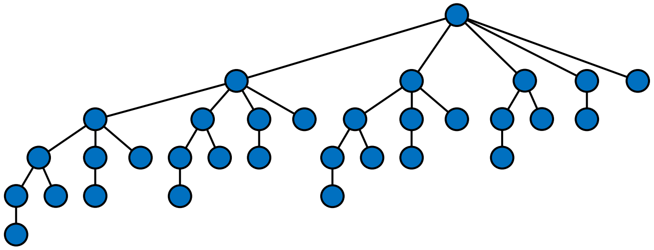
Lecture 12. Sorting and Selection

CpSc 212: Algorithms and Data Structures

Brian C. Dean



School of Computing

Clemson University

Spring, 2014

Comparison based sorting

Algorithm	**Runtime**	**Stable**	**In-Place?**
Bubble Sort	$O(n^2)$	Yes	Yes
Selection Sort	$O(n^2)$	Yes	Yes
Insertion Sort	$O(n^2)$	Yes	Yes
Merge Sort	$\Theta(n \log n)$	Yes	No
Randomized Quicksort	$\Theta(n \log n)$ w/high prob.	No*	Yes*
Deterministic Quicksort	$\Theta(n \log n)$	No*	Yes*
BST Sort	$\Theta(n \log n)$	Yes	No
Heap Sort	$\Theta(n \log n)$	No*	Yes*

- Sorting takes $\Omega(n \log n)$ in the comparison based model.

* = can be transformed into a stable, out-of-place algorithm

Comparison based sorting

Algorithm	Runtime	Stable	In-Place?
Bubble Sort	$O(n^2)$	Yes	Yes
Selection Sort	$O(n^2)$	Yes	Yes
Insertion Sort	$O(n^2)$	Yes	Yes
Merge Sort	$\Theta(n \log n)$	Yes	No
Randomized Quicksort	$\Theta(n \log n)$ w/high prob.	No*	Yes*
Deterministic Quicksort	$\Theta(n \log n)$	No*	Yes*
BST Sort	$\Theta(n \log n)$	Yes	No
Heap Sort	$\Theta(n \log n)$	No*	Yes*

- Sorting takes $\Omega(n \log n)$ in the comparison based model.
- Can we do better in a different model of computation?

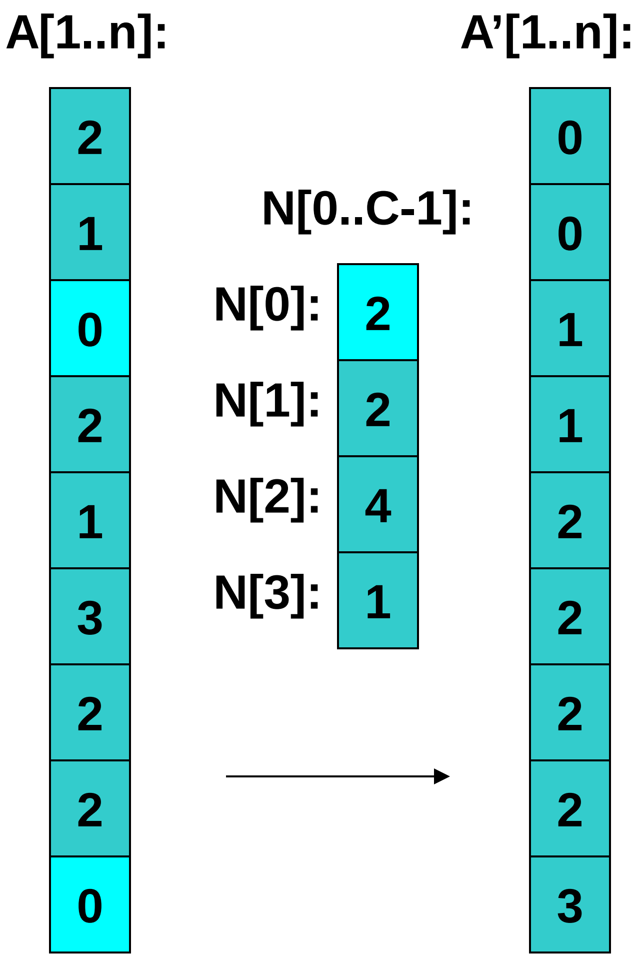
* = can be transformed into a stable, out-of-place algorithm

RAM Sorting Algorithms

- Suppose we are sorting n integers in the range 0…C – 1 in the RAM model of computation.
- **Counting sort**: O(n + C) time.
  - Sorts integers of magnitude C = O(n) in linear time.
- **Radix sort**: $O(n \max(1, \log_n C))$ time.
  - Sorts integers of magnitude $C = O(n^k)$, k = O(1), in linear time.

Counting Sort

- Scan A[1..n] in O(n) time and build an array N[0..C-1] of element counts.
- By scanning N, we then reconstruct A in sorted order in O(n + C) time.
- Ideally suited for C = O(n).
- Not in-place.
- Stable, if we're careful…



Radix Sort

- Write elements of A[1..n] in some base (radix), r. Typically, we set r = n.
- Sort on each digit, starting with the **least** significant, using a **stable** sort.
- # digits = $\log_n C$, which is constant if $C = n^{O(1)}$.
- Runtime O(n) if $C = n^{O(1)}$. (recall word size assumptions w/RAM)
- Stable, not in-place

A[1..n]:

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

Selection

- **Selection** is the problem of locating the kth largest element in an unsorted array / linked list.
- The kth largest element is also called the kth **order statistic**.
- Common values of k:

  k = 1: minimum

  k = n: maximum

  k = n/2: median
- *select* operation in a binary search tree.
- It's easy to find the min and max in Θ(n) time.
- For the median, we can easily achieve Θ(n log n) time by first sorting the array, but can we do it faster?

“QuickSelect”

- To select for the item of **rank** k in an array A[1..n].
- As in quicksort, pick a pivot element and partition A in linear time:

Elements < pivot	pivot	Elements > pivot

- After partitioning, the pivot element ends up being placed where it would in the sorted ordering of A
  – So we know the rank, r, of the pivot!
  – If k = r, the pivot is the element we seek and we’re done.
  – If k < r, select for the element of rank k on the left side.
  – If k > r, select for the element of rank k – r on the right side.
- Just like quicksort, except we only recurse on one subproblem instead of both.

QuickSelect: Choosing a Pivot

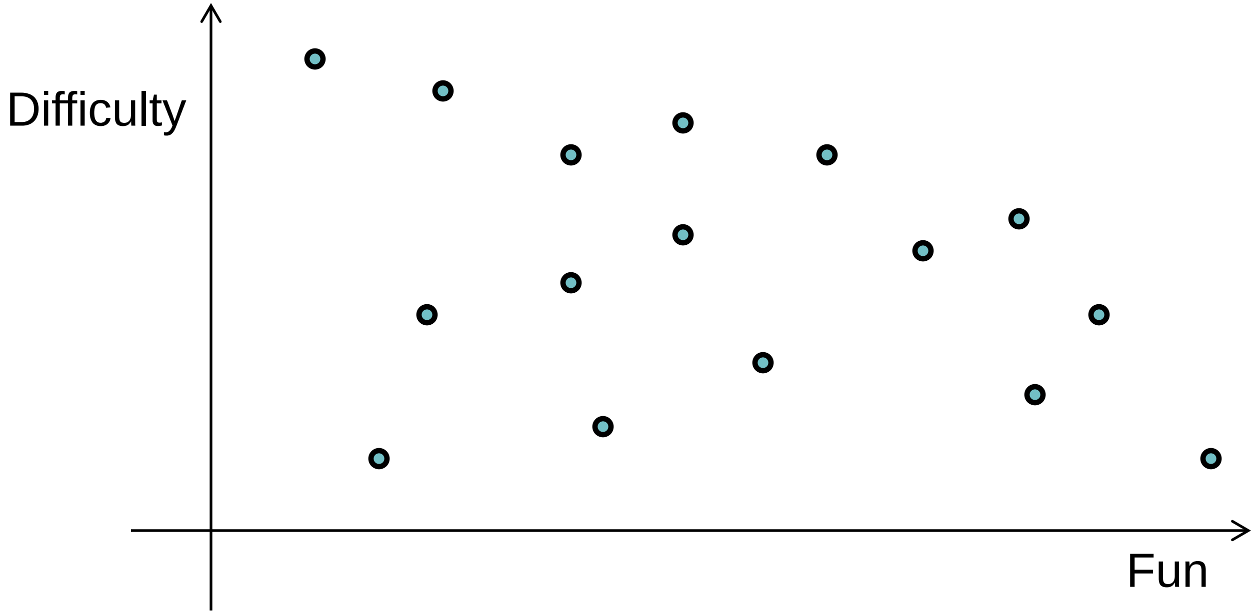
- As in quicksort, the difficulty with quickselect is choosing a good pivot.
- The ideal choice would be the median element, but then again we're trying to compute the median with this algorithm!
- Good choice: choose a random element as the pivot.
  - Intuition: "on average" we split the problem into two reasonably large pieces. And if we always manage to split into reasonably large pieces, we're solving a recurrence like $T(n) = T(n/2) + \Theta(n)$ or $T(n) = T(9n/10) + \Theta(n)$, the solution of which is $T(n) = \Theta(n)$.
  - The **expected** running time of quicksort is $\Theta(n)$ even though the worst case running time is $\Theta(n^2)$.

Deterministic Selection: Applications

- The median is an ideal partitioning point for divide and conquer algorithms.
- Median can be found in O(n) worst case time using a cool divide and conquer algorithm.
- Example: with quicksort, we can now find the median and partition in linear time in the worst case, so this gives us a $\Theta(n \log n)$ deterministic version of quicksort.
  - Can it still be made to operate in place?
  - The hidden constant is slightly large, since the hidden constant in the $\Theta(n)$ runtime for deterministic selection is also a bit large.

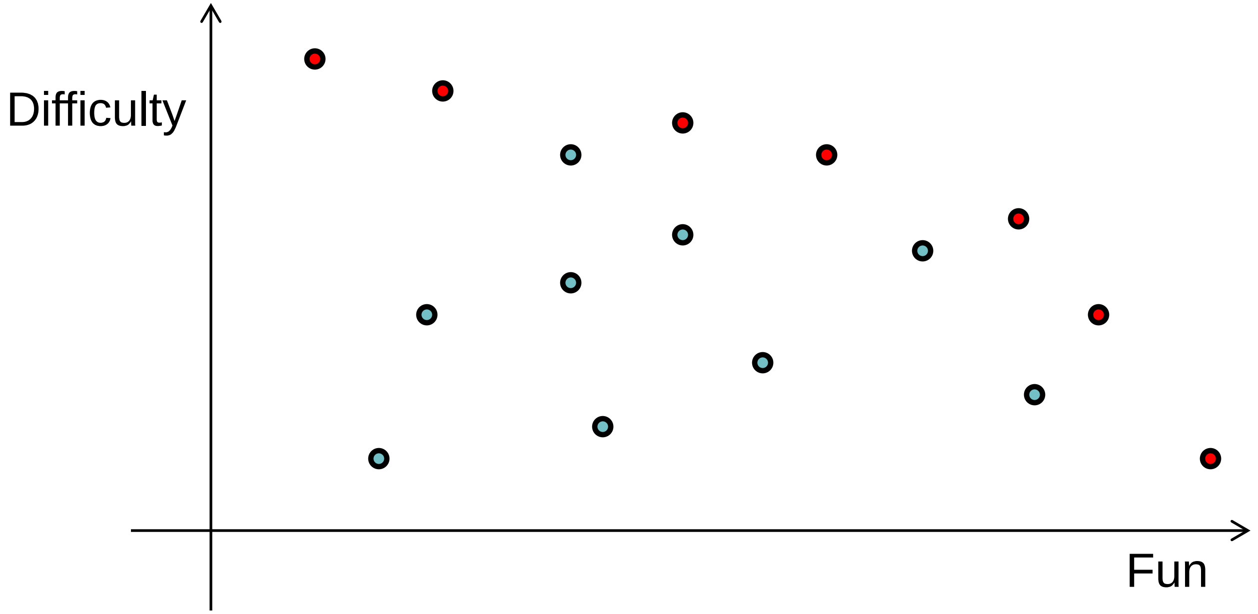
Pareto Optimal Solutions

- In optimization, it is often useful to find the non-dominating points.
- Points that are not dominated in both x and y axis.



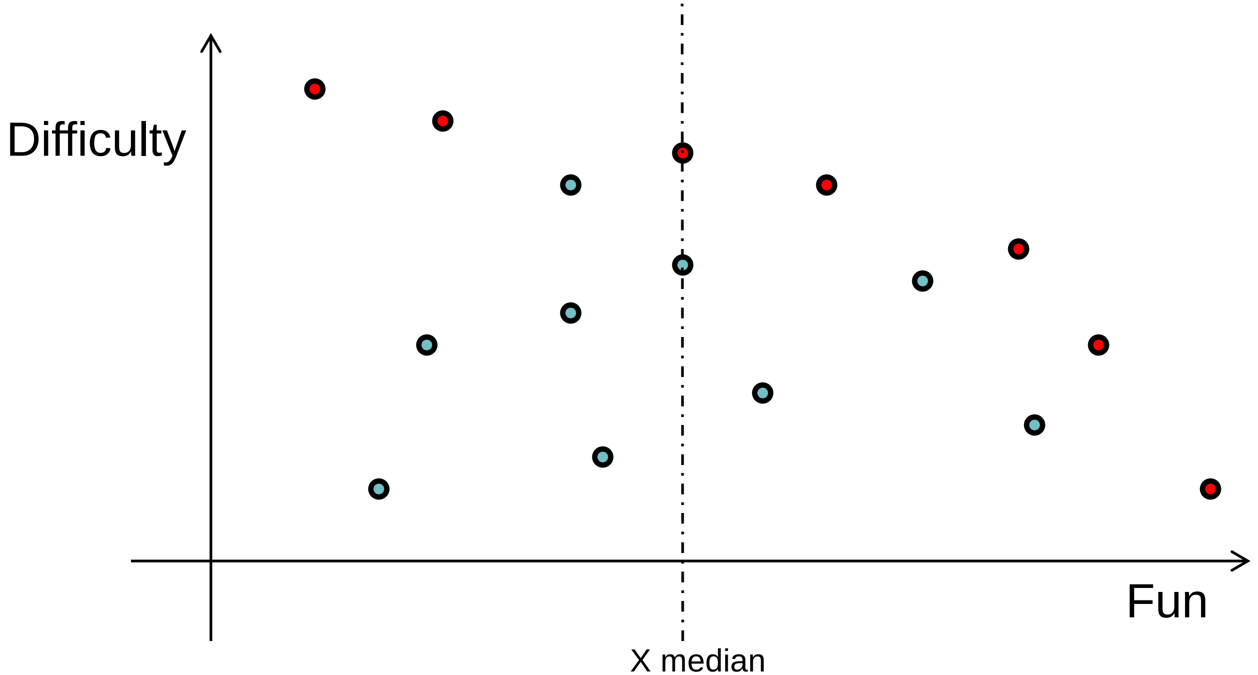
Pareto Optimal Solutions

- In optimization, it is often useful to find the non-dominating points.
- Points that are not dominated in both x and y axis.
- Also known as pareto optimal solutions.



Pareto Optimal Solutions

- In optimization, it is often useful to find the non-dominating points.
- Points that are not dominated in both x and y axis.
- Also known as pareto optimal solutions.
- Find the median and recursively find the pareto optimal solutions in both halves.
- Merge the solutions by deleting all solutions in the left half that are greater than the max in the right half. $T(n) = 2T(n/2) + O(n) = O(n \log n)$.



Pareto Optimal Solutions

- In optimization, it is often useful to find the non-dominating points.
- Points that are not dominated in both x and y axis.
- Also known as pareto optimal solutions.
- Find the median and recursively find the pareto optimal solutions in both halves.
- Merge the solutions by deleting all solutions in the left half that are greater than the max in the right half. $T(n) = 2T(n/2) + O(n) = O(n \log n)$.
- Much simpler – Sort all points on x. Scan from right to left and add points where we reset the max. Also $O(n \log n)$ time.