Sorting Algorithms

Introduction

Sorting Problem

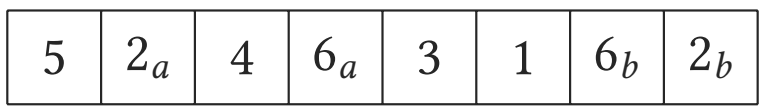
Sorting Problem

Given a sequence $A = \langle a_1, \ldots, a_n \rangle$, find a permutation $\langle a'_1, \ldots, a'_n \rangle$ of A such that $a'_1 \leq \ldots \leq a'_n$.

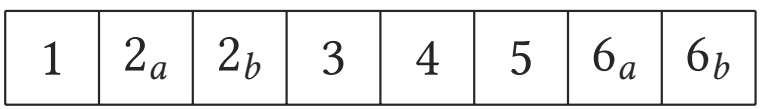
Stable Sorting

Stable Sorting

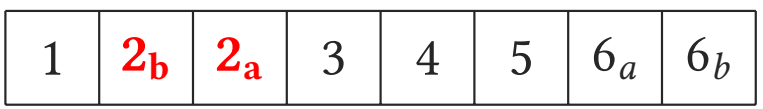
A sorting algorithm is *stable*, if the relative ordering of equal elements is preserved.



Input



Stable



Unstable

In-Place Sorting

In-Place Sorting

A sorting algorithm is *in-place*, if it requires only $\mathcal{O}(1)$ additional memory.

Notes

- Sometimes $\mathcal{O}(\log n)$ is also accepted as in-place.
- Counting requires $\mathcal{O}(\log n)$ space.
- In this class: A word ($\mathcal{O}(\log n)$ bits) requires $\mathcal{O}(1)$ space.

Recursive Calls

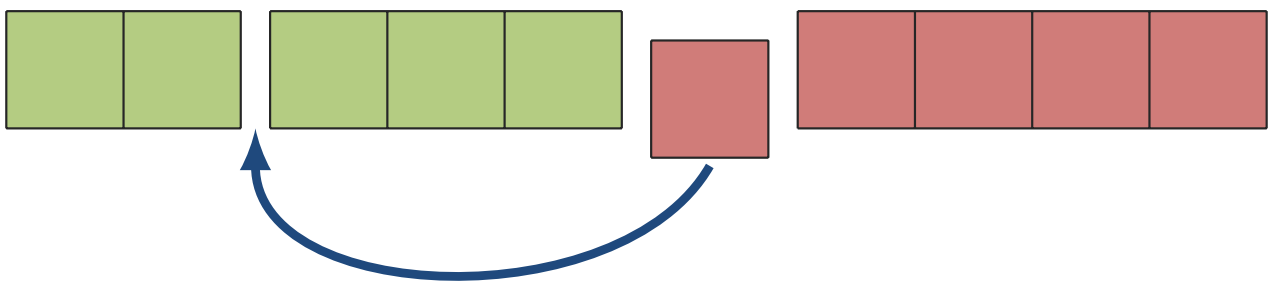
- Calling a function requires space!
- Local variables and return address are stored on the system stack.

Insertion Sort

Insertion Sort

Idea

- Sorted and unsorted part of data.
- Pick an element from the unsorted part.
- Insert it at the correct place in the sorted part.



Insertion Sort

Input: An array A with size n.

1 **For** $i = 1$ **To** $n - 1$
2 Set $j := i - 1$
3 **While** $j \geq 0$ and $A[j] > A[j+1]$
4 Swap $A[j]$ and $A[j+1]$.
5 $j := j - 1$

Properties

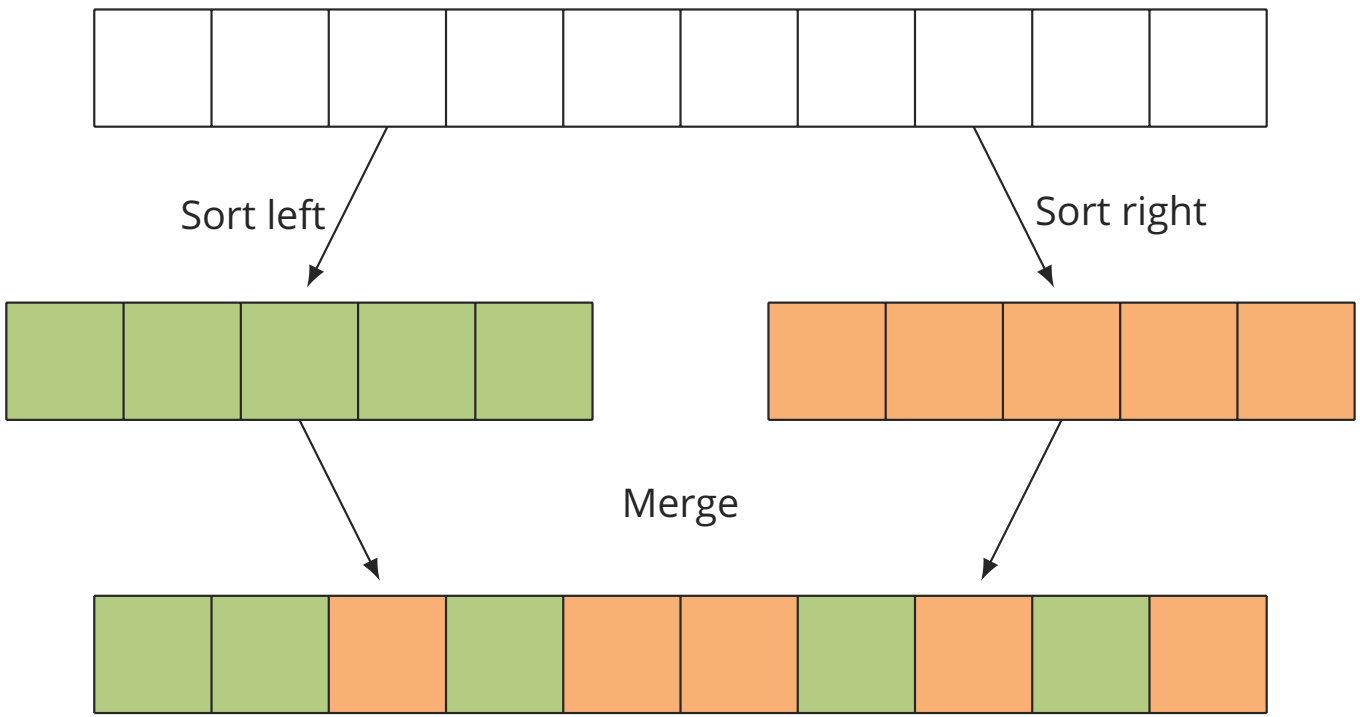
- In-place, Stable
- $\mathcal{O}(n^2)$ comparisons, $\mathcal{O}(n^2)$ swaps
- Close to linear if nearly completely sorted.

Merge Sort

Merge Sort

Idea

- Split data in two parts.
- Sort each part.
- Merge sorted parts together.



Merge Sort

Input: An array A of size n and two indices start and end with

$$0 \leq \text{start}, \text{end} < n.$$

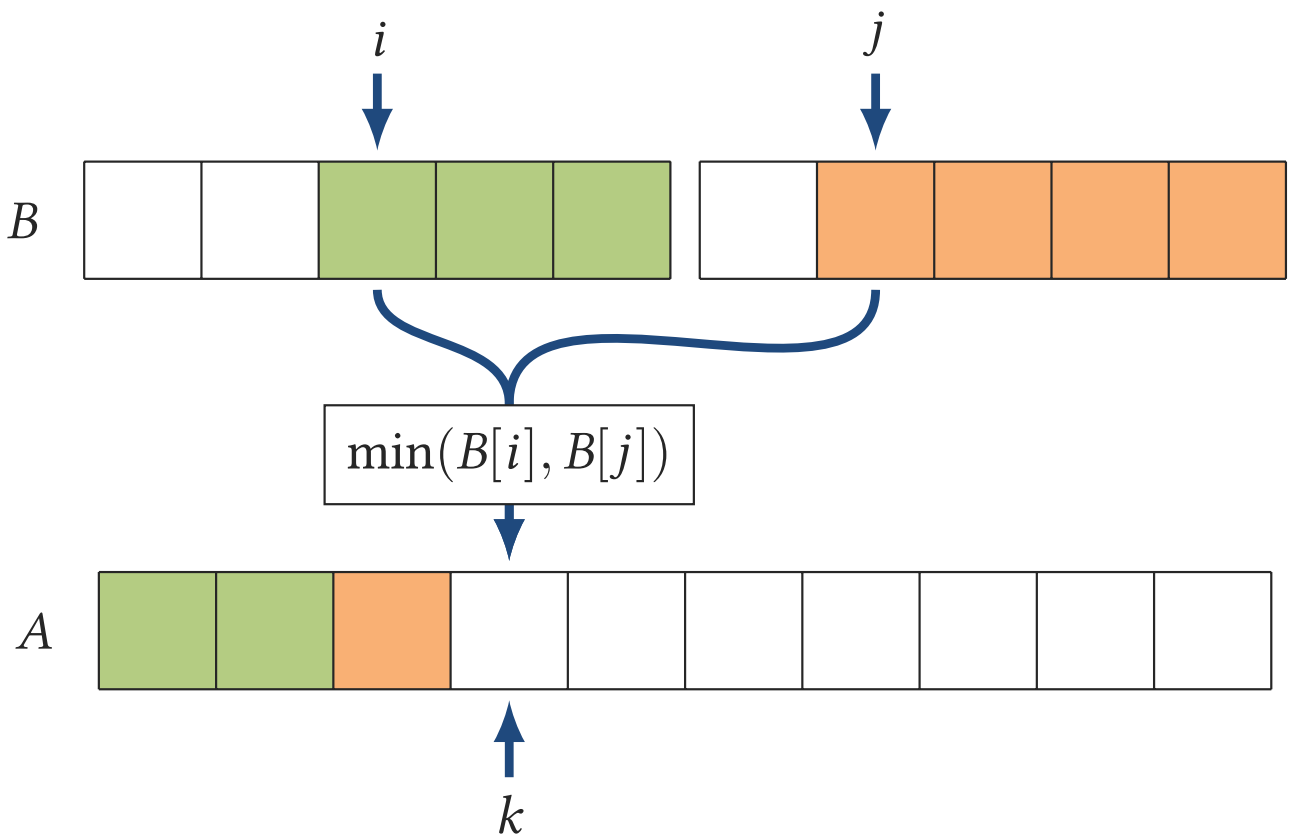
```
1 Procedure MergeSort (A, start, end)
2     If start < end Then
3         mid := start + ⌊(end − start)/2⌋
4         MergeSort(A, start, mid)
5         MergeSort(A, mid + 1, end)
6         Merge(A, start, mid, end)
```

Question: How do we merge?

Merging

Idea

- Write left and right part into array B.
- Step by step write minimum value from B back into A.



Merging

```
1 Procedure Merge (A, start, mid, end)
2     Set i := 0, m := mid − start, and k := start.
3     Copy A[start] to A[end] into an array B.
4     For j := m + 1 To end − 1 − start
5         While i ≤ m and B[i] ≤ B[j]
6             Set A[k] := B[i], i := i + 1, and k := k + 1.
7         Set A[k] := B[j] and k := k + 1.
8     While i ≤ m
9         Set A[k] := B[i], i := i + 1, and k := k + 1.
```

Merge Sort

Runtime

- Each call of *MergeSort* calls itself two times for half the array.
- Merging costs $\Theta(n)$ time.
- Thus, $T(n) = 2T\left(\frac{n}{2}\right) + n$, i. e., $T(n) \in \Theta(n \log n)$

Other Properties

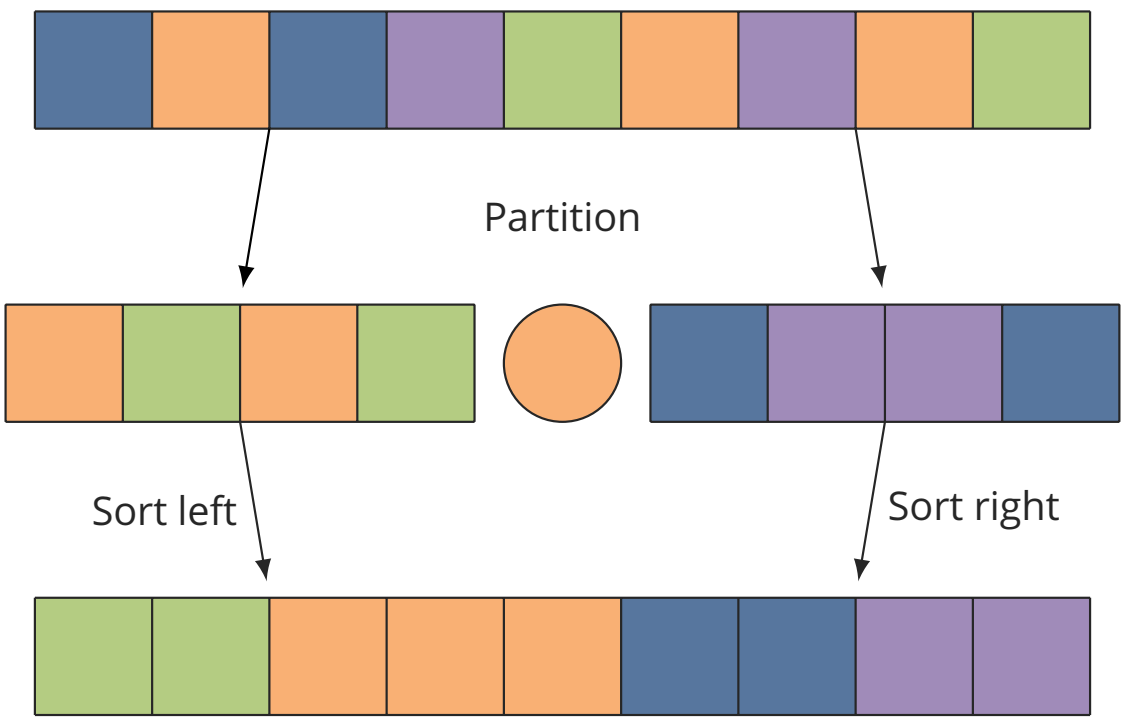
- Not in-place, $\mathcal{O}(n)$ extra memory required
- Stable (depending on merge implementation)
- $\mathcal{O}(n \log n)$ swaps.

Quicksort

Quicksort

Idea

- Pick pivot element p.
- Partition data in two subsets: elements smaller than p and elements larger than p.
- Sort each part.



Quicksort

Input: An array A of size n and two indices start and end with

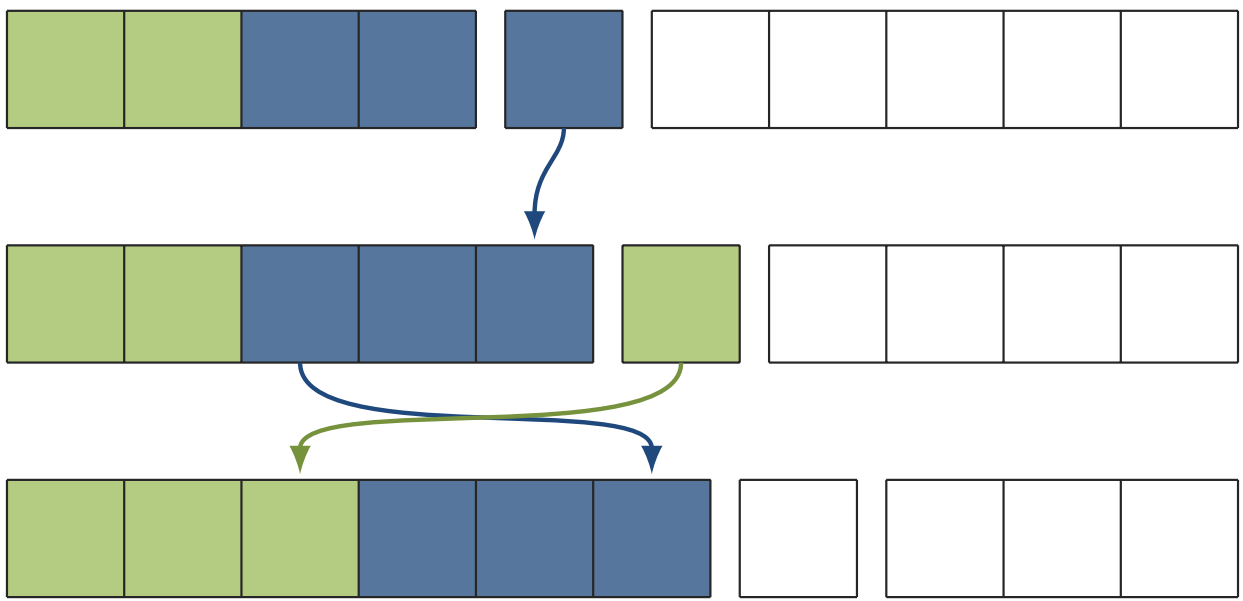
$$0 \leq \text{start}, \text{end} < n.$$

```
1 Procedure Quicksort (A, start, end)
2     If start < end Then
3         mid := Partition(A, start, end)
4         Quicksort(A, start, mid − 1)
5         Quicksort(A, mid, end)
```

Partition

Strategy 1

- Iterate from left to right
- Grow smaller and larger partition from the left.
- If element larger than p, nothing to do.
- If element is smaller than p, exchange with leftmost larger element.



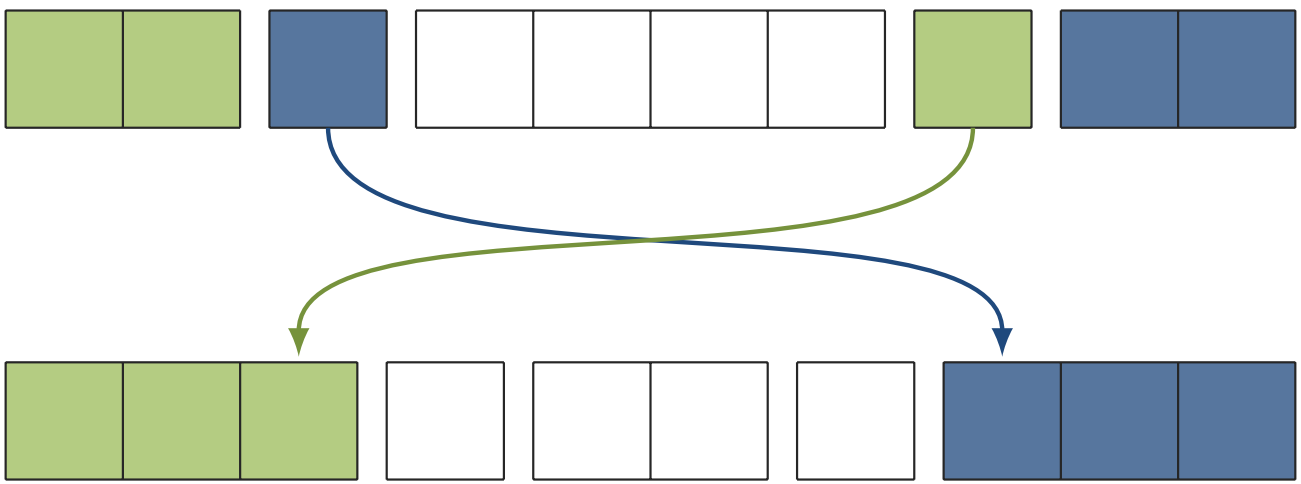
Partition

```
1 Procedure Partition (A, start, end)
2     Pick a pivot-element p and swap it with A[end].
3     Set j := 0.
4     For i := 0 To end − 1
5         If A[i] < p Then
6             Swap A[j] and A[i].
7             Set j := j + 1.
8     Swap A[j] and A[end].
9     Return j
```

Partition

Strategy 2

- Explore data from both sides.
- Grow smaller part from the left and larger part from the right.



Partition

Strategy 3 (Dutch National Flag Problem)

- Partition into three groups: smaller, larger, and equal to p.
- Useful if there are multiple equal elements.
- Grow smaller/equal part from the left and larger part from the right.

Selecting Pivot Element

Ideal Case

- p splits data in two equally large parts.
- Doable in linear time, but constant factor is too large.

Bad Strategy

- Select first or last element.
  (leads to $\mathcal{O}(n^2)$ time if already sorted)

Better Strategies

- Select middle element.
- Select random element.
- Select median of three elements.

Runtime and Properties

Runtime

- Worst case: $\mathcal{O}(n^2)$
- Best case: $\mathcal{O}(n \log n)$
- Average case: $\mathcal{O}(n \log n)$
- Usually faster than other sorting methods (e. g. merge sort).

Other Properties

- Not in-place, *in average* $\mathcal{O}(\log n)$ extra space required (recursive calls)
  (if implemented correctly, $\mathcal{O}(\log n)$ extra space *in worst case*)
- Not stable

Exercises

Exercises

Outline a reasonable method of solving each of the following problems. Give the order of the worst-case complexity of your methods.

(a) You are given a pile of thousands of telephone bills and thousands of checks sent in to pay the bills. Find out who did not pay.

(b) You are given a list containing the title, author, call number and publisher of all the books in a school library and another list of 30 publishers. Find out how many of the books in the library were published by each company.

(c) You are given all the book checkout cards used in the campus library during the past year, each of which contains the name of the person who took out the book. Determine how many distinct people checked out at least one book.

Exercises

Use the partitioning idea of quicksort to give an algorithm that finds the median element of an array of n integers in expected $\mathcal{O}(n)$ time.