Exercise 5.1. Below is a series of optimization problems that takes as input an array $A[1..n]$ of integers, and asks for optimal subsequences of A satisfying certain properties. Design and analyze an algorithm for each of these problems, addressing items 1–5 from section 5.2.

Exercise 5.1.2. A sequence of numbers $x_1, \ldots, x_k$ is *convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \ldots, k-1$. Compute the length of the longest convex subsequence of A.

Recursive spec. Let $A[1..n]$ be a fixed array, and $cache[1..n][1..n]$ be an $n \times n$ matrix. Define the function $\texttt{convex}(i, j)$ to return the length of the longest convex subsequence ending in $A[j]$ and $A[i]$, with $i > j$. □

Recursive implementation.

convex(i, j):

1. If $j < 1$ or $i = 1$, then return 2
2. Else, return $\max\limits_{1 \leq k < j} \begin{cases} 1 + \text{convex}(j, k) & \text{if } A[i] - A[j] \geq A[j] - A[k] \\ 2 & \text{else} \end{cases}$

□

Dynamic Programming. We can utilize dynamic programming to reduce running time by filling an $\mathcal{O}(n^2)$ table $cache[1..n][1..n]$, in which the entry $cache[i][j]$ corresponds to $\texttt{convex}(i, j)$. □

Usage. To use this function, initialize some variable $tmp := 1$. Then iterate through the pairs (i, j) for which $1 \leq j < i < n$, and set $tmp = \max\{tmp, \texttt{convex}(i, j)\}$. Finally, return tmp. □

Analysis of running time. The function convex fills an $n \times n$ matrix without repeating any computations, which gives us $\mathcal{O}(n^2)$ subproblems.

Each subproblem takes $\mathcal{O}(n)$ time to complete in the worst case.

Thus our final time complexity is $\mathcal{O}(n^3)$. □

Exercise 5.1.4. Compute both

- the length of the longest increasing subsequences of A where the sum of integers is even,
- the length of the longest increasing subsequences of A where the sum of integers is odd.

Note. Based on solution key

Recursive spec. Returns a 2-tuple (a, b) where a and b represent the length of the LIS ending at i with sums of even and odd parity, respectively. $\square$

Recursive implementation.

`LIS-parity(`i`):`

1. If $A[i]$ is even: set $a := 1$ and $b := 0$
2. Else: set $a := 0$ and $b := 1$
3. For $1 \leq j \leq i - 1$:
   A. If $A[j] < A[i]$:
      1. Set $(\hat{a}, \hat{b}) :=$ `LIS-parity(j)`
      2. If $A[i]$ is even: set $a := \max(a, \hat{a} + 1)$ and $b := \max(b, \hat{b} + 1)$
      3. Else: set $a := \max(a, \hat{b} + 1)$ and $b := \max(b, \hat{a} + 1)$
4. return (a, b)

$\square$

Dynamic Programming. We can leverage dynamic programming by caching the result of the $\mathcal{O}(n)$ subcalls. $\square$

Usage. return $\max_{i \in [n]}$ `LIS-parity`(i) $\square$

Analysis of running time. The $\mathcal{O}(n)$ subcalls each take $\mathcal{O}(n)$ time, giving us a final time complexity of $\mathcal{O}(n^2)$. $\square$

Exercise 5.1.5. Suppose each entry in A is also colored red, white, or blue. We say that a sequence is *American* if the colors alternate red, white, blue, red, white, blue, ... The first number in the sequence can be any color. Compute the length of the longest increasing American subsequence of A. (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

Note. Based on solution key

Recursive spec. Returns longest increasing American subsequence of A ending at i. □

Recursive implementation. Assumes usage of subroutine `patriotic`(α, β) that returns true if color of $A[\alpha]$ follows color of $A[\beta]$ in sequence.

`LIS-USA`(i):

1. Define $len := 1$
2. For $1 \leq j < i$:
   A. If $A[j] < A[i]$ and `patriotic`(i, j):
      1. Set $len := \max\{len, \texttt{LIS-USA}(j) + 1\}$
3. return len

□

Dynamic Programming. We leverage dynamic programming by caching the result of the $\mathcal{O}(n)$ subcalls. □

Usage. return $\max\limits_{1 \leq i \leq n} \{\texttt{LIS-USA}(i)\}$ □

Analysis of running time. Each subcall takes $\mathcal{O}(n)$ time, so the final complexity is $\mathcal{O}(n^2)$. □

Exercise 5.1.6. A sequence $x_1, \ldots, x_k$ is a *palindrome* if the reversed sequence is the same; i.e., $(x_1, \ldots, x_k) = (x_k, \ldots, x_1)$. For example,

mom, dad, racecar, and gohangasalamiimalasagnahog

are all palindromes. Compute the length of the longest palindrome subsequence of A.

Note. Based on solution key

Recursive spec. Returns the length of the longest palindrome subsequence of A between indices i and j. □

Recursive implementation.

<u>`pal`(i, j):</u>

1. If $i = j$: return 1
2. return $\max\{2 + \texttt{pal}(i+1, j-1),\ \texttt{pal}(i+1, j),\ \texttt{pal}(i, j-1)\}$

□

Dynamic Programming. We leverage dynamic programming by caching the result of the $\mathcal{O}(n^2)$ subcalls. □

Usage. return $\texttt{pal}(1, n)$ □

Analysis of running time. Each subcall takes $\mathcal{O}(1)$ time, giving a final complexity of $\mathcal{O}(n^2)$ □

Exercise 5.2. Let $A[1..m]$ and $B[1..n]$ be two arrays. Each of the following problems asks for some aspect (e.g., the length) of a common subsequence of $A[1..n]$ and $B[1..n]$ satisfying or optimizing certain properties. (A common subsequence of A and B is a sequence that is a subsequence of both A and B.) Design and analyze an algorithm for each of these problems, addressing items 1–5 from section 5.2.

Exercise 5.2.1. Compute the length of the longest common subsequence of A and B.

Note. Based on solution key

Recursive spec. Returns length of longest common subsequence between $A[1..a]$ and $B[1..b]$. ☐

Recursive implementation.

$\texttt{common}(a, b)$:

1. If $a = 0$ or $b = 0$: return 0
2. If $A[a] = B[b]$: return $1 + \texttt{common}(a-1, b-1)$
3. return $\max\{\texttt{common}(a-1, b),\ \texttt{common}(a, b-1)\}$

☐

Dynamic Programming. Cache the solution to each of the $\mathcal{O}(n^2)$ subproblems ☐

Usage. return $2n - \texttt{common}(n, n)$ ☐

Analysis of running time. Each of the $\mathcal{O}(n^2)$ subcalls takes $\mathcal{O}(1)$ time, so the final comlexity is $\mathcal{O}(n^2)$ ☐

Exercise 5.2.2. Compute the length of the shortest common supersequence of A and B.

Note. Based on solution key

Recursive spec. Returns the length of the shortest palindrome subsequence between $A[a_1..a_2]$ and $B[b_1..b_2]$ □

Recursive implementation.

<u>`shortest-common-pal`(a_1, a_2, b_1, b_2):</u>

1. If $a_1 > a_2$ or $b_1 > b_2$: return 0
2. If $(a_1 = a_2$ or $b_1 = b_2)$ and $A[a_1] = B[b_1]$: return 0
3. If $A[a_1] = A[a_2]$ and $B[b_1] = B[b_2]$ and $A[a_1] = B[b_1]$: return $2 +$ `common-pal`$(a_1 + 1, a_2 - 1, b_1 + 1, b_2 - 1)$
4. return min $\begin{cases} \texttt{shortest-common-pal}(a_1 + 1, a_2, b_1, b_2) \\ \texttt{shortest-common-pal}(a_1, a_2 - 1, b_1, b_2) \\ \texttt{shortest-common-pal}(a_1, a_2, b_1 + 1, b_2) \\ \texttt{shortest-common-pal}(a_1, a_2, b_1, b_2 - 1) \end{cases}$

□

Dynamic Programming. Cache the solution to the n^4 subproblems □

Usage. return `shortest-common-pal`$(1, n, 1, n)$ □

Analysis of running time. Each of the n^4 subproblems takes $\mathcal{O}(1)$ time, so our final complexity is $\mathcal{O}(n^4)$ □

Exercise 5.2.3. Compute the length of the longest common subsequence of A and B that is a palindrome.

Note. Based on solution key

Recursive spec. Returns the length of the longest palindrome subsequence between $A[a_1..a_2]$ and $B[b_1..b_2]$ $\square$

Recursive implementation.

`longest-common-pal`(a_1, a_2, b_1, b_2):

1. If $a_1 > a_2$ or $b_1 > b_2$: return 0
2. If $(a_1 = a_2$ or $b_1 = b_2)$ and $A[a_1] = B[b_1]$: return 0
3. If $A[a_1] = A[a_2]$ and $B[b_1] = B[b_2]$ and $A[a_1] = B[b_1]$: return $2 + \texttt{common-pal}(a_1 + 1, a_2 - 1, b_1 + 1, b_2 - 1)$
4. return $\max \begin{cases} \texttt{longest-common-pal}(a_1 + 1, a_2, b_1, b_2) \\ \texttt{longest-common-pal}(a_1, a_2 - 1, b_1, b_2) \\ \texttt{longest-common-pal}(a_1, a_2, b_1 + 1, b_2) \\ \texttt{longest-common-pal}(a_1, a_2, b_1, b_2 - 1) \end{cases}$

$\square$

Dynamic Programming. Cache the solution to the n^4 subproblems $\square$

Usage. return `longest-common-pal`$(1, n, 1, n)$ $\square$

Analysis of running time. Each of the n^4 subproblems takes $\mathcal{O}(1)$ time, so our final complexity is $\mathcal{O}(n^4)$ $\square$

Exercise 5.10. ... [C]ompute the maximum number of sets that can be obtained in a game of Solitaire Set over the decks $A[1..m]$, $B[1..n]$, and $C[1..p]$. Design and analyze an algorithm for this problem.

Recursive spec. IDK!! □

Recursive implementation.

```
():
1. .
```

□

Dynamic Programming. □

Usage. □

Analysis of running time. □

Proof of correctness. □