# Assignment 2

## Algorithms & Complexity (CIS 522-01)

*Javier Arechalde*

February 9, 2018

# Stress Testing

## Model description

In this problem, we are stress-testing different models of glass jars, to determine the highes distance they can be dropped without breaking.

For the test setup we have a ladder with $n$ rungs, and as we stated before, we want to find the *highest safe rung*, that is the maximum distance we can drop a jar from, without it breaking. For this test, we have $k$ jars available, this number of jars, will change depending, on the requirements of each type of 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 two 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 from linear or binary, 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.

For running this test, 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 second jar broke.

## Pseudocode

**Algorithm 1** My implementation

At the beginning $r = 0$ and $k_1, k_2$ are not broken
We have $n$ rungs, and we chose to separate our rungs into $m$ divisions
**while** $k_1$ not broken **do**
    Start increasing distance in steps of $n/m$
    Saving last ring $r_0 = r$
    $r = r + n/m$
    **if** $r > bd$ **then**
        $k_1$ breaks at rung r
    **end if**
**end while**
We start our next iterations from r = r0
**while** $k_2$ not broken **do**
    r = r + 1
    **if** $r > bd$ **then**
        $k_2$ breaks at rung $r$
        We return $sr = r - 1$, safest rung distance for the jars not to break
    **end if**
**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')
```

```
#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.

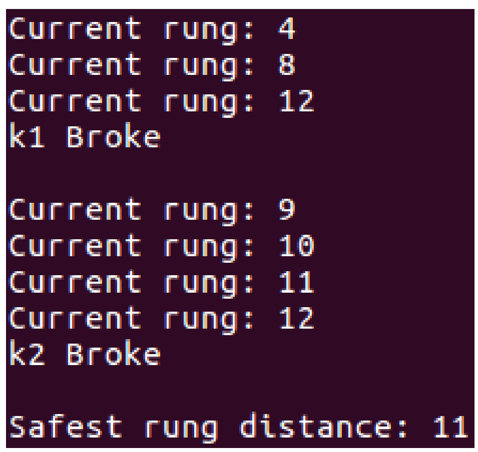


Figure 1: Results

## 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 reduced exponentially compared to 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 have $k$ jars to run our test, we will try this approach. One of the requirements for the solution is that the number of times we had to drop the jar exponientally decreases with each iteration. To fulfill this requirement, we decided to divide the jars into $n^{1/(k+1)}$ divisions, at each step, being $n$ the distance we are working with, being the whole rung distance at the beginning of our algorithm, and decreasing to the space between the point the jar broke, and the previous rung, with was a step size away from that rung. This way, at each iteration of our algorithm, the number of divisions we divide our rungs in will exponentially decrease at each step.

## Pseudocode

**Algorithm 2** My implementation

```
function TEST(start,end,k,jar)
    Divide the rungs into n^(1/k + 1) divisions
    Find the step size step = (start − end)/(numberofdivisions)
    while Jar didnt break do
        Increase drop distance by the step size step
        if Drop distance is greater than the breaking point then
            Jar k breaks
            Report highest safe rung, which would be sr = r − step
            if We still have jars available then
  TEST(k+1,previousrung,rungbroke)
            end if
        end if
    end while
end function
```

## Example

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

```
#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)
 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.

```
Number of rungs: 4294967296

Starting rung: 0 Ending rung: 4294967296
Divisions: 65536 Step size: 65536
Current Highest safe rung: 0

Starting rung: 0 Ending rung: 65536
Divisions: 40 Step size: 1625
Current Highest safe rung: 29250

Starting rung: 29250 Ending rung: 30875
Divisions: 6 Step size: 255
Current Highest safe rung: 29760

Starting rung: 29760 Ending rung: 30015
Divisions: 3 Step size: 84
Current Highest safe rung: 29928

Starting rung: 29928 Ending rung: 30012
Divisions: 2 Step size: 40
Current Highest safe rung: 29968

Starting rung: 29968 Ending rung: 30008
Divisions: 1 Step size: 1
Current Highest safe rung: 30000
```

Figure 2: Results

An important thing to note in this implementation, is that due to the nature of the algorithm, we may not be able to find the highest safe run, because our algorithm will skip it due to the step size. If we don't find a solution for this, we wont find the actual highest safe rung, but an approximation of it instead. My solution for this issue was to fix the step size when the number of divisions won't go lower than 1. This way, even though we are not fullfilling the requierement set in the problem for the last step, we will be able to fing the actual highest safe rung rather than an approximation.

### 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 run for this implementation will be $n^{1/2} + n^{1/3} + ... + n^{1/(k+1)}$ 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, and we want to separate then in two groups. Let's call this groups $A$ 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 an specific 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. Otherwise, we should order the pairs first.

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

```
while The pairs are consistent, for every tuple in m do
    Take one tuple m_i ∈ m
    if None of the tuple members have a group assigned then
        if Tuple in the same group then
            Assign group A to m_i[0]
            Assign group A to m_i[1]
        else if Tuple in different group then
            Assign group A to m_i[0].
            Assign group B to m_i[1].
        end if
    end if
    if One of the members in the tuple has a group assigned then[[
        if Tuple in the same group then
            Assign Group(m_i[0]) to m_i[1]
        else if Tuple in different group then
            Assign opposite Group(m_i[0]) to m_i[1]
        end if
    end if
    if Both members on the tuple have a group assigned already then
        if Tuple in the same group then
            if They have different groups assigned then
                Inconsistency!
            else
                continue
            end if
        else if Tuple in different group then
            if They have same groups assigned then
                Inconsistency!
            else
                continue
            end if
        end if
    end if
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
```

```
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.