CPSC 320 2017W1: Assignment 3

October 18, 2017

Please submit this assignment via GradeScope at `https://gradescope.com`. Be sure to identify everyone in your group if you're making a group submission (which we encourage!).

Submit by the deadline **Friday 27 Oct at 10PM**. For credit, your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**. Your group's submission **must**:

- Be on time.
- Consist of a single, clearly legible file uploadable to GradeScope with clearly indicated solutions to the problems. (PDFs produced via LaTeX, Word, Google Docs, or other editing software work well. Scanned documents will likely work well. **High-quality** photographs are OK if we agree they're legible.)
- Include prominent numbering that corresponds to the numbering used in this assignment handout (not the individual quizzes). Put these **in order** starting each problem on a new page, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the **ugrad.cs.ubc.ca e-mail addresses** of each member of your team. (No names are necessary.)
- Include at the start of the document the statement: "All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and GradeScope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff." (Go read those guidelines!)
- Include at the start of the document your outside-group collaborators' ugrad.cs.ubc.ca e-mail addresses. (Be sure to get those when you collaborate!)

1 eXtreme True And/Or False

Pre-Reading: The Very Busy 3-D Printer

The CS department buys a single 3-D printer and wants to develop a scheduling algorithm for it. We'll call their problem the Very Busy 3-D Printer Problem or VB3.

The input for a particular week is a list of n jobs for the printer. Each job is a pair of a positive integer duration t of the job and non-negative integer deadline d for the job. Once started, a job must be run to completion, which takes t minutes. To be useful, the job must be complete by the deadline d (also in minutes from the start of the week).

The output is a schedule: a list of $k \leq n$ of the jobs along with a non-negative integer start time s for each one, sorted in increasing order of start time. A valid schedule must ensure that no two jobs overlap (i.e., no jobs i and j exist in the schedule such that $s_i \leq s_j$ and $s_i + t_i > s_j$) and all jobs finish on time ($s_i + t_i \leq d_i$). Note that it is OK for a job's end time to match the next job's start time. The goal is to schedule as many of the jobs as possible.

For example, the input $(2, 5), (1, 3), (3, 4), (5, 15)$ represents four jobs. The first is of length 2 minutes with a deadline 5 minutes from the "zero" time. The second job is only 1 minute long and has a deadline 3 minutes after the zero time. Etc.

The optimal solution to this instance includes three of the four jobs. One option—represented with tuples of (s, t, d) start time, duration, and deadline—would be $(0, 2, 5), (2, 1, 3), (8, 5, 15)$. That runs the 2 minute job right away, followed by the one minute job. Then, at minute 8, it runs the 5 minute job. This would also be an optimal schedule with different jobs and timing: $(0, 1, 3), (1, 3, 4), (4, 5, 15)$.

FILL IN YOUR UGRAD ID and read the problem statement on the previous page (relevant for all but the first part). Then, follow the directions on this page.

Each of the following problems presents a scenario and a statement about that scenario. For each one, indicate by filling in the appropriate circle whether:

- The statement is **ALWAYS** true, i.e., true in *every* instance matching the scenario.
- The statement is **SOMETIMES** true, i.e., true in some instance matching the scenario but also false in some such instance.
- The statement is **NEVER** true, i.e., true in *none* of the instances matching the scenario.

Problems:

1. **Scenario:** A simple graph with $n \geq 3$. **Statement:** Any simple path that starts at an arbitrary vertex v and ends at an arbitrary vertex $u \neq v$ can be extended into a simple cycle by adding vertices after u.

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

2. **Scenario:** An instance of VB3 with $n \geq 2$. **Statement:** An optimal schedule $((s_1, t_1, d_1), \ldots)$ exists in which $s_1 = 0$ and for each subsequent pair of jobs in the schedule i and $i+1$, $s_i + t_i = s_{i+1}$.

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

3. **Scenario:** An instance of VB3 with $n \geq 1$. **Statement:** The job i with the maximum value of $d_i - t_i$ is in some optimal solution to the instance.

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

4. **Scenario:** An instance of VB3 with $n \geq 1$. **Statement:** This greedy algorithm produces an optimal solution to the instance: Begin with an empty schedule and time marker $m = 0$. In increasing order of job duration: if the next job i can be finished in time ($m + t_i \leq d_i$), add it to the schedule with start time m and then increase m by t_i.

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

5. **Scenario:** An instance of VB3 with $n \geq 1$ and with an optimal solution containing at least one job. **Statement:** This greedy algorithm produces an optimal solution to the instance: Begin with an empty schedule and time marker $m = 0$. In increasing order of job deadline: if the next job i can be finished in time ($m + t_i \leq d_i$), add it to the schedule with start time m and then increase m by t_i.

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

1.1 Quiz Solution

1. **SOMETIMES**
2. **ALWAYS**
3. **ALWAYS**
4. **SOMETIMES**
5. **SOMETIMES**

1.2 Assignment

On the assignment, for each of the problems above, please **briefly** justify the answer.

Justify an **ALWAYS** answer by giving a small instance that fits the scenario for which the statement is true and then briefly sketching the key points in a proof that the statement is true for all instances that fit the scenario.

Justify a **NEVER** answer by giving a small instance that fits the scenario for which the statement is **false** and then briefly sketching the key points in a proof that the statement is **false** for all instances that fit the scenario.

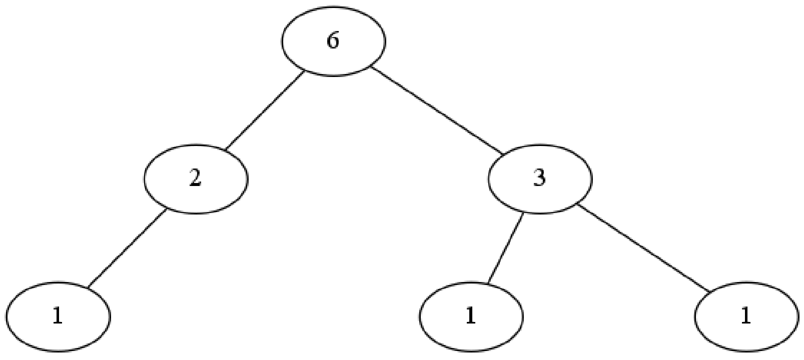
Justify a **SOMETIMES** answer by giving **two** small instances that fit the scenario: one for which the statement is true and one for which the statement is false.

IMPORTANT CAUTION: None of your example instances should critically rely on tie-breaking behaviour to illustrate what they are meant to show. **Ensure** that you briefly explain why each instance fits the scenario and makes the statement true or false, as needed.

2 Bestride the BST

We define a size-weighted BST (wBST) to be a binary search tree in which each node also has a weight: a count of the number of nodes in the subtree rooted at that node. A leaf (a single node with empty subtrees) has weight 1, a node with only one subtree which is itself a leaf has weight 2, a node with two leaves as subtrees has weight 3, etc.

The following binary tree is labelled with its weights (but omits the keys and values that would be in each node of a full wBST):



FILL IN YOUR UGRAD ID and read the definition on the previous page. Then, consider the following pseudocode for correcting a wBST's weights:

```
// A node has four fields: key, value, weight, and subtrees left and right.
// The only field of the special value EMPTY representing empty trees is EMPTY.weight = 0.
// (So, for a wBST b, b.weight will not produce an error, even if b is an empty tree.)
function WeightBST(root):
    if root != EMPTY:
        WeightBST(root.left)
        WeightBST(root.right)
        root.weight = root.left.weight + root.right.weight
```

1. There is a single, small error in the code above; fix it.

2. After you have corrected it, a proof of correctness of this algorithm would proceed by induction. This problem and the next two ask about such a proof.

   Which of these would be the type of tree to use for the base case in the proof? Fill in the circle next to the **best** answer.

   ◯ an incorrectly weighted tree
   ◯ a correctly weighted tree
   ◯ an empty tree
   ◯ a single leaf node
   ◯ a tree one node smaller than the current one
   ◯ the two subtrees of the current tree

3. Your induction hypothesis will assume a property applies to a particular type of tree. Which of these best describes the *type of tree* on which you make the induction hypothesis? Fill in the circle next to the **best** answer.

   ◯ an incorrectly weighted tree
   ◯ a correctly weighted tree
   ◯ an empty tree
   ◯ a single leaf node
   ◯ a tree one node smaller than the current one
   ◯ the two subtrees of the current tree

4. Your induction hypothesis will assume a property applies to a particular type of tree. Which of these best describes the *property* you will assume true for the type of tree you chose in the previous part? Fill in the circle next to the **best** answer.

   ◯ it is weighted correctly
   ◯ it is weighted incorrectly
   ◯ it is an empty tree
   ◯ it has a single leaf node
   ◯ it has one fewer nodes than the current tree
   ◯ it is a subtree of the current tree

5. Fill in the circle next to the best asymptotic bound on the runtime of this algorithm in terms of the number of nodes in the tree n.
   ◯ $O(1)$
   ◯ $O(\lg n)$
   ◯ $O(n)$
   ◯ $O(n \lg n)$
   ◯ $O(n^2)$

6. Fill in the blanks to correctly and efficiently complete the following code to count the number of keys in a (correctly weighted) wBST `root` that are strictly larger than the target value `lo`. (As is usual, no duplicate keys appear in the tree.)

```
procedure CountLarger(root, lo)
    if root is EMPTY then
        return 0
    else if root.key < lo then
        return [                    ]
    else if root.key = lo then
        return [                    ]
    else
        return [                    ]
    end if
end procedure
```

2.1 Quiz Solution

1. Change the root weight calculation to `root.weight = root.left.weight + root.right.weight + 1`.

2. an empty tree

3. the two subtrees of the current tree

4. it is weighted correctly

5. $O(n)$

6. No solution provided at this time.

2.2 Assignment

Note: For the proofs in this section, we will grade based on:

- A clear, correct, easily readable proof structure, which allows us to put our attention on individual parts of the proof. The structure must reflect both an understanding of induction and of the key features of the algorithms, recurrences, and properties being proven.

- Clear, concise, correct, and easily readable arguments in each section of the proof.

Now, solve the following:

1. Prove the correctness of the algorithm after the correction from the quiz solution.

   Specifically, prove that after the call to `WeightBST` on `root` completes, for each node and empty tree `v` in the subtree rooted at `root` the number of nodes in the subtree rooted at `v` is exactly `v.weight`.

2. Create a recurrence relation $T_W(n)$ describing the time taken by the `WeightBST` algorithm called on a wBST with n nodes. (You don't know the structure of the wBST; so, assume for the recursive case that the left subtree has k nodes.)

3. Prove inductively that $T_W(n) \leq cn + d$ for appropriate constants c and d.

4. Complete the `CountLarger` function so that it runs in $O(h)$ time on a tree with height h.

5. Prove the correctness of `CountLarger` by induction.

6. Write pseudocode for a short, clear, and efficient function `CountRange(root, lo, hi)` that counts the number of nodes in the correctly-weighted wBST `root` with keys k such that $lo < k \leq hi$. Make use of `WeightBST` or `CountLarger` as needed.

3 Preparing for Sasquatch (Part 1)

Every year, on Memorial Day weekend, the Gorge Amphitheater in Washington hosts the Sasquatch! Music Festival. Tickets are expensive, so if you go it's imperative to maximize your musical pleasure by attending as many performances as you can. Luckily, you're enrolled in cpsc320, which makes you an expert in festival planning!

A *performance* is represented by a pair (s, f) where s is its start time and f is its finish time (relative to the start of the festival). There are n performances over the three days, hosted across many stages. Your goal is to maximize the number of non-overlapping performances in your festival itinerary.

In each of the following greedy algorithms, answer "Yes" if the algorithm *always* constructs an optimal schedule, and answer "No" otherwise.

	Yes	No	Algorithm:
1	○	○	Choose the performance p that ends last, discard all performances that conflict with p, and recurse on the remaining performances.
2	○	○	Choose the performance p that starts last, discard all performances that conflict with p, and recurse on the remaining performances.
3	○	○	Choose the performance p with shortest duration $(f - s)$, discard all performances that conflict with p, and recurse on the remaining performances.
4	○	○	Choose a performance p that conflicts with the fewest other performances, discard all performances that conflict with p, and recurse on the remaining performances.
5	○	○	If no performances conflict, choose them all. Otherwise, discard a performance that conflicts with the most other performances and recurse on the remaining performances.
6	○	○	If any performance p completely contains another performance, discard p and recurse. Otherwise, choose the performance q that ends first, discard all performances that canflict with q, and recurse on the remaining performances.

3.1 Quiz Solution

1. NO

2. YES

3. NO

4. NO
5. NO
6. YES

3.2 Assignment

1. Prove that each of the answers above is correct.

   For a "No" answer: give a small instance (counterexample to the algorithm's correctness) where the greedy algorithm described fails to achieve an optimal solution. Be sure to briefly explain the optimal solution and the lower-valued solution found by the greedy algorithm. Your counterexample should **not** critically rely on tie-breaking behaviour of the algorithm to illustrate what it is meant to show.

   For a "Yes", prove your result. (Neither proof requires an exchange argument if you can think of a useful reduction.)

   Reminder, you are giving in order:

   (a) Counterexample
   (b) Proof
   (c) Counterexample
   (d) Counterexample
   (e) Counterexample
   (f) Proof

2. Suppose the organizers of the festival would like to schedule a set of times where things like raffle winners, camping rules, and merchandise tent hours could be announced to all the festival attendees. To do this, a broadcast is made simultaneously to every performance occurring at the time of the announcement. Describe and analyze a greedy algorithm to find the smallest set of broadcast times required to assure that every performance has at least one announcement.

   (a) Give 3 different trivial instances.
   (b) Give all of the meaningfully distinct instances for $n = 2$ performances.
   (c) Write pseudocode for a greedy algorithm to solve this problem. (As a simplifying hint, it's ok for announcements to be made exactly when a performance begins or ends.)
   (d) Prove that your greedy algorithm finds an optimal solution.

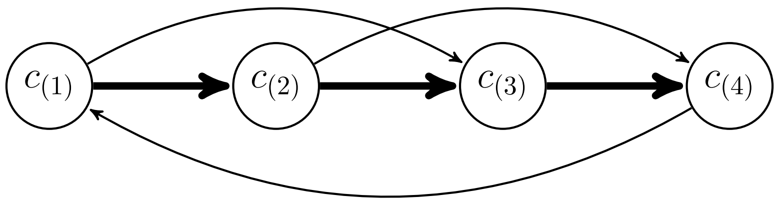
(Thanks to `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/07-greedy.pdf` for the great problems.)

4 Preparing for Sasquatch (Part 2)

Suppose there are n candidates running for a public office, and that voting has provided a tally that indicates, between any pair of candidates, which one is preferred. This is very naturally modeled as a directed graph with n vertices, and exactly one directed edge between every pair of vertices: $G = (V, E)$, with $V = c_1, \ldots, c_n$ and $\forall u, v \in V$ either $(u, v) \in E$, or $(v, u) \in E$, but not both. Such a graph is called a *tournament*.

A *ranking* is a permutation of candidates $c_{(1)} \ldots c_{(n)}$ so that edge $(c_{(i)}, c_{(i+1)}) \in E$ for all $1 \leq i < n$. In this problem we will show that a ranking always exists, *even if there are cycles in the original graph.* (An alternative way of representing the solution is a simple directed path through all the vertices.)

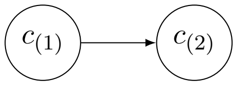
Here's an example of a tournament of size $n = 4$ with a ranking in bold:



The actual names of the candidates (or the labels on the vertices), do not matter in this problem.

A proof that a ranking is always possible in a tournament would proceed by induction. These questions are designed to help you gather your thoughts for completing that proof.

1. (We're doing this one for you!) There is only one instance for $n = 2$, drawn here:



2. Draw *all* instances for $n = 3$:

3. In each of the size 3 instances in the previous part, clearly indicate a ranking by drawing a dotted line along the directed path through all the vertices.

4. Consider an arbitrary tournament of size n. An appropriate inductive hypothesis would apply to ____________.
   - ○ a tournament of size $\frac{n}{2}$
   - ○ a directed, acyclic graph of size n
   - ○ a tournament of size $n - 1$
   - ○ a DFS tree of $n - 1$ edges
   - ○ a directed acyclic graph of size $\frac{n}{2}$
   - ○ a partition of the tournament into 3 subgraphs

5. The inductive hypothesis would conclude by saying the structure you specified in the previous part has a ____________.
   - ○ tournament
   - ○ ranking
   - ○ champion
   - ○ tree of shortest paths
   - ○ vertex whose out degree is 0
   - ○ directed acyclic graph of size $\frac{n}{2}$

6. Suppose you have a tournament of size m and you add a new vertex v, together with edges from v *to* each of the vertices in the tournament. Can you add the new vertex to the ranking? If so, where? If not, why not? (Choose the best answer to these questions.)

   ◯ Yes, anywhere!
   ◯ Yes, in position 1.
   ◯ Yes, in position $m + 1$.
   ◯ Maybe, depending on the structure of the existing tournament.
   ◯ No, because the new vertex must have both an incoming and an outgoing edge.

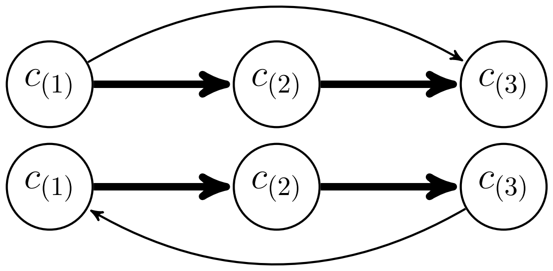
7. Suppose you have a tournament of size m and you add a new vertex v, together with edges *from* each of the vertices in the tournament to v. Can you add the new vertex to the ranking? If so, where? If not, why not? (Choose the best answer to these questions.)

   ◯ Yes, anywhere!
   ◯ Yes, in position 1.
   ◯ Yes, in position $m + 1$.
   ◯ Maybe, depending on the structure of the existing tournament.
   ◯ No, because the new vertex must have both an incoming and an outgoing edge.

4.1 Quiz Solution

1. Given above.

2. Here are the only two meaningfully different instances. Any other instance is just a renaming of these (and remember that the vertex labels don't matter).



$$\begin{tikzcd} c_{(1)} \arrow[r] \arrow[rr, bend left] & c_{(2)} \arrow[r] & c_{(3)} \end{tikzcd}$$

$$\begin{tikzcd} c_{(1)} \arrow[r] & c_{(2)} \arrow[r] & c_{(3)} \arrow[ll, bend left] \end{tikzcd}$$

3. See above.

4. a tournament of size $n - 1$

5. ranking

6. yes in position 1

7. yes in position $m + 1$

4.2 Assignment

1. To further prepare for your inductive argument, make a sketch representing the ranking for a tournament of size $n-1$, and consider an n^{th} vertex. Under what condition can it be inserted into position 1? Under what condition can it be inserted into the middle of the existing chain? Under what condition can it be inserted into position n? Must one of those conditions exist?

2. Assemble the answers from the quiz and the exploration in the previous part into a complete proof that, for any n, a tournament of size n always has a ranking.
3. Use the insights from your proof to design (and give in pseudocode!) an algorithm for finding a ranking, given a tournament.
4. Give and briefly justify a good big-O bound on the running time of the algorithm in the previous part in terms of the number of vertices n in the graph.