Gizmos (20 points) 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:

\[
\begin{align*}
\text{BestPrice}[n : \text{positiveinteger}, \text{Packages}[1..m] : \text{array of pairs (size: integer, price: real)}] & : \\
1. \quad \text{MinPrice} \leftarrow \text{inf}; \\
2. \quad \text{For } d = 1 \text{ to } m \text{ do:} \\
3. \quad \text{begin}; \\
4. \quad \quad \text{IF } \text{Packages}[d].\text{size} \geq n \text{ THEN } \text{TempPrice} \leftarrow \text{Packages}[d].\text{price} \\
5. \quad \quad \quad \text{ELSE } \text{TempPrice} \leftarrow \text{Packages}[d].\text{price} + \\
6. \quad \quad \quad \quad \text{BestPrice}(n - \text{Packages}[d].\text{size}, \text{Packages}); \\
7. \quad \quad \text{IF } \text{TempPrice} < \text{MinPrice} \text{ THEN } \text{MinPrice} \leftarrow \text{TempPrice}; \\
8. \quad \text{end}; \\
9. \text{Return } \text{MinPrice}. \\
\end{align*}
\]

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

Case 2: Buy package of 3. Cost: 8+BestPrice(3, Packages)
Case 2a: Buy package of 5 \( \hat{\mu} 3 \). Cost:12
Case 2b: Buy package of 3 =3. Cost:8
Case 2c: Buy package of 2. Cost:6+BestPrice(1, Packages)
Case 2cI: Buy package of 5 \( \hat{\mu} 1 \). Cost:12
Case 2cII: Buy package of 3 \( \hat{\mu} 1 \). Cost:8
Case 2cIII: Buy package of 2 \(i\). 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 \(i\). 4. Cost: 12
Case 3b: Buy package of 3 \(i\). Cost: 8+BestPrice(1, Packages)
Case 3bI: Buy package of 5 \(i\). 1. Cost: 12
Case 3bII: Buy package of 3 \(i\). 1. Cost: 8
Case 3bIII: Buy package of 2 \(i\). 1. Cost: 6

Case 3c: Buy package of 2. Cost: 6+BestPrice(2, Packages)
Case 3cI: Buy package of 5 \(i\). 2. Cost: 12
Case 3cII: Buy package of 3 \(i\). 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:

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

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 $\text{TempPrice} \leftarrow \text{Packages}[d].\text{price} + \text{MP}[N - \text{Packages}[d].\text{size}]$;
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).

**Non-consecutive sums** Given an array of integers, $A[1..n]$, find the subset $S$ of positions so that no two consecutive numbers $i$ and $i + 1$ are both in $S$, which maximizes $\sum_{i \in S} A[i]$.

A recursive algorithm could be based on deciding whether or not to include $A[1]$. It would look like:

1. $\text{BTMNCS}[A[1..n]]$
2. IF $n = 0$ return 0
3. IF $n = 1$ return $\max(0, A[1])$

We note that all of the recursive calls will be to subarrays of the form $A[I..n]$, $1 \leq I \leq n + 1$, where we use $I = n + 1$ to denote the empty array. Thus, we can use a matrix $\text{MNCS}[I]$ to store the answer to $\text{BTMNCS}[A[I..n]]$ to convert to dynamic programming. Since in the recursion, $I$ increases, we fill in this matrix in decreasing order of $I$. The base cases of an array of size 0 or 1 correspond to $I = n + 1$ and $I = n$.

This yields the following DP algorithm:
1. DPMNCS[A[1..n]]
2. Initialize MNCS[1..n + 1].
3. MNCS[n + 1] ← 0
4. MNCS[n] ← max(0, A[n]).
5. FOR I = n − 1 downto 1 do:
   6. MNCS[I] ← max(MNCS[I + 1], A[I] + MNCS[I + 2])
7. Return MNCS[1].

The inside of the loop is constant time, so this algorithm solves the problem in O(n) time.

Interleaving Consider two binary strings x₁, x₂, ..., xₙ and y₁, y₂, ..., yₘ. An interleaving of x and y is a string z₁...zₙ₊ₘ so that the bit positions of z can be partitioned into two disjoint sets X and Y, so that looking only at the positions in X, the sub-sequence of z produced is x and looking only at the positions of Y, the sub-sequence is y. For example, if x = 1010 and y = 0011, z = 10001101 is an interleaving because the odd positions of z form x, and the even positions form y. The problem is: given x, y and z, determine whether z is an interleaving of x and y.

We first give a recursive algorithm. z₁ must either be copied from x₁ or y₁. If it matches neither, we can terminate early. If it matches both, we need to consider both cases.

1. BTI[x₁..xₙ, y₁..yₘ, z₁...zₙ₊ₘ]
2. IF n = 0 THEN IF y₁..yₘ = z₁...zₘ return True, ELSE return False.
3. IF m = 0 THEN IF z₁..xₙ = z₁..zₙ return True, ELSE return False.
4. Return False.

Note that in this recursive calls, we always have a suffix of x, a suffix of y and a suffix of z. This would give three parameters, but note that the suffix of z is always of length equal to the sum of the two other input lengths. So the third parameter is determined by the other two. We’ll use Inter[I, J] to store whether zᵢ₊ⱼ₋₁...zₙ₊ₘ is an intertwine of xᵢ..xₙ and yⱼ..yₘ. As usual we use I = n + 1 to mean the first word is empty, and J = m + 1 for the second word being empty.

1. DPI[x₁..xₙ, y₁..yₘ, z₁..zₙ₊ₘ]
2. Initialize Inter[1..n + 1, 1..m + 1], an array of Booleans
3. FOR J = 1 to n + 1 do:
   4. IF yⱼ..yₘ = zᵢ₊ⱼ..zₙ₊ₘ THEN Inter[n + 1, J] ← True
      ELSE Inter[n + 1, J] ← False
6. FOR $I = 1$ to $n$ do:
7.  IF $x_I \ldots x_n = z_{m+I} \ldots z_{n+m}$ THEN $\text{Inter}[n+1, J] \leftarrow \text{True}$
8.  ELSE $\text{Inter}[n+1, J] \leftarrow \text{False}$
9. FOR $I = n$ downto 1 do: FOR $J = m$ downto 1 do:
10. $\text{Inter}[I, J] \leftarrow \text{False}$
11. IF $x_I = z_{J+I-1}$ AND $\text{Inter}[I+1, J]$ THEN $\text{Inter}[I, J] \leftarrow \text{True}$
12. IF $y_J = z_{J+I-1}$ AND $\text{Inter}[I, J+1]$ THEN $\text{Inter}[I, J] \leftarrow \text{True}$
13. Return $\text{Inter}[1, 1]$

The array is $nm$ in size, and each entry takes constant time, so the algorithms time is $O(nm)$.

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. $\text{BTShelves}[b[1..n]]$: array of books, $W$: real]:real;
2. IF $n = 0$ return 0;
3. IF $n = 1$ return $h_1$;
4. $\text{BestHeight} \leftarrow \text{inf}$
5. $\text{CurrentShelfHeight} \leftarrow h_1$;
6. $\text{CurrentShelfThickness} \leftarrow t_1$
7. $J \leftarrow 2$;
8. Until $J > n$ or $\text{CurrentShelfThickness} > W$ do:
9. begin;
10. $A \leftarrow \text{BTShelves}(b[J..n], W)$:{If we start a new shelf at $b_J$, how high will future shelving take us?}
11. IF $A + CurrentShelvesHeight < BestHeight$ THEN $BestHeight \leftarrow A + CurrentShelvesHeight$;

12. $CurrentShelvesThickness \leftarrow CurrentShelvesThick + t_J$

13. $CurrentShelvesHeight \leftarrow \max(CurrentShelvesHeight, h_J)$


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. $Shortest[n + 1] \leftarrow 0.$

4. $Shortest[n] \leftarrow h_n$;

5. FOR $I = n - 1$ TO 1 do:

6. begin;

7. $BestHeight \leftarrow \inf$

8. $CurrentShelvesHeight \leftarrow h_I$;

9. $CurrentShelfThickness \leftarrow t_I$

10. $J \leftarrow I + 1$;

11. Until $J > n$ or $CurrentShelfThickness > W$ do:

12. begin;

13. $A \leftarrow Shortest(J);$ {If we start a new shelf at $b_J$, how high will future shelving take us?}

14. IF $A + CurrentShelvesHeight < BestHeight$ THEN $BestHeight \leftarrow CurrentShelvesHeight$;

15. $CurrentShelvesThickness \leftarrow CurrentShelvesThick + t_I$

16. $CurrentShelvesHeight \leftarrow \max(CurrentShelvesHeight, h_J)$

17. $J \leftarrow J + 1.$

18. end;

19. $Shortest(I) \leftarrow Best$;

20. end;


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