Algorithms in Systems Engineering
IE170

Lecture 8

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - CLRS Chapter 7
- References
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Quicksort

- We now discuss a sorting algorithm called *quicksort* similar to one that we saw in Lab 2.
- The basic quicksort algorithm is as follows.
  - Choose a partition element.
  - Partition the input array around that element to obtain two subarrays.
  - Recursively call quick sort on the two subarrrays.
- Here is code for the basic algorithm.

```
void quicksort(Item* data, const int l, const int r)
{
   if (r <= l) return;
   int i = partition(data, l, r);
   quicksort(data, l, i-1);
   quicksort(data, i+1, r);
}
```

Partitioning

- One big advantage of quicksort is that the partitioning (and hence the entire algorithm) can be performed in place.
- Here is an in place implementation of the partitioning function.

Analyzing Quicksort

- Questions to be answered
  - How do we prove the correctness of quick sort?
  - Does quicksort always terminate?
  - Can we do some simple optimization to improve performance?
  - What are the best case, worst case, and average case running times?
  - How does quicksort perform on special files, such as those that are almost sorted?

Importance of the Partitioning Element

- Note that the performance of the algorithm depends primarily on the chosen partition element.
- Some questions
  - What is the "best" partition element to select?
  - What is the running time if we always select the "best" partition element?
  - What is the "worst" partition element to select?
  - What is the running time in the worst case?
  - What is the running time in the average case?

Choosing the Partitioning Element

- We would like the partition element to be as close to the middle of the array as possible.
- However, we have no way to ensure this in general.
- If the array is randomly ordered, any element will do, so choose the last element (this was our original implementation).
- If the array is almost sorted, this will be disastrous!
- To even the playing field, we can simply choose the partition element randomly.
- How can we improve on this?

More Simple Optimization

- Note that the check `if (j == l)` in the partition function can be a significant portion of the running time.
- This check is only there in case the partition element is the smallest element in the array.
- Here again, we can use the concept of a sentinel, introduced in Lecture 4.
- If we place a sentinel at the beginning of the array, we avoid this check.
- Another approach is to ensure that the pivot element is never the smallest element of the array.
- If we use median-of-three partitioning, then the partition element can never be the smallest element in the array.

Average Case Analysis

- Assuming the partition element is chosen randomly, we can perform average case analysis.
- The average case running time is the solution to the following recurrence.

$$T(n) = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} T(k-1) + T(n-k)$$

along with $T(0) = T(1) = 1$.

- Although this recurrence looks complicated, it's not too hard to solve.
- First, we simplify as follows.

$$T(n) = n + 1 + \frac{2}{n} \sum_{1 \leq k \leq n} T(k-1)$$

Average Case Analysis (cont.)

- We can eliminate the sum by multiplying both sides by n and subtracting the formula for T(n-1).

$$nT(n) - (n-1)T(n-1) = n(n+1) - (n-1)n + 2T(n-1)$$

- This results in the recurrence

$$nT(n) = (n+1)T(n-1) + 2n$$

- The solution to this is in $\Theta(n \lg n)$.
- In fact, the exact solution is more like $1.39n \lg n$.
- This means that the average case is only about 40% slower than the best case!

Duplicate Keys

- Quicksort can be inefficient in the case that the file contains many *duplicate keys*.
- In fact, if the file consists entirely of records with identical keys, our implementation so far will still perform the same amount of work.
- The easiest way to handle this is to do *three-way partitioning*.
- Instead of splitting the file into only two pieces, we have a third piece consisting of the elements equal to the partition element.
- Implementing this idea requires a little creativity.
- How would you do it?

Small Subarrays

- Another way in which quicksort, as well as other recursive algorithms can be optimized is by sorting small subarrays directly using insertion sort.
- Empirically, subarrays of approximately 10 elements or smaller should be sorted directly.
- An even better approach is to simply ignore the small subarrays and then insertion sort the entire array once quick sort has finished.

Stack Depth

- An important consideration with any recursive algorithm is the depth of the call stack.
- Each recursive call means additional memory devoted to storing the values of local variables and other information.
- In the worst case, quicksort can have a stack as deep as the number of elements in the array.
- One way to deal with this is to ensure that the smaller of the two subarrays is processed first.
- This does not affect the correctness.
- Even this idea will not work in a truly recursive implementation without compiler optimization.
- The most memory-efficient implementation is a non-recursive one that explicitly maintains the stack of subarrays to be sorted.

A Nonrecursive Quicksort

```
#include <stack>
void quicksort(Item* data, int l, int r)
{
   stack<int> s();
   int m(0), n(0);
   s.push(l); s.push(r);
   while (!s.empty()){
      m = s.pop(); n = s.pop();
      if (n <= m) continue;
      int i = partition(data, m, n);
      if (m-1 > n-i){
         s.push(m); s.push(i-1); s.push(i+1); s.push(n);
      }else{
         s.push(i+1); s.push(n); s.push(m); s.push(i-1);
      }
   }
}
```