Assignment 3

Algorithms & Complexity (CIS 522-01)

Javier Arechalde

February 21, 2018

1. Time-series data mining

1.1 Problem description

In this problem, we will be given two sequences of events, and we want to find out if the first sequence of events is a subsequence of the second given sequence. The events of the first sequence must appear in the same order in the second sequence too, but they dont necessarily need to be consecutive.

For example, we will be given the following sequence of events.

```
buy Yahoo, buy eBay, buy Yahoo, buy Oracle
```

We will also be given this other sequence, which may, or may not contain the first given sequence.

```
buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle
```

These two sequence of events will be named S' and S. Given these two sequence of events, S' of length m and S of length n each containing an event possibly more than once, we want to find in time $O(m+n)$ if S' is a subsequence of S, in the fastest way possible.

1.2 Proposed solution

Our proposed solution is to iterate over the two sequences S' and S simultaneously, to find if the first sequence is a subsequence of the second one. s We will start by taking the first element in S', then we will iterating over the S sequence to see if we can find that element. If we find that element in S, we move to the next element in S', and start iterating to search for this second element in the sequence S, starting from the S position right after the position on which we found the first element of S'. If we can't find that element in S, that means we iterated over all S and reached the final position in S without finding it. In that case we will return that S' is not a subset of S. If we find all the elements of S' in S, then we will return that S' is a subset of S.

1.3 Pseudo code

Algorithm 1 Checking if S' subset of S

```
1: We initialize $i_{pos} = j_{pos} = 0$
2: while We didnt reach the end of $S$ or $S'$ do
3:     Take $S(i_{pos})$
4:     for $j$ in range $j_{pos} \rightarrow length(S)$ do
5:         if $S'(i_{pos}) == S(j_{pos})$ then
6:             if $i_{pos} == length(S')$ then
7:                 $S'$ is a subsequence of $S$
8:             end if
9:             if $i_{pos} \neq length(S')$ then
10:                $i_{pos} + +$
11:                $j_{pos} = j$
12:                Break the loop
13:            end if
14:        end if
15:        if $S'(i_{pos}) \neq S(j_{pos})$ then
16:            $j + +$
17:        end if
18:    end for
19: end while
```

1.4 Example

We implemented the algorithm in *Python*, if we run the script, the algorithm will iterate over the two sequences to find if the first sequence is a subsequence of the second sequence. If it is it will print that S' is a subsequence of S, and the opposite otherwise.

```
#These are the two sets that we want to compare
s1 = ['buy Yahoo','buy eBay','buy Yahoo','buy Oracle']
s2 = ['buy Amazon','buy Yahoo','buy eBay','buy Yahoo','buy Yahoo','buy Oracle']

def comp(s1,s2):

 #Getting the length of the two sets
 l1 = len(s1)
 l2 = len(s2)

 #We initialize the operators
 ipos = 0
 jpos = 0
```

```
  while(jpos<l2 -1):
   el1 = s1[ipos]
   for j in range(jpos,l2):
    el2 = s2[j]

    #If the element is in that position of the list...
    if el1==el2:
     if(ipos==l1 -1):
      ipos = ipos+1
      print("S1 is a subset of S2")
      return
     elif(ipos!=l1):
      ipos = ipos+1
      jpos = j
      break #We exit the for loop

    #If we cant find the element in that position of the list
    #We move to the next element
    elif(el1!=el2):
     jpos = j
     continue

  #If the loop above didn't return that S1 is a subset of S2
  # we print the opposite
  print('S1 is not a subset of S2')
  return

comp(s1,s2)
```

After running over our implementation over a sample dataset, this are the results we obtained.

```
S1 is a subset of S2
```

1.5 Time complexity

In worst case scenario, the last element of S' will be in the last position of S, therefore, we would have iterated over both lists to check if S' is a subset of S. As the length of S' is m and the length of S is n, the time complexity of our implementation will be $O(m + n)$.

2. Competition scheduling

2.1 Problem description

In this problem, we are hosting a competition. In this competition, we plan to do a mini-thriatlon, in which contestants will have to swim 20 laps of a pool, then bike for 10 miles, and then run for 3 miles.

This competition must follow this rule: no more than one contestant can be swimming in the pool at a time.

Each one of the contestants has a projected *swimming time*, a projected *biking time*, and a projected *running time*. These are the projected times that will take each one of the constestants to complete each one of de different sections of the thriatlon.

Our goal is to design an efficient algorithm that produces an schedule whose competition's completion time is as small as possible.

In this problem scopes, each one of the contestants will be named c_i, and each one of them will have a $\{time_{swim}[i], time_bike[i], time_{run}[i]\}$ assigned that will be the projected finish time for each of the sections of the thriatlon.

2.2 Proposed solution

In this problem's case, we will have one resource that can be used only by one contestant at a time, and the rest of our resources can be shared without any limits by all the contestants. We have n requests to use the pool for an interval of time, being n the number of contestants.

Our implementation will consist in a *greedy* algorithm, that follows the *Earliest Deadline First* rule, that is sort the requests in ascending deadline order, and schedule first the jobs that end earlier. This may not be the best solution, but a greedy algorithm just chooses the 'most attractive' option at each step, that doesn't mean there is not another other possible solution that is optimal.

To sort the requests in ascending deadline order we will use **Quicksort** algorithm, that will take the contestants and their projected times as an input, and will return the same list sorted in increasing order.

Then our algorithm, will go contestant by contestant in the sorted list, assigning a start time and each time for each one of the sections that the contestant needs to complete in order to finish the mini-triathlon.

The output of our algorithm will be a set of scheduled intervals for each contestant $sched[i] = \{start_{swim}[i], end_{swim}[i], start_{bike}[i], end_{bike}[i], start_{run}[i], end_{run}[i]\}$ for each one of the contestants.

2.3 Pseudocode

Algorithm 2 Competition scheduling implementation

```
First we sort the students by completion time using QuickSort
function QUICKSORT(array)
    if Array length is 1 then
      return array
    end if
    if Array length $> 1$ then
        $pivot == array[0]$
        for Element in array do
            if $element < pivot$ then
                We add the element to the lowerlist
            end if
            if $element > pivot$ then
                We add element to the upperlist
            end if
            if d then
                Append element to the pivotlist
            end if
        end for
        upperlist = QUICKSORT(upperlist)
        lowerlist = QUICKSORT(lowerlist)
    end if
    Return lowerlist + pivotlist + upperlist
end function

Now we will start scheduling the students by increasing deadline
We initialize start time of swimming: $s_{swim} = 0$
while We didn't schedule all students do
    Get the timings for the corresponding participant
    Time swimming: $t_{swim}$
    Time biking: $t_{biking}$
    Time running $t_{running}$

    Swimming start and finish
    $f_{swim} = s_{swim} + t_{swim}$

    Biking start and finish
    $s_{bike} = f_{swim}$
    $f_{bike} = s_{bike} + t_{bike}$

    Running start and finish
    $s_{run} = f_{bike}$
    $f_{run} = s_{run} + t_{run}$

    We return the start and finish time for each one of the sections
end while
```

2.4 Example

In our implementation, we first calculate the total time that will take each one of the contestants to complete all the sections of the thriatlon. After this, we will sort the contestants in increasing order of completion time, and after they are sorted, we will pass them to the scheduler that will assign starting and finishing times for each one of the sections.

```
#We will first create a  list that contains the participants in the competition
#and their swimming time, biking time, and running time
contestants = [
(1,1,2,3), #Contestant #1
(2,4,6,6), #Contestant #2
(3,2,3,5),#...
(4,4,5,3)  #Contestant #N
]

#Deadlines list
deadline = [0]*len(contestants)

#We calculate the total deadline for swimming for each one of the contestants
for i in range(0,len(contestants)):
 contestant = contestants[i]
 deadline[i] = sum(contestant[1:4])

def quicksort(array,array0):
 upperlist = []
 lowerlist = []
 pivotlist = []

 upper0 = []
 lower0 = []
 pivot0 = []

 #If the length of the array is 1 it doesnt need sorting of any kind
 if len(array)<=1:
  return array, list(array0)

 #Else, we will separate the list into the upper elements and lower
 else:
  pivot = array[0]

  for i in range(0,len(array)):
   element = array[i]
   element0 = list(array0[i])
   if element<pivot:
    lowerlist.append(element)
    lower0.append(element0)
   elif element>pivot:
    upperlist.append(element)
    upper0.append(element0)
   else:
    pivotlist.append(element)
    pivot0.append(element0)

  upperlist,upper0 = quicksort(upperlist,upper0)
  lowerlist,lower0 = quicksort(lowerlist,lower0)

 return lowerlist + pivotlist + upperlist, list(lower0 + pivot0 + upper0)

#We will return an array with the indexes in the order that they need to be acce
schedule,sortcont = quicksort(deadline, contestants)

#We initialize the variables
s_swim = 0

#Now we will start scheduling the events
for contestant in sortcont:

 print("\n")
 print("Contestant␣ID:␣%i" %contestant[0])

 #The time that will take the contestant to finish each one of the sections
 t_swim = contestant[0]
 t_bike = contestant[1]
 t_run = contestant[2]

 #Swimming time
 f_swim = s_swim + t_swim

 #Biking time
 s_bike = f_swim
 f_bike = s_bike + t_bike

 #Running time
 s_run = f_bike
 f_run = s_run + t_run

 print("Start␣Swimming:␣%i␣End␣Swimming:␣%i" %(s_swim,f_swim))
 print("Start␣Biking:␣%i␣End␣Biking:␣%i" %(s_bike,f_bike))
 print("Start␣Running:␣%i␣End␣Running:␣%i" %(s_run,f_run))
```

```
#The next contestant will start swimming when the previous one is done
s_swim = f_swim
```

After running over our implementation over a sample dataset, this are the results we obtained.

```
Contestant ID: 1
Start Swimming: 0 End Swimming: 1
Start Biking: 1 End Biking: 2
Start Running: 2 End Running: 4


Contestant ID: 3
Start Swimming: 1 End Swimming: 4
Start Biking: 4 End Biking: 6
Start Running: 6 End Running: 9


Contestant ID: 4
Start Swimming: 4 End Swimming: 8
Start Biking: 8 End Biking: 12
Start Running: 12 End Running: 17


Contestant ID: 2
Start Swimming: 8 End Swimming: 10
Start Biking: 10 End Biking: 14
Start Running: 14 End Running: 20
```

2.5 Time complexity

In this problem, we used two different algorithms to assign the start and ending time for the different sections for each contestant. For **Quicksort** algorithm, the running time is $O(n \log n)$, and for our greedy scheduling algorithm, the running time will be $O(n)$, as we will have to iterate over all contestants.

Then the time complexity of our implementation will be $O(n \log n + n)$, but as $n \log n$ is an upper bound of n, we can conclude that the time complexity of our implementation is $O(n \log n)$.