Data Structures and Algorithms — CSci 230
Chapter 7 — Sorting

Overview

- We will cover Sections 7.1-7.3 and 7.5-7.7.
- The focus of the discussion will be *Quick Sort* in Section 7.7
- All of the algorithms will be described as sorting an array with values in locations 0 to $n-1$ into *ascending order*.

Sorting algorithms already covered

- *Insertion Sort*, covered in the Chapter 2 notes, is an $O(n^2)$ worst-case time sort. What happens, however, when the list is nearly sorted?
- *Merge sort*, covered in the Chapter 1 notes and review problems, is an $O(n \log n)$ worst-case time sort. It incurs a substantial overhead due to recopying the array after the merge step. When applying (a modified) Merge Sort to sorting a linked list, recopying is not necessary.

Heap Sort

- We developed the basic algorithm using `Build_Heap` and `Delete_Min` in Lab 8.
- It is easy to see that this is an $O(n \log n)$ algorithm.
- Several implementation details differ from the implementation in the lab:
  - Heap ordering places the *maximum* value at the root (location 0 of the array).
  - The first "deleted" item is placed in sorted position at the end of the array by swapping the value in location $n-1$ with the value at the root (which is at location 0) and then running `Percolate_Down` assuming there are $n-1$ values in the heap.
  - This is repeated $n-1$ times: at iteration i the root value is swapped with the value at location $n-i-1$ prior to running `Percolate_Down` assuming there are $n-i-1$ values in the heap.

Exercise

Hand simulate several iterations of *Insertion Sort*, *Merge Sort* and *Heap Sort* on the following array

0	1	2	3	4	5	6	7	8	9
9.5	3.5	9.1	14.4	8.7	17.8	6.4	9.3	7.5	1.9

Quick Sort — basic version

- Quick Sort is the fastest known sorting algorithm on average for arbitrary data.
- We will examine several versions.
- In the basic version, an element of the array is chosen as a *pivot*, and the remaining elements are *partitioned* into those less than the pivot and those greater. These two partitions are then sorted recursively.
- This idea is realized in the attached C++ code.

Exercises

1. Hand simulate `B_Q_Sort` on the array

0	1	2	3	4	5	6	7	8	9
9.5	3.5	9.1	14.4	8.7	17.8	6.4	9.3	7.5	1.9

   up until the first recursive calls are made. What are these recursive calls?

2. In the function `B_Q_Sort`, suppose we replaced the inner two `while` loops with the statements

   ```
   while( A[ i ] < Pivot ) i++;
   while( A[ j ] > Pivot ) j--;
   ```

   Would the function still work properly? As a hint, suppose all values from `A[Left]` to `A[Right]` are equal.

Analysis of Quick Sort

- Let $T(n)$ denote the number of array comparisons used in Quick Sort when $Right - Left + 1 = n$, and let k be the final position of the pivot after `Swap` outside the `for` loop. Then, we will show in class that

$$T(n) = T(k) + T(n-k-1) + n$$

- From this recursive equation, we will show that on average $T(n) = O(n \log n)$. You should be able to follow what's going on in the analysis, though it is not expected that you could have come up with it on your own.

Exercise

Analyze the worst-case for $T(n)$. First try to do this based on intuition and then more carefully. What condition of array `A` causes this?

Median-of-Three Quick Sort

- The worst-case behavior of Quick Sort can be made extremely rare, but not entirely eliminated, by different strategies for selecting the pivot:
  - choosing the pivot location randomly,
  - choosing the median of the left, middle and right values in the interval to be sorted.

  We will discuss the second in detail.
- Further speed enhancements may be obtained by stopping Quick Sort when the size of the interval is too small and then running Insertion Sort to bring array into final sorted order.
- These improvements are realized in the `Quick_Sort` function attached to the notes.

Exercises

1. Hand simulate `Q_Sort` with `Left=0` and `Right=14` on the following array. What are the recursive calls? Stop your simulation after identifying the recursive calls.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
90.1	16.2	50.5	40.7	15.0	89.7	36.8	3.0	14.2	30.9	28.6	50.1	32.3	21.1	27.5

2. Why does `Q_Sort` use `i` to swap at the end (with `Right-1`) whereas `B_Q_Sort` uses `j` to swap at the end (with `Left`)?

3. How would you modify `Q_Sort` to make it non-recursive?

Quick Select

- Some problems in statistics require finding the median or the kth ordered element of an array.
- The simplest thing to do would be to sort the array and just take the kth.
- Improved speed may be obtained by modifying Quick Sort to concentrate only on the partition that *must* contain the kth element. This results in an algorithm called "Quick Select", which runs in average case time $O(n)$.
- After looking at the code for Quick Select we will compare various algorithms experimentally.

Review Problems

1. The second version quick sort discussed in class uses the median-of-three technique to find the pivot and does not continue sorting when the length of the sublist (the difference between `Right` and `Left` is less than `Cutoff`). After this algorithm completes, insertion sort is run over the resulting array to bring the values into their final sorted position. What is the worst-case time required by insertion sort in this case? Justify your answer.

2. Here is a slightly different version of the Quick Select algorithm, which is much closer to the original version of Quick Sort. At the end of the function (including all recursive calls), the desired value will be in the kth location of the array.

```
Q_Select_One ( int A[ ], const int k, const int Left, const int Right )
{
    if( Left < Right ) {
        int Pivot = A [Left];
        unsigned int i = Left, j = Right + 1;

        for( ; ; ) {
            while( A[ ++i ] < Pivot && i<Right );
            while( A[ --j ] > Pivot );
            if( i < j )
                Swap( A[ i ], A[ j ] );
            else
                break;
        }
        Swap( A[ j ], A[ Left ] ); // Move pivot to sorted position.

        if( k < j )       Q_Select_One( A, k, Left, j-1 );
        else if( k > j )  Q_Select_One( A, k, j+1, Right );
    }
}
```

   (a) Assume the original call is made to this function with

   `Q_Select_One( A, 3, 0, 8)`

   with the following contents of `A`

0	1	2	3	4	5	6	7	8
14	24	8	16	32	71	26	25	12

   Show the contents of `A` near the end of the code, just before entering the conditional to check if a recursive call should be made. What recursive call, if any, is made?

   (b) What is the worst-case number of comparisons in `Q_Select_One` as a function of both n and k, where n is the value of `Right-Left+1` in the first call? When does this occur?

3. Solve the Quick Sort recursive function

$$T(n) = T(k) + T(n - k - 1) + n - 1$$

to yield a non-recursive form when $k = 1$. Assume that $T(0) = T(1) = 0$ and $T(2) = 2$. You may assume that n is even.