CSCI 3104 Fall 2022
Instructors: Prof. Grochow and Chandra Kanth Nagesh

Problem Set 11 SOLUTIONS

Due Date ..... **Tuesday** November 29, 2022
Name ..... **Your Name**
Student ID ..... **Your Student ID**
Collaborators ..... **List Your Collaborators Here**

Contents

Instructions **1**

Honor Code (Make Sure to Virtually Sign the Honor Pledge) **2**

25 Standard 25: Design a Dynamic Programming Algorithm (Synthesis Standard) **3**
25(a) 3
25(b) 4
25(c) 5
25(d) 6

27 Standard 27: Amortized Analysis & Doubling Lists **7**
27(a) 7
27(b) 8

Instructions

- The solutions **must be typed**, using proper mathematical notation. We cannot accept hand-written solutions. Useful links and references on LaTeXcan be found here on Canvas.
- You should submit your work through the **class Canvas page** only. Please submit one PDF file, compiled using this LaTeX template.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please submit this document with no fewer pages than the blank template (or Gradescope has issues with it).
- You are welcome and encouraged to collaborate with your classmates, as well as consult outside resources. You must **cite your sources in this document. Copying from any source is an Honor Code violation. Furthermore, all submissions must be in your own words and reflect your understanding of the material.** If there is any confusion about this policy, it is your responsibility to clarify before the due date.

- Posting to **any** service including, but not limited to Chegg, Reddit, StackExchange, etc., for help on an assignment is a violation of the Honor Code.
- You **must** virtually sign the Honor Code (see Section Honor Code). Failure to do so will result in your assignment not being graded.

Honor Code (Make Sure to Virtually Sign the Honor Pledge)

Problem HC. On my honor, my submission reflects the following:

- My submission is in my own words and reflects my understanding of the material.
- Any collaborations and external sources have been clearly cited in this document.
- I have not posted to external services including, but not limited to Chegg, Reddit, StackExchange, etc.
- I have neither copied nor provided others solutions they can copy.

In the specified region below, clearly indicate that you have upheld the Honor Code. Then type your name.

Honor Code. □

25 Standard 25: Design a Dynamic Programming Algorithm (Synthesis Standard)

Problem 25. The Placing Electrons on a Tree problem is defined as follows.

- Instance: A tree T, with root $r \in V(T)$.
- Solution: A subset $S \subseteq V(T)$ such that there are no edges (s, s') for $s, s' \in S$, and such that $|S|$ is as large as possible.

The *idea* is that you are placing electrons on the vertices of a tree, but electrons repel each other, so you cannot place two electrons adjacent to one another, with the goal being to place as many electrons as possible.

The goal of this problem set is to design a dynamic programming algorithm to solve the Placing Electrons on a Tree problem.

Problem 25(a)

(a) For each vertex $v \in V(T)$, let $L[v]$ denote the maximum number of electrons that can be placed on the subtree rooted at v. Write down a mathematical recurrence for $L[v]$. Clearly justify each case. *Hint:* your recurrence should involve the children and grandchildren of v. You may use notation such as "children(v)" to denote the set of children of v, and "grandchildren(v)" to denote the set of grandchildren of v. Remember to have base case(s).

Answer. If v is a leaf, then the subtree rooted at v is just the node v itself, for which the optimum solution is clearly $S = \{v\}$ so we have $L[v] = 1$ for any leaf v.

If v is not a leaf, then either v is in the optimal set, or not. If v is included, then its children cannot be, so they are skipped, but its grandchildren may. Thus if v is included, the best one can do for the subtree rooted at v consists of including v, together with whatever the optimal solutions are for the subtrees rooted at v's grandchildren. Since those subtrees are disjoint, we simply take the sum of their optimum values. On the other hand, if v is not included, then the best choice will be whatever the optimal solutions are for the subtrees rooted at v's children. Thus for non-leaves v we have the recurrence:

$$L[v] = \max \begin{cases} 1 + \sum_{u \in \text{grandchildren}(v)} L[u] \\ \sum_{u \in \text{children}(v)} L[u] \end{cases}$$

Putting these together, we get

$$L[v] = \begin{cases} 1 & \text{if } v \text{ is a leaf} \\ \max\{1 + \sum_{u \in \text{grandchildren}(v)} L[u], \sum_{u \in \text{children}(v)} L[u]\} & \text{otherwise.} \end{cases}$$

□

Problem 25(b)

(b) Clearly describe how to construct and fill in the lookup table. For the cell $L[v]$, clearly describe the sub-cases we consider, and which optimal sub-case we select. *Hint:* Similar to counting paths in a DAG, what order should you put the vertices in so that you can fill in the lookup table "in order"?

Answer. To fill in $L[v]$, we must first know $L[u]$ for every u that is either a child or grandchild of v. The subcases we consider are including v together with all of the optimal solutions of its grandchildren, or excluding v together with all the optimal solutions of its children. We thus start by filling in the table for the leaves, and then work our way up the tree. (Technically, one may define the *height* of a vertex in the tree as $h(v) = 1$ if v is a leaf, and $h(v) = 1 + \max\{h(u) : u \in \text{children}(v)\}$. Then we fill in the table in order of height.)

□

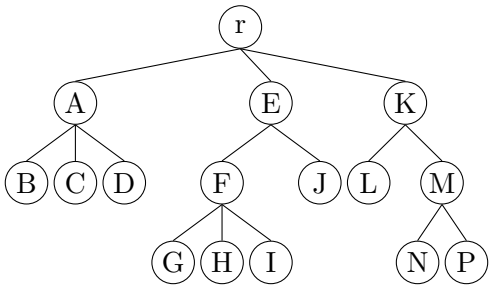
Problem 25(c)

(c) Let `back`$[v]$ denote another array storing "backpointer" information to enable you to reconstruct the optimal solution. Clearly describe what information should be stored in `back`[v] and how it is generated based on the optimal sub-cases selected in the computation of $L[v]$ (as you described in Problem2(b) above).

Answer. Since the computation of $L[v]$ considered only two cases, we just need a single bit to keep track of which case was chosen. We will use `back`$[v] = True$ to indicate that v was included in the optimal solution for the subtree rooted at v (that is, the optimum was to include v plus whatever the optimal solutions of its grandchildren were), and `back`$[v] = False$ to indicate that v was excluded in the optimal solution for the subtree rooted at v (that is, the optimum was to exclude v, plus whatever the optimal solutions of its children were). □

Problem 25(d)

(d) Work through an example of your algorithm using the following tree. Clearly show how to recover an optimal solution using your lookup table. You may hand-draw your lookup table, but your explanation must be typed.



Answer. Base cases:

- For all the leaves $v \in \{B, C, D, G, H, I, J, L, N, P\}$, we have $L[v] = 1$ and $\texttt{back}[v] = True$.
- For F, we have $L[F] = \max\{1, L[G] + L[H] + L[I]\} = \max\{1, 3\} = 3$, since F has no grandchildren. Since the optimal selection was when F was not chosen, we have $\texttt{back}[F] = False$.
- For A, we have $L[A] = \max\{1, L[B] + L[C] + L[D]\} = \max\{1, 3\} = 3$, since A has no grandchildren. Since the optimal selection was when A was not chosen, we have $\texttt{back}[A] = False$.
- For M, we have $L[M] = \max\{1, L[N] + L[P]\} = \max\{1, 2\} = 2$, since M has no grandchildren. Since the optimal selection was when M was not chosen, we have $\texttt{back}[M] = False$.
- For E, we have $L[E] = \max\{1 + L[G] + L[H] + L[I], L[F] + L[J]\} = \max\{4, 4\} = 4$, so either choice may be chosen. We arbitrarily choose to set $\texttt{back}[E] = False$.
- For K, we have $L[K] = \max\{1 + L[N] + L[P], L[L] + L[M]\} = \max\{3, 3\} = 3$, so either choice may be chosen. We arbitrarily choose to set $\texttt{back}[K] = False$.
- Finally, for the root r, we have $L[r] = \max\{1 + \sum_{v \in \{B,C,D,F,J,L,M\}} L[v], L[A] + L[E] + L[K]\} = \max\{1 + 1 + 1 + 1 + 3 + 1 + 1 + 2, 3 + 4 + 3\} = \max\{11, 10\} = 11$. Since the choice chosen was when r is included, we set $\texttt{back}[r] = True$.

The resulting table would look llke:

v	B	C	D	G	H	I	J	L	N	P	F	A	M	E	K	r
L[v]	1	1	1	1	1	1	1	1	1	1	3	3	2	4	3	11
back[v]	T	T	T	T	T	T	T	T	T	T	F	F	F	T/F	T/F	T

To reconstruct the optimal solution, we trace back through the back-pointers.

- Since $\texttt{back}[r] = True$, we put r into our optimal solution set S. We then recurse on the *grand*children of r, namely B, C, D, F, J, L, M.
- Among those grandchildren, B, C, D, J, L are leaves, so they all have $\texttt{back}= True$, and they all get included in S.
- This leaves only F and M. We have $\texttt{back}[F] = False$ and $\texttt{back}[M] = False$, so we leave F and M out of S, and then recurse on their children G, H, I and N, P.
- Since G, H, I and N, P are all leaves, they all have $\texttt{back}= True$, and thus get included in S.

Thus, our optimal solution S is $\{r, B, C, D, J, L, G, H, I, N, P\}$ (double-check: this indeed has size 11, which is the value we computed at $L[r]$).

□

27 Standard 27: Amortized Analysis & Doubling Lists

Problem 27. *Note: this is your second attempt on this standard—the first was on Midterm 2. You will also see it on a quiz and on the final. Because we pushed back the schedule on S27, despite having four chances at it, you only have to demonstrate proficiency once to get Standard 27 to count.*

A dynamic array-list is a data structure that can support an arbitrary number of append (add to the end) operations by allocating additional memory when the array becomes full. The standard process covered in class is to double (adds n more space) the size of the array each time it becomes full. You cannot assume that this additional space is available in the same block of memory as the original array, so the dynamic array must be copied into a new array of larger size. Here we consider what happens when we modify this process. The operations that the dynamic array supports are

- `Indexing A[i]`: returns the `i`-th element in the array
- `Append(A,x)`: appends `x` to the end of the array. If the array had `n` elements in it (and we are using 0-based indexing), then after `Append(A, x)`, we have that `A[n]` is `x`.

27(a)

a. Consider a dynamic array-list that adds 7 to its length (that is, increases length from n to $n+7$) each time it's full. Derive the amortized runtime of Append for this data structure. Clearly explain the reasoning behind your calculations.

Answer. Suppose N append operations are performed. Many of these have cost $\Theta(1)$, but every 7th one costs $\Theta(n)$ (where n is the size of the list at the time of the operation). Suppose N is one of these latter operations. Then that operation, together with the previous 7, cost $\Theta(N) + 7\Theta(1)$. The total cost of the N operations is thus

$$T(N) = \Theta(N) + 7 \cdot \Theta(1) + T(N-7) = T(N-7) + \Theta(N).$$

Solving this by unrolling gives us $T(N) = \Theta(N^2)$ (quick summary: it takes $N/7$ steps to get to the base case, we end up with a sum that looks like $\sum_{i=0}^{N/7} ci$, which adds up to $\Theta(N^2)$). Thus the amortized cost of each append operation is $\Theta(N^2)/N = \Theta(N)$ (which is pretty bad!). □

27(b)

b. Derive the amortized runtime of Append for a dynamic array that squares its length (that is, increases length from n to n^2) each time it's full. Derive the amortized runtime of Append for this data structure. Clearly explain the reasoning behind your calculations.

Answer. Suppose N append operations are performed. Many of these have cost $\Theta(1)$, but every so often one costs $\Theta(n)$ (where n is the size of the list at the time of the operation). Suppose N is one of these latter operations. Then that operation, together with the previous $N - \sqrt{N}$, cost $\Theta(N) + (N - \sqrt{N})\Theta(1)$, since the last time the list was increased in size was when it had size $\sqrt{N}$. The total cost of the N operations is thus

$$T(N) = \Theta(N) + (N - \sqrt{N}) \cdot \Theta(1) + T(\sqrt{N}) = T(\sqrt{N}) + \Theta(N).$$

We solve by unrolling:

$$\begin{aligned}
T(N) &= T(\sqrt{N}) + cN \\
&= (T(N^{1/4}) + c(\sqrt{N})) + cN && \text{(unroll)} \\
&= T(N^{1/4}) + c(N + \sqrt{N}) && \text{(simplify)} \\
&= T(N^{1/8}) + cN^{1/4} + c(N + \sqrt{N}) && \text{(unroll)} \\
&= T(N^{1/2^k}) + c(N + N^{1/2} + N^{1/4} + \cdots + N^{1/2^{k-1}}).
\end{aligned}$$

Solve for the base case:

$$\begin{aligned}
N^{1/2^k} &\le c' \\
(1/2^k) \log_2 N &\le \log_2 c' && \text{(take logs)} \\
\log_2 N &\le 2^k \log_2 c' \\
\log_2 \log_2 N &\le k + \log_2 \log_2 c' \text{(take logs)}
\end{aligned}$$

so we hit the base case when $k = \log \log N$, and then the result is

$$T(N) = \Theta(1) + \sum_{i=0}^{\log \log N} N^{1/(2^i)}.$$

Separating out the first term of this sum we get

$$T(N) = \Theta(1) + N + \sum_{i=1}^{\log \log N} N^{1/(2^i)}.$$

Now, since the terms of the sum are decreasing, we get an upper bound if we replace them all by the first one:

$$T(N) \le \Theta(1) + N + \sum_{i=1}^{\log \log N} N^{1/2}.$$

But now we can evaluate this more simply:

$$T(N) \le \Theta(1) + N + N^{1/2} \log \log N = \Theta(N).$$

Thus the amortized cost is $\Theta(N)/N = \Theta(1)$. □