CSE 421 Winter 2026: Section 6

February 12, 2026

Instructions: This section worksheet is designed to assist you in working through some example problems and developing your basics. You are encouraged to collaborate on the problems on this section as well as use the course's generative AI.

1 Mechanical calculations: Knapsack

The following is a Knapsack problem. Propagate the $(n+1) \times W$ dynamic programming table for this problem from lecture. Also, propagate the boolean table of include or exclude for this problem. Then output the optimal Knapsack.

Knapsack capacity $W = 10$.

item i	1	2	3	4	5
weight w_i	2	8	4	2	5
value v_i	5	3	2	7	4

2 Airline pilot scheduling with fatigue constraints

An airline is scheduling a single pilot's flight duty assignments over n days. If the pilot is assigned to fly on day i, they earn a known integer pay amount $A[i] \geq 0$; if the pilot is not assigned to fly, they earn 0 for that day. However, due to fatigue and safety regulations, the pilot may not be assigned to fly for three consecutive days. That is, for every $i \in \{1, 2, \ldots, n-2\}$, the schedule cannot assign the pilot to days $i, i+1, i+2$ all as flying days.

Given the array A of integers of length n, choose the days on which the pilot flies so as to maximize the total pay, subject to safety regulations. You may assume that all integer arithmetic operations (addition, subtraction, comparison, and assignment) take unit time.

Instructions: You are free to solve this problem however you like — the following "step-by-step" procedure is our general technique for thinking about algorithms.

1. **What is the problem asking?** It is hard to solve a problem if you don't know what you are supposed to do. Reread the problem slowly, highlighting key parameters, requirements, and the format of the "return type" that the answer needs to be provided in. Ask yourself the following questions:

- Are they are technical terms in the problem that you don't understand?
- What is the input type? (ex., Array, Graph (directed/undirected/weighted), Integer, something else)?
- What is the output type? Do I need to keep track of decisions I make (ex. assignments, weights, values, etc.) as I run the algorithm?
- For a word problem, can you extract the core algorithmic question?

2. **What are some good examples?** Come up with some examples in an effort to ensure that you truly understand the problem. Generate a few sample instances and calculate the optimal solution.

   Did you notice any patterns while coming up with the solution? If so, what were they, and can those "patterns" be expressed algorithmically? If so, what properties might we need to prove? At this stage, the goal is to just come up with a few (2-3) examples, but it is important to keep an eye on the goal.

   In terms of examples, our suggestion is to **not** focus on base or edge cases. Rather, we want to understand the general behavior of the algorithm. Come up with examples that are decently large ($n \approx 5 - 10$) and are sufficiently different. For example, if the problem is described by a graph, make sure that at least one example is *non-planar*.

3. **What is a baseline?** In a time-constrained setting (ex., interview or exam), it is good to come up with a "baseline" or "brute-force" algorithm. It may not be optimal, but it can help give you a baseline to compare any future improvement against.

4. **What problems does this remind me of?** A great starting point is identifying all the problems you have seen that are similar. Try thinking through the following questions:

   - Does this remind me of any prior problems? What techniques did we use there?
   - Does the problem look like a *standard* algorithm when turned on its head? For example, do I need to create a graph or adjust the data and *then* run a standard algorithm on the new graph/data?
   - Is there some structure to the input? For example, is it 1-dimensional?
   - Is there some structure to the output? If we need to make multiple selections or decisions, can we make them *greedily*?

5. **Write an algorithm.** Try running your algorithms against your examples. Did something click? If one of your algorithms seems to be working, construct an example problem that bucks your algorithm – meaning make sure every path through your example is explored. If you have an "if"

argument in your algorithm, make sure there exist examples that apply to both sides of the "if" statement. If you cannot come up with such an example, perhaps the "if" statement wasn't needed.

6. **Prove that your algorithm is correct.** As we explore more algorithms, we will also come up with more techniques for proving correctness. For this problem, there are a couple of equivalent proofs of correctness. Any will do; come up with the one that feels most natural for you to write.

3 Additional Problems

3.1 Longest increasing subsequence

Given an array A of integers of length n, find the longest sub-array of elements that is increasing. For this problem, a sub-array is any ordered subset of the entries; they do not need to be contiguous.

An example: $[10, -2, 5, 0, 3, 11, 8]$ has a longest increasing subsequence of 4 elements: $[-2, 0, 3, 8]$.

Assume comparisons of the integers requires $O(1)$ time (i.e., unit cost).

3.2 Longest Common Subsequence

Given two strings s with length m and t with length n, find the length of their longest common subsequence. A subsequence is a (possibly non-contiguous) substring. Assume character comparisons are unit time.

Examples:

1. Input s="backs", t="arches": the longest common subsequence is "acs", so the output should be 3.
2. Input s="skaters", t="hated": the longest common subsequence is "ate", so the output should be 3.

4 Solutions

4.1 Mechanical calculations: Knapsack

$V[i, W'] =$

$i \backslash W'$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	7	7	12	12	12	12	14	16	16
4	0	0	7	7	12	12	12	12	14	14	14
3	0	0	5	5	5	5	7	7	7	7	8
2	0	0	5	5	5	5	5	5	5	5	8
1	0	0	5	5	5	5	5	5	5	5	5
0	0	0	0	0	0	0	0	0	0	0	0

$\text{Inc}[i, W'] =$

$i \backslash W'$	0	1	2	3	4	5	6	7	8	9	10
5	F	F	F	F	F	F	F	F	F	T	T
4	F	F	T	T	T	T	T	T	T	T	T
3	F	F	F	F	F	F	T	T	T	T	F
2	F	F	F	F	F	F	F	F	F	F	T
1	F	F	T	T	T	T	T	T	T	T	T

Tracing the path back using Inc:

$$S[5, 10] = \{5\} \cup S[4, 5] = \{5, 4\} \cup S[3, 3] = \{5, 4\} \cup S[2, 3] = \{5, 4\} \cup S[1, 3] = \{5, 4, 1\} \cup S[0, 1] = \{5, 4, 1\}.$$

Thus, an optimal set is items $\{5, 4, 1\}$ for a total weight of 9 and a value of 16.

4.2 Airline pilot scheduling

1. **What is the problem asking?**

   Input: An integer array $A[1..n]$ with $A[i] \geq 0$ for each day i.

   Output: A *set of days* $S \subseteq \{1, 2, \ldots, n\}$ on which the pilot flies.

   Feasibility constraint (fatigue): The set S may not contain three consecutive days. Equivalently, for every $i \in \{1, 2, \ldots, n-2\}$, it is *not* the case that

   $$i \in S, \;\; i+1 \in S, \;\; i+2 \in S.$$

   (So the schedule is forbidden from having three flying days in a row.)

   Objective: Maximize the total pay $\sum_{i \in S} A[i]$. Integer arithmetic operations (addition, subtraction, comparison, assignment) take unit time.

2. **What are some good examples?**

   It is often helpful to represent a schedule by a *boolean vector* $x \in \{0, 1\}^n$, where

   $$x[i] = \begin{cases} 1 & \text{if the pilot flies on day } i, \\ 0 & \text{if the pilot rests on day } i. \end{cases}$$

   The constraint becomes: for every $i \in \{1, \ldots, n-2\}$, we cannot have $x[i] = x[i+1] = x[i+2] = 1$. The total pay is $\sum_{i=1}^{n} x[i] \cdot A[i]$.

Example 1. Let $A = [5, 1, 4, 9, 2, 6]$. One optimal choice is

$$x = [1, 0, 1, 1, 0, 1],$$

which corresponds to the set $S = \{1, 3, 4, 6\}$ and total pay $5 + 4 + 9 + 6 = 24$.

Example 2. Let $A = [3, 10, 3, 10, 3, 10, 3]$. A high-paying legal schedule is

$$x = [1, 1, 0, 1, 0, 1, 1],$$

corresponding to $S = \{1, 2, 4, 6, 7\}$ with total pay $3 + 10 + 10 + 10 + 3 = 36$.

Example 3. Let $A = [8, 7, 6, 1, 9, 2, 10, 1]$. A strong legal schedule is

$$x = [1, 1, 0, 1, 1, 0, 1, 0],$$

corresponding to $S = \{1, 2, 4, 5, 7\}$ with total pay $8 + 7 + 1 + 9 + 10 = 35$.

Pattern noticed: The legality of continuing the schedule depends only on how many consecutive 1's appear at the end of the prefix so far (namely 0, 1, or 2).

3. **What is a baseline?**

   A brute-force baseline is to enumerate all boolean vectors $x \in \{0, 1\}^n$ (equivalently all subsets $S \subseteq \{1, \dots, n\}$), discard any schedule that contains three consecutive 1's, and among the remaining ones return the S maximizing $\sum_{i=1}^{n} x[i]A[i]$.

   This takes 2^n iterations; checking the constraint and evaluating pay takes $O(n)$ time per schedule, for total time $O(n2^n)$.

4. **What problems does this remind me of?**

   This is a 1-dimensional "prefix DP" problem, similar in spirit to Fibonacci-style recurrences: The optimal solution for the first i days can be expressed using information from the first $i - 1$ days plus a constant amount of state that summarizes the "recent history" (here, the length of the current consecutive-flying streak).

5. **Write an algorithm.**

   Define a function $f(i, k)$ for $i \in \{0, 1, \dots, n\}$ and $k \in \{0, 1, 2\}$ as follows: $f(i, k)$ is the maximum pay obtainable from days 1 through i among all legal schedules whose consecutive-flying streak ending at day i has length k, where $k = 0$ means day i is a rest day.

   Base case:

   $$f(0, 0) = 0, \qquad f(0, 1) = f(0, 2) = -\infty.$$

Recurrence: For $i \geq 1$,

$$f(i,0) = \max\{f(i-1,0), f(i-1,1), f(i-1,2)\},$$
$$f(i,1) = f(i-1,0) + A[i],$$
$$f(i,2) = f(i-1,1) + A[i].$$

These transitions enforce the fatigue constraint by never allowing a streak of length 3.

Value of an optimal schedule:

$$\max\{f(n,0), f(n,1), f(n,2)\}.$$

Recovering the set S: Store a parent pointer for each state (i,k) indicating which previous state attained the maximum in the recurrence (for $k = 1, 2$ the predecessor is forced; for $k = 0$ it is whichever state attains the max). Then backtrack from the best state among $(n,0), (n,1), (n,2)$: include day i in S exactly when the state at day i has $k \in \{1,2\}$.

Running time: For each i we compute three values using $O(1)$ integer operations, so the total time is $O(n)$. The DP table uses $O(n)$ space (or $O(1)$ space if we only want the optimal value and do not backtrack).

6. **Prove that your algorithm is correct.**

   We prove by induction on i that for each $k \in \{0,1,2\}$, the quantity $f(i,k)$ equals the maximum total pay among all legal schedules on days $1..i$ that end with a consecutive-flying streak of length k (with $k = 0$ meaning day i is a rest day).

   Base case ($i = 0$): The empty schedule has total pay 0 and ends with streak length 0, so $f(0,0) = 0$. It is impossible to end day 0 with a flying streak, so $f(0,1) = f(0,2) = -\infty$.

   Inductive step: Fix $i \geq 1$ and assume the statement holds for $i-1$. We consider each k:

   - *Case $k = 0$.* Day i is a rest day, so the schedule on days $1..i-1$ may end with streak length 0, 1, or 2. Resting on day i adds 0 pay. Therefore, the best value is $\max\{f(i-1,0), f(i-1,1), f(i-1,2)\}$, which is exactly the recurrence for $f(i,0)$.
   - *Case $k = 1$.* Day i is a flying day and the ending streak length is 1, so day $i-1$ must be a rest day (streak length 0 at day $i-1$). Thus the best value is $f(i-1,0) + A[i]$, which is the recurrence for $f(i,1)$.
   - *Case $k = 2$.* Day i is a flying day and the ending streak length is 2, so day $i-1$ must be a flying day with ending streak length 1. Thus the best value is $f(i-1,1) + A[i]$, which is the recurrence for $f(i,2)$.

In each case, the recurrence exhaustively considers all legal ways to obtain the required ending streak length and chooses the maximum; by the inductive hypothesis, these candidate values are correct, hence so is $f(i, k)$. This completes the induction.

Finally, any legal schedule on $1..n$ must end with streak length $k \in \{0, 1, 2\}$, so the optimal total pay is $\max\{f(n, 0), f(n, 1), f(n, 2)\}$. The parent pointers reconstruct a schedule attaining this value, and the recovered set S is therefore optimal.

4.3 Additional problems

4.3.1 Longest increasing subsequence

Algorithm For each index $i \in \{1, \dots, n\}$, let $L[i]$ denote the maximum length of an increasing subsequence whose final element is $A[i]$. Also maintain an array $\text{pred}[i] \in \{1, \dots, n\} \cup \{\text{NIL}\}$ storing a predecessor index for reconstructing a subsequence.

Initialize $L[i] \leftarrow 1$ and $\text{pred}[i] \leftarrow \text{NIL}$ for all i. Then process indices $i = 1, 2, \dots, n$ in increasing order. When processing i, examine every earlier index $j < i$. For each j with $A[j] < A[i]$, the algorithm may extend an increasing subsequence ending at $A[j]$ by appending $A[i]$, obtaining a candidate length $L[j] + 1$. Set $L[i]$ to the maximum candidate length over all such j (or leave $L[i] = 1$ if no such j exists), and set $\text{pred}[i]$ to an index j achieving this maximum.

Finally, let $i^\star$ be any index maximizing $L[i]$. Output the subsequence obtained by starting at $i^\star$ and repeatedly following $\text{pred}[\cdot]$ until reaching NIL, then reversing the resulting list of values.

Proof of correctness We prove that for every index i, the value $L[i]$ computed by the algorithm equals the length of the longest increasing subsequence that ends at $A[i]$.

Fix an index i. Consider any increasing subsequence that ends at $A[i]$. If it has length 1, it consists only of $A[i]$, so $L[i] \geq 1$ is valid. Otherwise, it has some second-to-last element $A[j]$ with $j < i$ and $A[j] < A[i]$. In this case, the subsequence length equals

$$\big(\text{length of an increasing subsequence ending at } A[j]\big) + 1 \;\leq\; L[j] + 1.$$

The update rule for $L[i]$ is *exhaustive* over all possible choices of such indices $j < i$ with $A[j] < A[i]$: it considers every such j and takes the maximum value of $L[j] + 1$. Therefore, $L[i]$ is at least the length of every increasing subsequence ending at $A[i]$, hence $L[i]$ is an upper bound on the optimum.

Conversely, whenever the algorithm sets $L[i] = L[j] + 1$ for some j with $A[j] < A[i]$, this value is achievable by taking a longest increasing subsequence ending at $A[j]$ (which has length $L[j]$ by definition of $L[j]$) and appending $A[i]$. Thus $L[i]$ is also a lower bound on the optimum. Therefore, $L[i]$ equals the optimum length for subsequences ending at $A[i]$.

Every increasing subsequence ends at some index i, so the overall LIS length is $\max_i L[i]$. Choosing $i^\star$ attaining this maximum yields a longest increasing subsequence, and the predecessor pointers $\text{pred}[\cdot]$

reconstruct such a subsequence by following the maximizing transitions.

Running time For each i the algorithm compares $A[i]$ to all earlier $A[j]$, so there are $\sum_{i=1}^{n}(i-1) = O(n^2)$ comparisons and constant-time updates. Reconstruction takes $O(n)$. Total time is $O(n^2)$ and total space is $O(n)$.

4.3.2 Longest common subsequence

Algorithm Define $\mathrm{DP}[i, j]$ to be the length of the longest common subsequence between the prefixes $s[1..i]$ and $t[1..j]$. Initialize the base cases $\mathrm{DP}[0, j] = 0$ for all j and $\mathrm{DP}[i, 0] = 0$ for all i.

Fill the table in increasing order of i and j. For each pair (i, j) with $1 \le i \le m$ and $1 \le j \le n$:

- If $s[i] = t[j]$, set $\mathrm{DP}[i, j] \leftarrow \mathrm{DP}[i-1, j-1] + 1$.
- If $s[i] \neq t[j]$, set $\mathrm{DP}[i, j] \leftarrow \max\{\mathrm{DP}[i-1, j],\ \mathrm{DP}[i, j-1]\}$.

Output $\mathrm{DP}[m, n]$.

Proof of correctness We prove by induction on $i + j$ that $\mathrm{DP}[i, j]$ equals the LCS length of $s[1..i]$ and $t[1..j]$. If $i = 0$ or $j = 0$, one of the strings is empty, so the longest common subsequence length is 0; the initialization $\mathrm{DP}[0, j] = 0$ and $\mathrm{DP}[i, 0] = 0$ is correct.

Fix $i, j \ge 1$ and assume the claim holds for all pairs (i', j') with $i' + j' < i + j$.

If $s[i] = t[j]$, then greedily, there exists an optimal common subsequence that ends with this matching character. Removing that last character yields a common subsequence of $s[1..i-1]$ and $t[1..j-1]$, so the optimum value is

$$1 + \mathrm{LCS}(s[1..i-1], t[1..j-1]) \;=\; 1 + \mathrm{DP}[i-1, j-1],$$

where the equality uses the induction hypothesis. The algorithm sets $\mathrm{DP}[i, j]$ to exactly this value.

Whereas, if $s[i] \neq t[j]$, consider any common subsequence X of $s[1..i]$ and $t[1..j]$. Since the last characters differ, X cannot match both $s[i]$ and $t[j]$ as its final matched character. Thus, *exhaustively*, every valid X falls into one of two cases: either X does not use $s[i]$, in which case X is a common subsequence of $s[1..i-1]$ and $t[1..j]$, or X does not use $t[j]$, in which case X is a common subsequence of $s[1..i]$ and $t[1..j-1]$. Therefore, the optimum LCS length is

$$\max\{\mathrm{LCS}(s[1..i-1], t[1..j]),\ \mathrm{LCS}(s[1..i], t[1..j-1])\} \;=\; \max\{\mathrm{DP}[i-1, j],\ \mathrm{DP}[i, j-1]\},$$

where the second equality uses the induction hypothesis. The algorithm sets $\mathrm{DP}[i, j]$ to this maximum, so it is correct.

By induction, $\mathrm{DP}[i, j]$ is correct for all (i, j), and in particular $\mathrm{DP}[m, n]$ is the length of the longest common subsequence of s and t.

Running time The table has $(m+1)(n+1) = O(mn)$ entries, and each entry is computed in $O(1)$ time. Thus, the total time is $O(mn)$ and the total space is $O(mn)$.