

**Announcements:**

- We'll be running a preliminary testing run for Project #2 on Tuesday night.

**Today:**

- Sorting algorithms: why?
- Insertion, Shell's, Heap, Merge sorts
- Quicksort
- Selection
- Distribution counting, radix sorts

**Readings:** Today: *DS(IJ)*, Chapter 8; Next topic: Chapter 9.

- Sorting supports searching
- Binary search standard example
- Also supports other kinds of search:
  - Are there two equal items in this set?
  - Are there two items in this set that both have the same value for property X?
  - What are my nearest neighbors?
- Used in numerous unexpected algorithms, such as convex hull (smallest convex polygon enclosing set of points).

**Some Definitions**

- A sort is a *permutation* (re-arrangement) of a sequence of elements that brings them into order, according to some *total order*. A total order,  $\preceq$ , is:
  - **Total:**  $x \preceq y$  or  $y \preceq x$  for all  $x, y$ .
  - **Reflexive:**  $x \preceq x$ ;
  - **Antisymmetric:**  $x \preceq y$  and  $y \preceq x$  iff  $x = y$ .
  - **Transitive:**  $x \preceq y$  and  $y \preceq z$  implies  $x \preceq z$ .
- However, our orderings may allow unequal items to be equivalent:
  - E.g., can be two dictionary definitions for the same word: if entries sorted only by word, then sorting could put either entry first.
  - A sort that does not change the relative order of equivalent entries is called *stable*.

**Classifications**

- *Internal sorts* keep all data in primary memory
- *External sorts* process large amounts of data in batches, keeping what won't fit in secondary storage (in the old days, tapes).
- *Comparison-based* sorting assumes only thing we know about keys is order
- *Radix sorting* uses more information about key structure.
- *Insertion sorting* works by repeatedly inserting items at their appropriate positions in the sorted sequence being constructed.
- *Selection sorting* works by repeatedly selecting the next larger (smaller) item in order and adding it one end of the sorted sequence being constructed.

## Sorting by Insertion

- Simple idea:
  - starting with empty sequence of outputs.
  - add each item from input, *inserting* into output sequence at right point.
- Very simple, good for small sets of data.
- With vector or linked list, time for find + insert of one item is at worst  $\Theta(k)$ , where  $k$  is # of outputs so far.
- So gives us  $O(N^2)$  algorithm. Can we say more?

## Inversions

- Can run in  $\Theta(N)$  comparisons if already sorted.
- Consider a typical implementation for arrays:
 

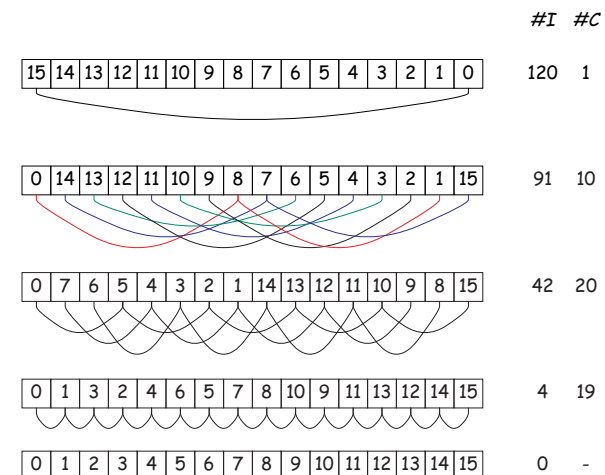
```
for (int i = 1; i < A.length; i += 1) {
    int j;
    Object x = A[i];
    for (j = i-1; j >= 0; j -= 1) {
        if (A[j].compareTo (x) <= 0) /* (1) */
            break;
        A[j+1] = A[j];
    }
    A[j+1] = x;
}
```
- #times (1) executes  $\approx$  how far  $x$  must move.
- If all items within  $K$  of proper places, then takes  $O(KN)$  operations.
- Thus good for any amount of *nearly sorted* data.
- One measure of unsortedness: # of *inversions*: pairs that are out of order (= 0 when sorted,  $N(N - 1)/2$  when reversed).
- Each step of  $j$  decreases inversions by 1.

## Shell's sort

**Idea:** Improve insertion sort by first sorting *distant* elements:

- First sort subsequences of elements  $2^k - 1$  apart:
  - sort items #0,  $2^k - 1, 2(2^k - 1), 3(2^k - 1), \dots$ , then
  - sort items #1,  $1 + 2^k - 1, 1 + 2(2^k - 1), 1 + 3(2^k - 1), \dots$ , then
  - sort items #2,  $2 + 2^k - 1, 2 + 2(2^k - 1), 2 + 3(2^k - 1), \dots$ , then
  - etc.
  - sort items # $2^k - 2, 2(2^k - 1) - 1, 3(2^k - 1) - 1, \dots$ ,
  - Each time an item moves, can reduce #inversions by as much as  $2^k + 1$ .
- Now sort subsequences of elements  $2^{k-1} - 1$  apart:
  - sort items #0,  $2^{k-1} - 1, 2(2^{k-1} - 1), 3(2^{k-1} - 1), \dots$ , then
  - sort items #1,  $1 + 2^{k-1} - 1, 1 + 2(2^{k-1} - 1), 1 + 3(2^{k-1} - 1), \dots$ ,
  - :
- End at plain insertion sort ( $2^0 = 1$  apart), but with most inversions gone.
- Sort is  $\Theta(N^{1.5})$  (take CS170 for why!).

## Example of Shell's Sort



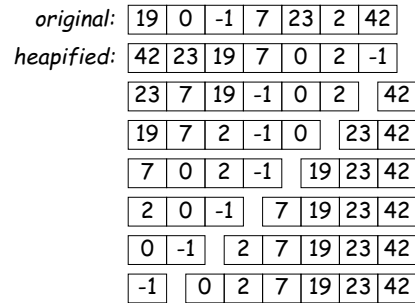
I: Inversions left.

C: Comparisons needed to sort subsequences.

## Sorting by Selection: Heapsort

**Idea:** Keep selecting smallest (or largest) element.

- Really bad idea on a simple list or vector.
- But we've already seen it in action: use heap.
- Gives  $O(N \lg N)$  algorithm ( $N$  remove-first operations).
- Since we remove items from end of heap, we can use that area to accumulate result:



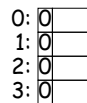
## Merge Sorting

**Idea:** Divide data in 2 equal parts; recursively sort halves; merge results.

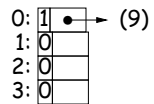
- Already seen analysis:  $\Theta(N \lg N)$ .
- Good for external sorting:
  - First break data into small enough chunks to fit in memory and sort.
  - Then repeatedly merge into bigger and bigger sequences.
  - Can merge  $K$  sequences of arbitrary size on secondary storage using  $\Theta(K)$  storage.
- For internal sorting, can use binomial comb to orchestrate:

## Illustration of Internal Merge Sort

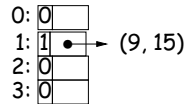
L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)



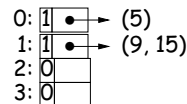
0 elements processed



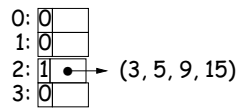
1 element processed



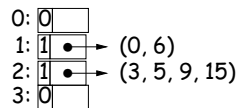
2 elements processed



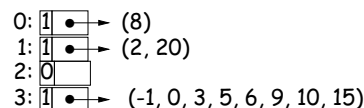
3 elements processed



4 elements processed



6 elements processed



11 elements processed

## Quicksort: Speed through Probability

**Idea:**

- Partition data into pieces: everything  $>$  a pivot value at the high end of the sequence to be sorted, and everything  $\leq$  on the low end.
- Repeat recursively on the high and low pieces.
- For speed, stop when pieces are "small enough" and do insertion sort on the whole thing.
- Reason: insertion sort has low constant factors. By design, no item will move out of its will move out of its piece [why?], so when pieces are small, #inversions is, too.
- Have to choose pivot well. E.g.: median of first, last and middle items of sequence.

## Example of Quicksort

- In this example, we continue until pieces are size  $\leq 4$ .
- Pivots for next step are starred. Arrange to move pivot to dividing line each time.
- Last step is insertion sort.

16 10 13 18 -4 -7 12 -5 19 15 0 22 29 34 -1\*

-4 -5 -7 -1 18 13 12 10 19 15 0 22 29 34 16\*

-4 -5 -7 -1 15 13 12\* 10 0 16 19\* 22 29 34 18

-4 -5 -7 -1 10 0 12 15 13 16 18 19 29 34 22

- Now everything is "close to" right, so just do insertion sort:

-7 -5 -4 -1 0 10 12 13 15 16 18 19 22 29 34

## Performance of Quicksort

- Probabalistic time:
  - If choice of pivots good, divide data in two each time:  $\Theta(N \lg N)$  with a good constant factor relative to merge or heap sort.
  - If choice of pivots bad, most items on one side each time:  $\Theta(N^2)$ .
  - $\Omega(N \lg N)$  in best case, so insertion sort better for nearly ordered input sets.
- Interesting point: randomly shuffling the data before sorting makes  $\Omega(N^2)$  time very unlikely!

## Quick Selection

**The Selection Problem:** for given  $k$ , find  $k^{\text{th}}$  smallest element in data.

- Obvious method: sort, select element # $k$ , time  $\Theta(N \lg N)$ .
- If  $k \leq$  some constant, can easily do in  $\Theta(N)$  time:
  - Go through array, keep smallest  $k$  items.
- Get probably  $\Theta(N)$  time for all  $k$  by adapting quicksort:
  - Partition around some pivot,  $p$ , as in quicksort, arrange that pivot ends up at dividing line.
  - Suppose that in the result, pivot is at index  $m$ , all elements  $\leq$  pivot have indices  $\leq m$ .
  - If  $m = k$ , you're done:  $p$  is answer.
  - If  $m > k$ , recursively select  $k^{\text{th}}$  from left half of sequence.
  - If  $m < k$ , recursively select  $(k - m - 1)^{\text{th}}$  from right half of sequence.

## Selection Example

**Problem:** Find just item #10 in the sorted version of array:

Initial contents:

51 60 21 -4 37 4 49 10 40\* 59 0 13 2 39 11 46 31  
0

Looking for #10 to left of pivot 40:

13 31 21 -4 37 4\* 11 10 39 2 0 40 59 51 49 46 60  
0

Looking for #6 to right of pivot 4:

-4 0 2 4 37 13 11 10 39 21 31\* 40 59 51 49 46 60  
4

Looking for #1 to right of pivot 31:

-4 0 2 4 21 13 11 10 31 39 37 40 59 51 49 46 60  
9

Just two elements; just sort and return #1:

-4 0 2 4 21 13 11 10 31 37 39 40 59 51 49 46 60  
9

Result: 39

## Selection Performance

- For this algorithm, if  $m$  roughly in middle each time, cost is

$$\begin{aligned}
 C(N) &= \begin{cases} 1, & \text{if } N = 1, \\ N + C(N/2), & \text{otherwise.} \end{cases} \\
 &= N + N/2 + \dots + 1 \\
 &= 2N - 1 \in \Theta(N)
 \end{aligned}$$

- But in worst case, get  $\Theta(N^2)$ , as for quicksort.
- By another, non-obvious algorithm, can get  $\Theta(N)$  worst-case time for all  $k$  (take CS170).

## Better than $N \lg N$ ?

- Can prove that if all you can do to keys is compare them then sorting must take  $\Omega(N \lg N)$ .
- Basic idea: there are  $N!$  possible ways the input data could be scrambled.
- Therefore, your program must be prepared to do  $N!$  different combinations of move operations.
- Therefore, there must be  $N!$  possible combinations of outcomes of all the if tests in your program (we're assuming that comparisons are 2-way).
- Since each if test goes two ways, number of possible different outcomes for  $k$  if tests is  $2^k$ .
- Thus, need enough tests so that  $2^k > N!$ , which means  $k \in \Omega(\lg N!)$ .
- Using Stirling's approximation,

$$m! \in \sqrt{2\pi m} \left(\frac{m}{e}\right)^m \left(1 + \Theta\left(\frac{1}{m}\right)\right),$$

this tells us that

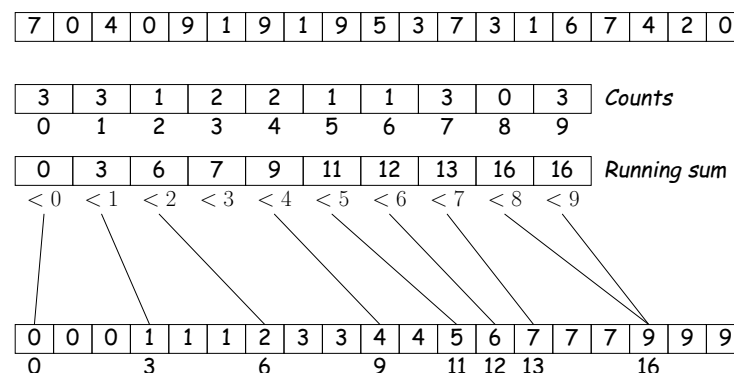
$$k \in \Omega(N \lg N).$$

## Beyond Comparison: Distribution Counting

- But suppose can do more than compare keys?
- For example, how can we sort a set of  $N$  integer keys whose values range from 0 to  $kN$ , for some small constant  $k$ ?
- One technique: count the number of items  $< 1, < 2, \dots$
- If  $M_p = \#$ items with value  $< p$ , then in sorted order, the  $j^{\text{th}}$  item with value  $p$  must be  $\#M_p + j$ .
- Gives linear-time algorithm.

## Distribution Counting Example

- Suppose all items are between 0 and 9 as in this example:



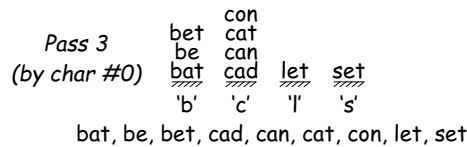
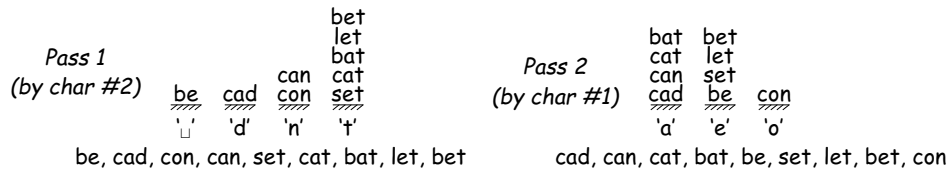
- "Counts" line gives # occurrences of each key.
- "Running sum" gives cumulative count of keys  $\leq$  each value...
- ... which tells us where to put each key:
- The first instance of key  $k$  goes into slot  $m$ , where  $m$  is the number of key instances that are  $< k$ .

## Radix Sort

**Idea:** Sort keys *one character at a time*.

- Can use distribution counting for each digit.
- Can work either right to left (LSD radix sort) or left to right (MSD radix sort)
- LSD radix sort is venerable: used for punched cards.

Initial: set, cat, cad, con, bat, can, be, let, bet



Last modified: Thu Oct 30 14:53:18 2008

CS61B: Lectures #27-28 21

## MSD Radix Sort

- A bit more complicated: must keep lists from each step separate
- But, can stop processing 1-element lists

A	posn
* set, cat, cad, con, bat, can, be, let, bet	0
* bat, be, bet / cat, cad, con, can / let / set	1
bat / * be, bet / cat, cad, con, can / let / set	2
bat / be / bet / * cat, cad, con, can / let / set	1
bat / be / bet / * cat, cad, can / con / let / set	2
bat / be / bet / cad / can / cat / con / let / set	

Last modified: Thu Oct 30 14:53:18 2008

CS61B: Lectures #27-28 22

## Performance of Radix Sort

- Radix sort takes  $\Theta(B)$  time where  $B$  is total size of the key data.
- Have measured other sorts as function of #records.
- How to compare?
- To have  $N$  different records, must have keys at least  $\Theta(\lg N)$  long [why?]
- Furthermore, comparison actually takes time  $\Theta(K)$  where  $K$  is size of key in worst case [why?]
- So  $N \lg N$  comparisons really means  $N(\lg N)^2$  operations.
- While radix sort takes  $B = N \lg N$  time.
- On the other hand, must work to get good constant factors with radix sort.

Last modified: Thu Oct 30 14:53:18 2008

CS61B: Lectures #27-28 23

## And Don't Forget Search Trees

**Idea:** A search tree is in sorted order, when read in inorder.

- Need *balance* to really use for sorting [next topic].
- Given balance, same performance as heapsort:  $N$  insertions in time  $\lg N$  each, plus  $\Theta(N)$  to traverse, gives

$$\Theta(N + N \lg N) = \Theta(N \lg N)$$

Last modified: Thu Oct 30 14:53:18 2008

CS61B: Lectures #27-28 24

## Summary

- Insertion sort:  $\Theta(Nk)$  comparisons and moves, where  $k$  is maximum amount data is displaced from final position.
  - Good for small datasets or almost ordered data sets.
- Quicksort:  $\Theta(N \lg N)$  with good constant factor if data is not pathological. Worst case  $O(N^2)$ .
- Merge sort:  $\Theta(N \lg N)$  guaranteed. Good for external sorting.
- Heapsort, treesort with guaranteed balance:  $\Theta(N \lg N)$  guaranteed.
- Radix sort, distribution sort:  $\Theta(B)$  (number of bytes). Also good for external sorting.