Sorting in Arrays

Sorting

Binary search works great but how do we create a sorted array in the first place?

Sorting algorithms:

- Selection sort: $O(n^2)$ time
- Merge sort: $O(nlog(n))$ time
- Quicksort: $O(n^2)$ time

SelectionSort

- Input: array of Comparables
- Output: same array sorted in ascending order
- Algorithm: assume N is size of array

  1. Examine all elements from 0 to (N-1), find the smallest one and swap it with the 0^{th} element of the input array.
  2. Examine all elements from 1 to (N-1), find the smallest one in that portion of the array and swap it with the 1^{st} element of the array.
  3. In general, at the i^{th} step, examine array elements from i to $(N-1)$, find the smallest one in that range, and exchange it with the i^{th} element of the array.

It is easy to show that selection sort requires *N*(N-1)/2* comparisons, if N is the length of the array to be sorted.

This is called an $O(N^2)$ algorithm since the leading term is N^2 (we also ignore constant coefficients in big-O notation).

Question: can we find a way to speed up selection sort?

Any time you have a $O(N^2)$ algorithm, it pays to break the problem into smaller subproblems, solve subproblems separately and assemble final solution.

Rough argument: suppose you break problem into k pieces. Each pieces takes $O((N/k)^2)$ time, so time for doing all k pieces is roughly $\frac{1}{k}O(N^2)$ time.

If we divide problem into two subproblems that can be solved independently, we can halve the running time!

Caveat: the partitioning and assembly process should not be expensive.

Can we apply this divide and conquer approach to sorting?

MergeSort

Quintessential divide-and-conquer algorithm

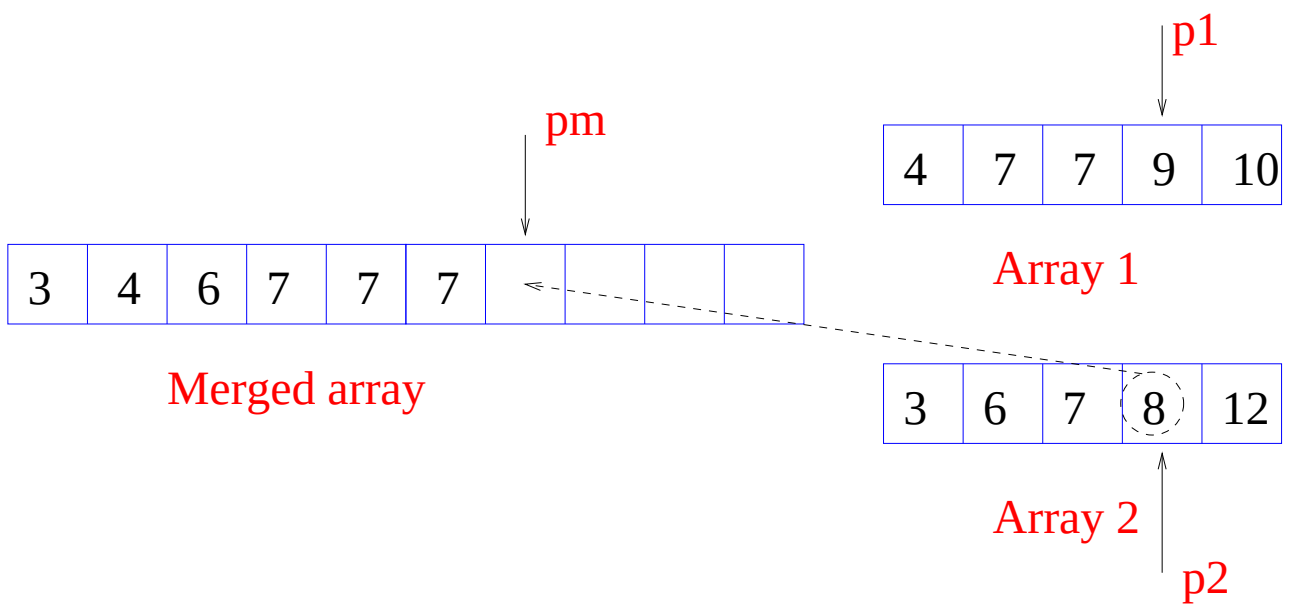
Divide array into two equal parts, sort each part and merge

Three questions:

- How do we divide array into two equal parts? Use indices into array
- How do we sort the parts? Call merge-sort recursively
- How do we merge sorted arrays? Let us write some code.

Merging sorted arrays a1 and a2

- Create an array m whose size = size of a1 + size of a2
- Keep three indices: p1 into a1, p2 into a2, pm into m, all initialized to 0.
- Compare first elements of a1 and a2, and move the smaller one (say it is from a2) into first element of m. Increment p2 and pr.
- In general, compare element a1[p1] with a2[p2] and move the smaller one into m[pm], incrementing the appropriate indices.
- If either a1 or a2 is emptied out, copy over all remaining elements of the other array into m.


p1
pm
4 7 7 9 10
Array 1
3 4 6 7 7 7
Merged array
3 6 7 8 12
Array 2
p2

Asymptotic complexity of merge sort: $O(n * log_2(n))$

Disadvantage of merge sort: need extra storage for temporary arrays

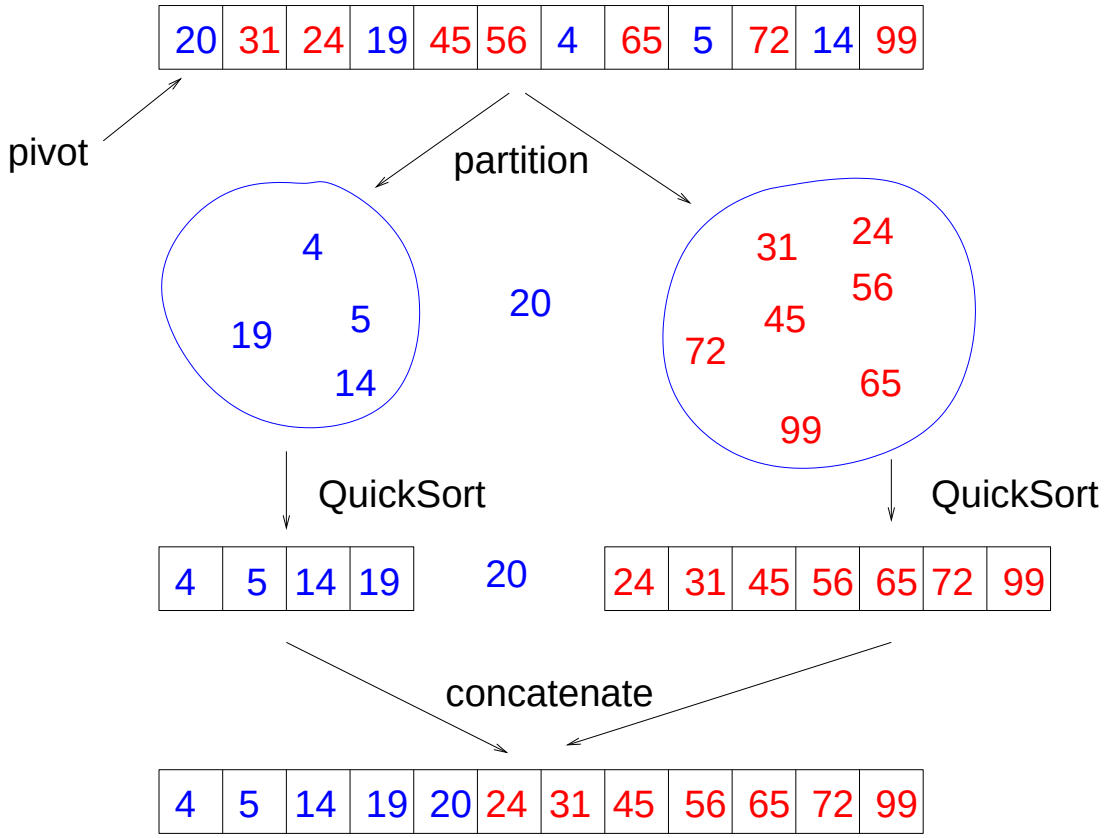
In practice, this can be a disadvantage even though merge sort is an *asymptotically optimal algorithms for sorting*.

Good sorting algorithm that does not use extra array storage: Quicksort

QuickSort

Intuitive idea:

- Given an array A and a *pivot value* v
- Partition array elements into two subarrays SX and SY
- SX contains only elements less than or equal to v
- SY contains only elements greater than v
- Sort SX and SY separately
- Concatenate sorted SX and sorted SY to produce result


20 31 24 19 45 56 4 65 5 72 14 99
pivot
partition
4
19
5
14
20
31
24
56
45
72
65
99
QuickSort
QuickSort
4 5 14 19
20
24 31 45 56 65 72 99
concatenate
4 5 14 19 20 24 31 45 56 65 72 99

Main advantage of QuickSort: sorting can be done *in-place* (no extra array needs to be created).

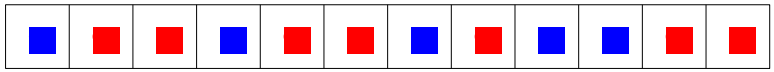
Key problem to be solved: how do we partition array in place?

If we can do this, we can have a Quicksort method of form

`void Quicksort(int[] A, int l, int h)` //sort values between l and h

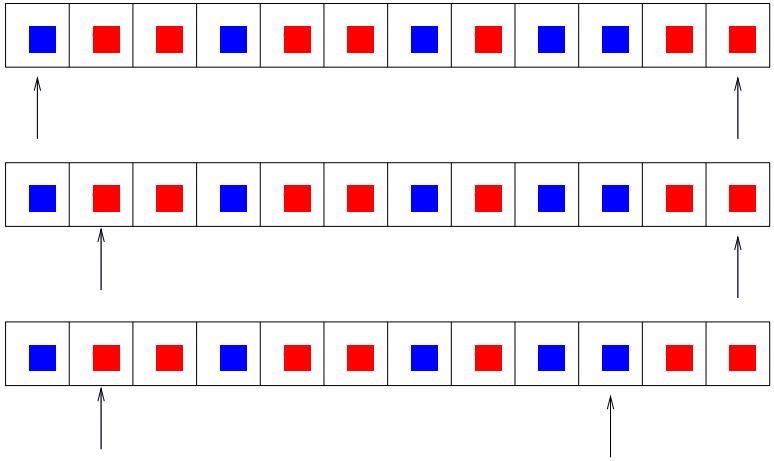
where `l` is lower bound of subarray to be sorted, and `h` is upperbound.

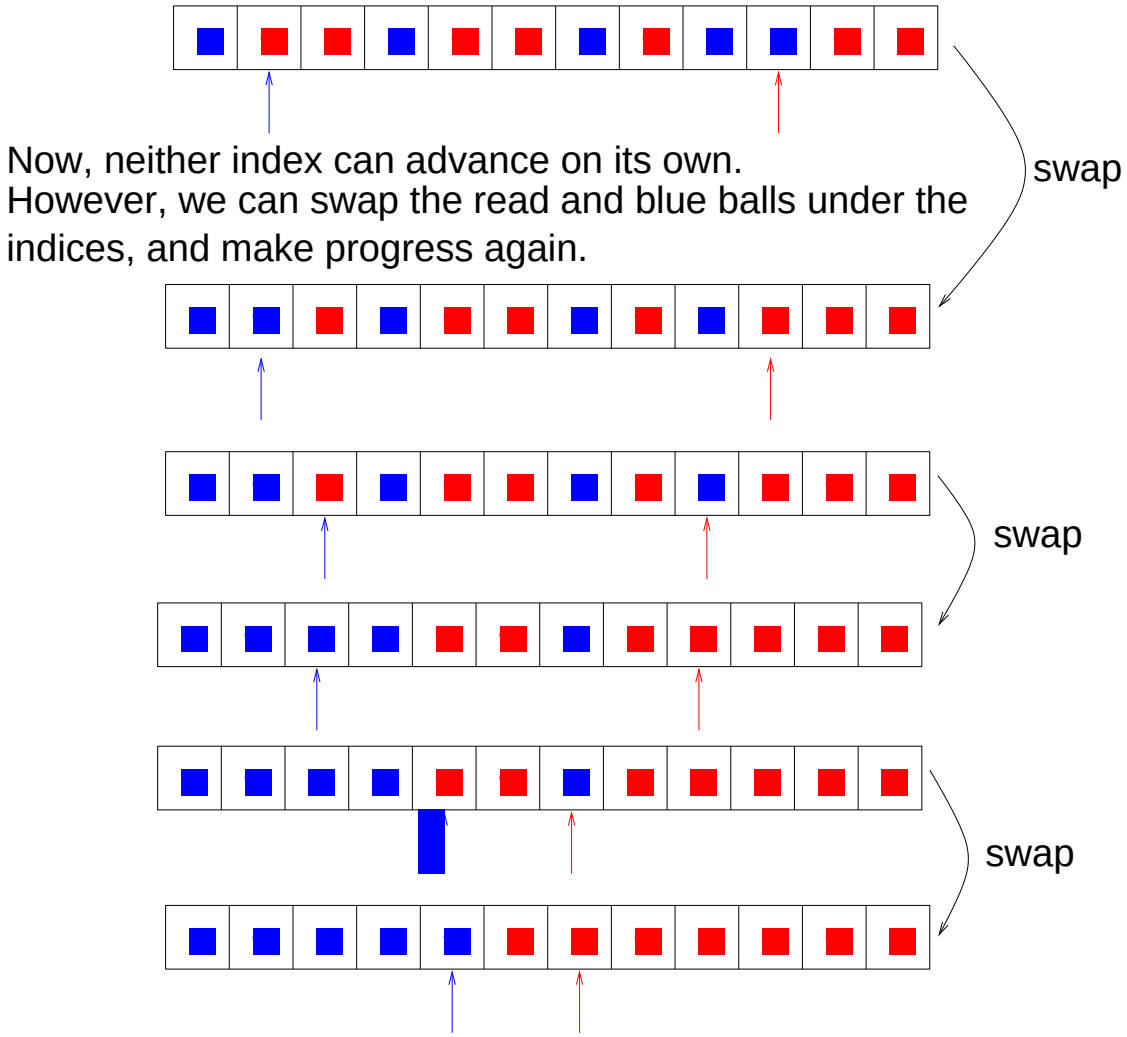
Inplace Partitioning

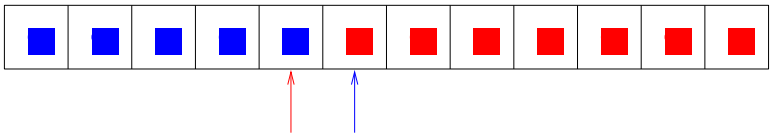


Can we get all blue balls to the left of all red balls?

Solution: keep two indices LEFT and RIGHT.
Invariant: all balls to left to LEFT index are certainly blue.
all balls to right of RIGHT index are certainly red.
Keep advancing the two indices till they cross.




Now, neither index can advance on its own.
However, we can swap the read and blue balls under the indices, and make progress again.
swap
swap
swap



Once indices cross over, partitioning is done.

What is analog of red/blue for quicksort?

All numbers less than pivot are blue.

All numbers greater than or equal to pivot are red.

Using this algorithm, we can partition in place!

Remaining question: what number do we choose for pivot?

Ideally, we would choose median value since that partitions array into two pieces of equal size.

However, median is expensive to compute exactly.

Heuristic: use first array value as pivot.

some people use middle element as pivot.

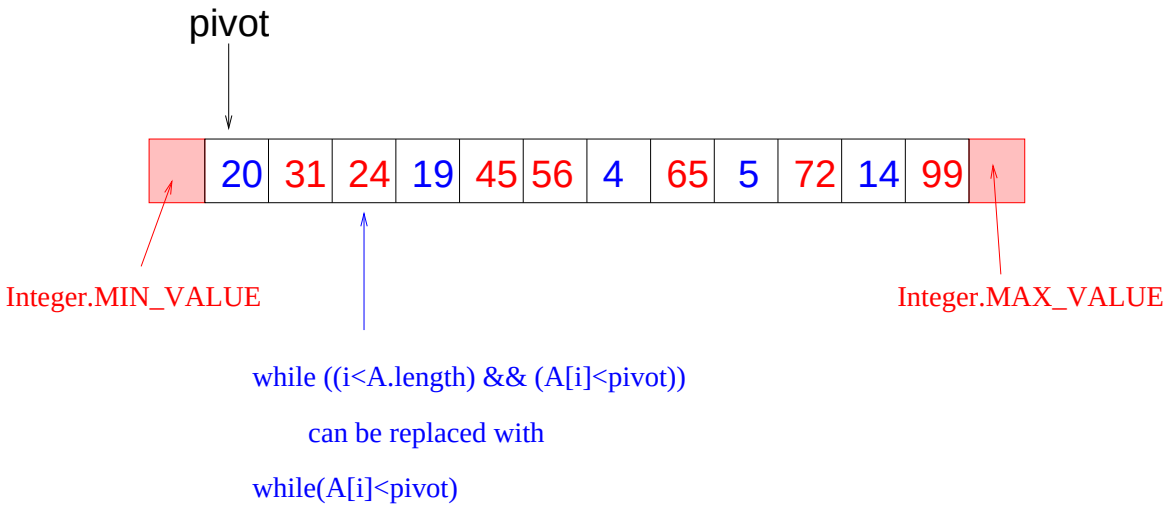
others use median of first, middle and last element

Notes on QuickSort code

Our Partition code is somewhat more inefficient than it needs to be.

Checks for *array out of bounds* can be done more efficiently with *sentinels*.

Sentinels



- Pad initial array with Integer.MIN_VALUE (-2147483648) and Integer.MAX_VALUE (2147483647)
- When advancing LEFT and RIGHT indices, Partition does not have to check that indices "fall off" the ends of the array.
- Recursive calls to Quicksort: sentinels are automatically there! (See picture below)

Summary

- Sorting methods discussed in lecture
  - Selection sort: $O(N^2)$
  - Merge sort: $O(Nlog(N))$ asymptotically optimal sorting method
  - Quicksort: $O(N^2)$ used in practice because it does not require extra storage for sorting
- There are many other sorting methods in the literature.
  - Heap sort (see later)
  - Shell sort
  - Bubble sort
  - Radix sort
  - ....