Student Name: Duc Tu Luong  
Last 3-digit ID: 122

# Homework #2

**Problem 1. Chapter 2, Exercise 8 (Stress Testing), Page 69.**

1. Model of the problem  
   You are doing stress testing using a ladder with *n* rungs and a various model of jars to determine the height from which they can be dropped and still not break. The highest rung from which you can drop a copy of the jar and not have it breakis called *the highest safe rung.* The problem is we can perform fewer drops if we are willing to break more jars. In this experiment, you are given a fixed “budget” of *k >= 1* jars, you must determine the highest safe rung, and can use at most *k* jars in doing so.
2. Suppose you are given a budget of k = 2 jars. Describe a strategy for finding the highest safe rung that requires you to drop a jar at most *f(n)* times, for some function *f(n)* that grows slower than linearly.
3. Now suppose you have a budget of *k > 2* jars, for some given *k*. Describe a strategy for finding the highest safe rung using at most *k* jar.
4. Overall idea of the algorithm  
   a) Let *n* be the number of rungs of the ladder, suppose *n=16*, then if we divide the ladder in blocks of , we will have 4 blocks of 4 rungs for each block. The algorithm will test the smallest rung in each block (we only test one rung for each block) until we have a crashed jar. Now we have the highest rung possible that the jar will break if we drop, we need to run a loop inside that block to test again to find out the exact rung that the jar can be dropped without being crashed.

b) If k > 2, we can start dropping the jars from the heights that are multiple of . Thus, the first jar should crash at most times. Then, if the first jar breaks at say , we know that the highest safest rung should lie between and . Now we recursively repeat the algorithm for *k – 1* jars between and .

1. Pseudo code  
   a)  
   Let n be the number of rungs

Let b be the number of blocks, b =

Round b, let h be the current rung and h = b

While (h <= n)

Drop jar from rung hth

If jar is crashed

Let s be the lowest rung in the block, and s = (h - b + 1)

For (i = s; i < h; i++)

Drop jar from rung ith

If jar is crashed

h = i - 1;

return h;

Endif

Endfor

Endif

h = h + b;

Endwhile

If h > n

For(i = h - b + 1; i <= n; i++)

Drop jar from rung ith

If jar is crashed

h = i - 1;

return h;

Endif

Endfor

Endif  
  
b)  
Let b be the block of rung

Let n be the number of rungs

Let k be the number of jars

Let h be the highest safest rung

Let ht be the temporary highest safest rung

Test(h, n, k) {

If k > 1

Let b = round( n / ( ( k - 1 ) / k ) )

Let h = b

while (h <= n)

Drop jar from rung hth

If jar is crashed

Let ht = h

Let h = 0

Let k = k - 1

Test(h, ht, k)

Endif

Let h = h + b

Endwhile

If h > n

Test(h - b, n - (h - b), k)

Endif

Else

For(i = ht - b, i++, i < ht)

Drop jar from rung ith

If jar is crashed

Let h = i – 1

return h

Endif

Endfor

Endif

}

1. a) Suppose we have 9 rungs, *n = 9*, then we have blocks, 3 rungs in each block. Let *h = b = 3*, at the first run of the algorithm, we drop jar from rung 3rd, if the jar crashed, we know rung 3rd is the upper bound limit rung which we can drop jar without being crashed. Now we have one more jar to drop, we loop inside block of rung from rung 1st to rung 2nd, if the jar crash at any rung, we know for sure the previous rung is the highest safe rung we need to find. If we finish the algorithm without crashing any jar, we can say that the highest safest rung *h = n = 9*.   
     
   There is also a case the number of rungs are not nicely square root, for e.g. *n = 10*. In such cases, after running through all blocks without crashing any jars, we still have some leftover rungs to test, that’s why after finishing while loop, if the current *h* is bigger than *n*, we need to run a loop from *h – b* to *n* find the highest safe rung.

b) In this case, we run the algorithm recursively. We have more jars to drop, which means we need to divide the ladder in more blocks. Specifically, let b = , we run the test similar to the first algorithm, but instead, we call the function Test() recursively with 3 parameters and k = 1 is the condition to escape the recursive function. Each time we crash a jar, we obviously decrease k by 1, and we now have a temporary variable ht to store current highest safe rung, we then call function Test() recursively but pass on different variables because we have a smaller range to drop jars and less jars to drop.

1. Time complexity  
   a) In the first loop of this algorithm, we run times to find out the maximum rung that the jar can drop, for the second loop, we also run times to find out the exact rung, so for the whole process, the time complexity is . This algorithm grows slower than linearly.   
   b) The algorithm will run which is less than . So the time complexity should be O().

**Problem 2. Chapter 3, Exercise 4 (Butterfly Studies), Page 107.**

1. Model of the problem  
     
   We have n butterflies which need to be categorized into either A or B. For each pair of specimens i and j, we study them carefully side by side. If we are confident enough, then we label the pair (i, j) either “same” or “different”. And if we don’t have enough information, we will give no judgement for that pair, which is called *ambiguous*.

So now we have the collection of n specimens and a collection of m judgments (either "same" or "different") for the pairs that were not declared to be *ambiguous*. We will declare the m judgments to be *consistent* if it is possible to label each specimen either A or B in such a way that for each pair *(i, j)* labeled "same," it is the case that i and j have the same label; and for each pair (i, j) labeled "different," it is the case that i and j have different labels.

Give an algorithm with running time *O(m + n)* that determines whether the m judgments are consistent.

1. Overall idea of the algorithm:  
     
   We have a set of n specimens and m judgements; we need to show if the set of judgements are consistent or not. We construct an undirected graph G = (V, E), each specimen is a vertex. There is an edge between vi and vj if there is a clear judgement about those specimens.

After that, we randomly select some vertex as the starting node s. We perform Breadth-First Search algorithm on Gi starting at si. For each vertex vk that is visited from vj, consider the judgement made on the specimens (vj, vk). If the judgement was “same”, label vk the same as vj and if the judgement was “different”, label vk the different from vj. There are specimens that are not labeled by any judgments. These specimens maybe labeled randomly. Once the labeling is complete we may double check on the list of judgements for consistency.

1. Pseudo code:

Randomly pick a starting node s and label it A

Mark s as “Visited”

Define a set R = {s}.

Define layer L(0) = {s}.

For (i = 0; i < n; i++)

For each node u in L(i)

Consider each edge (u, v)

If v is not marked “Visited”

Mark v “Visited”

If the judgement (u, v) was “same”

label v the same as u

else if the judgement was “different”

label v different from u

Endif

Add v to R and update layer L(i+1)

Endif

Endfor

Endfor

Endfor

For each edge (u, v)

If the judgement was “same” and u and v have different labels

Judgements were not consistent

Else if the judgement was “different” and u and v have the same labels

Judgements were not consistent

Endif

Endfor

1. Time complexity  
   Because we use BFS algorithm, so the time complexity should be 0(m + n)