Math 55 - Spring 2004 - Lecture notes #24 - April 22 (Thursday)

Goals for today: Applications to computer science
searching a list
collisions in a hash table
load balancing

EX: Suppose we have a list of n distinct items L(1),...,L(n), and want an algorithm that takes an input x and
(1) returns i if L(i)=x,
(2) returns n+1 otherwise
An obvious algorithm is "linear search"

```
i=0
repeat
   i=i+1
until L(i)=x or i=n+1
```

Suppose x is chosen at random from a sample space S with probability function P. What is the expectation of the operation count C of this algorithm, i.e. how many times is the line "i=i+1" executed? If we run this algorithm many times, E(C) tells us how long it will take "on average", and sigma(C) tells us how variable the running time will be.

The answer depends on S and P:

EX 1: Suppose S = {L(1),...,L(n)} and P(L(i))=1/n; this means each input x must be in the list L, and is equally likely.

```
Then E(C) = 1*(1/n) + 2*(1/n) + 3*(1/n) + ... + n*(1/n)
          = (n+1)/2
and  V(C) = E(C^2) - (E(C))^2
          = 1^2*(1/n) + 2^2*(1/n) + 3^2*(1/n) + ... + n^2*(1/n)
            - ((n+1)/2)^2
          = (n^3/3 + O(n^2))*(1/n) - (n^2/4 + O*(n^2))
          = n^2/12 + O(n)
```

so sigma(C) = n/sqrt(12) + O(1) ~ .29*n + O(1)

EX 2: Suppose S = {L(1),...,L(n),z}, and P(L(i))=p/n, P(z)=1-p
i.e. the probability that x is not on the list is 1-p

ASK&WAIT: What is E(C)? How does it depend on p?

ASK&WAIT: What are V(C) and sigma(C)? How do they depend on p?

EX 3: Suppose P(L(i)) = p(i) are all different.
In what order should we search the L(i) to minimize E(C)?
ASK&WAIT: Suppose n=2, and p(2) >> p(1); which item should we search first?

Thm: Searching list in decreasing order of p(i) minimizes E(C).

Proof: Suppose that we do not search list in decreasing order
of p(i); then we will show that there is a different search
order with smaller E(C). Let q(1),...,q(n) be the probabilities
of the items in the order they are searched, so q(1),...,q(n)
is a permutation of p(1),...,p(n). Suppose that q(1),...,q(n)
is not in decreasing order; this means that for some j
q(j) < q(j+1).
Let r = 1 - p(1) - p(2) - ... - p(n) be the
probability that the search item is not in the list.

Then C1 = E(C) for search order 1,...,j,j+1,...,n
= sum_{i=1 to n} i*q(i) + (n+1)*r
and C2 = E(C) for search order 1,...,j+1,j,...,n
(i.e. with j+1 searched before j)
= sum_{i=1 to n except j and j+1} i*q(i)
+ j*q(j+1) + (j+1)*q(j) + (n+1)*r
Then C1 - C2 = j*q(j) + (j+1)*q(j+1) - j*q(j+1) - (j+1)*q(j)
= q(j+1) - q(j)
> 0
In other words, searching j+1 before j lowers the expected cost E(C)
So unless q(j)>q(j+1) for all j, i.e. unless the list is sorted,
there is a better order to search in.

EX 4: Suppose P(L(i)) = c*q^i, q<1, where c is chosen so that
sum_{i=1 to n} P(L(i)) = 1, i.e. x guaranteed to be in list
ASK&WAIT: What is c?
ASK&WAIT: What is E(C)?
ASK&WAIT: What happens to E(C) as n grows?

Recall Binary Search:
Assume L(1) < L(2) < ... < L(n), by sorting if necessary
istart = 1; iend = n
repeat
middle = floor((istart + iend)/2)
if x < L(middle) iend = middle-1
if x > L(middle) istart = middle+1

until x = L(middle) or iend < istart
Cost = O(log_2 n) steps, because each time the list to search is at most half as long.

ASK&WAIT? What is faster for large n, linear search or binary search?

But in the worst case, linear search will take n steps, much slower than binary search.

ASK&WAIT: Can you think of an algorithm that takes O(1) steps on average, and O(log n) at worst, i.e. has the advantages of both kinds of search?

The next two topics are based on notes from CS 70.

EX The next question we ask is about hash tables. A hash table is a data structure that lets us store and look up data items in close to constant time no matter how many items n there are in the hash table (unlike linear search or binary search, which cost O(n) and O(log n) respectively).
The hash table uses a so-called hash function f(x) which takes a data item x and computes an index f(x) between 1 and n = hash table size. The data item x is then stored at H(f(x)), that in the f(x)-th location of the hash table. If more than one data item x1,x2, ... all have the same index i = f(x1)=f(x2)=... (called a "collision") then all these data items are stored in a linked list starting at H(i).

So for a hash table to have the attractive property of taking a constant amount of time to find a data time, these linked lists should be very short, ideally with one data item each (no collisions). This means that the hash function f(x) should "spread" all the data items out as evenly as possibly over all n hash table entries.

To model the behavior of a hash table, we will model our ideal hash function f(x) as independently picking a random hash table location for each x, where each location is chosen with equal probability 1/n. We then ask how many items the hash table can contain before the probability that the hash function picks the same location for two items (i.e. is no longer "perfect") exceeds some probability, say 1/2 (we could pick any number we like).

```
The purpose of this is to pick the size n of the hash table we need
to store a given number m of data items without collision.

So what we want to compute is
 P(after m items are inserted randomly into n table entries, no
   collisions occur) =
  P(1st item causes no collision) *
  P(2nd item causes no collision) * ... *
  P(3rd item causes no collision) * ... *
  ...
  P(mth item causes no collision)
       ... by independence of each choice

 = 1 *         ... collision impossible on first item
   (n-1)/n *   ... n-1 equally likely free locations for 2nd item
   (n-2)/n *   ... n-2 equally likely free locations for 3rd item
   ...
   (n-m+1)/n   ... n-m+1 equally likely free locations for mth item

 = (n-1)!/[(n-m)!*n^(m-1)]

 We want to choose m depending on n such that this probability
 is about 1/2. We will use Stirling's formula for this:

     n! ~ sqrt(2*pi) * n^(n+ 1/2) * e^(-n)

 to get

  1/2 ~ (n-1)! / (n-m)! / n^(m-1)
      ~  sqrt(2*pi) * (n-1)^(n- 1/2) * e^(-n+1) /
        [sqrt(2*pi) * (n-m)^(n-m+1/2)* e^(-n+m) * n^(m-1) ]

            ... cancelling sqrt(2*pi), substituting m = a*n,
            ... and factoring n^(...) out

      ~ n^(n-1/2) * (1 - 1/n)^n * (1 - 1/n)^(1/2) * e /
        [ n^(n*(1-a)+1/2) * (1-a)^((1-a)*n + 1/2) * e^(a*n) *
          n^(a*n - 1) ]

            ... cancelling n^(...) and using (1- 1/n)^n -> 1/e

      ~ (1-a)^(-1/2) * ((1-a)^(1-a)*e^a)^(-n)
```

taking logs yields

```
log 1/2 = -1/2*log(1-a) -n*log((1-a)^(1-a)*e^a)
        = -1/2*log(1-a) - n*[(1-a)*log(1-a) + a]

For this to be near log(1/2), the factor multiplying n
has to shrink like 1/n, i.e. a has to be tiny, so we
use use the Taylor expansion for log(1-a):

    log(1-a) = - a - a^2/2 - ... ~ -a

to get that the log is

  log(1/2) ~  1/2*a - n*[(1-a)*(-a) + a]
           ~  1/2*a - n*[a^2]
  or, solving for a:
         a ~ sqrt(log(2))/sqrt(n)
           ~ .8326/sqrt(n)
  or, solving for m = a*n
         m ~ .8326*sqrt(n)
```

In other words, we would want the size n of the hash table to be about the square of the number m of data items to be sure of no collisions with probability 1/2. With any other probability, we would have gotten a similar result with a slightly different constant.

EX: We consider "load balancing." For example, suppose you run a web service (like Google) to which large numbers of requests regularly stream in, and which need to be assigned to processors. A typical algorithm takes each incoming request and randomly picks a processor to assign it to. The question, given m requests and n processors, is will each processor have roughly the same amount of work to do, i.e. will the load be balanced? (The reason this is often done instead of having a centralized processor evenly divide the work among processors is that the centralized processor becomes a bottleneck.)

A similar question would be this: suppose you are a spammer, and randomly send m email messages to n recipients. Does each recipient get about the same number of spam messasges?

```
This is similar to the last example, but there we wanted
each processor (or hash table entry) to get at most one
request (or data item) with probability 1/2. Here, we will
instead ask the following question: Given m requests assigned
randomly to n processors what is the smallest value of k such
that
     P(some processor gets k or more requests) <= 1/2.
In other words, we have a good chance (1/2) that no processor
has more than k requests to handle. (We could change the
constant 1/2 to .1 or .01 if we wanted to be more sure.

ASK&WAIT: Would changing 1/2 to a smaller value make the answer k larger
          or smaller? why?

          We will approximate this as follows. Consider just processor 1.
          The probability that processor 1 gets k or more requests is
          the same as the probability that after flipping a biased coin
          m times where the coin
             comes up "assign to processor 1"       with probability 1/n, and
             comes up "assign to another processor" with probability (n-1)/n
          the number of times the coin comes up "processor 1" is
          k, k+1,..., m. This  probability is
             P(processor 1 has at least k requests) =
                P_1(k,m,n) =
                  sum_{j=k to m} C(m,j) (1/n)^j * ((n-1)/n)^(m-j)
          Note that the analogous function for any other processor,
          say P_i(...) for processor i, is the same. Therefore, the
            P(some processor will have at least k requests)
              = P(proc 1 has at least k requests or
                  proc 2 has at least k requests or ...
                  proc n has at least k requests)
             <= P(proc 1 has at least k requests) +
                P(proc 2 has at least k requests) + ...
                P(proc n has at least k requests)
                  ... because the probability of a union of events
                  ... it at most the sum of the individual probabilities.
                  ... We only get an upper bound because the events can
                  ... overlap, i.e. more than 1 processor may
                  ... simultaneously have more than k requests)
               = n*P_1(k,m,n)
                  ... since all the functions P_i(k,m,n) are the same
```

Suppose we choose k as small as possible (and depending on m and n) so that

P_1(k,m,n) <= 1/(2*n)

Then

P(some processor will have at least k requests) <= n/(2*n) = 1/2

as desired.

The Central Limit Theorem can be used, and we discover (in the special case of m=n), that the value of k is quite small, namely k ~ 2*log n / log log n. So if n = 10^6, then the probability is 1/2 that no processor has more 11 requests. If 250M pieces of spam are randomly mailed to 250M recipients, the probability is 1/2 that no one will get more than 12 pieces of spam.