Sorting

Lots of Sorting Applications. Some are:

Commercial Applications:

data stored ordered by one "key". Processing requires it ordered by another key(s).

- e.g., enumerate hash tabled
- e.g., transactions on e-commerce site ordered by server arrival timestamp. Sort by "expiry date on credit card" to send out "card about to expire email"
- e.g., iPhone app-usage log ordered by timestamp of usage user wants to see app list ordered by frequency of use. Sort the log by app.
  e.g., "messaging sites" ordered by timestamp, but want to know: what's trending, who are the chattiest users, who is the chattiest country, etc. Sort by these.

Operating Systems Research

- e.g., Complete N jobs, each requiring T(N) units of processing time.
- Must schedule to maximize customer satisfaction by minimizing average job completion times.
- e.g., M processors and N jobs. Must schedule so that last job to complete finishes as soon as possible.
- Algorithms to accomplish these require sorting (and re-sorting) by time-to-completion, T(N), etc.

Simulations:

- e.g., weather prediction, financial markets, traffic flow, urban planning, etc.
  usually require events/items sorted on various keys

Graph Algorithms:

- e.g., shortest path through network, fastest path through network, etc.
- algorithms require sorting by "weights" (e.g., bandwidth, cost of fuel)

Huffman Compression:

- sort by frequencies

Order Statistics:

e.g.:

- Efficient (speed up) Searching - How can we efficiently test whether element, k, is in set S?
- Uniqueness Testing - How can we test if the elements of a given collection of items, S, are all distinct?
- Deleting Duplicates - How can we remove all but one copy of any repeated elements in S?
- Median/Selection - How can we find the k-th largest item in set S?
- Frequency Counting - Which is the most frequently occurring element in set S, i.e., the mode?
- Reconstructing the Original Order - How can we restore the original arrangement of a set of items after we permute them for some application?
- Set Intersection/Union - How can we intersect or union the elements of two containers?
- Finding a Target Pair - How can we test whether there are two integers, x,y in S, such that x+y = z for some target z ?

Sorting Effciency

1) number of comparisons
2) number of data moves

O(nlogn) – best average case (comps) for comparison-based sorts
O(n) – best average case (adat moves) for address-based sorts

Types of sorts:

Insertion-type sorts:

- start with empty container
- insert items one-by-one (in order in container)
- Tree Sort, Insertion Sort

Priority Q-type sorts:

- insert items into P.Q.
- remove one-by-one => get sorted order
- Heap Sort, Selection Sort

Divide and Conquer-type sorts:

- divide unsorted part into 2 parts
- sort each part and recombine
- Quick Sort, Merge Sort

Diminishing Increment-type sorts:

- Shell sort

Transposition-type sorts:

- Bubble sort

Address-type sorts:

- items are not compared to each other
- categorized based on specific properties
- Radix Sort, Proxmap Sort
-

Sorting Animations:

As far as they may help one understand algorithms covered in text/class.
(Note: tests assume algorithms as covered in text/class.)

Human subject Insertion/Selection/Merge:
http://www.youtube.com/watch?v=INHF_5RIxTE&feature=related

Robot QS vs BS sort-off:
https://www.youtube.com/watch?v=aXXWXz5rF64&feature=iv&src_vid=H5kAcmGOn4Q&annotation_id=annotation_2512573901

Robot QS vs MS sort-off:
https://www.youtube.com/watch?v=es2T6KY45cA&feature=iv&src_vid=H5kAcmGOn4Q&annotation_id=annotation_2685924271
etc.

Enable java/script: http://www.csse.monash.edu.au/~dwa/Animations/index.html

Mergesort can help visualize the recursion:
http://www.ee.ryerson.ca/~courses/coe428/sorting/sorting.html

Insertion Sort

- A divides into 2 parts: Left Hand Sides (LHS) is sorted, Right-Hand-Side(RHS) is not
- each step:
  - get next from RHS (x) (and remove)
  - find spot in LHS it should go
  - shuffle if necessary
  - insert x

e.g.

```
Sorted    Unsorted
     5|3 9 6 1 7          A[1]   (3)
R    5 _|9 6 1 7
S    _ 5|9 6 1 7
I    3 5|9 6 1 7
R    3 5 _|6 1 7          A[2]   (9)
S    3 5 _|6 1 7
I    3 5 9|6 1 7
     3 5 6 9|1 7          after A[3] done (6)
     1 3 5 6 9|7          after A[4] done (1)
     1 3 5 6 7 9          after A[5] done (7)
             Sorted   Unsorted
```

Insertion Sort Algorithm

```
A - Array
n - size of array
insertionSort(A, n)

   for i from 1 to n - 1 by 1

       key = A[i];
       j = i-1;

       Loop while  j >= 0 && A[j] > key
           A[j+1] = A[j];
           j = j-1;

       A[j+1] = key;
```

Analysis of Insertion Sort

Comparisons
Worst Case: compare A[i] to all items to left of it (reverse ordered list)

A[1] → 1 comp (to A[0])
A[2] → 2 comps
…
A[n-1] → n-1 comps
total 1+2+.. (n+1) $= \frac{(n-1)n}{2}$ => O(n^2)

Average: as above, but for each A[i]
do average of 1, 2, 3 .. I comps $= \frac{i(i+1)}{2i} = \frac{(i+1)}{2}$
total $\sum_{i=1}^{n-1} \frac{(i+1)}{2} = \frac{1}{2}[(n-1) + \sum_{i=1}^{n-1} \mathrm{i}\,] = \frac{1}{2}[(n-1) + \frac{(n-1)n}{2}]$ => O(n^2)

Best: at each step 1 comp
1 + 1+ 1 … + 1 = n -1 => O(n)

Data Moves
Worst Case: (reverse ordered list)

- each step shuffle all items in "sorted"

A[1] → 1 shuffle and insert item in hole = 2
A[2] → 1 shuffle and insert item in hole = 3
…
A[n-1] → n-1 shuffle and insert item in hole = n
total $\frac{(n+1)n}{2} - 1$ => O(n^2)

Average: A[1] average of 1 shuffle and insertion in hole = 2
A[2] average of 2 shuffle and insertion in hole = 3
…
A[n-1] average of n-1 shuffle and insertion in hole = n
=> O(n^2)

Best: (ordered list)
No shuffling. Code may do the 1 "insert" so 1 + 1 + 1 … + 1 = n-1 => O(n)
OR O(1) if no "copy over"

Quick Sort

idea:

- [unsorted list]. Choose pivot
- rearrange list such that
  [(< pivot) | pivot | (> pivot)]
- pivot is in final position
- Run quick sort on (< pivot) and (> pivot)\
  NOTE: can “re-arrange” by starting from both ends and swapping if out-of-place

Choose Pivot 8

5, 10, 3, 2, 7, 8, 9, 15, 1, 4, 20,

5, 4, 3, 2, 7, 1, 9, 15, 8, 10, 20,

1, 2, 3, 4, 7, 5, 9, 15, 8, 10, 20,

1, 2, 3, 4, 7, 5, 9, 15, 8, 10, 20,

1, 2, 3, 4, 5, 7, 9, 15, 8, 10, 20,

1, 2, 3, 4, 5, 7, 9, 15, 8, 10, 20,

1, 2, 3, 4, 5, 7, 8, 15, 9, 10, 20,

1, 2, 3, 4, 5, 7, 8, 9, 15, 10, 20,

1, 2, 3, 4, 5, 7, 8, 9, 10, 15, 20,

Quick Sort Algorithm

Quicksort from Standish (does not move pivot into center)

```
Partition (array A, i, j)

  pivpos=(i+j)/2
  pivot = A[ pivpos ]  //middle key
  Loop
   while ( A[i] < pivot ) i++
   while ( A[*j] > pivot ) j--
   if (*i <= *j )
      temp = A[*i]
      A[*i]=A[*j]
      A[*j]=temp; //swap i, j
      i++
      j--

   until i <= j;


QuickSort (array A, m, n) {
  if (m<n)
    i=m
    j=n
    Partition (A, i, j)
    QuickSort(A,m,j)
    QuickSort(A,i,n)
```

Analysis of Quick Sort

Best Case: when choose pivot, list is divided equally in half
(1/2 n) p (1/2n)

Comparisons:

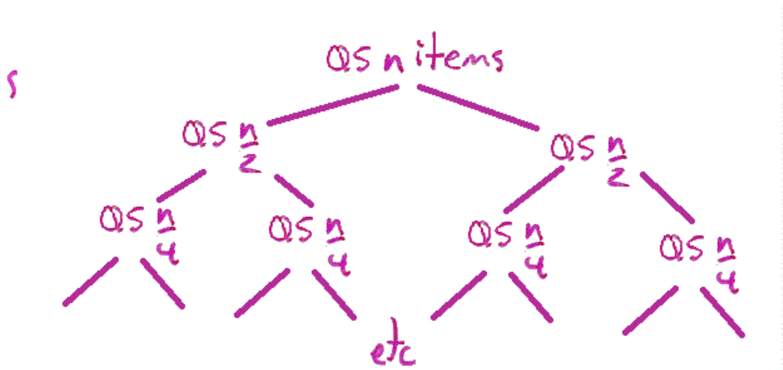
1st call to QS $\approx n$ comps (comp each with pivot) + 2 recursive calls

on each $\frac{1}{2}$ $\approx \frac{n}{2} + \frac{n}{2} = n$ total comps + 4 recursive calls

on each $\frac{1}{4}$ $\approx \frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} = n$ total comps + 8 recursive calls

…

on each level $\approx n$ comps



How many levels?
≈ height of recursive tree
≈ how many x can cut list length n in half? log(n)
=? O(n log n) comps
ore recurrence relations: C(n)= n + 2C (n/2) , C(2) = 2

Worst: pivot largest/smallest in list [p | rest of list]
divide lists length 0 + n -1

each step is approximately n comps
n+ (n-1) + (n-2) … 1 => O(n^2)

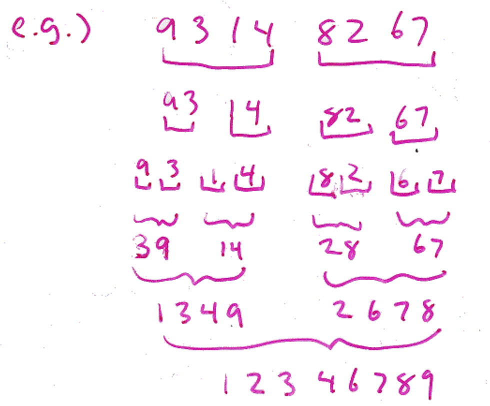
Average: recurrence relations(test) O(nlogn)

Merge Sort

idea:

- if list has 1 item, return
- divide list in half
- MergeSort each half
- merge the 2 halves into one

e.g.)



Merge Sort Algorithm

```
Merge (left, right)
  create temp array T
  loop while !empty(left && !empty(right)
    if first(left) < first(right)
      Append(T, first(left))
      removeFirst(left)
    else
      Append(T, first(right))
      removeFirst(left)
  if !emppty(left)
    Append(T, leftOver(left))
  if !emppty(right)
    Append(T, leftOver(right))

  return T
```

Analysis of Merge Sort

Worst Case: in half each time which is about logn height of call tree
each level (except leafs) total is about n comps so O(nlogn)

Best Case: same but n/2 comps so O(nlogn)

Average Case: O(nlogn)

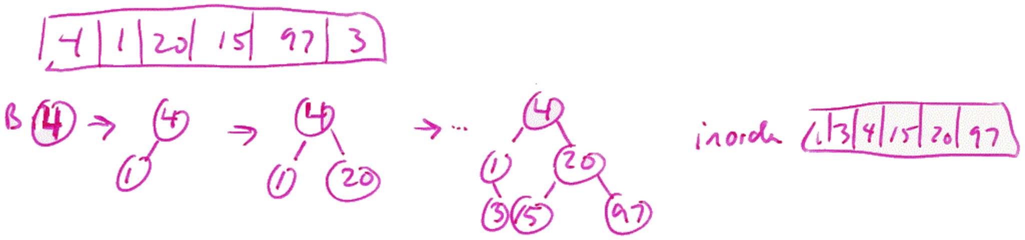
[Generally proved that for any comparison-based sort, fastest average time comparisons is O(nlogn)]

Tree Sort

idea:

- unsorted array A (A[0] … A[n-1])
- create BST B
- for each I, insert(B, A[i])
- In-order traversal of B

e.g.)



Analysis of Tree Sort

- insertion of BST is O(logn)
- insert n items and thus generally O(nlogn)
  comps = log1 + log2 + log3 + ... + log(n-1) = nlogn => O(nlogn) best/average

Note: In-order traversal takes no comps.

Worst O(n^2) – degenerate tree

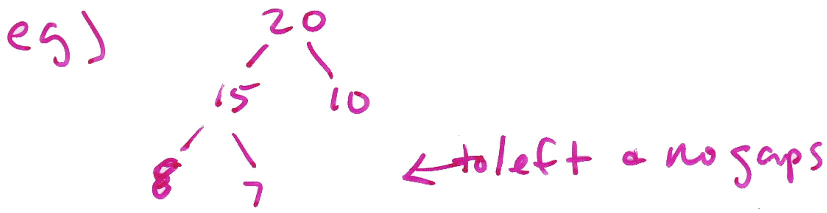
Data Moves

- move each item from A to B :n
- move each item from B to A :n
- 2n => O(n)

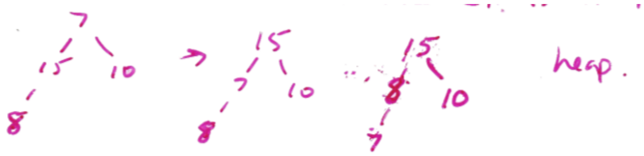
Heap Sort

Heap:

- What is a Heap? Heap is a binary tree such that
  - value of node >= value of kids (descendants)
  - every level is full except for possibly the last, but items on last level are filled inserted as far left as possible



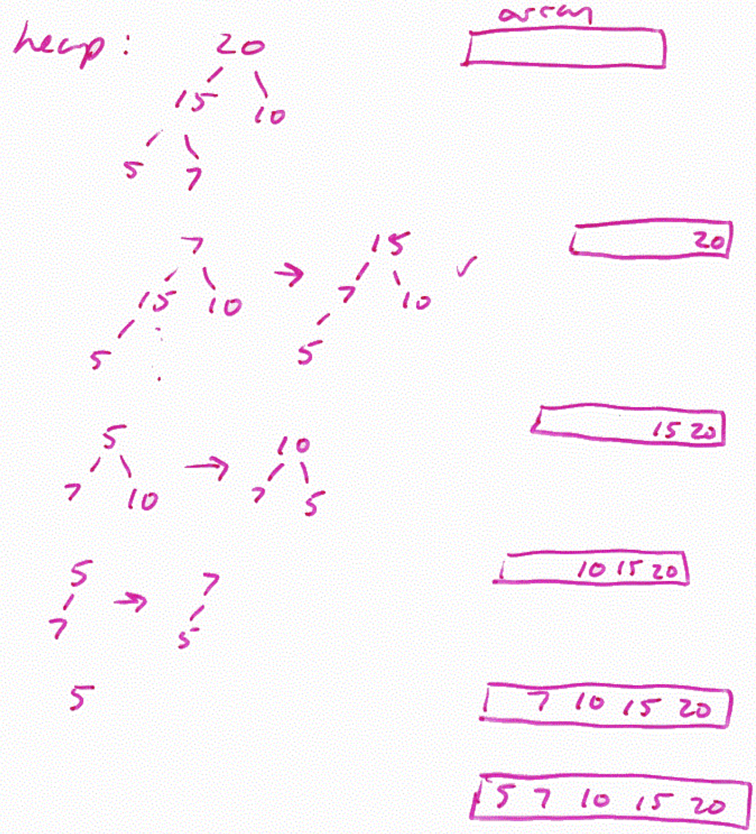
Note: largest item in tree must be at root
if remove root, re-heapify by:
moving "last" item to root "7"
keep swapping node with largest of kids until it is in place



idea:

- make data into heap
- while items left in tree
  - remove root and put in "final" array
  - replace root by "last" item
  - re-heapify

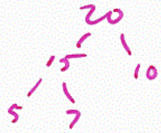
if fill "final array" back-to-front, it is in sorted order once tree is empty e.g) 15 20 10 5 7 ✓



Can do this "in" array.

store root at node 1 A[1]
left child of node in A[i] is at A[2i]
right " " A[2i+1]

eg) A: | 20 | 15 | 10 | 5 | 7 | =



each step swap "last" in heap with root + re-heapify

$\underbrace{|7\ 15\ 10\ 5}_{\text{heap part}}|20|$ re-heapify |15 7 10 5|20|

swap |5 7 10||15 20| re-heapify |10 7 5||15 20|

swap ||5 7||10 15 20| re-heapify ||7 5||10 15 20|

just swap 1st 2 ||5 7 10 15 20| ✓

Analysis of Heap Sort

- Comparisons and data moves are similar
- insert n items and thus generally O(nlogn)

Data Moves:

Worst case: to transform A into Heap takes O(n) (from Trees notes)
swap – constant K2
re-heapify heap into I – 1 nodes – at most log(i-1) saps, (height of heap)

Total:

O(n) + K2log(n-1)+k2+k2log(n-2)+…+k2log(2)+k2

$= k2 \sum_{i=2}^{n-1} \log(i) + (n-2)k2 + OO(n) + k2$

$< k2nlog(n) + (n-1)k2 + O(n)$ => O(nlogn)

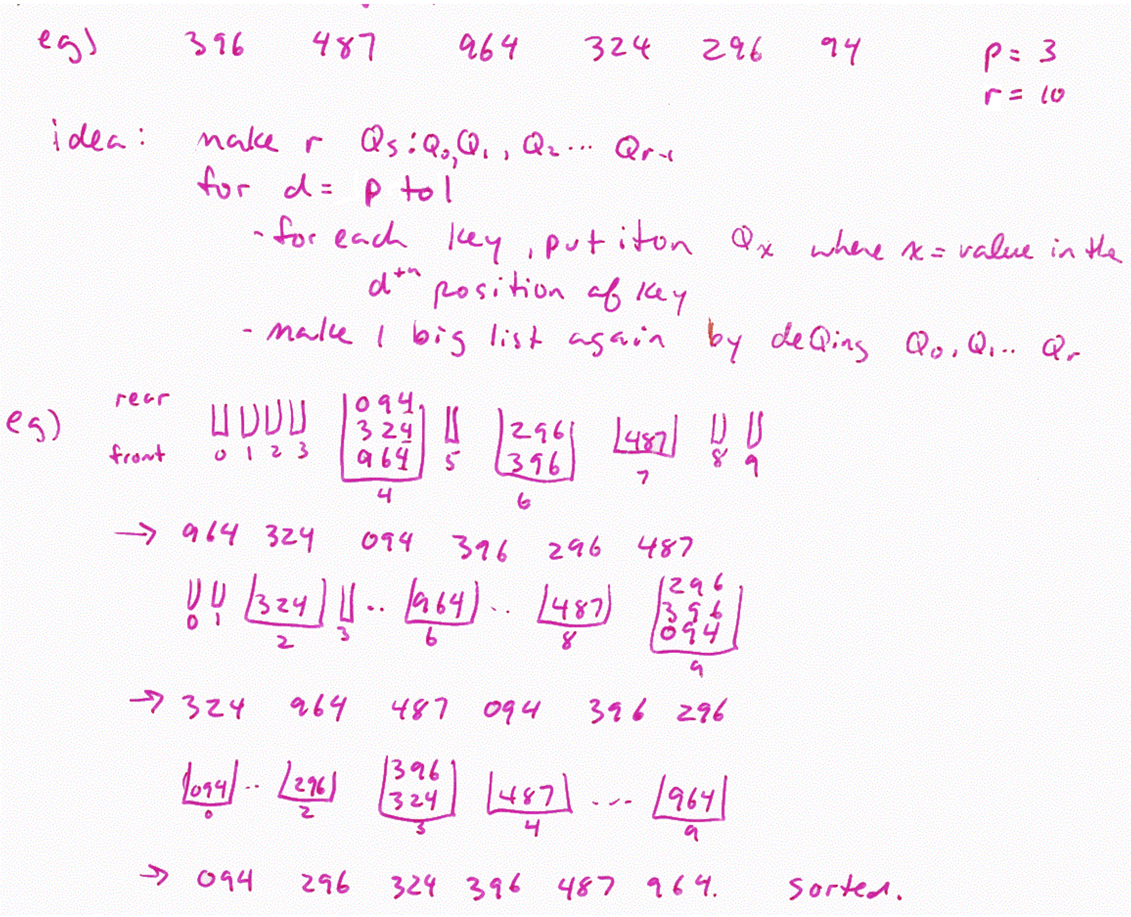
Worst, Average, Best time

Radix Sort

idea:

- radix (e.g. 129 radix is 10, 111010 radix is 2)

let p = max number of digits in keys to be sorted
let r = radix of keys



eg) 396 487 964 324 296 94 $p = 3$ $r = 10$

idea: make r Qs: $Q_0, Q_1, Q_2 \dots Q_{r-1}$
for $d = p$ to 1
- for each key, put it on Q_x where x = value in the d^{th} position of key
- make 1 big list again by deQing $Q_0, Q_1 \dots Q_r$

eg)

	0	1	2	3	4	5	6	7	8	9
rear					094					
					324		296			
front					964		396	487		

→ 964 324 094 396 296 487

0	1	2	3	..	6	..	8	9
								296
								396
		324			964		487	094

→ 324 964 487 094 396 296

0	..	2	3	4	...	9
			396			
094		296	324	487		964

→ 094 296 324 396 487 964. sorted.

Sorted p passes where each pass moved n items to Qs and from Q => 2pn => O(n)

No comparisons, only data moves

Stability of Sorts

- A sorting method is **stable** if it is:
    - preserves relative order of equal keys

e.g.) e- commerce site

- transactions put on array as arrive (ordered by timestamp)
- application needs to process by province so sort by province.
- if sort is **unstable**, timestamp ordering is not necessarily preserved within prince

original		Sorted by province (unstable)	
ONT	08:00:00	AB	08:01:32
AB	08:00:03	AB	08:00:03
ONT	08:01:00	AB	08:02:21
NB	08:01:09	BC	08:02:04
AB	08:01:32	NB	08:01:09
BC	08:02:04	ONT	08:01:00
ONT	08:02:11	ONT	08:02:11
AB	08:02:21	ONT	08:00:00

Stable Sorts: Insertion Sort, Merge Sort, Radix, BSTs

Unstable: Quick Sort, Heap Sort

Stability of Sorts

	Best	Average	Worst
Quick Sort	nlogn	nlogn	n^2
Merge Sort	nlogn	nlogn	nlogn
Heap Sort	nlogn	nlogn	nlogn
Insertion Sort	n	n^2	n^2
Tree Sort	nlogn	nlogn	n^2
Radix Sort	pn	pn	pn

Is Radix Sort Best?

- if p is large, pn maybe no better than nlogn or n^2
- if space is limited, Rardix sort is bad (all those Qs)

Quick sort is about 2x faster than Heap Sort and Merge sort in practice
Is Quick Sort Best?

- space is limited, Quick Sort and Merge Sort are bad because lots of stack space or recursive calls) – Heap sort is best
- if guarantee required (e.g. real-time applications), Quick Sort is bad O(n^2) worst case – Heap or Merger Sorts is best
- if Stability requires – Merge Sort is best

If A is already mostly in order – Insertion Sort is good => approximately O(n) best time vs O(nlogn)

No Single method is better than all others in all situations