Assignment 2

Algorithms & Complexity (CIS 522-01)

Javier Arechalde

February 9, 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.

In our implementation, we will try a different approach, so we make the best use of the two jars that we have available, while keeping our solution growing slower than linearly. We will divide the set of n rungs into m parts, each one of these parts containing n/m parts.

First we will use the first jar iterating over the m parts until the jar breaks, by steps of size n/m. Once the first jar breaks, we will then start from the previous rung (distanced n/m from the rung on which the jar broke) with the next jar, increasing the rungs by 1, then the maximum safe rung will be the previous rung on which the jar broke.

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

In this section, we will show our implementation, and we will run it over a sample set.

```
#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')
```
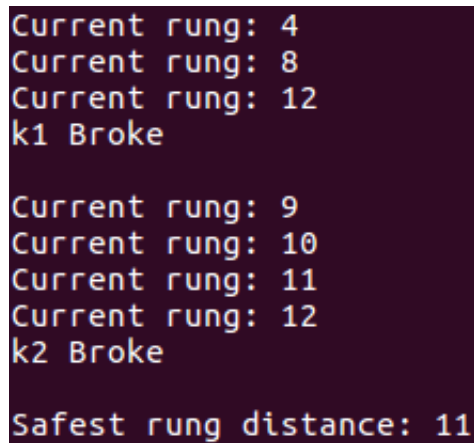
2

```
#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 over the sample set.


```
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

Time complexity analysis

As we stated before, in our solution, we will divide the ladder into m divisions. This way, our algorithm will take $(m + n/m)$ steps at most, so then the time complexity of our implementation will be $O(m + n/m)$. This can be explained because in the worst case scenario, we will need to go over all the m divisions to find the rung on which the first jar breaks, and then go to the start of the previous division before it broke, then iterate towards next division, on steps of 1, which is n/m steps away, at most, from the next division.

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.

In case, we have k jars to run our test, we will try a different approach. One of the requirements of the algorithm design was that each next iteration was exponiently slower than the previous one. Then, our plan is to start dividing the rungs into $n^{(1/2)}$ divisions, and when we find the point at which the jar breaks, take that section that goes from the previous division and the point on which the jar broke and divide that section into $n^{(1/3)}$ divisons, find that breaking point, and continue increasing the denominator in the exponent, so the number of checks we have to make exponentially decreases between iterations, while we get closer to the solution.

Pseudocode

Algorithm 2 My implementation

1: **function** TEST(start,end,k,jar)
2:     Divide the rungs into $n^{(}1/k+1)$ divisions
3:     Find the step size $step = (start - end)/(number of divisions)$
4:     **while** Jar didnt break **do**
5:         Increase drop distance by the step size $step$
6:         **if** Drop distance is greater than the breaking point **then**
7:             Jar k breaks
8:             Report highest safe rung, which would be $sr = r - step$
9:             **if** We still have jars available **then**
        TEST(k+1,previousrung,rungbroke)
10:             **end if**
11:         **end if**
12:     **end while**
13: **end function**

Example

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

```python
#Number of rungs
n = 2**32

print('Number of rungs: %i\n' %n)

jar = 'OK' #Set the status of the initial jar to OK
bd = 30000 #Breaking distance
r0 = 0 #Starting rung
kmax = 12 # Maximum number of jars for the test

def test(start,end,k,jar):
 print('Starting rung: %i Ending rung: %i' %(start,end))
 jar = 'OK' #We set the status of the current jar to OK
 r = start #Starting rung
 #We calculate the step size and the number of divisions
 exp = (1./(k+1))
 div = (end-start)**(exp)
 step = (end-start)/div

 #Convert the number of divisions and steps to integers
 div = int(div)
```

```python
step = int(step)

#If the number of divisions is only one, we will fix the step
#size to 1
if div == 1:
  step = 1

print('Divisions: %i Step size: %i' %(div, step))

#While the jar is not broken, we increase the drop by the step size
while (jar=='OK'):
  r0 = r #Save the current safest rung
  r = r + step #Current rung
  if (r>bd):
    jar = 'RIP'
    print('Current Highest safe rung: %i\n'  %r0)
    k = k +1
    #While we still have jars available, and the number of divisions is greater
    #than one, we continue to search
    if (k<= kmax and div >1):
      test(r0,r,k,jar)

test(0,n,1,jar)
```

Here are the results we obtained by running the algorithm over the sample dataset.

Figure 2: Results

Important note here, is that due to the requirements of this implementation, we might not be able to find the real highest safe rung, because the step size may skip the actual highest safe rung. To solve this I fixed step size to 1 for when the number of divisions is just one, this way we can find the highest safe rung, even though it may not suffice the requirements of this problem.

Time complexity analysis

For this problem, we recursively reduced the number of rungs we have to check to find the highest safe rung, then the time complexity for this implementation will be $n^{1/2} + n^{1/3} + ... + n^{k+1}$ so the complexity in big-O notation will be given

by $O(n^{1/2})$.

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 3 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,'S')]

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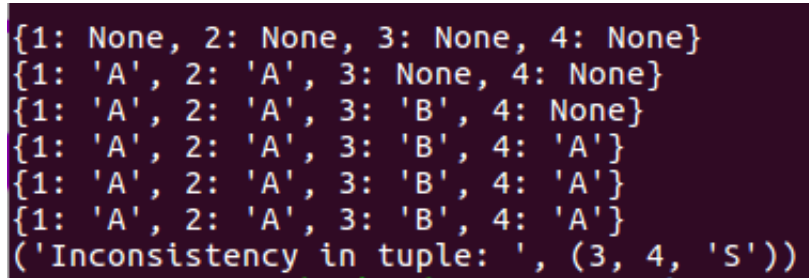
```
      n[ t1 ] = 'B'
      continue
     else :
      n[ t1 ] = 'A'
      continue

  #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
```

Here are the results we obtained by running the algorithm over the sample dataset.

```
{1: None, 2: None, 3: None, 4: None}
{1: 'A', 2: 'A', 3: None, 4: None}
{1: 'A', 2: 'A', 3: 'B', 4: None}
{1: 'A', 2: 'A', 3: 'B', 4: 'A'}
{1: 'A', 2: 'A', 3: 'B', 4: 'A'}
{1: 'A', 2: 'A', 3: 'B', 4: 'A'}
('Inconsistency in tuple: ', (3, 4, 'S'))
```

Figure 3: Results

Time complexity analysis

Our algorithm will run over all the tuples, this takes $O(m)$ time, because m is the number of tuples in our dataset. Then, our algorithm will have to check at most, the n specimens, to assign this specimens a group, this would take $O(n)$ time. So our algorithm time complexity will be $O(m + n)$ then.