Math 55 - Spring 2004 - Lecture notes # 12 - Mar 2 (Tuesday)

Keep Reading Sections 3.1 - 3.5 (not 3.6)
(we will not cover 3.5 in detail, since it appears elsewhere in EECS curriculum, but just do a few examples; you don't have to know sorting algorithms)

Homework, due Mar 10

(1) Let f(0)=0, f(1)=1, f(n)=f(n-1)+f(n-2) for n >= 2.
Let r1 = (1+sqrt(5))/2, r2 = (1-sqrt(5))/2.
Prove by induction that f(n) = (r1^n - r2^n)/sqrt(5)

(2) Let cost(n) be the number of additions needed to compute f(n) by the following recursive algorithm:

```
func f(n)
   if n=0 return(1)
   else if n=1 return(1)
   else return(f(n-1)+f(n-2))
```

Use induction to prove that cost(n) = f(n)-1, which means that the cost of the recursive algorithm grows very fast, O(r1^n), i.e. about O(1.62^n)

(3) Suppose g(1) = 7, g(2) = 8, and g(n)=2*g(n-1)-2*g(n-2) for n>2.
Derive a closed form formula for g(n).

(4) 3.3-16, 18, 50, 60, 62

(5) 3.4- 8, 14, 38, 48, 50, 62

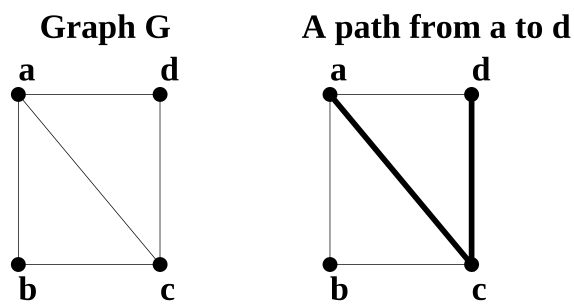
Goals for today: Continue induction proofs
Recursive functions

EX: So far we have been doing induction on numbers, showing P(n) is true if P(k) is true for numbers k smaller than n. But we can also do induction on other structures besides numbers, such as data structures that come up in programs. We illustrate with a structure called a tree, which is a special case of a graph.

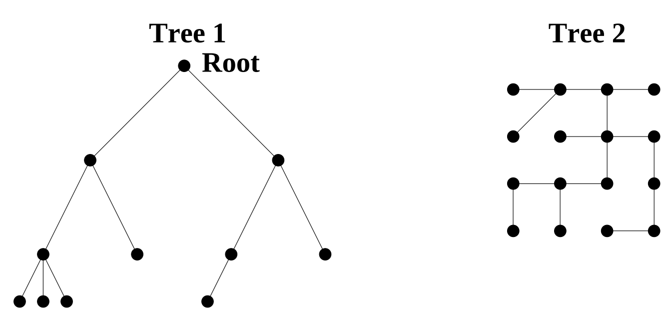
DEF A graph G = (V,E) consists of a nonempty set V of vertices, and a set E of edges connecting them. More formally, if a in V and b in V are vertices, then (a,b) in E means that there is an edge connecting a,b

EX: G = (V,E), V = {a,b,c,d}, E = {(a,b),(b,c),(c,d),(d,a),(a,c)}

```
DEF a path from node a to node c is a set of edges connected
    end to end starting at a and ending at c
EX: G as before, path from a to d consists of edges (a,c),(c,d)
```



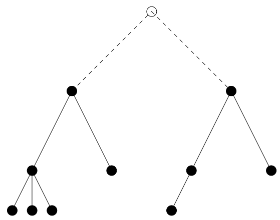

```
       EX: Other common graphs, questions people ask about them
           V = {cities}, E = {roads connecting them};
             what are shortest paths from one city to another?
           V = {computers}, E = {networks connecting them},
             what is available bandwidth for any two computers to communicate?
           V = {web pages}, E = {Whether one points to another}
             what is best answer to a web search?
           V = {people}, E = {Whether one person has met another}
             what is likely path of spread of disease?
ASK&WAIT: Other examples?
       DEF A tree is a graph with exactly one path between any two nodes.
           A rooted tree is a tree with a distinguished node called a
           root
ASK&WAIT: Is G a tree?
```




```
ASK&WAIT: Where is the root of Tree 2 above?
       Fact: if I remove the root from a tree T, and its k connecting edges,
             I am left with k unconnected subtrees T1,...,Tk (ie there is no
             path from any node in Ti to any node in Tj)
```

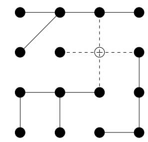
Tree 1

root and connecting edges removed



Tree 2

root and connecting edges removed



ASK&WAIT: what are k connected subtrees in figure above?

Proof: let r1,...,rk be the nodes connected to the root.
I need to show 2 things: that each Ti is unconnected to any other, and each Ti is a tree.
I use proof by contradiction: suppose node a in Ti and node b in Tj were connected; I will find a contradiction. Since a is connected to r and b is connected to r in T, there must be two paths from a to r in T (the one directly from a to r and the one via b); this contradicts the fact that T is a tree.
Now suppose that Ti is not a tree; I will find another contradiction. Ti not a tree means there are two nodes c and d in Ti with either 0 or >1 paths connecting them. If there are >1 paths connecting them in Ti, the same paths exist in T, so T must not be a tree. If there are no paths connecting them in Ti, then there are in particular no paths connecting them both to ri, and hence no paths in T connecting both to root; contradicting the fact that T was a tree.

Thm: Let T be any tree. Let E be the number of edges of T and N be the number of nodes. Then E = N-1.

Proof: We do two slightly different proofs.
First we do induction on trees, or more precisely the height H of a tree, the length of the longest path from the root to a leaf.
Base case: height H = 0 means that the tree consists of the root by itself (N=1) and no edges (E=0). Clearly E = N-1.
Induction step: Assume the result is true for trees up to a certain height H, and consider a tree T of height H+1. By the lemma, if we remove the root r we get k unconnected subtrees T1,...,Tk, whose roots are the vertices that were directly connected to r. The heights of these trees is at most H, so by the induction hypothesis

```
      #nodes(Ti) = #edges(Ti) + 1
    Thus
      #nodes(T) = SUM_{i=1}^k #nodes(Ti) + 1
                    ... the "+1" is to count the root r
                = SUM_{i=1}^k (#edges(Ti)+1) + 1
                    ... by induction hypothesis
                = SUM_{i=1}^k #edges(Ti) + k + 1
                = #edges(T) + 1
                    ... since the edges in T include the
                        edges in T1,...,Tk and the k edges
                        connecting T1,...,Tk to r

In the second proof, we do induction on N, the number of nodes
in the tree.
Base case: N=1 means there is one node, and no edges, so
           E=0 as desired. (This is the same base case as before.)
Induction step: Assume the result is true for trees of
up to N nodes, and consider a tree with N+1 nodes.
Remove any node r from T and its k adjacent edges, leaving
trees T1,...,Tk. The number of nodes in any Ti is at most N,
since we removed r, so the induction hypothesis applies, and
#nodes(Ti)-1=#edges(Ti). Also,
#nodes(T) = sum_{i=1}^k #nodes(Ti) + 1
                   ... the "+1" is to count the node r
          = sum_{i=1}^k (#edges(Ti) +1) +1
                   ... by induction hypothesis
          = sum_{i=1}^k #edges(Ti) +k+1
          = #edges(E) + 1 as desired.
                   ... since the edges in T include the
                       edges in T1,...,Tk and the k edges
                       connecting T1,...,Tk to r
```

```
Here is a Bogus proof: Why is it bogus?
    "Thm": All Berkeley students have the same color eyes.
    proof: We will use induction on n to prove that that if S
           is a set of n Berkeley students, then all students
           in S have the same color eyes.
           Base case (n=1): then any set S={student} consisting
                of one student has the property that all students
                in S have the same color eyes
           Induction step: Assume the result is true for n.
                Let S be any set of n+1 students:
```

```
                S = {s(1),s(2),...,s(n+1)}
                  = S1 U S2 where
                    S1 = {s(1),s(2),...,s(n)} and
                    S2 = {s(2),s(3),...,s(n+1)}
          By induction all students in S1 have the same eye color,
          since S1 has n members. Similarly all students in S2 have
          the same eye color. In particular, they all have the same
          eye color as s(2), say, since s(2) is in S1 and in S2.
          So all students in S have the same eye color.
```

A function f(n) where n is a nonnegative integer is defined recursively if
1) we give the value of f(0)
2) we give a rule for computing f(n) from f(n-1), when n >= 1

```
 EX: f(0) = 1, f(n) = n*f(n-1)
ASK&WAIT: what is a closed form formula for f(n)? proof by induction
           Analogous program:
             func f(n)
             if n=0
                return(1)
             else
                return n*f(n-1)
             endif
       What does it mean for a program to call itself?
    Ex: if n=3, f(3) computes 3*f(2) = 3*(2*f(1)) = 3*(2*1) = 3*2 = 6
ASK&WAIT: how many times is f() called when you call f(10)?
 EX: f(0) = 1, f(n) = a*f(n-1) = a^n, proof by induction
           Analogous program: func f(n) if n=0 return(1) else return a*f(n-1)
```

We can also define f(n) recursively via
1) we give the value of f(0), f(1), ... , f(k)
2) for n>k, we give a rule for computing f(n) from f(0),...,f(n-1)

```
 EX: Fibonacci numbers: f(0)=0, f(1) = 1, f(n)=f(n-1)+f(n-2)
           n= 0   1   2   3   4   5   6   7   8   9  10 ...
        f(n)= 0   1   1   2   3   5   8  13  21  34  55 ...
     Analogous program:
     func f(n)
        if n=0 return(1)
        else if n=1 return(1)
        else return f(n-1)+f(n-2)
     How much does it cost to compute f(n)?
```

```
Via loop:
  array g(n)
  g(0)=0, g(1)=1, for i=2 to n, g(i)=g(i-1)+g(i-2), end for
     Then g(i) = f(i)
     cost = # additions = n-1
Via recursive program above:
EX: what happens when you call f(4)? (show call tree)
     cost(n) = # additions to compute f(n) using recursion
     cost(0) = cost(1) = 0
     Otherwise, cost(n) = cost(n-1) + cost(n-2) + 1
         n     =  0  1  2  3  4  5  6  7  8  9 ...
       cost(n) =  0  0  1  2  4  7 12 20 33 54 ...
     cost(n)+1 =  1  1  2  3  5  8 13 21 34 55 ... = f(n)
       (you will prove this by induction in homework)
     Just how big is f(n)? Is it O(n)? To decide we use formula for
     f(n):
Via formula
    Define  r1 = (1+sqrt(5))/2 ~ 1.62 and
            r2 = (1-sqrt(5))/2 ~ -.62
    Then f(n) = (r1^n - r2^n)/sqrt(5)
    (you will prove this on homework, by induction)
In other words, f(n) grow like O(1.62^n), exponentially
```

```
ASK&WAIT: How much does evaluating this formula cost, compared to other ways?
     Evaluating formula cleverly MUCH cheaper than either O(n) or O(1.62^n)
     Derivation of formula for f(n)
     (1)  "Guess" that there is a solution of f(n)=f(n-1)+f(n-2) of form r^n
          for some constant r. We know there is, We only need to fine r.
          Plug in to get r^n = r^(n-1) + r^(n-2), and solve for r.
          r=0 is one possibility; otherwise divide by r^(n-2) to get
          r^2 = r + 1, a quadratic equation with solutions
             r1 = (1+sqrt(5))/2 and r2 = (1-sqrt(5))/2
     (2)  So r1^n and r2^n are solutions but neither satisfies f(0)=0,f(1)=1.
          But note that alpha*r1^n and beta*r2^n are also solutions for any
          alpha and beta, as is their sum alpha*r1^n + beta*r2^n.
          So seek alpha and beta such that
                alpha*r1^0 + beta*r2^0 = alpha + beta = 0 and
                alpha*r1^1 + beta*r2^1 = 1
          2 linear equations in 2 unknowns: solve them for alpha, beta to get
             -r1*(alpha+beta) + (alpha*r1 + beta*r2) = beta*(r2 - r1) = 1,
             so beta = 1/(r2 - r1) = -1/sqrt(5) and alpha = -beta = 1/sqrt(5)
          and f(n) = (r1^n - r2^n)/sqrt(5)
```

Formulas like this exist for any linear recurrence like
h(n) = a*h(n-1) + b*h(n-2) + c*h(n-3),
where a,b,c are constants and h(0),h(1),h(2) known; see chapter 5

EX: We can also describe "well-formed formulas (WFF)" or
"arithmetic expressions" recursively:
Well-formed: a+b, (a+b)/c, (a+b)/c + a^d, ((a+b)/c + a^d)/(a-d), ...
Not well-formed: a- , (ab+*(, ...
(a-a)/(a-a) is "well-formed", since we only care about the "syntax",
not the value of the formula
We can define these recursively as follows:
(1) Any single variable (a,b,c,...) or number (7, 3.1416,...) is a WFF
(2) If E is a WFF, so is (E)
(3) If E is a WFF, so is -E
(4) If E1 and E2 are WFF, so is E1+E2
(5) If E1 and E2 are WFF, so is E1-E2
(6) If E1 and E2 are WFF, so is E1*E2
(7) If E1 and E2 are WFF, so is E1/E2
(8) If E1 and E2 are WFF, so is E1^E2

EX: Shorthand notation:
E -> variable | number | (E) | -E | E+E | E-E | E*E | E/E | E^E
This is called a "grammar", and is used by compilers (CS164)

EX: (a+b)/c is WFF because it is gotten by applying the rules
(1) to a, (1) to b, (4) to a+b, (2) to (a+b), (1) to c, (7) to (a+b)/c
Order in which we apply rules is usually represented by a "parse tree":

```
          /(7)
         / \
        /   \
       /     \
      ( )(2)  c(1)
       |
       +(4)
      / \
     /   \
    a(1)  b(1)
```

EX: 1 goal of the compiler (parser) is to take an expression (a+b)/c
and either
1) produce the parse tree, and the corrsponding rules for each part, or
2) decide there is no parse tree, and print a "syntax error" message

Use this to prove (by induction) that any WFF has as many "("s as ")"s.
Proof: Base case: a variable or number has 0 parentheses

```
          Induction case: take each rule (2)-(8) and confirm that the
             number of "("s and ")"s stays equal:
                Rule (2): Number of "("s is number of "("s in E + 1, and
                          number of ")"s is number of ")"s in E + 1, so
                          if E had equal numbers of each, so does (E)
                Rule (3): numbers of parentheses does not change
                Rules (4)-(8): numbers of parentheses is the sum of
                          those in E1 and E2, so if there were equally many
                          "(" and ")" in E1 and E2, the same is true when
                          you combine them

   EX: Analysis of Euclidean Algorithm:
          x=a,y=b,  ... assume x >= y, swap them otherwise
          while y != 0
             r = x mod y
             x = y
             y = r   ... still true that x >= r
          end while
          return gcd = x

       Recursive version of same algorithm:
          func gcd(x,y)
            if x<y, swap x and y
            if y = 0 then return(x)
            else return( gcd( y, x mod y ) )
       How many times is the while loop executed?
       Def: Let N(x) denote the number of bits needed to represent
        the nonnegative integer x (N(x) = floor(log_2 x) + 1 for x>0, N(0)=1).
       Theorem: The number of times the loop in the Euclidean Algorithm is
         executed is bounded by N(a)+N(b) <= log_2 a + log_2 b + 2
       Proof: We will use induction on N(x)+N(y), showing it decreases by
         at least one after each pass through the loop,
         so it must stop by the time N(x)+N(y)=2 if not sooner.
ASK&WAIT: Base case: What are x and y if N(x)+N(y) reaches 2?
      Induction step:
      Let xo and yo denote the (old) values of x and y at the beginning
      of the loop, and xn and yn denote the (new) values of x and y at the end.
      We need to show that N(xn)+N(yn) <= N(xo)+N(yo)-1, because then by
      induction the number of passes through the loop to compute gcd(xo,yo) will
      be bounded by
          (N(xn)+N(yn)) + 1 = (N(xo)+N(yo)-1) + 1 = N(xo) + N(yo) as desired.
      Case 1: Suppose that N(xo)=N(yo), neither xo nor yo = 0.
```

ASK&WAIT: When we divide xo = q*yo + r using the Division Algorithm, what is q?
Then r=xo-yo has at most N(xo)-1 bits since the leading bit cancels, and so
N(xn) + N(yn) = N(yo) + N(r) <= N(yo) + N(xo)-1 as desired.

Case 2: Suppose that N(xo) > N(yo). Then r<yo has at most N(yo) bits, so
N(xn) + N(yn) = N(yo) + N(r) <= N(yo) + N(yo) < N(xo) + N(yo)
<= N(xo) + N(yo) -1 as desired.

Note: Lame's Theorem in book has slightly different bound, but same idea