CS 2

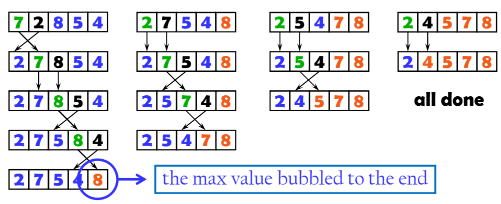
Introduction to Programming Methods

First Foray: Sorting

How do you sort a bunch of integers?

- setup: an array A of int, of size A.length
  - later on, we'll deal with arbitrary types, not just int
- now, how do we proceed?
  - not surprisingly, many algorithms to pick from
- let's start with a simple one: Bubble Sort
  - repeatedly step through the list, looking at pairs of adjacent items and swapping them if needed

Bubble Sort Live



- can you optimize it a bit?
  - do we always need to go to the end?

Bubble Sort Code [Java]

```java
public static void bubbleSort(int[] A)
{
  int i, temp;
  int n_remain = A.length-1;           // elements from 0 to n_left-1 are left to sort
  while (n_remain>0) {                 // until done
    last = 0;                          // initialize the last one swapped
    for (i = 0; i<n_remain-1; i++) {   // bubbling up
      if (A[i] > A[i+1]) {             // if out of order...
        temp = A[i];                   // ...then swap the two
        A[i] = A[i+1];
        A[i+1] = temp;
        last = i;        }             // & remember which one was last swapped
    }
    n_remain = last;                   // from n_left and up is sorted
  }
}
```


Questions to think about

Efficiency

- assuming that we only count #comparisons
  - what's the best case scenario?
  - what's the worst case scenario?
- can we do better than that?
  - what's the theoretical limit of #comparisons?

Generality

- "template" sorting code
  - that works for any type of value?
  - you just saw that last week...

Computational Complexity

How to analyze an algorithm?

- time to code, time to debug, and time to run
  - but different inputs/machine/memory size/coding details/... lead to different timings
- more abstract way: expected performance
  - without knowing the environment, or even the code!
  - interested in how *scalable* the algorithm is
    - think "order of magnitude" for large inputs
  - often counts number of operations for an input size of n
    - example: the simplest way to compute an average of n values will take about n operations (n-1 additions, 1 division)

Complexity Analysis Notation

Big-O notation

- only cares about most significant term in n
  - details do not matter for large n… (asymptotic behavior)
  - n+4 operations → O(n); n^2+3n ops → O(n^2); 10 ops → O(1); n!+4→ O(n!) and you are in trouble…
  - we'll say that an algorithm has constant / linear / quadratic / {…} complexity based on its O(.)
- more formally:
  - O(f(n)) means Time(n) ≤ C . f(n) for large enough n
  - Ω(g(n)) means C. g(n) ≤ Time(n) for infinitely many n
  - Θ(h(n)) means O(h(n)) and Ω(h(n))

What Is It Good For?

Predicting computational times

- if your algorithm is O(n) and it takes 1 second for n=1000, how long will it take for n=100,000?
  - ~100 s.
- what if it's O(n^2)?
  - ~10,000 s.
- …if nothing worse happens for large n's
  - running out of memory, you playing angry birds while it computes, etc…, can make it slower than expected

Additional Considerations

Let's go back to Bubble Sort

- what is its complexity?
- well, hard to say…
  - best case scenario? worst case scenario? average case?

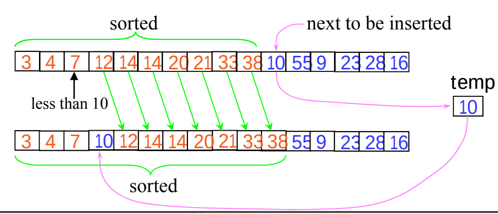
Usually, one uses worst case scenario

- n-1 + n-2 + n-3 +…+1 ops = ??
  - 1+2+3+…+n = n(n+1)/2 (easy to prove)
- So O(n^2) it is.
- best case? "average"?

Different Sorting, Better Sorting

Slew of quadratic-time sorting methods

- insertion sort
  - insert k^{th} element within first sorted (k-1) elements



Different Sorting, Better Sorting

Slew of quadratic-time sorting methods

- insertion sort
  - insert k^{th} element within first sorted (k-1) elements
- see also selection sort
  - select smallest and move to front; repeat

Now here's a stupid idea…

- n twice larger, complexity 4 times larger
- but half smaller → ¼ the time complexity
- so what about splitting the problem in 2?
  - and somehow "merge" the two halves quickly…

Bottom-up Merge Sort

A faster approach (suppose n=2^m for now)

- idea: split into smaller sets, then merge
  - m multiple passes; array divided into smaller subarrays of size 2^k (k=0…m-1) then adjacent subarrays are merged
  - merging two sorted lists is fast (complexity?)
  - so timing is improved!
    - code a bit messy (indices not trivial to get right the first time)
    - total complexity?

http://andreinc.net/2010/12/26/bottom-up-merge-sort-non-recursive/

But… much simpler if we use **recursion**

- divide and conquer—more on this later

Before then…

Assuming a comparison-based method… what is the best complexity we can get?

- for n numbers, how many permutations?
  - $n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1 = n!$
- the sorted list is only ***one*** of these n! combos
- each comparison kills half the permutations
  - e.g., for 1,2,3: (123), (132), (213), (231), (312), (321)
  - if A[1]<A[2], then we are left with (123), (132), (231)
  - like the "20 questions" game… (can find one out of 2^{20})
- so sorting is $\Omega(\log_2 n!) = \Omega(n \log n)$ (Stirling's)