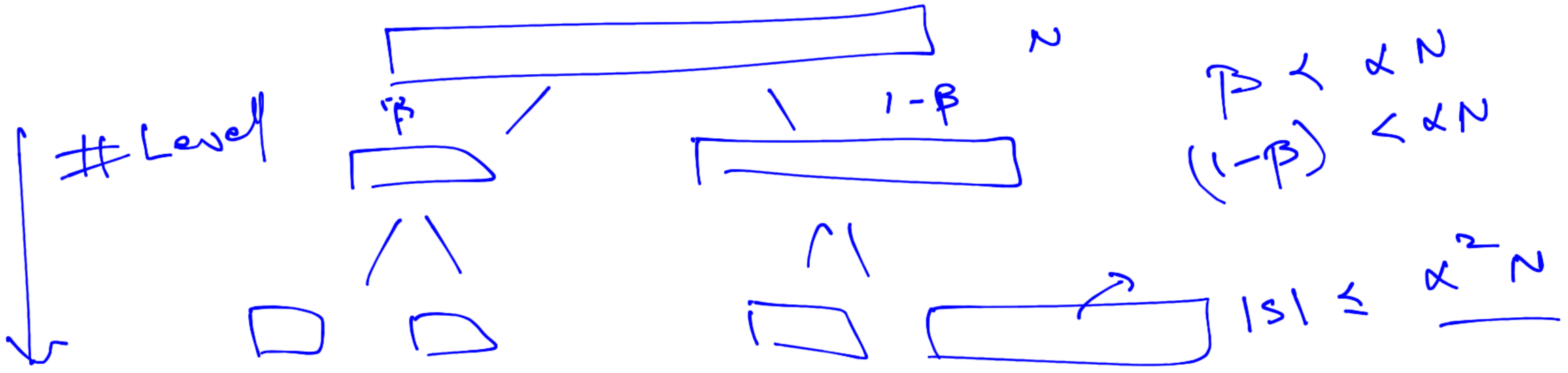
$\leq \frac{c(1-\beta)N}{} + T((1-\beta)\cdot\beta'' N) + T((1-\beta)(1-\beta'' N))$

$\leq cN + cN + \sum_{\alpha=1}^{4} T(\alpha_i N)$ $\quad \sum_{i=1}^{4} \alpha_i = 1$

$= cN + cN + cN + \sum_{k=1}^{8} T(\alpha_k' N)$

Quick Sort Analysis: K Levels of Recursion

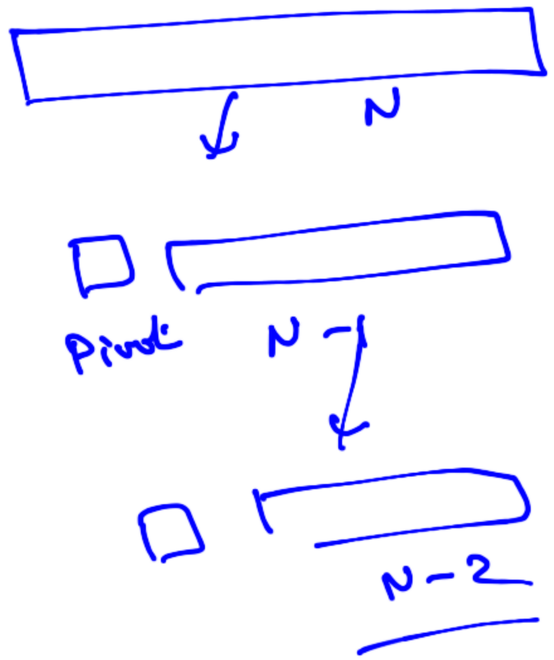
Assume: $|S_1| \leq \alpha N$ and $|S_2| \leq \alpha N$ for some α in (0.5, 1)



Quick Sort Analysis: Worst Case

Algorithm always chooses the worst pivot – one sub-array is empty at each recursion

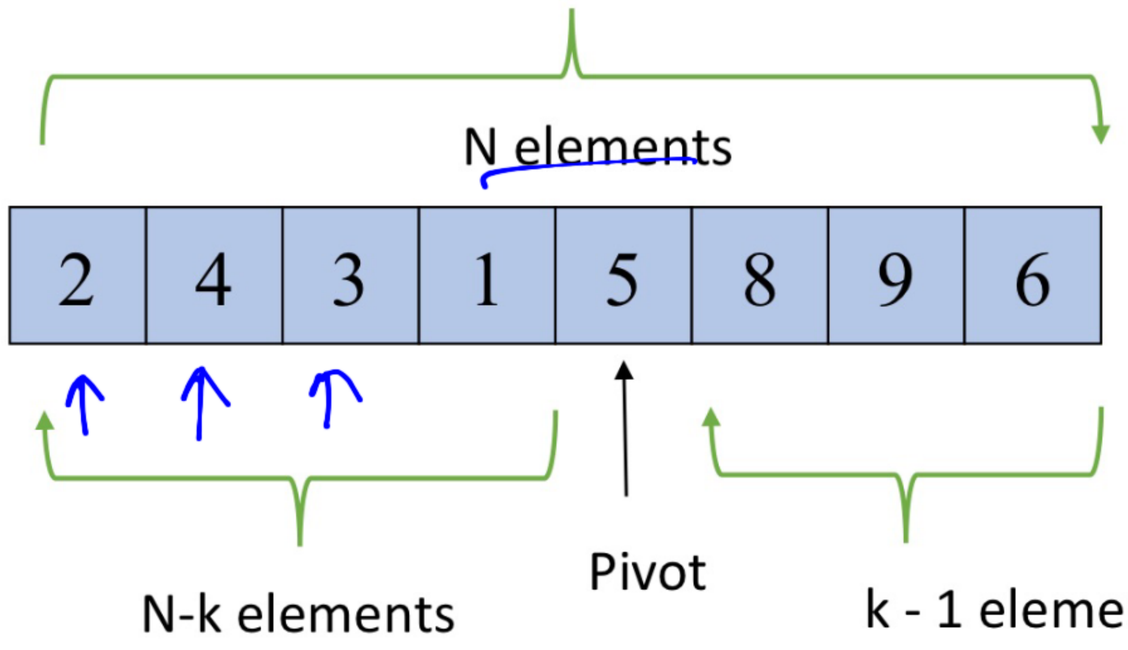
- T(N) = b for N = 1
- T(N) = cN + T(N-1) + T(0)
- = cN + c(N-1) + T(N-2)
- ≤ cN + c(N-1) + c(N-3) + … c(1) + T(1)
- ≤ b +c(1+2+…+N)
- T(N) = $O(N^2)$



Quick Sort Analysis

- Key to good performance is to pick a good pivot
- Balance the two sets S_1 and S_2
- Some pivoting strategies
  - First element
  - Last element
  - Middle element
  - Median: How to find median without sorting in linear time
  - Random element of the array – Average case complexity O(N log(N))

Quick Sort with Random Pivot



If all the elements are unique and the pivot is chosen randomly, then all possibilities for k = 1 to N are equally likely

$$T(N) = \sum_{k=1}^{N} \frac{1}{N}[T(N-k) + T(K-1) + cN]$$

T(1) = b; T(0) = 0;

Randomized Quick Sort Analysis

$$T(N) = \sum_{k=1}^{N} \frac{1}{N} [T(N-k) + T(K-1) + cN]$$

T(N) is the expected time to sort an input of size N

Expectation is taken over all possible random numbers generated by the program

T(1) = b; T(0) = 0;

Analysis

$$T(N) = \sum_{k=1}^{N} \frac{1}{N}[T(N-k) + T(K-1) + cN]$$

$$T(N) = \frac{2}{N}\sum_{k=0}^{N-1} T(K) + cN \qquad (1)$$

$$T(N-1) = \frac{2}{N-1}\sum_{k=0}^{N-2} T(K) + c(N-1)$$

$$\frac{N-1}{N}T(N-1) = \frac{2}{N}\sum_{k=0}^{N-2} T(K) + c\frac{(N-1)^2}{N} \qquad (2)$$

Eq (1) – Eq(2) gives: $T(N) = \frac{N+1}{N}T(N-1) + c\frac{2N-1}{N}$

Analysis

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

$$T(N) \leq \frac{N+1}{N} T(N-1) + 2c$$

$$T(N) \leq \frac{N+1}{N-1} T(N-2) + 2c\left(\frac{n+1}{n} + 1\right)$$

$$T(N) \leq \frac{N+1}{N-2} T(N-3) + 2c\left(\frac{n+1}{n-1} + \frac{n+1}{n} + 1\right)$$

$$\leq \frac{N+1}{2} T(1) + 2c(n+1)\left(\frac{1}{3} + \frac{1}{4} + \cdots \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}\right)$$

Bounded by log(N)

T(N) = O(N log(N))

More Details

- How to pick a pivot?
- How to partition without using extra memory?
- How to deal with case when many elements are equal to the pivot?

More Details

- How to pick a pivot?
  - Random element of the array
- How to partition without using extra memory?
  - Think
- How to deal with case when many elements are equal to the pivot?
  - Relax strict inequality

Quick Sort Algorithm

Input: An array S of numbers

Output: An array containing S in ascending order

Algorithm:

1. If the number of elements in **S** is 0 or 1, then return S. The array is sorted.
2. Pick a random pivot element *v* from **S**
3. Partition **S**-{*v*} into two disjoint subsets
   - $\mathbf{S}_1$ = {all values ~~$x < v$~~ $x \leq v$}, and
   - $\mathbf{S}_2$ = {all values ~~$x > v$~~ $x \geq v$}.
4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

In-Place Partitioning

- Pick a pivot
- Swap pivot with last element (A[N-1])
- Set pointers i and j to start and end (-1) of array
- Increment i until you hit element A[i] > pivot
- Decrement j until you hit element A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross
- Swap pivot (at A[N-1]) with A[i]

Example

Suppose 6 is picked as pivot

8	1	4	9	**6**	3	5	2	7	0

Swap pivot with the last element

8	1	4	9	**0**	3	5	2	7	**6**

Initialize pointers i, j

i

j

8	1	4	9	**0**	3	5	2	7	**6**

Now swap A[i] and A[j]

2	1	4	9	**0**	3	5	8	7	**6**

Move pointers i and j

2	1	4	9	**0**	3	5	8	7	**6**

2	1	4	5	**0**	3	9	8	7	**6**

Iteration stops when i and j cross. Now swap A[i] with pivot (A[N-1])

2	1	4	5	**0**	3	9	8	7	**6**

j **i**

Putting it Together

```
void Qsort(int A[], int left, int right) { // In place Quick-Sort
    // Quick sort elements in A[left..right] in ascending
    int pivot, pi;

    pi = random(left, right); // Pick a random element index to pivot
    pivot = A[pi];
    swap(A[pi], A[right]);
    pi = partition(A, left, right); // Partition the array A
    // Using the element A[right] as the pivot
    // Returns the index of pivot
    // S1 is A[left..pi-1]; S2 is A[pi+1..right]
    Qsort(A, left, pi-1);
    Qsort(A, pi+1, right);
}
```

Partition

```
int partition(int A[], int left, int right) {
     // Parititions A[left..right-1] using the pivot A[right]
     // Returns i such that A[i] is pivot and
     // For all j < i A[j] <= A[i] and
     // For all j > i A[j] >= A[i]
     int i = left, j = right-1;
     int pivot = A[right];

     do {
          while (A[i] <= pivot) i++; // TBD: Handle when i>right
          while (A[j] >= pivot) j--; // TBD: Handle when j < 0
          if (i < j) swap(A[i], A[j]);
     } while (i < j);
     swap(A[i], A[right]);
     return i;
}
```

Homework

- Work out iterative version of merge sort
- Work out iterative version of quick sort
  - Hint: You may need to use stacks for iterative version