1. Describe and analyze an algorithm that determines, given an array of target colors $C[1..n]$ as input, the minimum number of nickels that Huck must spend to have his fence completely painted.

Solution: First we observe that we only need to consider painting plans that satisfy the following *nesting constraint*:

> *After painting any interval of slats, the first and last slat in that interval will never be painted again.*

Suppose to the contrary that we decide to paint interval $i..j$ and then later repaint slat i. Replacing $i..j$ with the shorter interval $i+1..j$ does not change any slat's final color. Moreover, this modified plan either has the same cost as the original plan (if $i < j$) or lower cost (if $i = j$). Thus, if the original plan has minimal cost, so does the modified plan.

Any painting plan that satisfies this constraint also satisfies all of the following conditions:

- For each painted interval, the first and last slats have the same target color, and we paint the interval that color.
- Every pair of painted intervals is either properly nested (——) or disjoint (— —). (This is why I call it the "nesting constraint".)
- Just before any interval is painted, every slat in that interval has the same color.
- Without loss of generality, slat 1 is painted on the first day.

Let $Nick(i, j, c)$ denote the minimum number of nickels needed to paint slats i through j, assuming those slats start with color c. We need to compute $Nick(i, n, \emptyset)$, where $\emptyset$ is a special "color" that means "unpainted". The nesting constraint implies that this function satisfies the following recurrence:

$$Nick(i,j,c) = \begin{cases} 0 & \text{if } i > j \\ Nick(i+1,j,c) & \text{if } c = C[i] \\ Nick(i,j-1,c) & \text{if } c = C[j] \\ \min \left\{ \begin{array}{l} 1 + Nick(i,i'-1,c) \\ \quad + Nick(i'+1,j'-1,C[i']) \\ \quad + Nick(j'+1,j',c) \end{array} \middle| \begin{array}{l} i \le i' \le j' \le j \\ C[i'] = C[j'] \end{array} \right\} & \text{otherwise} \end{cases}$$

The last case of the recurrence considers all possible intervals $i'..j'$ that could be painted on the first day.

We can memoize this function into an $n \times n \times k$ array, where k is the number of colors. We can fill the array with three nested loops, decreasing i and increasing j in the two outer loops (in either order) and considering colors c in any order in the innermost loop. Filling a single entry $Nick[i, j, c]$ takes $O(n^2)$ time. So the resulting algorithm runs in $\boldsymbol{O(n^4k) = O(n^5)}$ ***time***. ⟨⟨*This is worth 7 points.*⟩⟩

We can improve this running time by a factor of n by taking further advantage of the nesting constraint. In any painting plan for the interval $i..j$ that satisfies the nesting constraint, either slat i starts with its final color and thus is never painted, or slat i is painted exactly once, without loss of generality on the first day.

So let's "redefine" $Nick(i,j,c)$ to denote the minimum number of nickels needed to paint slats i through j, assuming those slats start with color c *and the first interval we paint (if any) starts with the first slat in the range $i..j$ whose target color is not c*. This formulation implies the following modified recurrence:

$$Nick(i,j,c) = \begin{cases} 0 & \text{if } i > j \\ Nick(i+1,j,c) & \text{if } c = C[i] \\ \min \left\{ \begin{array}{l|c} 1 + Nick(i+1, j'-1, C[i]) & i \le j' \le j \\ \quad + Nick(j'+1, j, c) & C[i] = C[j'] \end{array} \right\} & \text{otherwise} \end{cases}$$

Again, the last case considers all possible intervals $i..j'$ that could be painted on the first day.

We can memoize this function into an $n \times n \times k$ array, which we can fill with three nested loops, decreasing i and increasing j in the outer two loops, and considering c in any order in the innermost loop. For each i, j, and c, we can compute $Nick[i,j,c]$ in $O(n)$ time, so the whole algorithm runs in $\boldsymbol{O(n^3k) = O(n^4)}$ ***time***. ⟨⟨*This is worth 10 points.*⟩⟩

But as a student pointed out in office hours on Wednesday, we can improve even further! For reasons that will become clear shortly, suppose we add a sentinel value $C[n+1] = \emptyset$ to the end of our target color array.

Lemma 1. *Suppose we evaluate $Nick(1,n,\emptyset)$ using the recurrence given above. Then for every recursive call $Nick(i,j,c)$ that is actually performed, we have $c = C[j+1]$.*

Proof: I claim more strongly that for all indices $1 \le i \le n$ and $1 \le j \le n$, if we evaluate $Nick(i,j,C[j+1])$ using the recurrence above, then for every recursive call $Nick(\tilde{i},\tilde{j},\tilde{c})$, we have $\tilde{c} = C[\tilde{j}+1]$. This claim implies the lemma, because $\emptyset = C[n+1]$.

We prove this claim by induction[a] as follows. Fix any indices i and j.

- If $i > j$, then $Nick(i,j,C[j+1])$ makes no recursive calls, so the claim is satisfied vacuously.
- Otherwise, if $c = C[i]$, we recursively call $Nick(i+1,j,C[j+1])$, which satisfies the claimed invariant. Thus, the induction hypothesis implies that all deeper recursive calls satisfy the claimed invariant.
- Otherwise, we recursively call $Nick(i+1,j'-1,C[j])$ and $Nick(j'+1,j,C[j+1])$, for at least one index j'. All of these recursive calls satisfy the claimed invariant, so by the induction hypothesis, all deeper recursive calls satisfy it as well. □

Lemma 1 implies that we can remove the third parameter from our function and define $Nick(i,j) = Nick(i,j,C[j+1])$. In other words, let $Nick(i,j)$ denote the minimum

number of nickels needed to paint slats i through j, *assuming those slats start with color* $C[j+1]$.

We can still assume without loss of generality that the first painted interval starts as far to the left as possible, so our simpler function satisfies a simpler recurrence:

$$Nick(i,j) = \begin{cases} 0 & \text{if } i > j \\ Nick(i+1,j) & \text{if } C[i] = C[j+1] \\ \min \left\{ \begin{array}{c} 1 + Nick(i+1, j'-1) \\ + Nick(j'+1, j) \end{array} \middle| \begin{array}{c} i \le j' \le j \\ C[i] = C[j'] \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into an $n \times n$ array, which we can fill with two nested loops, decreasing i and increasing j in either order. For each i and j, we can compute $Nick[i,j]$ in $O(n)$ time, so the whole algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ⟨⟨*This is worth 15 points.*⟩⟩

[a] either structural induction on the recursion tree, or induction on the interval length $j - i + 1$, whichever you like more

Rubric: 10 points, standard dynamic programming rubric; scale for slower or faster algorithms as described above. These are not the only correct algorithms with these running times. In hindsight, I'm sure there is a simpler way to derive the final recurrence, but that's how I got there. I have no reason to believe that $O(n^3)$ is the best possible running time for this problem.

In office hours, a few students suggested variants of the following greedy heuristic: *Always choose the longest possible starting interval*. This heuristic does not always return the best plan. For example, given the target colors $[1,2,3,2,1,3,2,1,2,3]$, the greedy heuristic always yields plans that cost 7 nickels. For example, we could start with the longest interval between 1s:

1 2 3 2 1 3 2 1 2 3

Or we could start with the longest interval between 2s:

1 2 3 2 1 3 2 1 2 3

But the unique optimal plan costs only 6 nickels:

1 2 3 2 1 3 2 1 2 3

Unfortunately, I don't see any monotonicity in the memoization array that would support a speedup through either Knuth/Yao or SMAWK. In particular, the unique optimal plan for the color sequence $[1,2,3,2,1,3,2,1,\mathbf{2,3}]$ starts with a shorter first interval than the unique optimal plan for the prefix $[1,2,3,2,1,3,2,1]$.

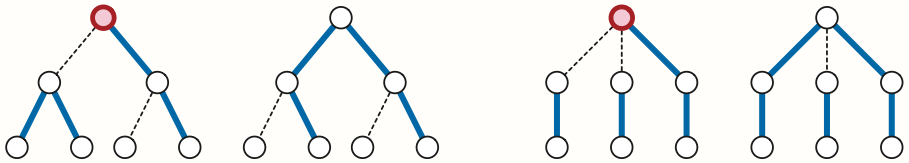
1 2 3 2 1 3 2 1

2. Let T be an arbitrary tree—a connected undirected graph with no cycles—each of whose edges has some positive weight. Describe and analyze an algorithm to cover the vertices of T with disjoint paths whose total length is as large as possible.

Solution: Choose an arbitrary root vertex r in the input tree T, and direct all edges away from r. I will write $w \downarrow v$ to denote that w is a child of v.

For any node v and boolean p, let $MaxLen(v, p)$ denote the maximum length of the longest disjoint paths that cover v and all its descendants, with the constraint that v is an endpoint of the path that contains it if $p = \text{TRUE}$ (and no constraint if $p = \text{FALSE}$). We need to compute $MaxLen(r, \text{FALSE})$.

Consider the trees in the figure below; all edges have weight 1. If v is the root of the tree on the left, then $MaxLen(v, \text{TRUE}) = MaxLen(v, \text{FALSE}) = 4$. If v is the root of the tree on the right, then $MaxLen(v, \text{TRUE}) = 4$ and $MaxLen(v, \text{FALSE}) = 5$. (In all cases, the paths in the figure are not the only optimal paths.)



(To save space, I'll write T and F from now on instead of TRUE and FALSE.) To develop the recurrence, let's first consider the case where $p = T$. In the optimal path cover, the path containing v either consists entirely of v or it extends down to one of v's children (but we don't know which one). So we have the first half of a recurrence:

$$MaxLen(v, T) = \max \left\{ \begin{array}{c} \displaystyle\sum_{w \downarrow v} MaxLen(w, F) \\ \displaystyle\max_{w \downarrow v} \left(MaxLen(w, T) + \sum_{x \downarrow v, x \neq w} MaxLen(x, F) \right) \end{array} \right\}$$

(The base case $MaxLen(v, T) = 0$ when v is a leaf is implicit here, because the empty sum is 0 and the empty max is $-\infty$.) Evaluating this recurrence naively would require two nested loops over the children of v, which takes $O(\deg(v)^2)$ time plus the time for recursive calls.

But the sums we need to evaluate are almost identical! Instead of summing $MaxLen(x, F)$ for all but one child w in the inner loop, we can save time by summing $MaxLen(x, F)$ over all children *once*, and then adding $MaxLen(w, T) - MaxLen(w, F)$ for each child w. Define the helper functions

$$NoPath(v) = \sum_{x \downarrow v} MaxLen(x, F)$$

$$Diff(v) = MaxLen(v, T) - MaxLen(v, F)$$

Then we have a simpler recurrence

$$MaxLen(v, T) = NoPath(v) + \max\left\{0, \max_{w \downarrow v} Diff(w)\right\}$$

Now the two loops (over x and w) are in series rather than nested, so we can evaluate $MaxLen(v, T)$ in $O(\deg(v))$ time (again, not counting time spent in recursive calls).

The logic for $p = F$ is similar: In the optimal path cover, the path through v extends to either zero, one, or two of v's children.

$$MaxLen(v, F) = NoPath(v) + \max\left\{\begin{array}{c} 0 \\ \max_{w \downarrow v} Diff(w) \\ \max_{w \downarrow v, x \downarrow v, w \neq x} \big(Diff(w) + Diff(x)\big) \end{array}\right\}$$

Naively, the last maximum would again be implemented as two nested loops, but that's overkill; finding the two largest elements of a list in linear time is straightforward. Thus, we can evaluate $MaxLen(v, T)$ in $O(\deg(v))$ time, not counting time spent in recursive calls.

Finally, we can memoize these mutual recurrences into two new fields at each vertex of the tree. As usual, we can evaluate the recurrence in postorder via depth-first search from the root. The resulting algorithm spends $O(\deg(v))$ at each vertex v—or equivalently, $O(1)$ time at each edge—so the overall running time is $\boldsymbol{O(n)}$, where n is the number of vertices in T. ■

Rubric: 10 points, standard dynamic programming rubric. Maximum scores for slower solutions as below; scale partial credit:

- $O(n \log n)$: 9 points
- $O(n^2)$ time: 8 points
- $O(n^2 \log n)$: 7 points
- $O(n^3)$ time: 6 points

Maximum credit for any submission should depend on its *actual* worst-case running times, not the running time it reports! (If there is a discrepancy between these two time bounds, take off points for incorrect analysis.) For example, if a submission loosely bounds the running time of an algorithm by writing $\sum_v O(\deg(v)) \leq \sum_v O(n) = O(n^2)$, that algorithm *actually* runs in $\sum_v O(\deg(v)) = O(E) = O(n)$ time.

3. Describe and analyze an algorithm to find the length of the longest palindrome path in a directed acyclic graph with labeled vertices.

Solution: We start by topologically sorting the input dag G, because that's *always* the first thing one does with a dag. Topological sort labels the vertices with consecutive integers from 1 to V, so that every edge points from a lower label to a higher label. Let $label[i]$ denote the label of the ith vertex in topological order.

For any two vertices i and ℓ (identified by their indices in topological order), let $LPPL(i,\ell)$ denote the length of the longest palindrome that is a label of a path from i to ℓ in G. (*LPPL* is a mnemonic for "Longest Palindrome Path Length".) We need to compute $\max_{i,\ell} LPPL(i,\ell)$.

The *LPPL* function obeys the following recurrence.

$$LPPL(i,\ell) = \begin{cases} -\infty & \text{if } label[i] \neq label[\ell] \\ 1 & \text{if } i = \ell \\ \max\left\{\max\{2 + LPPL(j,k) \mid i{\to}j, k{\to}\ell\},\ 2\right\} & \text{if } i{\to}\ell \in E \\ \max\{2 + LPPL(j,k) \mid i{\to}j, k{\to}\ell\} & \text{otherwise} \end{cases}$$

The first two base cases are straightforward. The third case includes a base case for the single edge $i{\to}\ell$, which is necessary for even-length palindromes, but also considers other possible paths from i to ℓ.

To argue that the recurrence is correct, we must show that if there are *no* paths from i to ℓ, then $LPPL(i,\ell) = -\infty$. We can assume $label[i] = label[\ell]$, since otherwise $LPPL(i,\ell) = -\infty$ by definition. Similarly, we can assume $i \neq \ell$ and $i{\to}\ell \neq E$, since otherwise there is a trivial path from i to ℓ. Thus, $LPPL(i,\ell)$ is determined by the last case of the recurrence. There are three cases to consider:

- If vertex i is a sink, then the outer max is over an empty set of edges, so $LPPL(i,\ell) = \max\emptyset = -\infty$.
- Similarly, if vertex ℓ is a source, then $LPPL(i,\ell) = \max_{i\to j} \max\emptyset = -\infty$.
- Otherwise, for every pair of edges $i{\to}j$ and $k{\to}\ell$, there is no path from j to k, so the inductive hypothesis implies that $LPPL(j,k) = -\infty$. $LPPL(i,\ell) = 2 - \infty = -\infty$ follows immediately.

We can memoize the *LPPL* function into a two-dimensional array $LPPL[1..V, 1..V]$, indexed by the vertices in topological order.[a] Each subproblem $LPPL[i,\ell]$ depends only on other subproblems $LPPL[j,k]$ with $i < j$ and $k < \ell$, so we can fill the array with two nested loops, decreasing i in one loop and increasing ℓ in the other; the nesting order of the two loops doesn't matter. Once the array *LPPL* is filled, we return the largest value that appears anywhere in that array.

The initial topological sort takes $O(V+E)$ time. Translating the for-loops into sums give us the following upper bound on the time to fill the *LPPL* array, ignoring

constant factors:

$$\sum_i \sum_\ell \left(1 + \sum_{i\to j} \sum_{k\to \ell} 1\right) = \sum_i \sum_\ell 1 + \sum_i \sum_\ell \sum_{i\to j} \sum_{k\to \ell} 1 = V^2 + E^2$$

(The double sum considers each ordered pair (i, ℓ) of vertices exactly once; similarly, the quadruple sum considers each ordered pair $(i{\to}j, k{\to}\ell)$ of edges exactly once.) Finding the largest entry in the finished *LPPL* array takes $O(V^2)$ time. We conclude that the entire algorithm runs in $\boldsymbol{O(V^2 + E^2)}$ ***time***. ■

[a]Alternatively, we could build and memoize into a product graph, similar to the next solution. But we can't easily memoize into vertices of the input graph G, because subproblems are specified by *pairs* of vertices. I suppose we could allocate a one-dimensional memoization array at each vertex of G, but the asymmetry of that solution makes it weird.

Solution (longest path in a dag): Let $label(v)$ denote the label of any vertex v of the input graph $G = (V, E)$. We construct a new directed acyclic graph $G' = (V', E')$ as follows. First the vertices V' are defined as

$$V' = \{(u, v) \mid u, v \in V \text{ and } label(u) = label(v)\} \cup \{s\}$$

(Here s is an artificial source vertex.) V' contains at most $O(V^2)$ vertices. The edge set E' is the union of three sets E_1, E_2, and $E_=$, where

$$E_1 = \{s{\to}(v, v) \mid v \in V\}$$
$$E_2 = \{s{\to}(v, w) \mid (v, w) \in V' \text{ and } v{\to}w \in E\}$$
$$E_= = \{(v, w){\to}(u, x) \mid (v, w), (u, x) \in V' \text{ and } u{\to}v, w{\to}x \in E\}$$

Every edge in E_1 has weight 1, and every edge in $E_2 \cup E_=$ has weight 2. E' contains at most $O(V + E + E^2) = O(E^2)$ edges.

Now every directed path in G' that starts at s and has length ℓ corresponds to a palindrome path in G that visits exactly ℓ vertices. For example, the path $s{\to}(a, a){\to}(b, c){\to}(d, e){\to}(f, g)$ in G', which has length 7, corresponds to the palindrome path $f{\to}d{\to}b{\to}a{\to}c{\to}e{\to}g$ in G. In particular, the longest palindrome path in G corresponds to the longest path in G' that starts at s.

We can construct G' by brute force in $O(V^2 + E^2)$ time. Then we can compute the longest path in G' that starts at s in $O(V' + E') = O(V^2 + E^2)$ time using the depth-first-search/dynamic-programming algorithm described in class and in the textbook. The overall algorithm runs in $\boldsymbol{O(V^2 + E^2)}$ ***time***. ■

Rubric: 10 points.

- Dynamic programming: standard rubric.
- Graph traversal: 2 for vertices + 2 for edges + 2 for noting correspondence between paths in G' and palindrome paths in G + 2 for algorithm + 2 for running time

No penalty for implicitly assuming the graph is (weakly) connected and reporting the running time as $O(E^2)$.