Standard dynamic programming rubric. 10 points =

- 3 points for a clear and correct English description of the recursive function(s) you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - — 1 for naming the function "OPT" or "DP" or any single letter.
  - – No credit if the description is inconsistent with the recurrence.
  - – No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - – No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - – An English explanation of the *recurrence* or *algorithm* does not qualify. We want a specification of *what* your function is supposed to return, not (here) an explanation of *how* that value is computed.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive backtracking algorithm.
  - + 1 for base case(s). —½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 for recursive case(s). —1 for each *minor* bug, like a typo or an off-by-one error.
  - — 2 for greedy optimizations without proof, even if they are correct.
  - – **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details
  - + 1 for describing an appropriate memoization data structure (***not*** a hash table).
  - + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)
  - + 1 for correct time analysis. (It is not necessary to state a space bound.)

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct.

  If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode that fills an array named "dp", with no explanation or time analysis, is worth at most 5 points out of 10***.
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points (= 70% of 3, rounded).

1. (a) Describe and analyze an algorithm to determine whether one array $P[1..k]$ occurs as two ***disjoint*** subsequences of another array $T[1..n]$.

Solution: To simplify the following case analysis, we add a sentinel character $P[0] = \#$ that does not appear anywhere else in X or Y.

For any indices i, j, and ℓ, we defined a boolean value $DisSS(i, j, \ell)$, which is TRUE if the prefixes $P[1..i]$ and $P[1..j]$ occur as disjoint subsequences of the prefix $T[1..\ell]$, and FALSE otherwise. We need to compute $DisSS(k, k, n)$.

This function satisfies the following recurrence:

$$DisSS(i,j,\ell) = \begin{cases} \text{TRUE} & \text{if } i = 0 \text{ and } j = 0 \\ \text{FALSE} & \text{if } i + j > \ell \\ DisSS(i,j,\ell-1) & \\ \quad \vee \big((T[\ell] = P[i]) \wedge DisSS(i-1,j,\ell-1)\big) & \text{otherwise} \\ \quad \vee \big((T[\ell] = P[j]) \wedge DisSS(i,j-1,\ell-1)\big) & \end{cases}$$

We can memoize this function into a 3-dimensional array $DisSS[0..k, 0..k, 0..n]$. Each entry $DisSS[i, j, \ell]$ depends only on entries of the form $DisSS[\cdot, \cdot, \ell - 1]$. Thus, we can fill the array with three nested for loops, increasing ℓ in the outermost loop, and considering i and j in arbitrary order in the inner loops. The resulting algorithm runs in ***$O(nk^2)$ time***.

```
DisSS(P[1..k], T[1..n]):
  P[0] ← #
  for ℓ ← 0 to n
      for i ← 0 to k
          for j ← 0 to k
              if i + j = 0
                  DisSS[0, 0, ℓ] ← TRUE
              else if i + j > ℓ
                  DisSS[i, j, ℓ] ← FALSE
              else
                  DisSS[i, j, ℓ] ← DisSS[i, j, ℓ − 1]
                                   ∨ ((T[ℓ] = P[i]) ∧ DisSS[i − 1, j, ℓ − 1])
                                   ∨ ((T[ℓ] = P[j]) ∧ DisSS[i, j − 1, ℓ − 1])
  return DisSS[k, k, n]
```

I chose to recuse on prefixes here because that makes the final for-loops run in increasing order. If instead I defined $DisSS(i, j, \ell)$ to be TRUE if the suffixes $P[i..k]$ and $P[j..k]$ occur as disjoint subsequences of the suffix $T[\ell..n]$, I would get an equivalent dynamic programming algorithm with backward loops, with exactly the same running time. ■

Rubric: 5 points: standard dynamic programming rubric

(b) Describe and analyze an algorithm to compute the number of occurrences of one array $P[1..k]$ as a subsequence of another array $T[1..n]$.

Solution: For any indices i and j, let $NumSS(i,j)$ denote the number of times the prefix $P[1..i]$ appears as a subsequence of the prefix $T[1..j]$. We need to compute $NumSS(k,n)$. This function obeys the following recurrence:

$$NumSS(i,j) = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{if } i > j \\ NumSS(i,j-1) & \text{if } P[i] \neq T[j] \\ NumSS(i,j-1) + NumSS(i-1,j-1) & \text{if } P[i] = T[j] \end{cases}$$

These cases my require some explanation.

- The objects we are actually counting are functions $f : \{1,2,\ldots,i\} \to \{1,2,\ldots,j\}$ such that $f(\ell) > f(\ell-1)$ and $P[\ell] = T[f(\ell)]$ for all $1 \le \ell \le i$. For any set S, there is exactly one function $f : \varnothing \to S$: the empty function! This function vacuously satisfies whatever condition you want for all $1 \le \ell \le 0$. Also, 1 is the only value for $NumSS(0,j)$ that actually makes the recurrence correct.
- If $P[i] = T[j]$, then either each occurrence of $P[1..i]$ as a subsequence of $T[1..j]$ either includes $T[j]$ or omits $T[j]$, and these two choices are exclusive. If $P[i] \neq T[j]$, then each occurrence of $P[1..i]$ as a subsequence of $T[1..j]$ definitely excludes $T[j]$.

We can memoize this function into an array $NumSS[0..k, 0..n]$. Each entry $NumSS[i,j]$ depends only on entries $NumSS[\cdot, j-1]$ in the previous column, so we can fill the array using two nested loops, considering j in increasing order in the outer loop and considering i in arbitrary order in the inner loop. The resulting algorithm runs in $O(nk)$ *time*.

```
NumSS(P[1..k], T[1..n]):
  for j ← 0 to n
      NumSS[0, j] ← TRUE
      for i ← 1 to k
          if i > j
              NumSS[i, j] ← FALSE
          else if P[i] = T[j]
              NumSS[i, j] ← NumSS[i, j−1] + NumSS[i−1, j−1]
          else
              NumSS[i, j] ← NumSS[i, j−1]
  return NumSS[k, n]
```

■

Rubric: 5 points: standard dynamic programming rubric

2. Describe and analyze an algorithm to load the ferry as lightly as possible. *[See the homework handout for a detailed problem statement]*

Solution: Let L be the given ferry length, and let $len[1..n]$ be the given length array.

For any integers a, b, c, and m, let $Ferry(a, b, c, m)$ denote the minimum number of cars that can be loaded onto the ferry if

- lane 1 has a meters available,
- lane 2 has b meters available,
- lane 3 has c meters available, and
- there are $n - m + 1$ cars in the queue with lengths $len[m..n]$.

We need to compute $Ferry(L, L, L, 1)$. This function satisfies the following recurrence:

$$Ferry(a,b,c,m) = \begin{cases} 0 & \text{if } m > n \\ 0 & \text{if } len[m] > \max\{a,b,c\} \\ \infty & \text{if } a < 0 \text{ or } b < 0 \text{ or } c < 0 \\ 1 + \min \begin{cases} Ferry(a - len[m],\ b,\ c,\ m+1) \\ Ferry(a,\ b - len[m],\ c,\ m+1) \\ Ferry(a,\ b,\ c - len[m],\ m+1) \end{cases} & \text{otherwise} \end{cases}$$

We can memoize this function into an $L \times L \times L \times n$ array. We can fill this array in constant time per entry by *decreasing* the fourth index in the outermost loop, and considering the other three indices in any order in the inner loops. The resulting algorithm runs in at most $\boldsymbol{O(L^3 n)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. This is not the only correct evaluation order for this recurrence. This is not the only correct solution with this running time. There are several small improvements that are worth extra credit; see below. (Similar improvements apply to other solutions.)

Solution (+2 extra credit): Because the length of each vehicle is a positive integer, at most $3L$ vehicles can fit on the ferry. So if $n > 3L$, we can consider only the first $3L$ cars. This change improves the running time of our algorithm to $\boldsymbol{O(\min\{L^3 n, L^4\})}$. ■

Solution (+3 extra credit): We can speed up the previous algorithm by observing that in every recursive invocation of the *Ferry* function, the arguments satisfy the following identity:

$$3L - a - b - c = \sum_{i=1}^{m-1} len[i].$$

Both sides of this equation equal the total length of the first $m-1$ vehicles. Thus, one of the parameters of our recurrence is redundant. For any integers a, b, and m, let

Ferry2(a, b, m) denote the minimum number of cars that can be loaded onto the ferry under the following conditions:

- lane 1 has a meters available,
- lane 2 has b meters available,
- lane 3 has $3L - a - b - \sum_{i=1}^{m-1} len[i]$ meters available, and
- there are $n - m + 1$ cars in the queue with lengths $len[m..n]$.

We need to compute *Ferry2*$(L, L, 1)$. As a helper function, for any integer $0 \le i \le n$, let *TotalLen*(i) be the total length of the first i vehicles; this function obeys the following simple recurrence:

$$TotalLen(i) = \begin{cases} 0 & \text{if } i = 0 \\ TotalLen(i-1) + len[i] & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $TotalLen[0..n]$; we can easily fill this array in $O(n)$ time from left to right.

Now the *Ferry2* function obeys the following recurrence:

$$Ferry2(a, b, m) = \begin{cases} 0 & \text{if } m > n \\ 0 & \text{if } len[m] > \max\{a, b, c\} \\ \infty & \text{if } a < 0 \text{ or } b < 0 \text{ or } c < 0 \\ 1 + \min \begin{Bmatrix} Ferry2(a - len[m],\ b,\ m+1) \\ Ferry2(a,\ b - len[m],\ m+1) \\ Ferry2(a,\ b,\ m+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

where $c = 3L - a - b - TotalLen(m-1)$

We can memoize *Ferry2* into an $L \times L \times n$ array. Once the *TotalLen* array is filled, we can fill the *Ferry2* array in constant time per entry by *decreasing* the third index in the outermost loop, and considering the other two indices in any order in the inner loops. The resulting improved algorithm runs in $\boldsymbol{O(L^2 n)}$ ***time***. ■

Solution (+3 extra credit): We can speed up the previous algorithm by observing that in every recursive invocation of the *Ferry* function, the arguments satisfy the following identity:

$$3L - a - b - c = \sum_{i=1}^{m-1} len[i].$$

Both sides of this equation equal the total length of the first $m-1$ vehicles. Thus, one of the parameters of our recurrence is redundant.

For any integers a, b, and m, let *Ferry3*(a, b, c) denote the minimum number of cars that can be loaded onto the ferry when lane 1 has a meters available, lane 2

has b meters available, and lane 3 has c meters available. We need to compute $Ferry3(L, L, L)$.

Our algorithm will infer which car is at the front of the queue from the values of a, b, and c using the equation above. Specifically, at the start of the algorithm, we build an integer array $NextLen[0..3L]$ such that if $NextLen[a+b+c] > 0$, then $NextLen[a+b+c]$ is the length of the car the front of the queue. This subroutine runs in $O(L)$ time:

```
InitNextLen(len[1..n]):
  for i ← 1 to 3L
      NextLen[i] ← 0
  space ← 3L
  for i ← 1 to n
      NextLen[space] ← len[i]
      space ← space − len[i]
      if space < 0
          break
  return NextLen[1..3L]
```

Once the *NextLen* array is filled, the *Ferry3* function obeys the following recurrence:

$$Ferry3(a,b,c) = \begin{cases} 0 & \text{if } a+b+c = 0 \\ \infty & \text{if } NextLen[a+b+c] = 0 \\ \infty & \text{if } a<0 \text{ or } b<0 \text{ or } c<0 \\ 1 + \min\begin{Bmatrix} Ferry2(a-\ell,\ b,\ c) \\ Ferry2(a,\ b-\ell,\ c) \\ Ferry2(a,\ b,\ c-\ell) \end{Bmatrix} & \text{otherwise, where } \ell = NextLen[a+b+c] \end{cases}$$

We can memoize *Ferry3* into an $L \times L \times L$ array. We can fill the *Ferry3* array in constant time per entry using three nested loops, each of which *decreases* one of the array indices. The resulting improved algorithm runs in $\boldsymbol{O(L^3)}$ ***time***. ■

Solution (+5 extra credit): Because the length of each vehicle is a positive integer, at most $3L$ vehicles can fit on the ferry. So if $n > 3L$, we can consider only the first $3L$ cars. This change improves the running time of our algorithm to $\boldsymbol{O(\min\{L^2 n, L^3\})}$. ■

Non-solution: Neither of the natural greedy strategies for filling the ferry lead to optimal assignments:

- **Put each vehicle into the lane with the *least* available space:** Consider the input $L = 6$ and $len = [1, 1, 1, 6, 6]$.
  - The greedy strategy fits all five vehicles onto the ferry, without loss of generality in lanes $1, 1, 1, 2, 3$.
  - The optimal assignment loads the first three cars into three different lanes, after which the fourth vehicle doesn't fit anywhere.
- **Put each vehicle into the lane with the *most* available space:** Consider the input $L = 12$ and $len = [1, 2, 3, 4, 5, 6, 7]$.
  - The greedy algorithm loads all seven vehicles onto the ferry, without loss of generality in lanes $1, 2, 3, 1, 2, 3, 1$.
  - The optimal assignment loads the first six cars into lanes $1, 2, 3, 3, 2, 1$, after which the seventh vehicle doesn't fit anywhere.

♣

3. Fix a sorted array $Coin[1..n]$ of distinct positive integer coin values. For any integers k and T, let $Change(T,k) = \textsc{True}$ if some collection of exactly k coins (possibly including multiple coins with the same value) has total value T, and $\textsc{False}$ otherwise.

   (a) Describe a recurrence for $Change(T,k)$ in terms of $Change(T,k/2)$. (Assume k is a power of 2.)

Solution: In the recursive case, we consider all possible ways of splitting k coins with total value T into two subsets of $k/2$ coins. The base case is a boolean expression, not an assignment.

$$Change(T,k) := \begin{cases} (T = Coin[i] \text{ for some } i) & \text{if } k = 1 \\ \displaystyle\bigvee_{S=0}^{T} \big(Change(S,k/2) \wedge Change(T-S,k/2)\big) & \text{otherwise} \end{cases}$$

■

Rubric: 3 points

   (b) Describe an efficient algorithm to compute $Change(T,k)$, given the array $Coin[1..n]$, the target total T, and the allowed number of coins k as input. Analyze your algorithm as a function of the parameters n, T, and k.

Solution: First, we can memoize the function *Change* into a two-dimensional array $ChangeLog[0..T, 0..\log_2 k]$, where each entry $ChangeLog[S,i]$ stores the value $Change[S,2^i]$. We can fill this array in standard column-major order, increasing the second index i in the outer loop and considering the first index S in any order in the inner loop. Initializing the initial row $ChangeLog[\cdot,0]$ takes $O(T)$ time, and filling each later row by brute force takes $O(T^2)$ time, so the overall algorithm runs in $\boldsymbol{O(T^2 \log k)}$ time.

We can improve this algorithm further by observing that the recurrence in part (a) is essentially a convolution; the only differences are that (1) we only care about boolean values, not integers, and (2) we don't care about values of $Change(S,k)$ where $S > T$. For any index $\ell > 0$, we can compute $ChangeLog[\cdot,\ell]$ by invoking the following subroutine on two copies of $ChangeLog[\cdot,\ell-1]$:

```
ClippedBooleanConv(A[0..n], B[0..n]):
  C ← A * B        ⟨⟨standard convolution using FFTs⟩⟩
  for i ← 0 to n
      if C[i] > 0
          C[i] ← 1
  return C[0..n]
```

This subroutine runs in $O(T \log T)$ time, so the overall algorithm now runs in $\boldsymbol{O(T \log T \log k)}$ ***time***. ■

Rubric: 7 points = 4 for naive dynamic programming algorithm (2 for smaller array + 1 for evaluation order + 1 for running time) + 3 for convolution speedup

(c) **Extra credit:** Describe an efficient algorithm to compute the smallest number of coins with total value T.

You may assume that there is at least one collection of coins with total value T. Ideally, your algorithm should have exactly the same big-Oh running time as your algorithm for part (b), where now k is the output value (which is *not* necessarily a power of 2).

Solution: One immediate solution is a simple linear search for the smallest k such that $Change(T,k) = \textsc{True}$, using the algorithm from part (b) at each iteration. The linear search requires k iterations, so this algorithm runs in $\boldsymbol{O(Tk \log T \log k)}$ ***time***.

The first improvement idea is to replace the linear search with a binary search. We can't do this directly using the function $Change(T,k)$, because that function is not monotone in k; instead, we use the following closely related function

> $Change_{\leq}(T,k) = \textsc{True}$ if some collection of ***at most*** k coins has total value T, and $\textsc{False}$ otherwise.

The function $Change_{\leq}$ satisfies a nearly identical recurrence to our original function $Change$; the only difference is in the simplest base case!

$$Change_{\leq}(T,k) := \begin{cases} \textbf{(}T = 0\textbf{) or } (T = Coin[i] \text{ for some } i) & \text{if } k = 1 \\ \displaystyle\bigvee_{S=0}^{T} \big(Change_{\leq}(S,k/2) \wedge Change_{\leq}(T-S,k/2)\big) & \text{otherwise} \end{cases}$$

Adapting our algorithm from part (b) to compute $Change_{\leq}(T,k)$ in $O(T \log T \log k)$ time is straightforward. Now we can find the smallest k such that $Change_{\leq}(T,k) = \textsc{True}$ using binary search over the range $1 \leq k \leq T$, invoking our modified decision algorithm at each iteration. Our binary search requires $O(\log T)$ iterations, so the resulting algorithm runs in $\boldsymbol{O(T \log^2 T \log k)}$ ***time***.

We can reduce the number of iterations from $O(\log T)$ to $O(\log k)$ using *exponential search*. We first find the smallest integer ℓ such that $Change_{\leq}(T, 2^{\ell}) = \textsc{True}$. Because $2^{\ell-1} < k \leq 2^{\ell}$, we consider only $\lceil \log_2 k \rceil$ different values of ℓ. Then we perform a binary search within the range $2^{\ell-1} < k \leq 2^{\ell}$ for the smallest integer k such that $Change_{\leq}(T,k) = \textsc{True}$, again considering only $O(\log k)$ different values of k. Altogether, we invoke our decision algorithm only $O(\log k)$ times, not $O(\log L)$, so the overall algorithm runs in $\boldsymbol{O(T \log T \log^2 k)}$ ***time***.

Finally, we can reuse information computed in earlier invocations of the decision algorithm in later invocations. We use a simple generalization of the recurrence from part (a):

$$Change_{\leq}(T, x+y) := \bigvee_{S=0}^{T} \big(Change_{\leq}(S,x) \wedge Change_{\leq}(T-S,y)\big)$$

Specifically:

- In the exponential phase, we can compute $Change_{\leq}(\cdot, 2^{\ell})$ from $Change_{\leq}(\cdot, 2^{\ell-1})$ using a single convolution in $O(T \log T)$ time.
- In each iteration of the binary search, the length of the active search interval $lo < k \leq hi$ is always a power of 2. Suppose $hi - lo = 2^s$, so the midpoint of the interval is $mid = lo + 2^{s-1}$. We computed $Change_{\leq}(\cdot, 2^{s-1})$ during the exponential phase, and we computed $Change_{\leq}(\cdot, lo)$ in an earlier iteration of the binary search, so we can compute $Change_{\leq}(\cdot, mid)$ using a single convolution in $O(T \log T)$ time.

Our overall algorithm uses only $O(\log k)$ convolutions, and therefore runs in $\boldsymbol{O(T \log T \log k)}$ ***time***, matching our algorithm from part (b). ■

This algorithm was described by Timothy Chan and Qizheng He [1], who were inspired by a CS 473 homework problem posed by Sariel Har-Peled [3]. (I think Qizheng was a student in Sariel's 473.) A similar idea was published earlier by Oliver Serang [4]. Chan and He describe even faster algorithms that solve the decision problem in $O(T \log T)$ worst-case time and the minimum-coins problem in $O(T \log T \log\log T)$ worst-case time and $O(T \log T)$ expected time; their followup paper [2] describes several further extensions.

[1] Timothy Chan and Qizheng He. On the change-making problem. *Proc. 3rd SIAM Symposium on Simplicity in Algorithms*, 38-42, 2020.

[2] Timothy Chan and Qizheng He. More on change-making and related problems. *J. Computer and System Sciences* 124:159-169, 2022.

[3] Sariel Har-Peled. Absolutely not subset sum. CS 473 homework 3 problem 2, Fall 2018.

[4] Oliver Serang. The probabilistic convolution tree: Efficient exact Bayesian inference for faster LC-MS/MS protein inference. *PLOS ONE* 9(3):e91507, 2014.