More on sorting…

Mergesort (v2)
Quicksort

Mergesort “in place” in action

- Boxes with same color are in a single “run”
  - Specific color has no other meaning
- Runs get larger as the algorithm runs
  - Eventually, entire set is in one run!
- Algorithm works well with linked lists
  - No need to allocate extra arrays for merging!
Benefits of mergesort “in place”

- Algorithm may complete faster than standard mergesort
  - Requires fewer iterations if array is nearly sorted (lots of long runs)
  - Even small amounts of order make things faster
- No additional memory need be allocated
- No recursion!
  - Recursion can be messy if large arrays are involved
- Works well with linked lists
  - Standard mergesort is tougher with linked lists: need to find the “middle” element in a list
- May be less copying: simply rearrange lists

Quicksort: another recursive sort

- “Standard” mergesort requires too much memory
  - Extra array for merging
  - Alternative: use quicksort
- Basic idea: partition array into two (possibly unequal) halves using a pivot element
  - Left half is all less than pivot
  - Right half is all greater than pivot
- Recursively continue to partition each half until array is sorted
  - Elements in a partition may move relative to one another during recursive calls
  - Elements can’t switch partitions during recursion
How quicksort works

- Pick a pivot element
- Divide the array to be sorted into two halves
  - Less than pivot
  - Greater than pivot
  - Need not be equal size!
- Recursively sort each half
  - Recursion ends when array is of size 1
  - Recursion may instead end when array is “small”: sort using traditional O(n²) sort
- How is pivot picked?
- What does algorithm look like?

Quicksort: pseudocode

```c
quicksort (int theArray[], int nElem)
{
    // We're done if there's at most 1 element
    if (nElem <= 1)
        return;
    Choose a pivot item p from theArray[]
    Partition the items of theArray about p
    Items less than p precede it
    Items greater than p follow it
    p is placed at index pIndex
    // Sort the items less than p
    quicksort (theArray, pIndex);
    // Sort the items greater than p
    quicksort (theArray+pIndex+1, nElem-(pIndex+1));
}
```

Key question: how do we pick a “good” pivot (and what makes a good pivot in the first place)?
Picking a pivot

- Ideally, a pivot should divide the array in half
  - How can we pick the middle element?
- Solution 1: look for a “good” value
  - Halfway between max and min?
  - This is slow, but can get a good value!
  - May be too slow...
- Solution 2: pick the first element in the array
  - Very fast!
  - Can result in slow behavior if we’re unlucky
- Most implementations use method 2

Quicksort: code

```c
quicksort (int theArray[ ], int nElem) { 
  int pivotElem, cur, tmp;
  int endS1 = 0;
  // We’re done if there’s at most 1 element
  if (nElem <= 1) return;
  pivotElem = theArray[0];
  for (cur = 1; cur < nElem; cur++) {
    if (theArray[cur] < pivotElem) {
      // Swap cur and endS1 after incrementing endS1
      tmp = theArray[++endS1];
      theArray[endS1] = theArray[cur];
      theArray[cur] = tmp;
    }
  }
  theArray[0] = theArray[endS1];
  theArray[endS1] = pivotElem;
  // Sort the two parts of the array
  quicksort (theArray, endS1);
  quicksort (theArray+endS1+1, nElem-(endS1+1));
}
```
How fast is quicksort?

- Average case for quicksort: pivot splits array into (nearly) equal halves
  - If this is true, we need $O(\log n)$ “levels” as for mergesort
  - Total running time is then $O(n \log n)$

- What about the worst case?
  - Pick the minimum (or maximum) element for the pivot
  - $S_1$ (or $S_2$) is empty at each level
  - This reduces partition size by 1 at each level, requiring $n-1$ levels
  - Running time in the worst case is $O(n^2)$!

- For average case, quicksort is an excellent choice
  - Data arranged randomly when sort is called
  - May be able to ensure average case by picking the pivot intelligently
  - No extra array necessary!