============================================================================

CSC 263 Tutorial 1 Winter 2019

============================================================================

Consider the following algorithm to find the maximum element in a list.

FIND-MAX(L):

max = -oo # minus infinity

for k = 0 to len(L)-1:

if L[k] > max:

max = L[k]

return max

For this problem, we are interested in the number of times that variable max

gets assigned a value.

1. What is the average-case number of times that max is assigned a value by

algorithm FIND-MAX? Show your work -- in particular, define your sample

space and probability distribution clearly, show the steps in your

computation, and simplify your final answer.

answer：

Sample space:  
      - How to determine? Think about possible behaviours of the algorithm,  
        ensure at least one input for each behaviour -- ideally, aim for  
        exactly one input for each behaviour.  
      - For this problem? Behaviours = number of times max is updated.  
        Simplest sample space:  
            S\_n = {all permutations of [1,2,...,n]}  
  
    Probability distribution:  
      - Warning: Don't just fall back on "uniform"! Think about inputs  
        and whether any of them represent special cases. May not know how  
        likely each one is, in which case can parametrize the answer (assign  
        unknown but fixed probabilities to each special case, then make the  
        rest of the "regular" cases uniform).  
      - For this problem? No special case; OK to use uniform distribution:  
        each permutation equally likely (with probability 1/n!).  
  
    Random variables:  
      - In general: Define one random variable for quantity of interest.  
      - For this problem? Let X = number of times the statement "max = L[k]"  
        is executed on input L.  
  
    Expression:  
      - Figure out possible values of random variable over sample space,  
        then write down definition of expected value.  
      - For this problem? Possible values for X range from 1 to n.  
  
                    n  
            E[X] = SUM i \* Pr[X = i]  
                   i=1  
  
      - What next? Three possibilities...  
  
       (a) Figure out Pr[X = i] directly, then compute the sum.  
            In this case, Pr[X = i] is not obvious to compute. This does not  
            mean it's "impossible": we may be able to work on it until we  
            come up with a reasonable expression. But in this case, let's  
            try another possibility.  
  
        (b) Re-express the sum based directly on the sample space.  
            Ultimately, Pr[X = i] depends on individual input probabilities,  
            so rewrite the sum in terms of those probabilities directly:  
  
                E[X] =   SUM    X(L) \* Pr[L]  
                       L in S\_n  
  
            where X(L) is the value of X for input L.  
            Advantage: Pr[L] is easy because of our probability distribution  
            -- Pr[L] = 1/n!  
            Disadvantage: X(L) is complicated. As before, we may be able to  
            come up with a reasonable expression with enough work. But in  
            this case, let's try another possibility.  
  
        (c) Re-express the sum by using "indicator" random variables.  
  
            In general, when computing average case running times, we want  
            to compute E[X] where X "counts" the number of times certain  
            operations are executed. Trying to come up with exact  
            expressions for Pr[X = i] or for the value of X on individual  
            inputs is generally difficult and messy. Instead, try to break  
            up X into a number of other random variables X\_1,X\_2,...,X\_m,  
            each one of which "counts" only part of the total value -- these  
            are not technically always indicator variables, but they will  
            help us compute a final answer anyway. The trick is to define  
            the X\_i's appropriately so that two conditions hold:  
  
              - X = X\_1 + X\_2 + ... + X\_m;  
              - Each X\_i has only two possible values: 0 or 1.  
  
            Then, we can compute E[X] as follows:  
  
                E[X] = E[X\_1 + ... + X\_m]  
                     = E[X\_1] + ... + E[X\_m]  (by linearity of expectation)  
                     = Pr[X\_1 = 1] + ... + Pr[X\_m = 1]  
  
            Where the last equality holds because each X\_i can only be equal  
            to 0 or 1, so by definition:  
  
                E[X\_i] = 0 \* Pr[X\_i = 0] + 1 \* Pr[X\_1 = 1]  
                       = Pr[X\_i = 1].  
  
            Hint for this problem --   
              
                      { 1  if L[i] > L[0, 1,..., i-1],  
                M\_i = {  
                      { 0  otherwise.  
  
      - Using the indicator random variables from the hint, we have that  
        X = M\_0 + ... + M\_{n-1} (the number of times that "max = L[k]" is  
        executed is equal to the number of times that L[k] is larger than  
        each of the preceding elements in L).  
        Then, as explained above:  
  
                   n-1  
            E[X] = SUM Pr[M\_i = 1]  
                   i=0  
  
        But what's Pr[M\_i = 1}? It's the probability that L[i] is greater  
        than every element before it. Since each permutation of L is equally  
        likely, this is simply equal to 1/(number of elements in L[0...i])  
        = 1/(i+1). (Quick sanity check: when i = 0, the probability is  
        1/(0+1) = 1/1, which is correct -- "max = L[0]" is always  
        executed for every input; when i = n-1, probability is 1/(n-1+1) = 1/n,  
        which is also correct: "max = L[n-1]" is executed only if L[n-1] is the  
        largest element, which happens with probability 1/n.)  
  
        So now we have:  
  
                   n-1               n-1  1     n  1  
            E[X] = SUM Pr[M\_i = 1] = SUM --- = SUM - = H\_n ~ log n.  
                   i=0               i=0 i+1   j=1 j  
  
        (Because the n-th Harmonic number H\_n converges to log n as n tends  
        to infinity.)

2. What is the best-case number of times that max is assigned a value by

algorithm FIND-MAX? Show your work.

Lower bound: L[k] > max is always true on the very first iteration (when  
    k = 0 and max = -oo) so max is assigned a value at least once  
    for every input. This implies best-case >= 1.  
  
    Upper bound: When L = [n,...,2,1], then L[k] > max is true only on the  
    first iteration and max is assigned a value exactly once. This implies  
    best-case <= 1.  
  
    So best case = 1 (exactly).  
  
    WARNING: "Best case is when n = 0" is incorrect! Best case running time  
    is a \_function\_ with a value for every n -- the answer must be given for  
    some arbitrary n, NOT by picking one particular value.

3. What is the worst-case number of times that max is assigned a value by

algorithm FIND-MAX? Show your work.

Upper bound: L[k] > max is true at most once for each element. This  
    implies worst-case <= n.  
  
    Lower bound: When L = [1,2,...,n], then L[k] > max is true for every  
    element. This implies worst-case >= n.  
  
    So worst case = n (exactly).  
  
    NOTE: The following picture is a useful reminder of what it means to  
    prove upper/lower bounds on the best/worst case running time, in  
    general. Each 'x' represents the running time for one particular input  
    of size n, where n is some arbitrary size. Generally, for anything but  
    the simplest algorithms, we don't know exactly where every possible 'x'  
    falls -- in particular, we do not know exactly what is the worst-case  
    input and what is the best-case input.  
  
     time  
         ^  
         |               X -- upper bound on worst case  
         |  
         |               :  } worst case is somewhere around here  
         |  
         |               x -- lower bound on worst case  
         |  
         |               x  } any one particular input is both  
         |               x  } a lower bound on the worst case and  
         |               x  } an upper bound on the best case  
         |  
         |               x -- upper bound on best case  
         |  
         |               :  } best case is somewhere around here  
         |  
         |               X -- lower bound on best case  
         |  
        -+--- ... ---+---+---+---+---+---> input size  
         |               n (arbitrary)  
  
      - An upper bound on the worst case is the same as an upper bound on  
        the time for \_every\_ input. It does not have to be exactly equal to  
        the running time for any actual input (it's just a bound).  
  
      - A lower bound on the worst case is the running time for any one  
        particular input that is "large enough" (meaning it's within a  
        constant factor of our best upper bound on the worst case).  
  
      - A lower bound on the best case is the same as a lower bound on the  
        time for \_every\_ input. It does not have to be exactly equal to the  
        running time for any actual input (it's just a bound).  
  
      - An upper bound on the best case is the running time for any one  
        particular input that is "small enough" (meaning it's within a  
        constant factor of our best lower bound on the best case).

additional notes:

About Indicator Random Variables (IRV)

An indicator random variable X\_e of an event e indicates whether event e

happens or not, so it's defined as follows

{ 1 if e happens

X\_e = {

{ 0 if e doesn't happen

One important property of indicator random variables is the following equality

E[X\_e] = Pr(e happens)

You can easily prove this equality by trying to compute the expectation of X\_e.

2. For the analysis of the FIND-MAX algorithm, we define a sequence of IRVs in

the following way:

X\_k for indicating the following event:

e\_k: the assignment line max = L[k] is executed

where k takes value from L.length - 1 (i.e., n-1) down to 0, therefore the

expected total number of times that "max = L[k]" is executed is simply the

following sum

n-1 n-1 n-1

E[ SUM X\_k ] = SUM E[X\_k] = SUM P(e\_k happens)

k=0 k=0 k=0

So, now the only thing left is to figure out P(e\_k happens).

3. Figure out P(e\_k)

We have no idea what this probability is without defining the input

distribution. So here we choose the following distribution of the inputs: X is

the random permutation of [1,2,...,n], and each permutation is equally likely to

be chosen. The following analysis will be based on this assumption on the input

distribution.

As mentioned above the event e\_k is

e\_k: the assignment line max = L[k] is executed, which is equivalent to

L[k] > max, which is equivalent to

L[k] is the max among the first k+1 elements (note k starts from 0)

Since all permutations are equally likely, each one of the first k+1 elements

is \*equally likely\* to be the max, therefore the probability that L[k] is the

max among the first k+1 elements is:

1/(k+1)

So the desired average-case runtime becomes

n-1 n-1

SUM P(e\_k happens) = SUM 1/(k+1) = O(log n)

k=0 k=0

To see why the last equality is true, Google "harmonic series" if you are

interested.