CPSC 320 2018W2: Assignment 3

These problems touch on greedy algorithms and divide and conquer algorithms. You'll also work on related problems in your week 6 and 7 classes and tutorials.

Please follow the guidelines given in Assignment 1 for submission to Gradescope, and for group collaboration. Remember to provide short justifications for your answers. Submit by the deadline **Monday February 18, 2019 at 10PM**.

For this and future assignments, you must use LaTeXto prepare your answers. Easiest will be to use the .tex file provided. For questions where you need to select a circle, you can change `\fillinMCmath` to `\fillinMCmathsoln` for your choice of answer. Similarly, for a "fill in the box" question, you can change `\fillinblank{?}` to `\fillinblanksoln{Your solution here}`. (And so on.)

Please enclose each paragraph of your solution in `\soln{Your solution here...}`.

Your solution will then appear in dark blue, making it a lot easier for TAs to find the parts that you wrote.

1 You are such a cheapskate!

Your significant other has a list of n "special dates" for which you are expected to treat them to dinner. Associated with each special date s is a deadline d with $s \leq d$. In order to earn credit for date s, you should schedule the dinner within the window $[s, d]$. That is, the dinner must be on or after day s and on or before day d. Otherwise you will suffer unspeakable consequences. (For instances of the problem in which an inordinate number of special dates and deadlines lie in a very short window, you may have to schedule more than one dinner on the same date in order to meet the constraints; this is allowed!)

One dinner can credit up to k special days. For instance, suppose that $k = 2$, there is a special day on February 10 with a deadline of February 24, and another special day on February 14 with a deadline of February 18. Then you can buy your partner dinner on February 16 to credit both of these special dates, but can't cover any other special dates with this dinner.

Given a set P of windows (that is, pairs (s, d) with $s \leq d$) and a number $k \geq 1$, the following algorithm minimizes the number of dinner dates. (After all, you are very busy with your studies and cannot afford to take many evenings off.) The algorithm greedily schedules a dinner on the earliest deadline, then chooses as many pairs as possible (up to k) that are "covered" by this dinner, choosing those pairs with the earliest deadlines. The process is repeated on the remaining subproblem until all special dates are covered. The algorithm's output is a list of days on which you go out for dinner together.

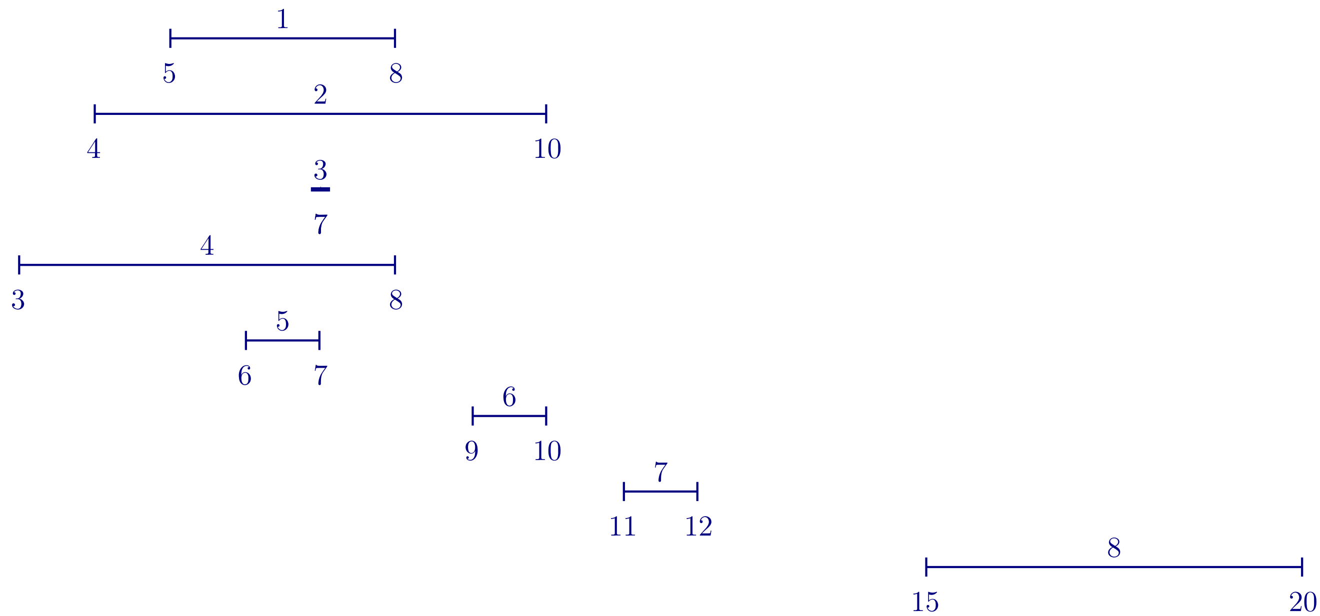
Algorithm Choose-Dinner-Dates(P, k)
  If $n = 0$
    Return the empty list
  Else
    Set $d_{\min}$ to be the earliest deadline of any pair in P
    $i = 0$
    While $i < k$ and there is some pair (s', d') in P with $s' \leq d_{\min}$
      Choose such a pair (s', d') with the earliest deadline (i.e., smallest d')
      Remove pair (s', d') from P
      Increment i
    Return $\{d_{\min}\}$ + Choose-Dinner-Dates(P, k)

1. Give a moderately-sized ($n \leq 8$, $k \leq 3$) instance, showing the dates where this greedy solution schedules the dinners, and a different optimal (not necessarily greedy) solution also scheduling a minimal number of dinners. A diagram may be helpful in illustrating your instance and solutions.

   Instance of "Special dates":

   $$P = [(5,8),(4,10),(7,7),(3,8),(6,7),(9,10),(11,12),(15,20)]$$

   $$k = 3$$

   Solution for Choose-Dinner-Dates:

   $$[7,7,10,12,20]$$

   Different solution:

   $$[7,8,10,11,15]$$

2. Show that Algorithm Choose-Dinner-Dates produces a valid solution, that is, one for which you earn credit for every special date.

   On each call to the algorithm we get the earliest deadline d_{min} of all P and remove from P at most k elements of P that have an $s <= d_{min}$. By doing this every element we delete from P will receive a credit for a date on d_{min}. This is because d_{min} is the earliest deadline in P, so we can cover from d_{min} all the "special dates" that have a start s less than or equal to d_{min}, because we know that their corresponding deadlines must be equal or greater than d_{min}.

3. Show that Algorithm Choose-Dinner-Dates produces an optimal solution, that is, one that minimizes the number of dinners scheduled.

   Consider the following solutions for the instance in question 1:

   $$S = [7,7,10,12,20] \text{ -> Choose-Dinner-Dates solution}$$

   We know that S will be sorted in increasing order because the algorithm on each call it selects d_{min} as the earliest deadline, so the next earliest deadline in the recursive call has to be equal or greater than d_{min}

   $$S' = [7,8,10,11,15] \text{ -> Different optimal solution}$$

We assume that S' will be sorted in increasing order
let $s = S[1]$
let $s' = S'[1]$
By switching s' with s, S' will still give us a valid solution, because we know there is no other element in P with an earlier deadline than s. So every element in P that has received a credit by having a date on s' will also receive a credit by having a date on s, because s' must be less than or equal to s.

If s' was greater than s then S' won't be a valid solution, because we know that there exists a "special date" in P that doesn't receive a credit by having a date on s', because s is the earliest deadline of all the dates in P.

We can repeat the swap on the elements $S[i]$ and $S'[i]$ at any index i and the solution S' will still be a valid solution and since we know S' is an optimal solution, then S has to be optimal.

4. What data structures could you use to ensure a running time of $O(n \log n)$? (Don't forget to justify your answer.)

   To ensure a running time of $O(n \log n)$ I would use a priority queue to store P. This will allow us to access the min element, i.e. the earliest deadline d_{min} in constant time, and to delete an element from it in $\log n$.

   So for the case in which $k = n$ and all elements have the same deadline, then the while will take $O(n)$ time by going through all elements, $O(1)$ to find the min deadline in P and $O(\log n)$ deleting at each iteration one, which will have a total running time of $O(n \log n)$.

5. A different greedy algorithm schedules a dinner on the latest start date and works backwards. Give pseudocode that does this. (You do not need to provide any justification or reasoning about your algorithm.)

Algorithm Different-Dinner-Dates(P, k)
  If $n = 0$
    Return the empty list
  Else
    Set $s_{\max}$ to be the latest start date of any pair in P
    $i = 0$
    While $i < k$ and there is some pair (s', d') in P with $d' \geq s_{\max}$
      Choose such a pair (s', d') with the latest start date (i.e., greatest s')
      Remove pair (s', d') from P
      Increment i
    Return $\{s_{\max}\}$ + Different-Dinner-Dates(P, k)

2 More on another spanning algorithm

This problem builds on your tutorial problem for week 6. Let $G = (V, E)$ denote a connected, undirected graph with $n \geq 2$ nodes and m weighted edges. Let $\mathrm{wt}(e)$ denote the weight of edge e of G. The following algorithm is similar but not identical to Kruskal's minimum spanning tree algorithm. (This version of the algorithm is interesting because it can be implemented efficiently on a multi-processor computer. Roughly this is because the steps for each connected component C can all be handled by different processors.)

Algorithm Spanning($G = (V, E)$, wt())

Let $G' = (V, E')$ where $E' = \emptyset$
While G' is not connected
    E-new $= \emptyset$
    For each connected component C of $G' = (V, E')$
        Find an edge $e = (u, v) \in E$ of minimum weight wt(e) that
        connects a node u in C to a node v that is *not* in C
        E-new $= E$-new $\cup \{e\}$
    $E' = E' \cup E$-new
Return G'

1. Explain why the algorithm always returns a tree on all inputs $G = (V, E)$ where all edges of E have different weights.

   The problem with this algorithm that causes a solution that is not a tree, occurs when two connected components, at the one iteration of the while loop, select different edges that connect them together. In this case a cycle is introduced to the solution because it creates two possible paths from one connected component to another one.

   If all edges of E have different weights then we know that for each pair of connected connected components there exists a single edge that has the lowest weight. Thus there is no way the algorithm chooses another edge rather than that one to connect them, both connected components must choose the same edge. So we can assume that when we connect two connected components there wont be a cycle introduced in the solution.

   This process is repeated for all the algorithm, guaranteeing no cycles if each edge in E has a different weight, which will give us a tree as a solution.

2. Explain why the tree returned by the algorithm is a minimum spanning tree on all inputs $G = (V, E)$ where all edges of E have different weights.

   A tree returned by the algorithm will always have all the minimum weighted edges because at each iteration when we add a new edge that will always be the minimum weighted edge that connects one connected component to another one. This will be repeated on all connected components that we are connecting on each iteration, ensuring that G' always have the minimum weighted edges.

   Thus our solution will be a tree (proved on previous question), which means we have the minimum number of edges that connect every single node in G, furthermore every edge in the tree will have the minimum weight on it, this characteristics are the ones that make the solution tree of this algorithm a minimum spanning tree.

3 Runs of zeros

A *run of 0's* in a binary string s of length at least 1 is a substring s' of s consisting only of 0's, such that each end of the substring is either adjacent to a 1 or is also the end of the whole string s. For example, the string 101100 has two runs of 0's, one of length 1 and one of length 2 and the string 000 has one run of 0's of length 3, namely the whole string itself.

Let $R(n)$ be the total number of runs of 0's, taken over all binary strings of size $n \geq 1$.

1. Give values for $R(1)$ and $R(2)$:

$R(1)$: $\boxed{1}$ $R(2)$: $\boxed{3}$

2. Provide a recurrence relation for $R(n)$, $n \geq 1$.

$$R(n) = \begin{Bmatrix} 1, \text{ when } n = 1 \\ 2R(n-1) + 2^{n-2}, \text{ when } n > 1 \end{Bmatrix}$$

3. Justify why your recurrence is correct.

   Base case:
   $R(n = 1) = 1$

   If $n = 2$
   $R(2) = 2R(1) + 2^{2-2} = 2(1) + 1 = 3$
   $R(n)$ provides a valid solution for n = 2

   Assuming $R(n)$ provides a valid solution for $n = k$, for a $n = k + 1$ we have
   $R(n = k + 1) = 2R(k) + 2^{k+1-2} = 2R(k) + 2^{k-1}$

   If we choose $k + 1 = 3$
   $R(n = 3) = 2R(2) + 2^{2-1} = 2(3) + 2 = 8$ Which is true

   If we choose $k + 1 = 4$
   $R(n = 4) = 2R(3) + 2^{3-1} = 2(8) + 4 = 20$ Which is also true

   So for every $k >= 1$ element this formula will hold.

4. Solve your recurrence, to express $R(n)$ as a function of n.

   $R(n) = 2R(n-1) + \lceil 2^{n-2} \rceil$

4 Nuts and bolts

This problem builds on your tutorial problem for week 7. You have to sort a bag of n nuts and n bolts by size, producing an output of n (nut, bolt) pairs that fit together. In part because the sizes are similar, and in part because you also want to watch videos while sorting, you are not relying on eyesight as you do this. So, the only way that you can tell if a particular bolt fits a particular nut is by trying to thread the bolt into the nut. You realize that you might be able to accomplish the task efficiently by using nuts and bolts that you match as a way to filter the rest. The following algorithm captures this idea.

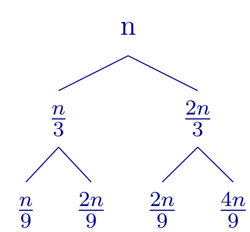
Algorithm NB-Quick(Nut-Set, Bolt-Set)
  If Nut-Set is empty, then
    Return the empty set
  Else If Nut-Set contains exactly one nut, say N, then
    Let B be the single bolt in Bolt-Set
    Return $\{(N, B)\}$
  Else
    Remove a nut, say N, from Nut-Set

Partner-found = False
Tried-Bolts = $\emptyset$
While not Partner-found
    Remove any bolt, say B, from Bolt-Set
    If bolt B threads into nut N then
        Partner-found = True
    Else
        Add B to Tried-Bolts
For each nut in Nut-Set
    If the nut is too loose for B
        Add it to the set Loose-Nuts
    Else add it to the set Tight-Nuts
For each bolt in Bolt-Set $\cup$ Tried-Bolts
    If the bolt is too large for N
        Add it to the set Large-Bolts
    Else add it to the set Small-Bolts
Return $\{(N, B)\} \cup$ NB-Quick(Loose-Nuts, Large-Bolts)
    $\cup$ NB-Quick(Tight-Nuts, Small-Bolts)

1. Consider the case where, at every recursive call, both of the sets Tight-Nuts and Loose-Nuts have size in the range $[n/k, (k-1)n/k]$, for some integer $k > 2$. Write a recurrence relation for the running time of this algorithm.

$$T(n) \le \begin{cases} c, & \text{when } n = 0 \text{ or } n = 1 \quad // \text{ base cases} \\ \boxed{\text{cn} + \text{T(n/3)} + \text{T(2n/3)}} & //\text{recursive case} \end{cases}$$

2. Solve your recurrence to get a good asymptotic (big-O) upper-bound on the running time of this algorithm, as a function of both n and k.



By the recursion tree above we know that the shallowest leaf of the tree is at level $\log_3 n$ on the other hand the deepest leaf of the tree will be at level $\log_{3/2} n$.

So we know that at most the algorithm will have to execute at most $\log_{3/2}$ times (to reach the deepest leaf), the running time of the base case is constant time and the running time done before calling the recursion function takes $O(n)$. n will vary on each recursive call but it will be either $(\frac{n}{3})^i$ or $(\frac{2n}{3})^i$ at the level i, so if we add the running time of all the recursive calls at depth i it will be $O(n)$.

Thus the algorithm has to do $O(n)$ operations at most $O(\log n)$ times, so the asymptotic upper-bound of all the algorithm is $O(n \log n)$.