CS211 Recitation: 27-29 April 2004
Analysis of execution time

We want to go over three things that students do not seem to grasp fully.

1. Definition of O(n) and how o use it formally.
2. Figuring out the execution time of a loopy program segment.
3. Figuring out the execution time of a recursive method.

1. Definition of O(n).

Students, you have to memorize this definition and be able to *use* it. See also Weiss chap. 5.4

Definition: Function f(n) is O(g(n)) if there are positive constants c and N0 such that

$$f(n) \le c*g(n)$$

for all n > N0.

Interpretation: Draw the curves f(n) and c*g(n) for positive n in the x-y plane. After point n = N0, the c*g(n) curve will ever after be above the f(n) curve.

Problems: You should be able to determine these by using the formal definition. We give one example.

1. Prove that function n+5 is O(n).

2. Prove that 2*n+5 is O(n).

3. Prove that $.5*n^2 + n$ is $O(n^2)$.

4. Prove that $5*n^2$ is $O(n^4)$.

5. Prove that 1/n is O(1).

6. Prove that log(n) is O(n).

7. Prove that n*log(n) is $O(n^2)$.

8. Prove that $8*n^2$ is $O(n^3 + n)$

9. Prove that $8*n^2 + 1000n$ is $O(n^2)$.

10. Make up 10 of your own interesting things (like the above 9) to prove or disprove.

We prove one of them here, number 2:
Theorem:

2*n+5 is O(n).

We have to find positive c and N0 such that for all n > N0,

2*n + 5 <= c*n

Choosing c = 2 doesn't work, because then it reads 2*n + 5 <= 2*n , which is false for all n > 0, so we have to choose a bigger c. Try c = 3. We have:

```
        2*n + 5 <= 3*n
=         <subtract 2*n from both sides>
        5 <= n.
```

This holds for all n>4, so choose N0 = 4.
Q.E.D. (Quit End Done)

2. Loopy program segments.

We deal only with worst-case or best-case execution time. Not average or expected time.

We should be able to find the order of execution time of a loop fairly easily. Consider a loop like this (it could also be a for-loop):

```
int k= 0;
while (expression) {
        S
        k= k+1;
}
```

We have to figure out the time statement S takes when k = 0, when k = 1, when k = 2, etc. and add these times up. Here, we may count the number of assignments in S, the number of array comparisons whatever seems to be the dominant part of S.

If S takes constant time —i.e. the time it takes does not depend on k—then the time is simply

(number of iterations) * (time for S)

Nested loops can complicate matters. You have to read carefully and understand the algorithm. We give two examples:

Consider insertion sort:

```
// Sort b[0..n-1]
// invariant: b[0..k-1] is sorted.
for (k= 0; k !=  n, k= k+1) {
    insert b[k] into its sorted position in the
    already sorted segment b[0..k-1]
}
```

The best case time occurs when the inner loop takes constant time, and this happens if the array is already in ascending order.

The worst case time occurs when the array is in descending order, because inserting b[k] into its sorted position takes time O(k). Add up 1 + 2 + 3 + … + n and you get n*(n+1)/2, giving O(n*n).

The next example shows the extreme care you must take in analyzing a program segment. Just because there are nested loops, the execution time is not necessarily n*n! Careful study of the following program segment will show that, even

though it has nested loops, the time is still O(n). Why?

```
/* Store in x the number of sections of equal elements of array b[0..n-1], for n>=0. For example, for the array b = (3,5,5,3,3,3,3,6,6,6), the value 4 is stored in x: there is a section of (one) 3's, then a section of 5's, then a section of 3's, and finally a section of 6's. */

int x= 0;  int k= 0;
/* inv P: 0 <= k <= n.
   x = no of sections of equal elements in b[0..k-1].
   the following holds:
        k=0  or  k=n  or  b[k-1] != b[k] */
while (k != n) {
   x= x+1;

   // Let v be the value in b[k] at this point.
   //  Increase k until either k = n  or  b[k] != v
   k= k+1;
   // invariant: 1 <= k <= n,   and
   // x = no. sections of equal elements in b[0..k-1],
   while (k != n  &&  b[k-1] = b[k]) {
      k= k+1;
   }
}
// R: x = no. sections of equal elements in b[0..n-1]
```

We will not give you a bunch of program segments to practice on (in determining their execution time). You have examples already. Look at all the algorithms we have developed in class and recitation. Look at algorithms in Weiss. Spend some time figuring out their order execution time.

3. Figuring out time of recursive methods

There is a formal and an informal way of figuring out the time required by a recursive function. The formal way requires coming up with a recurrence relation that defines the time and then solving the recurrence relation. We will not ask you to do this, but here is an example. (There was another example with mergesort in recitation 8.)
Consider this method.

```
/** Process each node of t in inorder */
public static void p(Tnode t) {
      if (t == null)
            return;
      p(T.left);
      process root t;
      p(T.right);
}
```

Assume that "process root t takes time K for some integer K (it takes constant time). The time for the empty tree is a constant K2. Then the time T(n) for a tree of n nodes can be written as follows:

$$T(0) = K2$$

$$T(n) = K2 + T(\text{size of left tree}) + K + T(\text{size of right tree})$$

One now has to solve this recurrence relation. We don't go into this here, because we do not require you to do it. That is a topic of CS280.

The less formal way of figuring out execution time of a recursive method is to judge the time based on what the method is doing (and how).

Consider the same algorithm. It is evident that each node of the tree is processed once. If the time to process a node is constant, then the algorithm takes time proportional to n, the number of nodes in the tree.

We had precisely this kind of problem on prelim 2, and many students got it wrong.

Consider this recursive function:

```
/** Set each b[k] of b[0..k] to sum of b[0..k].
    Example: sumit(b, 3) changes
    b = (3, 2, 4, 5, 1) to (3, 5, 9, 14, 1) */
public static void sumit(int[] b, int k) {
      if (k <= 0)  return;
      sumit(b, k-1);
      b[k]= b[k-1] + b[k];
}
```

The if-statement and assignment take constant time, say T. Each recursive call has its second argument reduced by 1, and we can see that there are calls with second argument k, k-1, k-2, …, 1, 0. Thus, the time it takes O(T*(k+1)), which is O(k) since T is a constant. It is a linear algorithm.

You have written many recursive methods in this class. Look at them and figure out their execution time. Here's one from assignment A5.

```
/** #12 = a list that represents the intersection of
    l1 and l2. precondition: no dups in l1 and l2 */
public static RList intersect(RList l1, RList l2){
    if (isEmpty(l1)) return null;
    if (!isMember(first(l1),l2))
        return intersect(rest(l1),l2);
    return prepend(first(l1),intersect(rest(l1),l2));
}
```

The membership test takes worst case time proportional to the length of l2. In total, this test is made for each element of l1. Hence, the time of this algorithm is at least O((length l1)*(length l2)).