Math 55 - Spring 2004 - Lecture notes # 5 - Feb 3 (Tuesday)

Read: Sections 2.1-2.3
Note: we will not cover binary search, sorting,
greedy algorithms, which are covered elsewhere (CS61B)

Homework : Due Feb 11 in section.

1) Show that if the first 14 positive integers
are placed around a circle in any order, there
exist 5 integers in consecutive locations
around the circle whose sum is 38 or greater.
Hint: Use the result of question 1.5-71.

2) A real number is called "algebraic" if
it is the root of some polynomial with
integer coefficients and degree at least 1.
Let A be the set of all algebraic numbers.
So A includes sqrt(2),
cuberoot((5-sqrt(2))/sqrt(3)), any other
such expression with roots and integers, and
many other real numbers besides.
This exercise will show that A is countable.

2.1) Show that if r is a root of a polynomial
with rational coefficients, it is also
a root of a polynomial with integer
coefficients. So we won't miss any real
numbers by restricting ourselves to
polynomials with integer coefficients.

2.2) Show that the set P_d of polynomials
of degree d >= 1 and with integer
coefficients is countable.
(A polynomial has degree d if it can be
written as
a_d*x^d + a_{d-1}*x^{d-1} + ... + a_1*x + a_0
with a_d nonzero)

2.3) Show that the set A_d of all real roots
of all polynomials in P_d is countable.

2.4) Show that the set A of all algebraic
numbers is countable.

2.5) Conclude that there are a great many
more real numbers that are not roots
of polynomials with integer coefficients
than real numbers that are roots of such
polynomials.

3) Simplify O(f(n)) where f(n) is given below. Your expression should be both as simple and accurate as possible (it should not overestimate f(n) by more than a constant factor). All logarithms are base pi.

```
f(n) =
  [ 9^(2^(n^.3)) - 2^(9^(n^.3)) ] *
  [ .99^(n^3) + log (log (log (log n ))) ] *
  [ (log (log n))^(log (log (log n))) +
    (log (log (log n)))^(log (log n)) ] *
  [ 3^n * n^4 - 4^n * n^3 ] *
  [ 89*n^4 + 1234 * n * (log n)^18 ]
```

4) Show that log_(1.75) 3.5 must be irrational. Hint: proof by contradiction

5) sec 2.2: 20, 36, 60, 62

6) sec 2.3: 8

7) Modify the algorithm in the last question (2.3-8) to compute the derivative of the given polynomial. How many additions and multiplications does your algorithm take (ignoring additions to increment the loop variable)?

Goals for today: Expressing algorithms
how do we measure, compare running times?
Big-O notation
Introduction to Complexity Theory

Algorithms:

ASK&WAIT: what does this program do?

```
prog1(integer n, integer array a(1),...,a(n))
   M = a[1]
   for i = 2 to n
      M = max(M,a(i))
   end for
   return M
```

ASK&WAIT: What does this program do?

```
prog2(integer n, integer array a(1),...,a(n))
   for i = 1 to n
      M = a(i)
```

```
              for j = 1 to n except i
                 if a(j)>M goto next
              end for
              return M
              next:
            endfor
ASK&WAIT: Which program do you think is faster? How much faster?
          How do we determine how long prog1 takes to run?
          Approach 1: run it and measure the run time in seconds,
ASK&WAIT: What are the pros and cons of this approach?
          Approach 2: for each operation performed by the program,
                    find out how many seconds it takes, and add them all up
                    n-1: max
                    n-1: increment i
                    n-1: test to see if loop is done
                    start up cost of M=a[1], etc...
ASK&WAIT: What are the pros and cons of this approach?
          Approach 3: it takes time proportional to n, ie. about k*n
                    for some constant k, large enough n that startup is small
ASK&WAIT: What are the pros and cons of this approach?
ASK&WAIT: How long does prog2 take to run? i.e. proportional to what
          function of n?
          If it depends on values of a(1),...,a(n), what is the worst case?
          Is prog2 or prog1 faster, in the worst case?
```

Big-O notation

Motivation: given a complicated function f(n), which
may represent how long a program runs on a problem of size n,
(or how much memory it takes), quickly approximate it by a much
simpler function g(n) that roughly bounds how fast f(n)
increases as a function of n

Ex: Consider f(n) = (pi+7)/3*n^2 + n + 1 + sin(n). When n is large,
n^2 is so much larger than n+1+sin(n) that we would like to ignore
these terms. Furthermore, (pi+7)/3 is just a constant, and
for many purposes we don't care exactly what the constant is.
So we'd like some notation to say that f(n) grows like
some constant time n^2: we will write "f(n) is O(n^2)".

Introduce notation to mean "proportional to n", or any other g(n):
DEF: Let f and g map integers (or real) to reals. We say that

```
     f(x) is O(g(x)) (read "Big-O of g") if there are
     constants C>0 and k>=0 such that |f(x)| <= C*|g(x)| whenever x>k
Intuitively, if f(x) grows as x grows, then g(x) grows at least as fast
EG: f(x) = 100*x and g(x)=x^2, then for x>100, g(x)>f(x) and f(x)=O(g(x))
EG:  if n>=1 then
     f(n) = (pi+7)/3*n^2 + n + 1 + sin(n)
         <= (pi+7)/3*(n^2 + n^2 + n^2 + n^2)
          = 4*(pi+7)/3*n^2
    so f(n) = O(n^2), with k=1 and C=4*(pi+7)/3 in the definition
```

(Draw pictures to illustrate)

```
Remark: Sometimes we write f(x) = O(g(x)), but this is misleading
        notation, because f1(x) = O(g(x)) and f2(x) = O(g(x))
        does not mean f1(x) = f2(x), for example
        x = O(x) and 2*x = O(x)
```

```
       EG: f(n) = run-time of prog1 for input of size n = O(n)
ASK&WAIT:  what is (worst case) running time of
           input: x, array a(1),...,a(n)
           found = false
           for i=1 to n
              if x = a(i)
                 found = true
                 exit loop
              endif
           end for
```

In some of most important applications of O(), we never have an
exact formula for f(n), such as when f(n) is the exact running
time of a program. In such cases all we can hope for is a
simpler function g(n) such that f(n) is O(g(n)).
But to teach you how to simplify f(n) to get g(n),
we will use exact expressions f(n) as examples.

```
       Goals of O() are
        1) simplicity: O(n^2) is simpler than (pi+7)/3*x^2+n+1+sin(n),
                       O(n) is simpler than actual run time of prog1
        2) reasonable "accuracy":
       EG: Consider f(x) = (pi+7)/3*n^2 + n + 1 + sin(n)
           f(n) is both O(n^2) and O(n^3)
ASK&WAIT: why?
```

```
ASK&WAIT:  Which is a "better" answer, to say f(n) is O(n^2) or O(n^3),
           since both are true?
  EX:  Suppose we have two programs for the same problem and want to
       pick the fastest. Suppose prog1 runs in exactly time 10*n and prog2
       runs in time n^2, so prog1 is clearly faster when n>10. But if we are
       "sloppy" and say that both run in time O(n^2), then we can't
       compare them.

       So we would like rules that make it easy to find simple
       and accurate g(x) so f(x)=O(g(x)) for complicated f(x)
       that avoid ever needing explicit values of C and k in
       the definition of Big-O

       Rule 1: if c is a constant, c*f(x) is O(f(x))
ASK&WAIT: what are C and k in definition of O() that proves this?
       EX given any a,b >0, we have log_a x = O(log_b x)
ASK&WAIT: why?

       Rule 2: x^a = O(x^b) if 0 < a < b
ASK&WAIT: what are C and k in definition of O() that proves this?

       Rule 3: If f1(x) = O(g1(x)) and f2(x) = O(g2(x)), then
                  f1(x)+f2(x) = O(h(x)) where h(x)=max(|g1(x)|,|g2(x)|)
               proof: |f1(x)| <= C1*|g1(x)| for x>k1 and
                      |f2(x)| <= C2*|g1(x)| for x>k2 means
              |f1(x)|+|f2(x)| <= C1*|g1(x)|+C2*|g2(x)| for x>max(k1,k2) so
                              <= C1*h(x)   +C2*h(x)    for x>max(k1,k2) so
                              <= (C1+C2)*h(x)          for x>max(k1,k2)
     EX: let f(x) = a_k*x^k + a_{k-1}*x^{k-1} + ... + a_1*x + a_0,
         be a polynomial of degree k, i.e. a_k is nonzero
           by Rule 1 each term a_j*x^j is O(x^j), so
           by Rule 2 each term is O(x^k), so
           by Rule 3 f(x) is O(x^k)
           In other words, for polynomials only the term with the
              largest exponent matters

       Rule 4: If f1(x) = O(g1(x)) and f2(x) = O(g2(x)), then
                  f1(x)*f2(x) = O(h(x)) where h(x)=g1(x)*g2(x)
ASK&WAIT: what are C and k in definition of O() that proves this?

       EG: f(x) = (x+1)*log(x-1) = f1(x)*f2(x)
                = O(x)*O(log(x)) = O(x*log x)
       Rule 5: if f(x) = O(g(x)) and a>0, then (f(x))^a = O((g(x))^a)
ASK&WAIT: what are C and k in definition of O() that proves this?

       Rule 6: (log_c x)^a = O(x^b) for any a>0, b>0, c>0
                in fact, the limit as x increases of (log_c x)^a / x^b is zero
       Proof: By rules 1 and 5 we can assume c is any convenient constant,
              say e=2.71828...
              If we can show log x = O(x^(b/a)) for any b,a>0, then taking
                the a-th power yields (log x)^a = O(x^b) (Rule 5)
              So try to show log x = O(x^d) for any d>0
              First we will show that as x increases,
                 f(x) = log x / x^d decreases, once x is large enough:
              Differentiate f(x), and show that for x large enough, f'(x)<0:
                 f'(x) = (x^d * (1/x) - d*x^(d-1)*log x)/x^(2*d)
                       = x^(-d-1)*(1 - log x^d )
                       < 0 if x > e^(1/d)
              Now we show that f(x) actually goes to zero as x increases.
              Since f(x) is decreasing, it is enough to show that there is
              an increasing sequence of values x1,x2,x3,... such that f(xi) -> 0
              Let x(i) = e^(i/d). Then
                 f(x(i+1)) = log e^((i+1)/d) / e^(i+1)
                           = log e^((i/d)*(i+1)/i) / ( e * e^i )
                           = ((i+1)/i)/e * log e^(i/d) / e^i
                           = ((i+1)/i)/e) * f(x(i))
                           <= (2/e) * f(x(i))
                                because (i+1)/i takes values 2 > 3/2 > 4/3 > ...
                           <  .74 * f(x(i))
                           <  .74^2 * f(x(i-1)) ...
                           <  .74^i * f(x(1))
                           =  .74^i * (1/(d*e)),
                              which goes to zero as i increases

ASK&WAIT:  simplify O(log x + sqrt(x))
ASK&WAIT:  simplify O((log n)^1000 + n^.001)

       Rule 7: x^a = O(b^x) for any a>0, b>1
           Proof: Just take logarithms (base 2, say), apply Rule 6:
                  log(x^a) = a*log x and log(b^x) = x*log b; we know that
                  for large enough x, x*log b is larger than a*log x,
                  so 2^(x*log b)=b^x is larger than 2^(a*log x) = x^a
       EG: There is a standard list of functions that appears frequently
           when computing running time of programs, and you should
           recognize them, and which is bigger than the other:
           O(1)
           O(log n)   = time to find an element in a sorted list, binary search
           O(n)       = time to find maximum
           O(n*log n) = time to sort n numbers, using good algorithm
           O(n^2)     = time to sort n numbers, using dumb algorithm
           O(n!) = O(1*2*3*...*n)

ASK&WAIT:  Simplify O((n+1)*log(sqrt(4n-2)) + log((n!)^2))
ASK&WAIT:  Simplify O(n^(log n) + 1.1^(sqrt n))
ASK&WAIT:  simplify O((log n)^(log n) - n^(log log n))
ASK&WAIT:  simplify O((2^n + n^3*log n)*(n^4 + (log n)^2) + 1.5^n*n^5)

 Some more definitions related to Big-O

 DEF: We say f(x) is Big-Omega(g(x)) if g(x) is O(f(x))

 Motivation: use g(x)=O(f(x)) to say a constant*f(x) is an
             upper bound on g(x)
             use f(x)=Big-Omega(g(x)) to say a constant*g(x) is a
             lower bound on f(x)

 DEF: We say f(x) and g(x) are of the "same order" (or f(x) = BIG_THETA( g(x) ))
      if f(x) = O(g(x)) and f(x) = Big-Omega(g(x)), i.e. for x > k,
      there are constants C1>0 and C2>0 such that
      C1*g(x) <= f(x) <= C2*g(x)

 EX:  2*n^2 + n + 1 = O(n^3) but BIG_THETA(n^2), because
            n^2 <= 2*n^2 + n + 1 <= 4*n^2    for n>=1

ASK&WAIT: If prog1 runs in time O(n) and prog2 runs in time O(n^2),
    then is prog1 is faster than prog2 for large enough n?

ASK&WAIT: If prog1 runs in time BIG_THETA(n) and prog2 runs in time
    BIG_THETA(n^2), then is prog1 is faster than prog2 for large enough n?

    "Complexity Theory" is the study of which how fast certain
    problems can be solved, expressed in terms like
    "Given an input of size n, this algorithm will run in time O(f(n))"
```

```
     EX: Given a list L of n numbers, and one other number s,
         linear search will decide if s is in L in time O(n)
     EX: Given a sorted list L of n numbers, and one other number s,
         binary search will decide if s in in L in time O(log n)
     EX: Given a list of n numbers, we can sort it using insertion sort
         (same algorithm you'd use to sort by hand) in time O(n^2)
ASK&WAIT: Do you know any other faster sorting algorithms?
     EX: Given a highway map labelled with distances between n towns,
         finding the shortest way to drive between every pair of towns
         (also called "all pairs shortest paths"), costs O(n^3)
         using the Floyd-Warshall algorithm (see CS170)
     DEF All these algorithms and many others are called "polynomial
         time algorithms" because they cost O(n^a) for some constant a.

     Here is another problem which cannot be solved in polynomial time
     as far as anyone knows:
        given any compound logical proposition such as
               q = p1 and p2 or not p3 ....
        using n proposition p1, p2, ... , pn, can q ever be True for
        any value of the p1,..,pn?
     This problem is called the "satisfiability" problem or SAT for short:
         can any values of p1,..,pn satisfy q, i.e. make it true?
     EX: if q = p1 and p2 and not p3
         then setting p1 = True, p2 = True and p3 = False makes q = True
     EX: if q = (p1 or p2) and (not p1 or not p2) and
                (not p1 or p2) and (p1 or not p2)
         then no matter what values p1 and p2 have, q is False
     Here is an obvious algorithm to solve this problem:
         evaluate q for all possible values of p1, p2,..., pn
         if q is ever True then the answer is yes (q can be true) else no
ASK&WAIT: What is the cost of this algorithm, at least?
     What may be surprising is that no significantly better algorithm
     is known, i.e. no algorithm that runs in polynomial time, O(n^a)
     for some constant a. They all run in "exponential time" (maybe
     faster than 2^n, but still exponential)

     One of the most famous open problem in mathematics (and computer
     science) is the question of whether any polynomial time algorithm
     for SAT can exist. The problem is also sometimes asked as
     "does P = NP or not?". Here P is the set of problem you can
     solve in polynomial time, and NP is a larger class including SAT.
     CS 170 and especially CS 172 talk about this question in more detail.
```