Lecture III. Recurrence Reminder & Sorting

In the previous lecture we saw two recurrences: T(n) = 4 T(n/2) + O(n) and T(n) = 3 T(n/2) + O(n). For both of them, we computed the overall running time by analyzing the recurrence tree, and concluded that the overall running time was the same as the running time at the bottom level. In general, however, the amount of work at each subsequent level may not increase – it may stay the same, or even become smaller. In these two cases, the overall running time will be the sum of the runtimes at each level, and the runtime at the top level, respectively. Intuitively, this result makes sense: in algorithms we only worry about the running time of the bottleneck of the overall algorithm. The Master Theorem (which you learned in CSC 263) makes this intuition explicit:

Given a program whose running time is described by the recurrence $T(n) = a*T(n/b) + O(n^d)$, for some constants a > 0, b > 1, d >= 0 then:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

The proof of this theorem is pretty easy to see from the recursion tree: consider the level k. It will have a^k independent sub-problems, each of size n/b^k. So the total work will be

$$a^k*O(n/b^k)^d = O(n)^d*(a/b^d)^k$$

We will want the sum of these over all k:

$$O(n)^d * \Sigma\ (a/b^d)^k$$

Because the summation is a geometric series, its sum is easy to find. If it is decreasing ($b^d > a$, or $d > \log_b a$, the sum is just a constant times the first term, giving the $O(n^d)$ runtime. If the ratio is one, the amount of work done at each level stays constant, so the runtime is just $O(n^d)$ times the number of levels (log n). if The ratio grows, it is the last term that dominates, and the overall runtime becomes $O(n^{\log_b a})$.

We have already seen an example of the algorithm from the 3rd class (multiplication). An example from the first class, that you all know, but may not have thought about quite in this way is binary search: we look at an element in the array (spending time O(1)) and then recurse on either of the halves. This can be described by the recurrence T(n) = T(n/2) + O(1). Since a=1, b=2 and d=0, this is the 2nd case, giving us running time O(log n). Another such example is sorting. Most likely you have learned previously about the quicksort algorithm. As a quick refresher, the algorithm works by picking a random element of the array as the *pivot*. All of the elements less then the pivot are moved to the front of the array, and all of the elements greater are moved to the end.

Here I will present another similar algorithm: mergesort. Unlike quicksort, which works top-down, doing quite a bit of work to split the input into parts, while the merging of the individual results is

trivial, mergesort works bottom-up, with splitting done trivially, while the merge of the various parts is more time-consuming. Here is the pseudocode:

```
void mergesort (int* array, int size) { //int* is C for array of ints. It is
                                        // just a pointer to the first element
  if (size <= 1) return;
  mergesort(array, size/2);
  mergesort(&(array[size/2]), size – size/2); //This is nasty (yet efficient)
                                              // way of saying 2nd half.
  merge(array, size);
}
```

So we have sorted the left and right sides of the array. Now we need to merge the results into a single array with all of the values. We can do it using a simple single pass:

```
void merge(int* array, int size) {// first and second halves presorted
   int* helper = new int[size];
   int i=0, j = size/2, n=0;
   while (i < size/2 && j < size) {
      if (a[i] < a[j] || j == size) { helper[n] = a[i]; i++; n++; }
      else { helper[n] = a[j]; j++; n++; }
   }
   memcpy(array, helper, size*sizeof(int)); //copy stuff back
   delete helper; // in C/C++ you should remember to delete your garbage
}
```

That's it! The running time of this algorithm is O(n log n), as you can easily see from the recurrences. Now you may ask, why do we need to learn two types of sort – we already know one, so shouldn't we just teach you the *best* sort, and go on? Well, it turns out there is not such thing as the best sort. It (as many other things) depends exactly what you are trying to sort. Quicksort is typically faster if you are given an array of integers *in memory*, where you can quickly go to an arbitrary location very quickly. It also requires less memory – the merge method, you may have noticed, required a helper array, so if you are trying an array that barely fits in memory, mergesort is not a good idea. Counter-intuitively, it is a good idea when working with data that has *no hope of fitting in memory*. Why? Because data that does not fit in memory is stored on spinning metallic disks. They take a while to spin to the correct location for a particular bit to be read or written. However once they are at some location, they can keep reading/writing very quickly, including large amounts of data. Because mergesort does all of it's writing in adjacent positions, the sorting of files on disk can be implemented extremely efficiently. This is one of the main reasons that the unix sort command uses mergesort, rather than quicksort (which is faster for smaller inputs). Furthermore, even slower sorting techniques (for example insertion sort) is very useful for sorting small arrays (e.g. as a subroutine for on of the recursive sorts once the input is small), or for sorting arrays that are *almost* in order. The running time of insertion sort is bounded by the total distance that all of the elements need to go to transform the input array into the output array.

Now let us do an example from the 1st class of recurrences, where we do more work up-front then in any subsequent stage: given a set of numbers, finder their median. We'll actually solve a slightly more general form, called k^{th} order statistic, which returns the k^{th} largest from a set of numbers.

Again, we will use a divide and conquer strategy, quite similar to quicksort. The key intuition is that since we do not need to sort all of the elements, we don't need to recurse on both sides of the pivot: for example if we want the 50th element, and after doing the initial splitting it turned out that the pivot was the 37th, we now know that we want an element from the 13th element from the right side. Here is the code:

```
int selection (int* array, int size, int k) { // we want the kth elem.
   int pivot = array[random(size)];
   int rank = partition(array, size, pivot); // standard partition from qsort
   if (rank > k) { return selection (array, rank, k); }
   else {return selection(&(array[rank]) , size-rank, k-rank); }
}
```

What is the running time? The partition algorithm takes O(n) time, and there is only one subjob. The size of the subjob is not known, however it turns out (almost) not to matter! If we go back to the 3 rules, d = 1 and a = 1. So for any constant b > 0, $\log_b(1) = 0$. So the overall runtime is going to be O(n).

Back to sorting. In CSC 263 you have learned that we actually cannot sort an array of numbers in time less then O(n log n). This is mainly true. This cannot be done, unless you know something else about the numbers you are sorting, for example how they are distributed, or if they are from a particular limited range. If they are, you can use a time/memory tradeoff by making a table of all possible values and recording which ones have been seen:

```
int countingsort(char* a) {
   int i, j, counts[256];  //assume it is initialized to zero
   for (i =0; i < strlen(a); i++) {
      counts[a[i]]++; //in C, chars are just small numbers
   }
   for (i=0; i < 256; i++) {
      while (counts[i] != 0) {
         counts[i]--;
         a[j] = (char) i;
      }
   }
}
```

While counting sort is short and elegant, many of the other methods that try to take advantage of some features of the data are much more complicated. And usually not really necessary: the biggest saving that is possible is O(log n), since you will never have a running time which is less than linear. However O(log(n)) is really not that much. While technically it is not a constant, If you assume that you have a 1GB input dataset, log (1GB) = 30this may mean the difference between waiting for your program for a day versus a month, for the bulk of applications the difference will not be noticeable: if something has to be instantaneous, odds are the data is so small, that an extra log will not be noticeable, while if you have to wait for something very large to finish, you can often win back a large fraction of the runtime by taking advantage of many computers simultaneously. In practice, you will most often have access to many machines, and many of the algorithms we have discussed can be effectively sped-up by running your program on all of them simultaneously.