Assignment 2

Algorithms & Complexity (CIS 522-01)

Javier Arechalde

February 7, 2018

Stress Testing

Model description

In this problem, we are doing some stress-testing on various models of glass jars, to determine the highest distance they can be dropped without breaking.

We have a ladder with n rungs, and we want to find the *highest safe rung*, that is the distance that we described in the last paragraph. We also have k jars, and this number of available jars, will be limited depending on the "budget" for the test.

a.

In this case our budget is limited to $k = 2$ and we want to find a solution $f(n)$ that grows slower than linearly. The breaking distance for the first jar k_1 is bd and the breaking distance for the second jar k_2 is bd too, as they are models of the same jar.

The current rung we are dropping our jars from is r, and the highest safe run will be assigned to sr.

Overall idea

In case we are given 2 jars, $k = 2$, we will use one algorithm with a different approach, because if we use linear search, our solution will grow linearly, but if we use binary search, we will exceed the number of available jars we have for this problem.

So in our solution, we will divide the ladder into m divisions. This way, our algorithm will take $(m + n/m)$ steps at most. This can be explained because in the worst case scenario, we will need to go over all the m divisions to find the highest safe rung, and then go to the start of the previous division before it broke, and go to the next division, which is n/m steps away from the previous division.

Pseudocode

Algorithm 1 My implementation

1: At the beginning $r = 0$ and k_1, k_2 are not broken
2: We have n rungs, and we chose to divide our rungs into $m = 4$
3: **while** k_1 not broken **do**
4:     Start increasing distance
5:     Saving last ring $r_0 = r$
6:     $r = r + n/m$
7:     **if** $r > bd$ **then**
8:         k_1 breaks at rung r
9:     **end if**
10: **end while**
11: We start our next iterations from r = r0
12: **while** k_2 not broken **do**
13:     r = r + 1
14:     **if** $r > bd$ **then**
15:         We return $sr = r - 1$, safest rung distance for the jars not to break
16:     **end if**
17: **end while**

Example

Now, we will prove that our algorithm works, by implementing it and running it.

```
#We have two jars that are not broken k1,k2
k1 = 'Ok'
k2 = 'Ok'

bd = 11 #Breaking distance
n = 16 #In this case we have n rungs, where n = 16
r0 = r = 0 #Starting rung
m = 4 #We divide our rungs in 4 equal parts


while(k1 == 'Ok'):
  r0 = r
  r = r0 + n/m
  print('Current_rung:_%i ' %r)
  if(r>bd):
   k1 = 'RIP'
   print('k1_Broke\n')
```

```
#The  safest  rung  before  break  point  will  be
r  =  r0

while(k2 == 'Ok'):
 r = r+1
 print('Current_rung:_%i' %r)
 if(r>bd):
    k2 ='RIP'
    print('k2_Broke\n')
    sr = r-1
    print('Safest_rung_distance:_%i' %sr)
```

In the image shown below, you can see the results of running this algorithm, in an example setup.


```
Current rung: 4
Current rung: 8
Current rung: 12
k1 Broke

Current rung: 9
Current rung: 10
Current rung: 11
Current rung: 12
k2 Broke

Safest rung distance: 11
```

Figure 1: Results of running algorithm

Time complexity analysis

b.

In this case, out budget is limited to k jars, where $k > 2$. We want to find the highest safe rung using at most k jars. For each jar k_i the number of times we drop whis jar should be less than the number of times we dropped the previous jar k_{i-1} so $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$.

The current rung we are dropping our jars from is r, and the highest safe run will be assigned to sr.

Overall idea

In case we are given 2 jars, $k = 2$, we will use one algorithm with a different approach, because if we use linear search, our solution will grow linearly, but if we use binary search, we will exceed the number of available jars we have for this problem.

So in our solution, we will divide the ladder into m divisions. This way, our algorithm will take $(m + n/m)$ steps at most. This can be explained because in the worst case scenario, we will need to go over all the m divisions to find the highest safe rung, and then go to the start of the previous division before it broke, and go to the next division, which is n/m steps away from the previous division.

Butterfly Studies

Model description

In this problem, we have n butterflies, we want to separate then in two groups, let's call them A and B. It doesn't matter in which group we classify each one of the butterflies, because we only want to separate them in two groups, we don't need to put them in the correct group.

To complete this task, we are given a m comparisons that dictaminate if the pair of specimens i, j belong to the same group, or they are in two separate groups. This number of m comparisons, is smaller than the possible number of pairs in the set of specimens, which is $n(n - 1)/2$, because some pairs are ambiguous, which means that we are not sure if the pair belongs to the same group or not.

We want to determine if this set of m comparisons is consistant, and thus, we are separating the butterflies coherently.

Overall idea

We will have a dictionary containing the n different specimens that we want to separate. This way, we can check in only O(1), the group each specimen its assigned.

We will have a set of m tuples that dictaminates if both specimens are in 'S' (Same group) or 'D' (different group). We also assume that this set of tuples

comes in order, starting first with the pairs that contain specimen 1, then the pairs that contain specimen 2, etc.

We will start going tuple by tuple, if none of them have a group assigned, we will assign them to the same group, or to a different group, depending the notation on that tuple. If only one member of the tuple has a group assigned, we will assign the other tuple to the same group or the different one, according to the notation on that tuple. In the end, if both tuples have a group already assigned, we will proceed to check if this new notation is consistent or not. In the case it's not consistent, the whole classification is unconsistent, and we stop our algorihtm.

Pseudocode

Algorithm 2 My Implementation

1: **while** The pairs are consistent, for every tuple in m **do**
2:     Take one tuple $m_i \in m$
3:     **if** None of the tuple members have a group assigned **then**
4:         **if** Tuple in the same group **then**
5:             Assign group A to $m_i[0]$
6:             Assign group A to $m_i[1]$
7:         **else if** Tuple in different group **then**
8:             Assign group A to $m_i[0]$.
9:             Assign group B to $m_i[1]$.
10:         **end if**
11:     **end if**
12:     **if** One of the members in the tuple has a group assigned **then**
13:         **if** Tuple in the same group **then**
14:             Assign $Group(m_i[0])$ to $m_i[1]$
15:         **else if** Tuple in different group **then**
16:             Assign opposite $Group(m_i[0])$ to $m_i[1]$
17:         **end if**
18:     **end if**
19:     **if** Both members on the tuple have a group assigned already **then**
20:         **if** Tuple in the same group **then**
21:             **if** They have different groups assigned **then**
22:                 Inconsistency!
23:             **else**
24:                 continue
25:             **end if**
26:         **else if** Tuple in different group **then**
27:             **if** They have same groups assigned **then**
28:                 Inconsistency!
29:             **else**
30:                 continue
31:             **end if**
32:         **end if**
33:     **end if**
34: **end while**

Example

To prove that our algorithm works, we implemented it in Python, and we ran it over a sample dataset.

#Here is the list of all the elements

```python
n = {1:None,2:None,3:None,4:None}

#Possible tuples
m = [(1,2,'S'),(1,3,'D'),(1,4,'S'),(2,3,'D'),(2,4,'S'),(3,4,'D')]

for tuple in m:
 t0 = tuple[0]
 t1 = tuple[1]
 assign = tuple[2]

 print(n)

 #If both of them has a group assigned
 if n[t0] != None and n[t1] != None:
  if assign == 'S':
   if n[t0] == n[t1]:
    continue
   else:
    print('Inconsistency in tuple: ',tuple)
    break
  elif assign == 'D':
   if n[t0] != n[t1]:
    continue
   else:
    print('Inconsistency in tuple: ',tuple)
    break

 #If one of them has a group assigned
 elif n[t0] != None or n[t1] != None:
  if n[t0] != None:
   if assign == 'S':
    n[t1] = n[t0]
    continue
   elif assign == 'D':
    if n[t0] == 'A':
     n[t1] = 'B'
     continue
    else:
     n[t1] = 'A'
     continue
  elif n[t1] != None:
   if assign == 'S':
    n[t0] = n[t1]
    continue
   elif assign == 'D':
    if n[t0] == 'A':
```

```
        n[t1] = 'B'
        continue
      else:
        n[t1] = 'A'
        continue

  #If none of them have a group assigned
  elif n[t0] == n[t1] == None:
    if assign == 'S':
      n[t0] = 'A'
      n[t1] = 'A'
      continue
    else:
      n[t0] = 'A'
      n[t1] = 'B'
      continue
```

Time complexity analysis