HW3-solutions

March 11, 2019

0.1 Homework 3

0.1.1 - Amit Makashir

0.1.2 Question 1

Suppose two teams A and B are playing a match to see who is the first to win n games (from given n). Assume that A and B are equally competent, so each has a 50% chance of winning any particular game. Suppose they have already played i+j games, of which A won i and B has won j. Given an efficient algorithm to compute the probability that A will go on to win the match. For example, if i=n-1 and j=n-3 then the probability that A will win the match is 7/8, since it must win any of the next three games.

0.1.3 Solution:

The basic idea for this approach is to create two paths each time, one where A wins the current game and one where B wins. In the both the cases, we then recursively keep exploring each path until either A or B wins the match and propagate these results back to our starting point

```
# Store all the results that have been calculated in this dictionary
winning_probs = {}

def probability(i,j,n):
    pA = 0.5  # Probability of A and B winning a match is 0.5
    pB = 0.5

    if i == n:
        return 1
    elif j == n:
        return 0

    # If the result for this match was calculated, use that
    # Else calculate and store in dictionary
    if (i,j) not in winning_probs:
        # There are two cases here, either A wins this match or B
        Awinsthis = pA*probability(i+1,j,n)

        # B wins this game, but ultimately A wins the match
        Bwinsthis = pB*probability(i,j+1,n)

        winning_probs[(i,j)] = Awinsthis + Bwinsthis

    return winning_probs[(i,j)]
```

0.1.4 Question 2

Give an O(nt) algorithm for the following task. Input: a list of n positive integers a1, a2, . . . , an; a positive integer t. Question: does some subset of the ai's add up to t?

0.1.5 Solution:

We can use a Dynamic programming approach where we can either use the current element and or drop it and check the if total sum parameter becomes zero

In this approach, we basically create a lookup table with n rows and t columns. Therefore our time complexity for this algorithm is $O(nt)$

```
subset_found = {}
subset_indexes = []
t = "Total sum for the subset"
A = "array of ints (given)"
i = "Find this subset till this index"

def getSumSubset(i,t,A):

    if t > 0 and i < 0: # If we used all the elements and there is still total sum
        return False

    elif t == 0: # if rest of total sum is zero
        return True

    if t < 0 : # If the total sum becomes negative
        return False

    # There are 2 parameters to max function:
    # 1. Consider this element
    # 2. Don't consider this element
    # If this was calculated before, use it
    if (i,j) not in subset_found:
        subset_found[(i,t)] = max(getSumSubset(i-1,t-A[i],A),getSumSubset(i-1,t,A))

    return subset_found[(i,t)]
```

0.1.6 Question 3

Consider the following problem: given an array A with n numbers A[1], A[2], . . . , A[n], find two numbers A[i] and A[j], i < j, whose difference A[j] – A[i] is maximum. a) devise a divide-

and-conquer algorithm to solve the problem; what is the running time complexity? B) devise a dynamic programming algorithm to solve it. What is the running time complexity.

0.1.7 Solution:

Divide and Conquer: We divide the array into two parts each time till we have just one element on each side. Everytime we can check if: 1. the leftside has maxDifference elements 2. the rightside has maxDifference elements 3. the min element is from left side and max element from right side

This approach is similar to Merge sort, where we keep dividing things till the smallest unit. In this case, we we divide by half every time, so we would expect to have $log_2(n)$ steps to reach smallest element, and then to calculate min,max a linear search. So the algorithm would have a time complexity of $O(nlog(n))$.

```
def maxDifference(A):
    n = len(A)

    if n <= 0 :
        return -inf

    # Check the max Difference on Right and left side
    left = maxDifference(A[:n])
    right = maxDifference(A[n:])

    # Calculate the difference between min element from left and max from right
    minL,maxR = getMinMax(left,right)
    diffLR = minR - minL

    return max(left,right,diffLR)
```

Dynamic Programming: We can check which was the min. element before the current index for the given array. We then calculate difference for all and check which difference is max.

As we are storing the result of min element before this element for every element, we have avoided an extra iteration. Overall, we iterating the list 2 times so the algorithm complexity is $O(2n)$ or simply $O(n)$.

```
# Let A be our array of numbers

def maxDifference(A):
    min_elements = []
    n = len(A)

    for i in range(n):
        # If this is the first element, just push to the array
        if len(min_elements) == 0:
            min_elements.append(i)

        # Else compare this element and previous min element,
        # and store the min amongst these
        else:
            prev_min_index = min_elements[-1]

            # Check if the last element is smaller the min element before that
            if A[i-1] > A[prev_min_index]:
                min_elements.append(prev_min_index)
            else:
                min_elements.append(i-1)

    max_diff = 0
    min_index = 0
    max_index = 0

    for i in range(n):
        diff = A[i] - A[min_elements[i]]

        # If this difference is max till now, update the max diff and indexes
        if diff > max_diff:
            max_diff = diff
            min_index = min_elements[i]
            max_index = A[i]

    return max_diff,min_index,max_diff
```

0.1.8 Question 4

Alice wants to organize a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five people whom they don't know. Given as input the list of n people and the list of pairs who know each other, devise an algorithm to output the best choice of party invitees.

0.1.9 Solution:

We first converted the list of pairs to a dictionary of known people, let's call it "connections". We also made an invite list which was initialized to all the people before the passing to the function.

We can iterate through every person on the invite list and check if that person meets the two conditions set by Alice. If the person does meet those conditions, we move onto next person. If the person doesn't meet any one of those conditions or both we remove this person from the invite list and call this function again. This means, we are starting our process again without the person we removed in our last operation.

Ultimately, after iterating through all person till they meet those conditions we will have a list of people that meet Alice's requirement and we would invite them to the party.

```
people_to_invite = people   # add all people to the invite list

def partyInvites(people_to_invite,connections,people):
    '''
    people => list of all people Alice is considering to invite

    connections => dictionary of known people made out of the
    list of pairs of people who know each other

    eg: Connections = {"A": ["B","C"]},
    A knows B and C
    '''

    for person in people_to_invite:
        # We'll have two functions to check if this person meets Alice's requirement

        # True if this person knows 5 or more people in the invite list,
        # else false
        condition1 = meetsKnowCondition(person,people_to_invite,connections)

        # True if this person does not know 5 people or more in the invite list,
        # else false
        condition2 = meetsNotKnowCondition(person,people_to_invite,connections)

        if condition1 and condition2: # If both are True
            continue  # Go to the next person
        else: # If one or both condition are False
            people_to_invite = removePerson(person,people_to_invite)
            partyInvites(people_to_invite,connections,people)

    return people_to_invite
```

0.1.10 Question 5

Consider the following variation on the Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

0.1.11 Solution:

The solution told by Professor in class wouldn't directly apply here because of the circular nature of the problem. The jobs appear periodically instead of linearly in time. We can apply the greedy algorithm taught in class if we somehow convert the problem to 1D time.

The challenge here is due the jobs that span over the midnight. Without these jobs, the problem can be solved with the greedy algorithm where we choose the jobs with earliest finish times. As we are sorting jobs for every midnight job, this algorithm is at least $O(n^2 log(n))$

Algorithm:

1. Convert the jobs from 12 hour format to 24 hour format
2. Get the list of all jobs that span over midnight
3. For every such job (and without the other midnight spanning jobs), solve remaining problem greedily
4. The greedy approach chooses the earliest finish times and removes all conflicting jobs till we have all compatible jobs

```
# jobs = [(start,end)] ... in 24 hour format

def maxJobs(jobs):

    max_no_jobs = 0
    max_jobs = []

    # getMidnightJobs() returns a list of jobs spanning over midnight
    # and all the other jobs (sorted asc on finish times)
    midnight_jobs,other_jobs = getMidnightJobs(jobs)

    # Iterate over all jobs in the midnight
    for job in midnight_jobs:
        chosen_jobs = other_jobs

        i = 0  # Counter to keep track of while iterations
        # and job to pick in the next iteration

        # Check if there are any conflicting jobs in "chosen_jobs" list,
        # if so return True
        while isConflict(chosen_jobs):

            # In the first iteration of while loop, choose
            # the first job (sorted asc on finish times)

            # After removing all the conflicts to the first job,
            # the second will be choosen and so on

            # getConflicts() returns a list of jobs conflicting with the given job
            conflicts = getConflicts(chosen_jobs[i])

            # removeConflicts() returns a list of jobs after removing the conflicts
            chosen_jobs = removeConflicts(conflicts,chosen_jobs)

            i += 1
```

```python
        chosen_jobs.append(job) # Add the currently iterating midnight job
        # to the list of jobs chosen

        if len(chosen_jobs) > max_no_jobs:
            max_no_jobs = len(chosen_jobs)
            max_jobs = chosen_jobs

    return max_jobs
```