Gizmos Consider the following problem. You wish to purchase (at least) \( n \) identical gizmos. Gizmos come in packages of different sizes and different prices. You can buy any number of packages of each size, as long as the total number is at least \( n \). You wish to find the minimum total price of such a set of packages.

The input is given as \( n \) and an array \( \text{Packages}[1..m] \), where each \( \text{Package}[i] \) has a positive integer field \( \text{Package}[i].\text{size} \) and a positive real field \( \text{Package}[i].\text{price} \) giving the number of gizmos in the package and the price of the package.

A recursive algorithm to solve this problem is:

BestPrice[\( n \); positive integer, \( \text{Packages}[1..m] \) : array of pairs (\( \text{size} \): integer, \( \text{price} \): real)].

1. \( \text{MinPrice} \leftarrow \infty; \)
2. For \( d = 1 \) to \( m \) do:
3. begin:
4. IF \( \text{Packages}[d].\text{size} \geq n \) THEN \( \text{TempPrice} \leftarrow \text{Packages}[d].\text{price} \)
5. ELSE \( \text{TempPrice} \leftarrow \text{Packages}[d].\text{price} + \text{BestPrice}(n – \text{Packages}[d].\text{size}, \text{Packages}); \)
6. IF \( \text{TempPrice} < \text{MinPrice} \) THEN \( \text{MinPrice} \leftarrow \text{TempPrice}; \)
7. end;
8. Return \( \text{MinPrice} \).

**Part 1: 2 points** Show the recursion tree of the above algorithm on the following input: \( n = 6 \), packages: buy 5 for \$12, 3 for \$8 or 2 for \$6.

Case 1: Buy package of 5. Cost: 12+BestPrice(1,\text{Packages})
Case 1a: Buy package of 5 \( \notin \) 1. Cost:12
Case 1b: Buy package of 3 \( \notin \) 1. Cost:8
Case 1c: Buy package of 2 \( \notin \) 1. Cost:6

Case 2: Buy package of 3. Cost: 8+BestPrice(3,\text{Packages})
Case 2a: Buy package of 5 \( \notin \) 3. Cost:12
Case 2b: Buy package of 3 =3. Cost:8
Case 2c: Buy package of 2. Cost:6+BestPrice(1,\text{Packages})
Case 2cI: Buy package of 5 \( \notin \) 1. Cost:12
Case 2cII: Buy package of 3 > 1. Cost:8
Case 2cIII: Buy package of 2 > 1. Cost:6
Case 2 returns 8+8 =16
Case 3: Buy package of 2. Cost: 6+BestPrice(4,Packages)
Case 3a: Buy package of 5 > 4. Cost:12
Case 3b: Buy package of 3 > 1. Cost:8+BestPrice(1,Packages)
Case 3bI: Buy package of 5 > 1. Cost:12
Case 3bII: Buy package of 3 > 1. Cost:8
Case 3bIII: Buy package of 2 > 1. Cost:6
Case 3c: Buy package of 2. Cost:6+BestPrice(2,Packages)
Case 3cI: Buy package of 5 > 2. Cost:12
Case 3cII: Buy package of 3 > 2. Cost:8
Case 3cIII: Buy package of 2 =2. Cost:6
Minimum=6. Case 3b returns 6+6 =12.
Case 3 minimum is 12. Case 3 returns 6+12 =18.
Overall minimum is Case 2, 16, which is returned by the main procedure.

Part 2: 3 points Give a bound on the worst-case number of recursive calls the recursive algorithm could make in terms of $n$ and $m$.
There are at most $m$ recursive calls, and each reduces the value of $n$ by at least 1. This gives a tree of fan-out $m$ and depth at most $n$, so a total of $O(m^n)$ recursive calls.
Assume all packages have distinct sizes. (If not, we could just delete all but the least expensive package of a given size.) Then we make in terms of $n$, $T(n) = T(n - size_1) + T(n - size_2) + ... + T(n - size_m) \leq T(n-1) + T(n-2) + ...$. We can use induction to prove $T(n) \in O(2^n)$.
(See last answer key.)

Part 3: 10 points Give a dynamic programming version of the recurrence.
Note that only the value of $n$, not the set of Packages, changes in recursive calls, and that takes on values from 1...$n$. Also, $n$ decreases in each recursive call. So we should fill in an array of one dimension of size $n$, in increasing order of $n$. This leads to:
DPBestPrice[n : positiveInteger, Packages[1..m] : array of pairs (size: integer, price: real)]
1. Initialize $MP[1..n]$.
2. FOR $N = 1$ to $n$ do:
3. \( \text{MinPrice} \leftarrow \text{inf}; \)
4. FOR \( d = 1 \) to \( m \) do:
5. IF \( \text{Packages}[d].\text{size} \geq N \) THEN \( \text{TempPrice} \leftarrow \text{Packages}[d].\text{price} \)
6. ELSE \( \text{TempPrice} \leftarrow \text{Packages}[d].\text{price} + MP[N - \text{Packages}[d].\text{size}]; \)
7. IF \( \text{TempPrice} < \text{MinPrice} \) THEN \( \text{MinPrice} \leftarrow \text{TempPrice}; \)
8. \( \text{MP}[N] \leftarrow \text{MinPrice}. \)
9. Return \( \text{MP}[n] \).

**Part 4: 3 points** Give a time analysis of this dynamic programming algorithm, in terms of \( n \) and \( m \).

There are two nested loops, one going from 1 to \( n \), the other from 1 to \( m \), which gives a total time of \( O(nm) \).

**Part 5: 2 points** Show the array that your algorithm produces on the above example.

1. \( 6 = \min(6, 8, 12) \)
2. \( 6 = \min(6, 8, 12) \)
3. \( 8 = \min(6+\text{MP}[1]=12, 8, 12) \)
4. \( 12 = \min(6+\text{MP}[2]=12, 8+\text{MP}[1]=14, 12) \)
5. \( 12 = \min(6+\text{MP}[3]=14, 8+\text{MP}[2]=14, 12) \)
6. \( 16 = \min(6+\text{MP}[4]=18, 8+\text{MP}[3]=16, 12+\text{MP}[1]=18) \)

For each of the following three problems, describe the fastest dynamic programming algorithm you can find, and give a time analysis (in terms on any of the given parameters).

**One Dimensional Clustering: 20 pts** You are given \( n \) real numbers \( r_1, r_2, ..., r_n \) and an integer \( 1 \leq k \leq n \). You want to find \( k \) disjoint intervals \( I_1 = [a_1, b_1], I_2 = [a_2, b_2], ..., I_k = [a_k, b_k] \) so that each \( r_i \in I_j \) for some \( j \), in a way that minimizes the sum of the squares of the length of the intervals, \( \sum_{j=1}^{k} (b_j - a_j)^2 \).

We can give a recursion based on the question, where does the first cluster end? Let’s assume the numbers are sorted from smallest to largest. (If not, we sort them in \( O(n \log n) \) time using mergesort.)

\( \text{BTClusters}[R[1..n], k] \).

1. IF \( k \geq n \) return 0.
2. IF \( k = 1 \) return \( (R[n] - R[1])^2 \). {Only choice is cluster \( (R[1], R[n]) \)}
3. Best \( \leftarrow \infty \)
4. FOR \( I = 1 \) to \( n \) do:
5. \[ \text{ThisCost} \leftarrow (R[I] - R[1])^2 + BTClusters[R'[1..n], k-1] \]
   \{Case when first cluster is \((R[1], R[I]).\} \}

6. \[ \text{IF ThisCost} < \text{Best} \text{ THEN } \text{Best} \leftarrow \text{ThisCost} \]

7. Return \text{Best}.

The sub-problems we solve are of the form \(BTClusters[R'[J..n], K]\) where
\(1 \leq J \leq n\) and \(1 \leq K \leq k\). So we will use a \(n \times k\) matrix \(C[I, K]\) to hold \(BTClusters[R'[J..n], K]\). \(K\) decreases as we make recursive calls so we fill this matrix in increasing value of \(K\).

DPClusters\([R[1..n], k]\).

1. Initialize \(C[1..n, 1..k]\).
2. FOR \(J = 1\) to \(n\) do:
3. \(C[J, 1] \leftarrow (R[n] - R[J])^2\)
4. FOR \(K = 2\) to \(k\) do:
5. FOR \(I = 1\) to \(n - K + 1\) do:
6. \(\text{Best} \leftarrow \infty\)
7. FOR \(I = J\) to \(n\) do:
8. \[ \text{ThisCost} \leftarrow (R[I] - R[J])^2 + C[J + 1, K - 1] \]
   \{Case when first cluster is \((R[J], R[I]).\} \}
9. \[ \text{IF ThisCost} < \text{Best} \text{ THEN } \text{Best} \leftarrow \text{ThisCost} \]
10. \(C[J, K] \leftarrow \text{Best}.\)
11. Return \(C[1, k]\).

The above algorithm takes time \(O(n^2k)\) for the three nested loops.

**Library storage** A library has \(n\) books that must be stored in alphabetical order on adjustable height shelves. Each book has a height and a thickness. The width of the shelf is fixed at \(W\), and the sum of the thicknesses of books on a single shelf must be at most \(W\). The next shelf will be placed on top, at a height equal to the maximum height of a book in the shelf. Give an algorithm that minimizes the total height of shelves used to store all the books. You are given the list of books in alphabetical order, \(b_i = (h_i, t_i)\), where \(h_i\) is the height and \(t_i\) is the thickness, and must organize the books in that order.

The better algorithm for that problem branched on, "What is the first word on the next line?". The equivalent is "What is the first book on the next shelf?"

A back-tracking approach computing the smallest height based on this idea gives:
1. BTShelves[b[1...n]]: array of books, W: real; real;
2. IF n = 0 return 0;
3. IF n = 1 return h_1;
4. BestHeight ← inf
5. CurrentShelfsHeight ← h_1;
6. CurrentShelfThickness ← t_1
7. J ← 2;
8. Until J > n or CurrentShelfThickness > W do:
9. begin;
10. A ← BTShelves(b[J..n], W);{If we start a new shelf at b_J, how high
     will future shelving take us?}
11. IF A + CurrentShelfsHeight < BestHeight THEN BestHeight ←
     A + CurrentShelfsHeight;
12. CurrentShelfThickness ← CurrentShelfsThick + t_J
13. CurrentShelfsHeight ← max(CurrentShelfsHeight, h_J)
15. end;
16. Return BestHeight;

Note that all the sub-problems are to solve the same problem for books
J...n. This gives a dp solution:

1. DPShelves[b[1..n], W]
2. Initialize Shortest[1...n+1];
3. Shortest[n + 1] ← 0.
4. Shortest[n] ← h_n;
5. FOR I = n − 1 TO 1 do:
6. begin;
7. BestHeight ← inf
8. CurrentShelfsHeight ← h_I;
9. CurrentShelfThickness ← t_I
10. J ← I + 1;
11. Until J > n or CurrentShelfThickness > W do:
12. begin;
13. A ← Shortest(J);{If we start a new shelf at b_J, how high will future
     shelving take us?}
14. IF $A + \text{CurrentShelfsHeight} < \text{BestHeight}$ THEN $\text{BestHeight} \leftarrow \text{CurrentShelfsHeight}$;
15. $\text{CurrentShelfsThickness} \leftarrow \text{CurrentShelfsThickness} + t_j$
16. $\text{CurrentShelfsHeight} \leftarrow \max(\text{CurrentShelfsHeight}, h_j)$
17. $J \leftarrow J + 1.$
18. end;
19. $\text{Shortest}(I) \leftarrow \text{Best}$;
20. end;

In the worst-case, this could take $O(n^2)$, because of the two nested loops.

**Protein Bonding : 20 pts** Let $\Sigma$ be a finite set of amino acids, and let $w = w_1...w_n$ be a sequence of acids from $\Sigma$. For $\sigma, \sigma' \in \Sigma$, let $b(\sigma, \sigma')$ be the strength of a bond between the two types of acids, a non-negative real number. A bonding of the sequence is a partial matching between positions in the word so that matched pairs can be connected with lines drawn below the word without lines crossing. Equivalently, it should satisfy: there are no two bonded pairs $i_1, j_1$ and $i_2, j_2$ with $i_1 \leq i_2 \leq j_1 \leq j_2$. The total bond strength is the sum over all bonded positions $i, j$ of the bond strength $b(w_i, w_j)$. Give as efficient as possible algorithm to find the bonding of a proteine sequence that maximizes the total bond strength. (We know an $O(n^3)$ algorithm.)

Identify a bonding with the set of pairs of positions matched, each pair listed in increasing order. Let $B$ be a bonding of $w_1...w_n$. If $B$ does not match position 1 with anything, it is also a bonding on positions 2...$n$. Conversely, any bonding on positions 2...$n$ is also a bonding on 1...$n$

If $B$ matches position 1 with position $i$, then every match $(j, k)$ in $B$ must either have $2 \leq j < k \leq i - 1$ or $i + 1 \leq j < k \leq n$, since if $j < i < k$, we have $1 \leq j < i < k$, which violates our constraints. Let $B_{\text{inside}}$ be the matching on 2..$i - 1$ of edges in $B$ of the first type, and $B_{\text{outside}}$ of the second type. $B_{\text{inside}}$ is a bonding on 2..$i - 1$, and $B_{\text{outside}}$ is a bonding on $i + 1..n$, and $\text{Totalstrength}(B) = \text{Totalstrength}(B_{\text{inside}}) + \text{Totalstrength}(B_{\text{outside}}) + \text{strength}(w_1, w_i)$. Conversely, let $B_I$ be any bonding on 2..$i - 1$, and $B_O$ any bonding on $i + 1..n$. Then $B = (1, i) \cup B_O \cup B_I$ is a bonding on 1..$i$ of total strength $\text{Totalstrength}(B_O) + \text{Totalstrength}(B_I) + \text{strength}(w_1, w_i)$.

This gives us the following recursive backtracking algorithm:

$TS(w_1...w_n \in \Sigma^n$, strength[$\Sigma$][$\Sigma$]: array of non-negative real numbers): non-negative real number.

1. IF $n \leq 1$ return 0; \{need two positions to bond\}
2. \textit{Non - bond} \leftarrow TS(w_2...w_n, strength).
3. \textit{Best} \leftarrow \textit{Non - bond}.
4. FOR \( i = 2 \) to \( n \) do:
5. \hspace{1em} \text{begin;}
6. \hspace{2em} \textit{CurrentCase} \leftarrow TS(w_2...w_{i-1}, strength)+TS(w_{i+1}...w_n)+strength(w_1,w_i)
7. \hspace{2em} \text{IF} \textit{Best} < \textit{CurrentCase} \text{ THEN} \textit{Best} \leftarrow \textit{CurrentCase}.
8. \hspace{1em} \text{end;}
9. \text{Return} \textit{Best}.

Then we note that this recursive procedure never changes \textit{strength} and only calls itself on consecutive subsequences \( w_i...w_j \) of the original sequence. The value of \( i \) is always greater in a sub-call, so bottom-up can be: in decreasing order of \( i \). This allows us to create the following dp version of the above recursion: \textit{DPTS}(w_1...w_n \in \Sigma^n, strength|\Sigma|\Sigma;: array of non-negative real numbers): non-negative real number.

1. Initialize array \( total[1..n][1..n] \) to 0’s. {includes base case}
2. FOR \( I = n - 1 \) to 1 do :
3. \hspace{1em} FOR \( J = I + 1 \) to \( n \) do:
4. \hspace{2em} \text{begin;}{\text{compute strength for} \ w_I...w_J} \}
5. \hspace{2em} \textit{Non - bond} \leftarrow total(I + 1, J)
6. \hspace{2em} \textit{Best} \leftarrow \textit{Non - bond}.
7. \hspace{2em} FOR \( K = I + 1 \) to \( J \) do:
8. \hspace{3em} \text{begin;}
9. \hspace{4em} \textit{Curr} \leftarrow total(I + 1, K - 1) + total(K + 1, J) + strength(w_I,w_K)
10. \hspace{4em} \text{IF} \textit{Best} < \textit{Curr} \text{ THEN} \textit{Best} \leftarrow \textit{Curr}.
11. \hspace{3em} \text{end;}
12. \hspace{2em} total[I,J] \leftarrow \textit{Best}
13. \hspace{2em} \text{end;}
14. \text{Return} \textit{total}[I][n].

The total time is \( O(n^3) \) since we have three nested loops, each with up to \( n \) iterations.

**Implementation:** Implement both the memoized and dynamic programming version of the longest common subsequence problem. Time them on random strings of length \( n \) for an alphabet with four symbols (e.g., A, G, C and T), with a wide variety of lengths \( n \). (Plot time and \( n \) on a log-log
scale.) Compare their two performances, and give an explanation for any differences.

Although the two will both scale the same, the dynamic programming should usually have a much better constant, due to better locality of reference and no overhead for recursion.