Sorting Algorithms

Lecture 7

CS321

Todays Lecture

- Assignment 2 is due Tuesday Feb 17th 11:59pm
- HW1 out on Monday Feb 15th.
- Monday 15th is President's Day: no class.

Sorting Compared

Method/Structure	Space	Complexity	Time
Insertion Sort/Array	O(n)	Simple	$O(n^2)$
Quicksort/Array	O(2*n)	Complex	$O(n^2)$
Heapsort/Tree	O(n)	Complex	O(n*log(n))
Mergesort/Array	O(2*n)	Simple	O(n*log(n))
Counting Sort/Array	O(2*n)	Simple	O(3*n)

MergeSort

- Pseudocode for Mergesort

```
void mergesort(A,i,j)
{
   // handle base case
   if ((j-i) == 1)
   {
     if (A[i] > A[j]) swap(A[i],A[j]);
     return;
   }
   mid = (j-i)/2 + i; // calculate mid point

   mergesort(A,i,mid); // sort the left part of the array
   mergesort(A,mid+1,j); // sort the right half of the array
   combine(A,i,mid,j);  // combine the two sublists

}
```

Bolt Sort – AKA Quicksort

- Problem: Sorting a set of Nuts and Bolts
- You have N nuts, and N bolts.
- For every bolt there is a matching nut.
- You can test a bolt against a nut.
- You can not compare a bolt to a bolt.
- You can not compare a nut to a nut.
- What is an efficient way to match all bolts and nuts?

Quicksort

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        if p > 0 then
            quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Why?

- Why would we use Quicksort versus mergesort?
- It’s average case performance is typically n*logn
- For a large majority of cases it actually completes faster than mergesort and heapsort.
- It’s typical constant is better.

Counting Sort

- Pseudocode for Counting Sort

- Int [] countsort(A, length) // not a recursive method
- {
- // create count array size of A.
- counter = new int[10]; // numbers are limited to what range?
-
- // now we count the numbers
- for i = 0 to length:
- counter[A[i]] += 1;
-
- // no we convert counter to a rolling count.

  – for i = 1 to length:
- counter[i] = counter[i] + counter[i-1];

- // now do the sort
- //
- for i = length downto 0:
- counter[A[i]] -= 1; B[counter[A[i]] = A[i];
- }

Radix Sort

- Sort by using multiple passes of counting sort on the individual digits of the numbers to be sorted.

- Requires that our counting sort be ‘stable’.
- This means counting sort does not alter the order of digits that are equal.