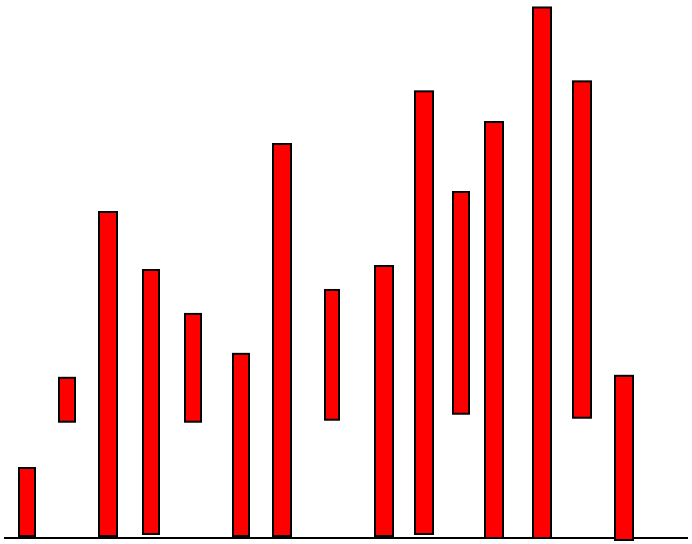
CompSci 100e
Program Design and Analysis II



April 26, 2011

Prof. Rodger

Announcements

- Things due this week:
  - APTs due today, Apr 26
  - Extra credit assignments due Wed, Apr 27
  - No late assignments accepted after Wed night!
- Today
  - Test 2 back – solutions posted on calendar page
  - Balanced Trees
  - Sorting

Final Exam

- Final Exam – Wed, May 4, 7-10pm
  - Same room, old Chem 116
  - Covers topics up through today
  - Closed book, closed notes
  - Can bring 4 sheets of paper with your name on it
- Study - practice writing code on paper
  - From tests this semester, from old tests
  - From classwork, labs, assignments, apts….
- Will have different office hours til exam
  - will post on front page of CompSci 100e web page
  - Subject to change, check before coming over

Sorting: From Theory to Practice

- Why study sorting?
  - Example of algorithm analysis in a simple, useful setting
  - Lots of sorts
    - Compare running times
    - Compare number of swaps
- http://www.sorting-algorithms.com/

Sorting out sorts

- Simple, $O(n^2)$ sorts --- for sorting n elements
  - Selection sort --- n^2 comparisons, n swaps, easy to code
  - Insertion sort --- n^2 comparisons, n^2 moves, stable, fast, can finish early
  - Bubble sort --- n^2 everything, easiest to code, slowest, ugly

- Divide and conquer sorts: O(n log n) for n elements
  - Quick sort: fast in practice, $O(n^2)$ worst case
  - Merge sort: good worst case, great for linked lists, uses extra storage for vectors/arrays

- Other sorts:
  - Heap sort, basically priority queue sorting O(n log n)
  - Radix sort: doesn’t compare keys, uses digits/characters
  - Shell sort: quasi-insertion, fast in practice, non-recursive

Selection sort: summary

- Simple to code n^2 sort: n^2 comparisons, only n swaps
- Repeat: Find next min, put it in its place in sorted order

```
void selectSort(String[] a) {
    int len = a.length;
    for(int k=0; k < len; k++){
        int mindex = getMinIndex(a,k,len);
        swap(a,k,mindex);
    }
}
```

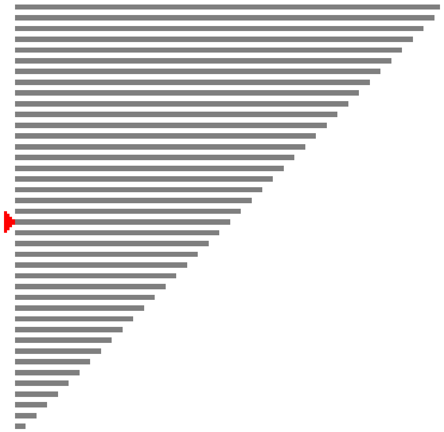
$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = O(n^2)$$

- # comparisons
  - Swaps?
  - Invariant:

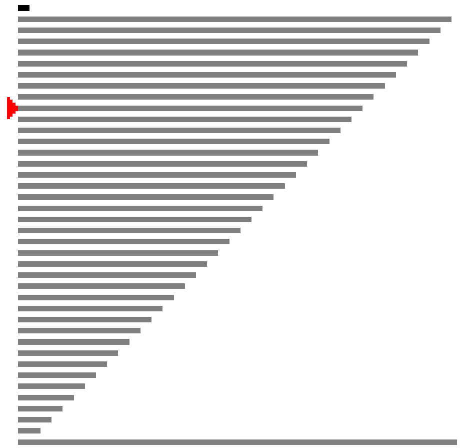
Sorted, won't move final position	?????

SelectionSort

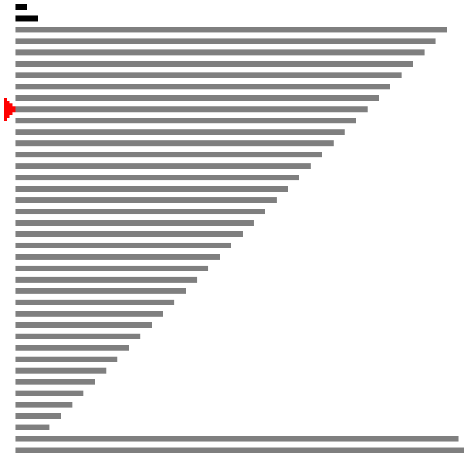
- Start



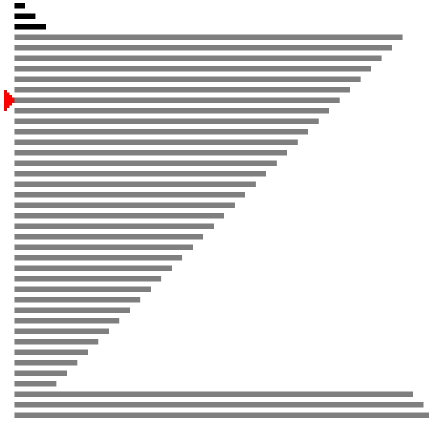
starting 2nd pass



- Starting 3rd pass



starting 4th pass



Insertion Sort: summary

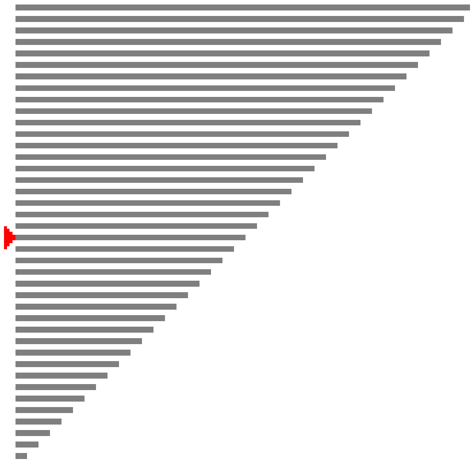
- Stable sort, $O(n^2)$, *good on nearly sorted vectors*
  - Stable sorts maintain order of equal keys
  - Good for sorting on two criteria: name, then age

```
void insertSort(String[] a){
    int k, loc; String elt;
    for(k=1; k < a.length; ++k) {
        elt = a[k];
        loc = k;
        // shift until spot for elt is found
        while (0 < loc && elt.compareTo(a[loc-1]) < 0) {
            a[loc] = a[loc-1];   // shift right
            loc=loc-1;
        }
        a[loc] = elt;
    }
}
```

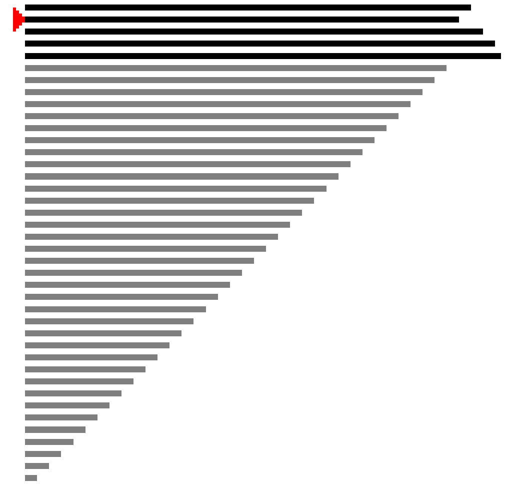
Sorted relative to each other	**?????**

Insertion Sort

- Start

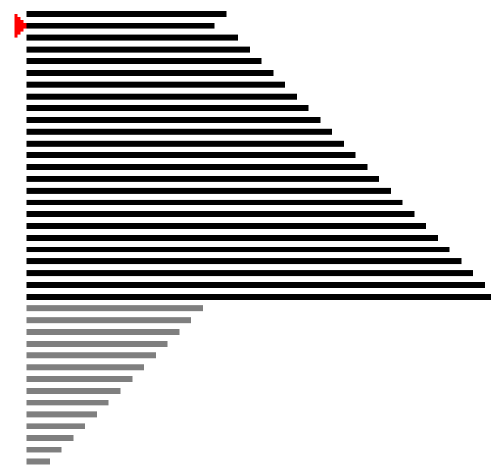


in 4th pass



- Several later passes

after more passes



Bubble sort: summary of a dog

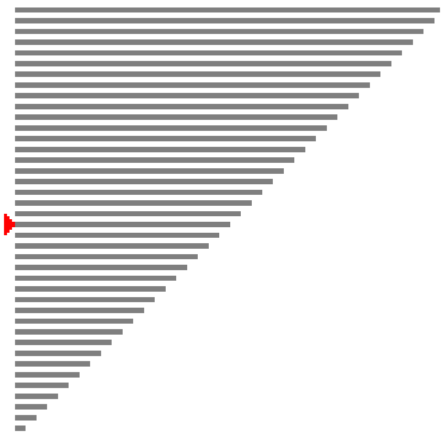
- For completeness you should know about this sort
  - Really, really slow (to run), really really fast (to code)
  - Can code to recognize already sorted vector (see insertion)
    - Not worth it for bubble sort, much slower than insertion

```
void bubbleSort(String[] a){
    for(int j=a.length-1; j >= 0; j--) {
        for(int k=0; k < j; k++) {
            if (a[k] > a[k+1])
                swap(a,k,k+1);
        }
    }
}
```

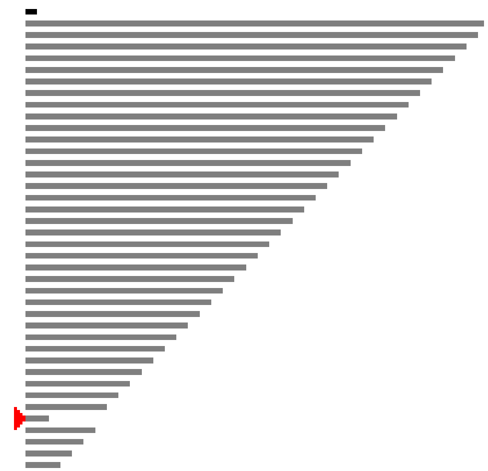
- “bubble” elements down the vector/array

Bubble sort

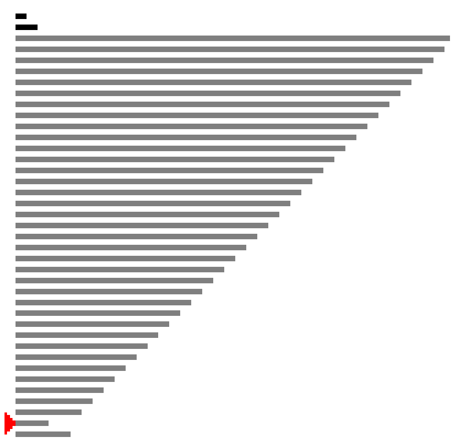
- Start



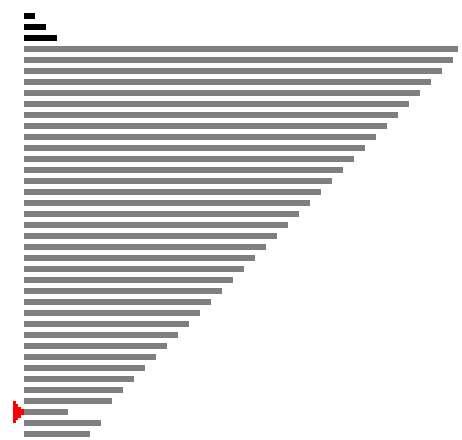
starting 2nd pass



- Starting 3rd pass



starting 4th pass



Summary of simple sorts

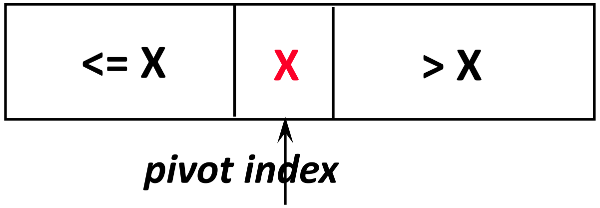
- Selection sort has n swaps, good for “heavy” data
  - moving objects with lots of state, e.g., …
    - In C or C++ this is an issue
    - In Java everything is a pointer/reference, so swapping is fast since it's pointer assignment
- Insertion sort good on nearly sorted data, stable!
  - Also foundation for Shell sort, very fast non-recursive
  - More complicated to code, but relatively simple, and fast
- Bubble sort is a travesty? But it's fast to code if you know it!
  - Can be parallelized, but on one machine don’t go near it

Quicksort: fast in practice

- Invented in 1962 by C.A.R. Hoare, didn't understand recursion
  - Worst case is $O(n^2)$, but avoidable in nearly all cases
  - In 1997 Introsort published (Musser, introspective sort)
    - Like quicksort in practice, but recognizes when it will be bad and changes to heapsort

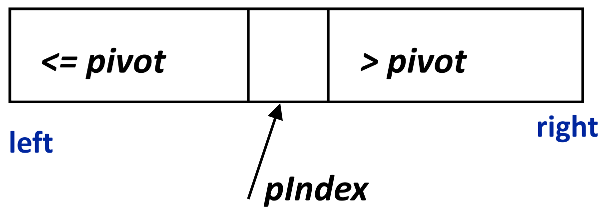
```
void quick(String[], int left, int right){
    if (left < right) {
        int pivot = partition(a,left,right);
        quick(a,left,pivot-1);
        quick(a,pivot+1, right);
    }
}
```

- Recurrence?

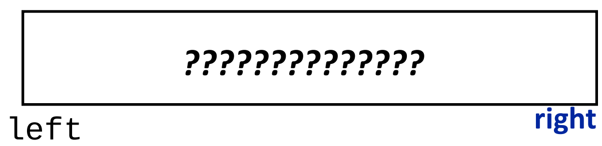


Partition code for quicksort

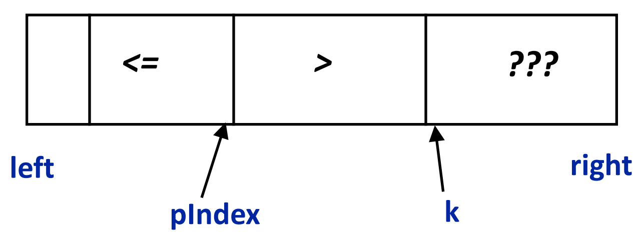
what we want



what we have



invariant



- Easy to develop partition

```
int partition(String[] a,
                    int left, int right)
{
     string pivot = a[left];
     int k, pIndex = left;
     for(k=left+1, k <= right; k++) {
        if (a[k].compareTo(pivot) <= 0){
           pIndex++;
           swap(a,k,pIndex);
        }
     }
     swap(a,left,pIndex);
}
```

- loop invariant:
  - statement true each time loop test is evaluated, used to verify correctness of loop
- Can swap into a[left] before loop
  - Nearly sorted data still ok

Analysis of Quicksort

- Average case and worst case analysis
  - Recurrence for worst case: T(n) =
  - What about average?
- Reason informally:
  - Two calls vector size n/2
  - Four calls vector size n/4
  - … How many calls? Work done on each call?
- Partition: median of three, then sort
  - Avoid bad performance on nearly sorted data

Analysis of Quicksort

- Average case and worst case analysis
  - Recurrence for worst case: T(n) = **T(n-1) + T(1) + O(n)**
  - What about average? **T(n) = 2T(n/2) + O(n)**
- Reason informally:
  - Two calls vector size n/2
  - Four calls vector size n/4
  - … How many calls? Work done on each call?
- Partition: median of three, then sort
  - Avoid bad performance on nearly sorted data

Merge sort: worst case O(n log n)

- Divide and conquer --- recursive sort
  - Divide list/vector into two halves
    - Sort each half
    - Merge sorted halves together
  - What is complexity of merging two sorted lists?
  - What is recurrence relation for merge sort as described?

    `T(n) =`

- Advantage of array over linked-list for merge sort?
  - What about merging, advantage of linked list?
  - Array requires auxiliary storage (or very fancy coding)

Merge sort: worst case O(n log n)

- Divide and conquer --- recursive sort
  - Divide list/vector into two halves
    - Sort each half
    - Merge sorted halves together
  - What is complexity of merging two sorted lists?
  - What is recurrence relation for merge sort as described?

  T(n) = **T(n) = 2T(n/2) + O(n)**

- Advantage of array over linked-list for merge sort?
  - What about merging, advantage of linked list?
  - Array requires auxiliary storage (or very fancy coding)

Merge sort: lists or arrays or …

- Mergesort for arrays

```
void mergesort(String[] a, int left, int right){
    if (left < right) {
         int mid = (right+left)/2;
         mergesort(a, left, mid);
         mergesort(a, mid+1, right);
         merge(a,left,mid,right);
    }
}
```

- What's different when linked lists used?
  - Do differences affect complexity? Why?

- How does merge work?

Summary of O(n log n) sorts

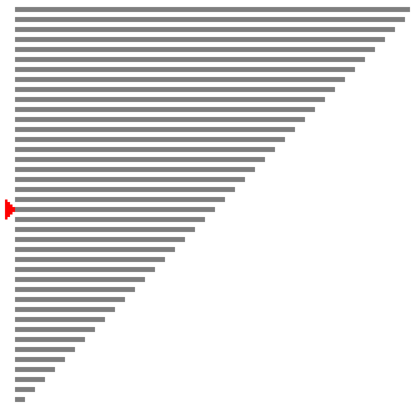
- Quicksort straight-forward to code, very fast
  - Worst case is very unlikely, but possible, therefore …
  - But, if lots of elements are equal, performance will be bad
    - One million integers from range 0 to 10,000
    - How can we change partition to handle this?
- Merge sort is stable, it's fast, good for linked lists, harder to code?
  - Worst case performance is O(n log n), compare quicksort
  - Extra storage for array/vector
- Heapsort, good worst case, not stable, coding?
  - Basically heap-based priority queue in a vector

Other sorts

- Shellsort
  - Divide and conquer approach then insertion sort kicks in
  - Named after?
- Timsort
  - Sort in python
  - Named after?
  - Derived from mergesort and insertionsort
  - Very fast on real world data, using far fewer than the worst case of O(n log n)

ShellSort

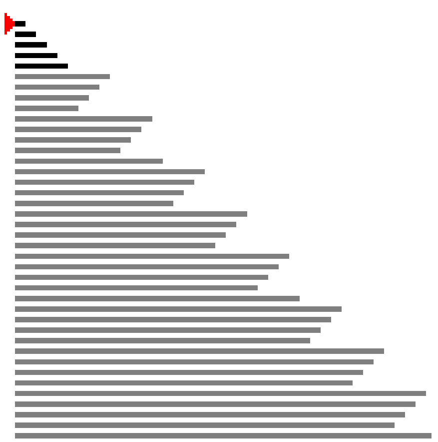
- Start



starting 2nd pass

- Starting 3rd pass

starting 4th pass



Sorting in practice

- Rarely will you need to roll your own sort, but when you do …
  - What are key issues?

- If you use a library sort, you need to understand the interface
  - In C++ we have STL
    - STL has `sort`, and `stable_sort`
  - In C sort is complex to use because arrays are ugly
  - In Java guarantees and worst-case are important
    - Why won't quicksort be used?
- Comparators allow sorting criteria to change

Non-comparison-based sorts

- lower bound: Ω(n log n) for comparison based sorts (like searching lower bound)
- bucket sort/radix sort are not-comparison based, faster asymptotically and in practice
- sort a vector of ints, all ints in the range 1..100, how?
  - (use extra storage)
- radix: examine each digit of numbers being sorted
  - One-pass per digit
  - Sort based on digit

23 34 56 25 44 73 42 26 10 16

0 1 2 3 4 5 6 7 8 9

10 42 23 73 34 44 25 56 26 16

0 1 2 3 4 5 6 7 8 9

10 16 23 25 26 34 42 44 56 73

Non-comparison-based sorts

- lower bound: Ω(n log n) for comparison based sorts (like searching lower bound)
- bucket sort/radix sort are not-comparison based, faster asymptotically and in practice
- sort a vector of ints, all ints in the range 1..100, how?
  - (use extra storage)
- radix: examine each digit of numbers being sorted
  - One-pass per digit
  - Sort based on digit

23 34 56 25 44 73 42 26 10 16

						16			
			73	44		26			
10		42	23	34	25	56			
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

10 42 23 73 34 44 25 56 26 16

		26							
	16	25		44					
	10	23	34	42	56		73		
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

10 16 23 25 26 34 42 44 56 73