Lecture 10: Recursion Analysis & Binary Search

PIC 10B

Todd Wittman

Recursion Analysis

- Analyzing the running time T of a recursive algorithm is generally harder than analyzing a non-recursive one.
- The most common strategy is to write the run time as a function of N: T(N). This indicates the time needed to process N items.
- By tracing carefully through the recursion, we can write down a recurrence relation for the algorithm. For example,

  $T(N) = T(N-1) + 1$
- Then we repeat the recurrence

  $T(N) = [T(N-2)+1] +1 = T(N-2) + 2$

  $T(N) = [T(N-3)+1] + 2 = T(N-3) + 3$
- Look for a pattern: $T(N) = T(N-k) + k$
- By tracing the pattern all the way to the base case T(1), we can determine the running time of the algorithm.

  $T(N) = O(N)$

Analyzing the Factorial

- Last class, we computed the factorial recursively.

```
int factorial (int N) {
        if (N==0) return 1;
        return N*factorial(N-1);
}
```

- Checking the if statement and multiplying by N is O(1) operation.

  Time for factorial(N) = Time for factorial(N-1) + O(1)

- Dropping the Big O for the moment, we get the recurrence:

$$T(N) = T(N-1) + 1$$

Analyzing the Factorial

- Repeatedly plug in the recurrence.

$$T(N) = T(N-1) + 1$$
$$= T(N-2) + 1 + 1 = T(N-2) + 2$$
$$= T(N-3) + 1 + 2 = T(N-3) + 3$$

- So the pattern is:

  $T(N) = T(N-k) + k$

- Let k = N.

  $T(N) = T(0) + N$

- T(0) is the time to compute the base case factorial(0), which is just O(1).

  $T(N) = 1 + N = O(N)$

- The non-recursive version also runs in O(N) time.

Analyzing the Palindrome

- Last class we saw the algorithm for checking for palindromes like "racecar".

```
bool isPalindrome (string s)  {
   if (s.length() <= 1)  return true;
   return (s[0]==s[s.length()-1])
        && isPalindrome(s.substr(1,s.length()-2));
 }
```

- What is the recurrence for T(N) with N=#letters?

$$T(N) = T(N-2) + 1$$

- What's the pattern?

$$T(N) = T(N-2k) + k$$

- Letting k = N/2, we get T(N) = O(N).

Sec 10.6: Fibonacci's Rabbits

- Last class we showed how to compute the Nth Fibonacci number.

```
int Fibonacci (int N) {
   if (N==1) return 1;
   if (N==2) return 1;
   return Fibonacci(N-1) + Fibonacci(N-2);
}
```

- What's the recurrence?

$$T(N) = T(N-1) + T(N-2) + 1$$

- The time recurrence looks just like the Fibonacci recurrence relation!

Fibonacci's Rabbits

- Sometimes we can't write out the full pattern, but we can still figure out how many computations we're making.

$T(N) = T(N-1) + T(N-2) + 1$ Lvl1

$= [T(N-2)+T(N-3)+1]+[T(N-3)+T(N-4)+1]+1$

$= T(N-2) + T(N-3) + T(N-3) + T(N-4) + 3$ Lvl2

- If we repeat the recurrence, we're going to get 8 T's on level 3. Then 16, 32, and so on...
- So we get 2^k T's at level k.
- To get down T(N-1) to the base case T(2), we'll need to go to level k = N-2.
- We'll have 2^{N-2} T's there, so $T(N) = O(2^N)$.

Permutations

- Ready for the monster?

```
vector<string> permute (string word) {
    vector<string> result;
    if (word.length() == 1) {
        result.push_back(word);
        return result;
    }
    for (int i=0; i<word.length(); i++) {
        string shorter = word.substr(0,i)
                + word.substr(i+1,word.length()-i-1);
        vector<string> shortWords = permute(shorter);
        for (int j=0; j<shortWords.size(); j++) {
            string longer = word[i]+shortWords[j];
            result.push_back(longer);
        }
    }
    return result;
}
```

O(1)

Loops N times

T(N-1)

size=(N-1)!

$T(N) = N\,T(N-1) + N!+1$

Permutations

- Repeat the recurrence...

T(N) = N T(N-1) + N! + 1
= N [(N-1) T(N-2) + (N-1)! + 1] + N! + 1
= N (N-1) T(N-2) + 2 N! + 2

- Repeat once more....

T(N) = N (N-1) [(N-2) T(N-3) + (N-2)! + 1] + 2 N! + 2
= N (N-1) (N-2) T(N-3) + 3 N! + 3

- So the pattern is...

T(N) = N (N-1)...(N-k+1) T(N-k) + k N! + k

- Let k=N to get down to base case N=0 letters.

T(N) = N (N-1)...1 T(0) + N N! + N = N! + N N! + N

- Run time T(N) = O(N*N!)
- If you think about how many letters we have to print out, this is the bare minimum running time.

Sec 11.6: Linear Search

- Suppose we want to find the position of the first occurrence of a value in a integer vector.
- Let's return -1 if the number is not found.

```
int search (vector<int> v, int val) {
    for (int i=0; i < v.size(); i++)
        if (v[i] == val)
            return i;
    return -1;
}
```

- If there are N items in the list, this is an O(N) algorithm.
- This is called a *linear* or *sequential search*.
- If the list is unsorted, this is the best we can do.

Sec 11.7: Binary Search

- But what if the list is sorted?
- Certainly we don't have to check every word in the dictionary to look up a word.
- A basic strategy is to repeatedly halve our search range until we find the value: *binary search.*
- For example, look for 5 in the list of 9 #s below.

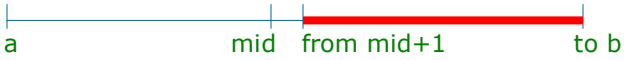
v = 1 1 3 5 8 10 18 33 42

- We would first start in the middle: 8
- Since 5<8, we know we can look at just the first half: 1 1 3 5
- Looking at the middle # again, narrow down to 3 5
- Then we stop when we're down to one #: 5

Sec 11.7: Binary Search

- So if we're searching in the range [a,b], first look at the middle position: mid = (a+b)/2

If val > v[mid], look at right half [mid+1,b].



If val < v[mid], look at left half [a,mid-1]

We've found val when we see val==v[mid].

If val is not in the list, we'll see the endpoints cross.

Sec 11.7: Binary Search

- There is a recursive procedure here.

```
int search (vector<int> v, int from, int to, int val) {
    if (from>to) return -1;   //val not found
    int mid = (from+to)/2;
    if (v[mid] == val)
        return mid;
    else if (val > v[mid])
        return search(v,mid+1,to,val);
    else
        return search(v,from,mid-1,val);
}
```

- Call function: int pos = search(v,0,v.size()-1,val);

Sec 11.7: Binary Search

- Recursion eats up memory because it has to create local copies of all the parameters.
- We have to pass parameters by value if they change at the recursion step.
- But does our dictionary vector v ever change?
- Changing it to pass by reference introduces a *significant* speed-up.

```
int search (const vector<int>& v, int from, int to, int val) {
```

- For this week's HW, I want you to figure out just how much speed-up you get with this little trick.

Sec 11.7: Binary Search

- Let's analyze the run time.
- It takes O(1) time to do the comparisons, then it cuts the search range in half.

$T(N) = T(N/2) + 1$

- Repeat the recurrence...

$T(N) = T(N/4) + 2$

$= T(N/8) + 3$

$\ldots = T(N/2^k) + k$

- Round up N to nearest power of 2: $N \leq 2^m$.

$T(N) \leq T(2^m/2^k)+k$

- Let $k = m$.

$T(N) \leq T(2^m/2^m)+m = T(1)+m = 1+m = O(m)$

- If $N=2^m$, then $m=\log N$. So $T(N) = O(\log N)$

Non-Recursive Binary Search

```
int non_recursive_binary_search (vector<int> v, int val) {
    int from = 0;
    int to = v.size()-1;
    int mid;
    while (from <= to) {
        mid = (from+to)/2;
        if (val == v[mid])
            return mid;
        else if (val > v[mid])
            from = mid+1;
        else
            to = mid-1;
    }
    return -1;
}
```

This version's running time is also O(logN), but it should be much faster than the recursive version of binary search.

For your HW...

- The binary search algorithm is easy to template. So you can search a vector of ints, doubles, strings, etc.
- For your HW, you have to write the isInDictionary() function for the text parsing routine.
- A linear search would be very slow: O(N).
- A binary search should be faster: O(logN).
- Theoretically, the binary searches are both O(logN) and the linear search is O(N). But Big O hides the constants.
- Should we use the recursive or non-recursive binary search?
- The search comparison program is online to help you decide.