Recurrence (Rosen, 6th edition, Sections 7.1, 7.3 – related to coding) Carol Zander

Recurrence relationships

A recurrence relationship is an equation for a sequence of numbers, where each number (except for the base case) is given in terms of previous numbers in the sequence. These are useful for determining the complexity of recursive algorithms. We will only consider recurrence equations from the standpoint of recursive programming.

Consider factorial:

```
int fact(int n) {
   if (n < 0) return -1;
   if (n <= 1) return 1;
   return n * fact(n-1);
}
```

Let T(n) be the running time for fact(n). If n ≤ 1, the running is constant, O(1). If n > 1, T(n) depends upon a version of itself, but with a smaller value. This is expressed using a recurrence equation:

$$T(n) = \begin{cases} O(1) & \text{if } n \le 1 \\ O(1) + T(n-1) & \text{if } n > 1 \end{cases}$$

Consider n > 1:

```
T(n) = 1 +  T(n-1)
           \______/
            1 +  T(n-2)
                \______/
                 1 +  T(n-3)
                     \______/
                      1 +  T(n-4)
                          \______/

                             . . .
                                1 + T(2)
                                   \____/
                                    1 + T(1)
                                       \___/
                                         1

     = 1 + 1 + 1 + 1 + . . . + 1
       \_____________________/
              n of them

T(n) = O(n)
```

Consider the iterative solution:

```
int fact(int n) {
   if (n < 0) return -1;
   int answer = 1;
   for (int i = 2; i <= n; i++)
      answer *= i;
   return answer;
}
```

This is also O(n), but with much less overhead so its constant, its c in the big-O definition, is smaller.

Now consider quickSort (hoareSort):

```
partition(a, pivot, up, down);
swap(a[up], a[high-1]);            // put pivot in rightful position
quickSort(a, low, up-1);
quickSort(a, up+1, high);
```

Consider the analysis, assuming a random pivot (no median-of-three partitioning) and no cutoff for small arrays. The running time is equal to the linear time spent in the partitioning plus the running time of the two recursive calls. (The pivot selection takes only constant time.) If the pivot is at position i , then the first recursive call acts on i elements, and the second recursive call acts on n-i minus another one for the pivot. This gives the recurrence equation

```
        /  O(1)                      if n <= 1
T(n) = {
        \  O(n) + T(i) + T(n-i-1)    if n > 1
```

Worst-case Analysis

The pivot is the smallest element all the time. Then i =0. Consider when n > 1:

```
T(n) = n +  T(n-1)
           \______/
            (n-1) + T(n-2)
                   \______/
                    (n-2) + T(n-3)
                           \______/
                             . . .
                                  2 + T(1)
                                     \___/
                                      1
```

Adding it all up yields

$$T(n) = \sum_{i=1}^{n} i = n(n+1)/2 = O(n^2)$$

Best-case Analysis

In the best case, the pivot is in the middle. To simplify the math, assume a larger $n = 2^k$, if necessary, so that the two halves of the array are each exactly half of the size of the original array. Include the pivot in one of the halves of the array. Consider when n > 1:

T(n) = n + T(n/2) + T(n/2)

```
T(n) = n +  T(n/2)                +          T(n/2)
           \______/                          \______/
   n/2 + T(n/2)  +   T(n/2)                n/2 +   T(n/2)  +  T(n/2)
         \____/       \____/                        \____/     \____/
n/4+T(n/4)+T(n/4)  n/4+T(n/4)+T(n/4)     n/4+T(n/4)+T(n/4)  n/4+T(n/4)+T(n/4)
         ...
```

$$T(n) = n + 2(n/2) + 4(n/4) + 8(n/8) + \ldots + 2^k(n/2^k)$$

$$= n + n + n + n + \ldots + n$$

(k = log n of them)

So T(n) = O(n log n)

Average-case Analysis (you are not responsible for this proof)

For the average case, assume that each of the sizes for the subarray is equally likely, and therefore has probability 1/n. This assumption is valid for our pivoting and partitioning strategy (it may not be valid for other implementations that do not preserve randomness of the subarrays). With this assumption, the average value of T(i) and T(n - i - 1) (where i is the number of elements in the subarray of the first recursive call) are both

$$(1/n) \sum_{j=1}^{n-1} T(j)$$

$$T(n) = (2/n) \left(\sum_{j=1}^{n-1} T(j) \right) + n$$

The recurrence equation when n > 1.
The first term is the T(i)+T(n-i+1)

$$nT(n) = 2 \left(\sum_{j=1}^{n-1} T(j) \right) + n^2$$

Multiply both sides by n.

To remove the summation, to telescope, we also need T(n) with its last summation term taken out, and T(n-1):

$$nT(n) = 2 \left(\sum_{j=1}^{n-2} T(j) \right) + 2T(n-1) + n^2$$

Take out last summation term.

$$(n-1)T(n-1) = 2 \left(\sum_{j=1}^{n-2} T(j) \right) + (n-1)^2 = 2 \left(\sum_{j-1}^{n-2} T(j) \right) + n^2 - 2n + 1$$

Subtract the second equation from the first equation:

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

Rearrange and drop the insignificant big-O-wise 1 (which is really O(1)) at the end:

$$nT(n) = (n-1+2)T(n-1) + 2n$$
$$nT(n) = (n+1)T(n-1) + 2n$$
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

Now telescope

```
T(n)/(n+1) =      T(n-1)/n + 2/(n+1)
                  \_______/
                 T(n-2)/(n-1)+2/n          by finding T(n-1)
                 \___________/
                T(n-3)/(n-2)+2(n-1)        by finding T(n-2)
                . . .
               T(2)/3 which is T(1)/2 + 2/3
           =   T(1)/2 + 2/(n+1) + 2/n + 2/(n-1) + 2/(n-2) + ... + 2/3
```

$$T(n)/(n+1) = T(1)/2 + 2\sum_{i=3}^{n+1} 1/i$$

The sum is about $\log_e(n+1) + \gamma - 3/2$, where $\gamma \approx 0.577$ (Euler's constant),
so T(n)/(n+1) = O(log n) and T(n) = O(n log n) .