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: $O(n\log_2(n))$ 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 in that part of the array an 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 element in that range, and exchange with the i^{th} element of the array.
  4. Done when i = (N-1).

SelectionSort

- Easy to show SelectionSort requires N*(N-1)/2 comparisons, if N is the length of the array to be sorted
- SelectionSort is an $O(N^2)$ algorithm since the leading term is N^2 (ignore constant coefficients in big-O notation)
- Question: can we find a way to speed up SelectionSort?

Speed Up SelectionSort?

- When you have a $O(N^2)$ algorithm, often pays to break problem into smaller subproblems, solve subproblems separately, and then assemble final solution.
- Rough argument: suppose you break problem into k pieces. Each piece takes $O((N/k)^2)$ time, so time for doing all k pieces is roughly $k * O(N^2/k^2) = 1/k * O(N^2)$ time.
- If we divide problem into two subproblems that can be solved independently, we can halve the running time!
- Caveat: the partitioning and assembly processes should not be expensive.
- Can we apply this divide and conquer approach to sorting?

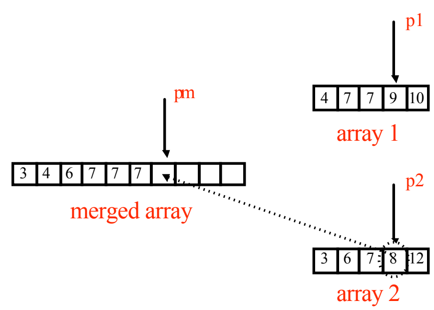
MergeSort

- Quintessential divide-and-conquer algorithm
- Divide array into equal parts, sort each part, then merge
- Three questions:
  - Q1: How do we divide array into two equal parts?
  - A1: Use indices into array.
  - Q2: How do we sort the parts?
  - A2: call MergeSort recursively!
  - Q3: How do we merge the sorted subarrays?
  - A3: Have to write some (easy) 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
- Initialize all three indices to 0 (start of each array)
- Compare element a1[p1] with a2[p2], and move smaller into m[pm].
- Increment the appropriate indices (p1 or p2), and pm.
- If either a1 or a2 is empty, copy remaining elements from the other array (a2 or a1, respectively) into m.

Merging Sorted Arrays

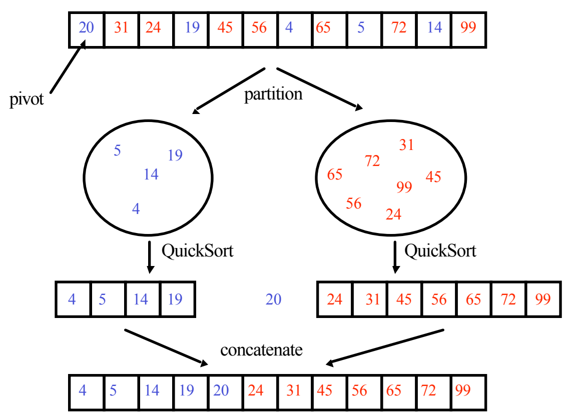


MergeSort

- Asymptotic complexity: $O(n\log_2(n))$
- Much faster than $O(n^2)$
- Disadvantage: need extra storage for temporary arrays
- In practice, this can be a serious disadvantage, even though MergeSort is *asymptotically optimal for sorting*.
- Can do MergeSort in place, but this is *very* tricky.
- Good sorting algorithms that do not use extra storage?
- Yes. Quicksort.

QuickSort

- Intuitive idea:
  - Given an array A to sort, and a pivot value P
  - Partition array elements into two subarrays, SX and SY
  - SX contains only elements less than or equal to P
  - SY contains only elements greater than P
  - Sort subarrays SX and SY separately
  - Concatentate (not merge!) sorted SX and SY to produce result
  - Sort(A) = Sort(SX<=P) + Sort(SY>P)
  - Divide and conquer if size SX and SY is about half size A
  - Concatentation is easier than merging

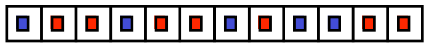


QuickSort

- Main advantages:
  - Divide and conqueror
  - Sorting can be done *in-place* (no extra arrays need to be created)
- Key problems:
  - How do we partition array in place?
  - How do we pick pivot to split array roughly in half?
- If we can partition in place, can have a quickSort method of the form:

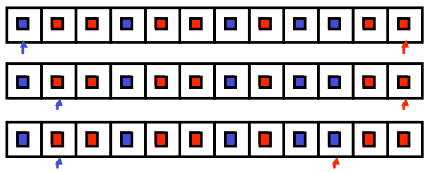
  void quickSort (int[] A, int low, int high)
  //quicksort values in array between low and high

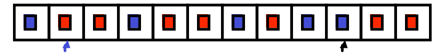
In-place Partitioning



How can we move all the blues to the left of all the reds?

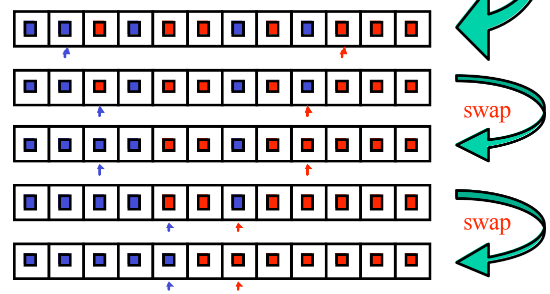
1. Keep two indices, LEFT and RIGHT.
2. Initialize LEFT at start of array and RIGHT at end of array.
3. Invariant: all elements to left of LEFT are blue
   all elements to right of RIGHT are red
4. Keep advancing indices until they pass, maintaining invariant

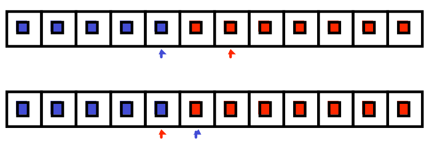




Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

swap





- Once indices cross partitioning is done.
- If you replace blue with '>' and red with '≥', this is exactly what we need for quicksort partitioning.
- Notice that after partitioning, array is partially sorted.
- Recursive calls on partitioned subarrays will sort subarrays.
- No need to concatenate arrays since we partitioned in place.

QuickSort

- How to pick pivot?
  - ideal pivot is median since this splits data in half
  - unfortunately, computing median exactly is expensive
  - Heuristics:
    - Use first value in array as pivot
    - Use middle value in array as pivot
    - Use median of first, last, and middle values in array as pivot
- QuickSort is not as efficient as SelectionSort for very small N. Often switch to simpler sort method when a subarray has length < small_N.
- Worst case for QuickSort is when array already is sorted is pivot is first element in array!

Summary

- Sorting methods discussed in lecture:
  - SelectionSort: $O(N^2)$
    - easy to code
    - sorts in place
  - MergeSort: $O(N\log(N))$
    - asymptotically optimal sorting method
    - most implementations require extra array allocation
  - Quicksort: $O(N^2)$
    - behaves more like $N\log(N)$ in practice
    - sorts in place
- Many other sorting methods in literature:
  - Heap sort (later in CS211)
  - Shell sort
  - Bubble sort
  - Radix sort
  - …