Chapter 6– Solutions

6.1. *Subproblems:* Define an array of subproblems $D(i)$ for $0 \leq i \leq n$. $D(i)$ will be the largest sum of a (possibly empty) contiguous subsequence ending exactly at position i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$ and update the $D(i)$'s in ascending order according to the rule:

$$D(i) = \max\{0, D(i-1) + a_i\}$$

The largest sum is then given by the maximum element $D(i)^*$ in the array D. The contiguous subsequence of maximum sum will terminate at i^*. Its beginning will be at the first index $j \leq i^*$ such that $D(j-1) = 0$, as this implies that extending the sequence before j will only decrease its sum.

Correctness: The contiguous subsequence of largest sum ending at i will either be empty or contain a_i. In the first case, the value of the sum will be 0. In the second case, it will be the sum of a_i and the best sum we can get ending at $i-1$, i.e. $D(i-1) + a_i$. Because we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

Running Time: The running time for this algorithm is $O(n)$, as we have n subproblems and the solution of each can be computed in constant time. Moreover, the identification of the optimal subsequence only requires a single $O(n)$ time pass through the array D.

6.2. *Subproblems:* Define subproblem $D(i)$ for $0 \leq i \leq n$ to be the minimum total penalty to get to hotel i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$ and compute the remaining $D(i)$ in ascending order using the recursion:

$$D(i) = \min_{0 \leq j < i} \{(200 - a_j)^2 + D(j)\}$$

To recover the optimal itinerary, we can keep track of a maximizing j for each $D(i)$ and use this information to backtrack from $D(n)$.

Correctness: To solve $D(i)$ we consider all possible hotels j we can stay at on the night before reaching hotel i: for each of these possibilities, the minimum penalty to reach i is the sum of the cost of a one-day trip from j to i and the minimum penalty necessary to reach j. Because we are interested in the minimum penalty to reach i, we take the minimum of these values over all j's.

Running Time: The running time is $O(n^2)$, as we have n subproblems and each takes time $O(n)$ to solve, as we need to compute the minimum of $O(n)$ values. Moreover, backtracking only takes time $O(n)$.

6.3. *Subproblems:* Define subproblem $D(i)$ to be the maximum profit which Yuckdonald's can obtain from locations 1 to i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$, and use the following rule to solve the other subproblems:

$$D(i) = \max\{D(i-1), p_i + D(i^*)\}$$

where i^* is the largest index j such that $m_j \leq m_i - k$, i.e. the first location preceding i and at least k miles apart from it.

Correctness: Note that, if location i is not used, the maximum profit $D(i)$ must equal $D(i-1)$. Otherwise, if location i is used, the best we can hope for is the sum of p_i and the maximum profit from the remaining locations we are still allowed to open before i, i.e. $D(i^*)$.

Running Time: This algorithm solves n subproblems; each subproblem requires finding an index i^*, which can be done in time $O(\log n)$ by binary search on the ordered list of locations, and computing a maximum of two values, which can be done in constant time. Hence, the running time is $O(n \log n)$.

6.4. a) *Subproblems:* Define an array of subproblems $S(i)$ for $0 \leq i \leq n$ where $S(i)$ is 1 if $s[1 \cdots i]$ is a sequence of valid words and is 0 otherwise.

Algorithm and Recursion: It is sufficient to initialize $S(0) = 1$ and update the values $S(i)$ in ascending order according to the recursion

$$S(i) = \max_{0 \leq j < i} \{S(j) : \texttt{dict}(s[j+1 \cdots i]) = \texttt{true}\}$$

Then, the string s can be reconstructed as a sequence of valid words if and only if $S(n) = 1$.

Correctness and Running Time: Consider $s[1 \cdots i]$. If it is a sequence of valid words, there is a last word $s[j \cdots i]$, which is valid, and such that $S(j) = 1$ and the update will cause $S(i)$ to be set to 1. Otherwise, for any valid word $S[j \cdots i]$, $S(j)$ must be 0 and $S(i)$ will also be set to 0. This runs in time $O(n^2)$ as there are n subproblems, each of which takes time $O(n)$ to be updated with the solution obtained from smaller subproblems.

b) Every time a $S(i)$ is updated to 1 keep track of the previous item $S(j)$ which caused the update of $S(i)$ because $s[j+1 \cdots i]$ was a valid word. At termination, if $S(n) = 1$, trace back the series of updates to recover the partition in words. This only adds a constant amount of work at each subproblem and a $O(n)$ time pass over the array at the end. Hence, the running time remains $O(n^2)$.

6.5. a) There are 8 possible patterns: the empty pattern, the 4 patterns which each have exactly one pebble, and the 3 patterns that have exactly two pebbles (on the first and fourth squares, the first and third squares, and the second and fourth squares).

b) Number the 8 patterns 1 through 8, and define $S \subseteq \{1, 2, \ldots, 8\} \times \{1, 2, \ldots, 8\}$ to be all (a, b) such that pattern a is compatible with pattern b. For each pattern, there are a constant number of patterns that are compatible (for example, every pattern is compatible with the empty pattern).

Sub-problems and Recursion: We consider the sub-problem $L[i, j]$, $i = 0, 1, 2, \ldots, n$ and $j \in \{1, 2, \ldots, 8\}$ to be the maximal value achievable by pebbling columns $1, 2, \ldots, i$ such that the final column has pattern j. It is easy to see that:

$$L[i+1, j] = \max_{(k,j) \in S} L[i, k]$$

The base case is $L[0, j] = 0$ for all j. In order to recover the optimal placement, we should also maintain a back-pointer: $P[i+1, j]$ is the value of k such that $(k, j) \in S$ and $L[i, k]$ is maximal.

Algorithm and Running Time: For $i = 0, 1, \ldots, n$, for $j = 1, 2, \ldots, 8$, compute $L[i, j]$ and $P[i, j]$ using the recurrence. Note that $L[i, j]$ and $P[i, j]$ can be computed using the recursion in constant time since we only need to check a constant number of possible k. The value of the optimal placement is given by $\max_j L[n, j]$.

To recover the optimal placement, let j^* be the value of j for which $L[n, j]$ is maximal. Then, column n should be pebbled using pattern j^*. Then, column $n-1$ should be pebbled using pattern $P[n, j^*]$, column $n-2$ with pattern $P[n-1, P[n, j^*]]$, and so on.

The running time of this algorithm is $O(n)$ because compute the recurrence takes $O(n)$ time and backtracking takes $O(1)$ time per column.

6.6. *Subproblems:* Let $x_1 \cdots x_n$ be the factors of the product in the order they appear. We define $C(i, j)$ to be the set of all possible values of the substring $x[i, \cdots, j]$ under all possible different parenthesizations.

Algorithm and Recursion: We can compute $C(i, i+s)$ by considering all positions j where we can break the expression into two products:

$$C(i, i+s) = \bigcup_{i \leq j < i+s} \{xy : x \in C(i, j), y \in C(j+1, i+s)\}$$

We can then recursively construct all $C(i, i+s)$ in increasing order of s.

Correctness and Running Time: The assignment to $C(i, i+s)$ is correct as it considers all possible ways of splitting $x_i \cdots x_{i+s}$ after x_j into two parenthesized expressions. The running time of the algorithm is $O(n^3)$ as there are $O(n^2)$ subproblems and each takes time $O(n)$.

6.7. *Subproblems:* Define variables $L(i, j)$ for all $1 \leq i \leq j \leq n$ so that, in the course of the algorithm, each $L(i, j)$ is assigned the length of the longest palindromic subsequence of string $x[i, \cdots, j]$.

Algorithm and Recursion: The recursion will then be:

$$L(i, j) = \max\{L(i+1, j), L(i, j-1), L(i+1, j-1) + \text{equal}(x_i, x_j)\}$$

where $\text{equal}(a, b)$ is 1 if a and b are the same character and is 0 otherwise, The initialization is the following:

$$\begin{aligned} \forall i, 1 \leq i \leq n\ , &\quad L(i, i) = 0 \\ \forall i, 1 \leq i \leq n-1\ , &\quad L(i, i+1) = \text{equal}(x_i, x_i + 1) \end{aligned}$$

Correctness and Running Time: Consider the longest palindromic subsequence s of $x[i, \cdots, j]$ and focus on the elements x_i and x_j. There are then three possible cases:

- If both x_i and x_j are in s then they must be equal and $L(i, j) = L(i+1, j-1) + \text{equal}(x_i, x_j)$
- If x_i is not a part of s, then $L(i, j) = L(i+1, j)$.
- If x_j is not a part of s, then $L(i, j) = L(i, j-1)$.

Hence, the recursion handles all possible cases correctly. The running time of this algorithm is $O(n^2)$, as there are $O(n^2)$ subproblems and each takes $O(1)$ time to evaluate according to our recursion.

6.8. *Subproblems:* For $1 \leq i \leq n$ and $1 \leq j \leq m$, define subproblem $L(i, j)$ to be the length of the longest common substring of x and y *terminating* at x_i and y_j. The recursion is:

$$L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{if equal}(x_i, y_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

The initialization is, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$\begin{aligned} L(0, 0) &= 0 \\ L(i, 0) &= 0 \\ L(0, j) &= 0 \end{aligned}$$

The output of the algorithm is the maximum of $L(i, j)$ over all $1 \leq i \leq n$ and $1 \leq j \leq m$.

Correctness and Running Time: The initialization is clearly correct. Hence, it suffices to prove the correctness of the recursion. The longest common substring terminating at x_i and y_j must include x_i and y_j: hence, it will be 0 if these characters are different and $L(i-1, j-1) + 1$ if they are equal. The running time is $O(mn)$ as we have mn subproblems and each takes constant time to evaluate through the recursion.

6.9. *Subproblems and Recursion:* Let the string be indexed $x_1 \cdots x_n$. Let $0 = c_0 \leq c_1 \leq c_2 \leq \cdots \leq c_m \leq c_{m+1} = n$ be the locations of the $m+2$ cuts, which include the beginning and end of the string(i.e. c_i is the index after which the ith cut takes place). Let us the define the following subproblems for all i, j such that $0 \leq i < j \leq m+1$:

$$L(i,j) = \text{minimum cost of breaking } x[c_i+1, \cdots, c_j] \text{ at positions } c_{i+1}, \cdots, c_{j-1}$$

Notice then that the solution to our problem will be given by the value $L(0,n)$.

Algorithm and Recursion: Initialize the dynamic program by setting $L(i,i+1) = 0$ for all i, $0 \leq i \leq n$. This is correct as the string $x[c_i+1, \cdots, c_{i+1}]$ needs no further cutting. The recursion to evaluate the values $L(i,j)$ is then, for $j > i+1$:

$$L(i,j) = \min_{i<k<j} \{L(i,k) + L(k,j) + (c_j - c_i)\}$$

Correctness and Running Time: The recursion is correct as it is equivalent to selecting the first cut k to make to minimize the cost of breaking up $x[c_i+1, \cdots, c_j]$. The running time is $O(n^3)$, as we have $O(n^2)$ subproblems, each of which takes $O(n)$ time to solve.

6.10. One way to solve this problem is by dynamic programming.
Subproblems: Define $L(i,j)$ to be the probability of obtaining exactly j heads amongst the first i coin tosses.

Algorithm and Recursion: By the definition of L and the independence of the tosses, it is clear that:

$$L(i,j) = p_i L(i-1,j-1) + (1-p_i) L(i-1,j) \quad j = 0, 1, \ldots, i$$

We can then compute all $L(i,j)$ by initializing $L(0,0) = 1$, $L(i,j) = 0$ for $j < 0$, and proceeding incrementally (in the order $i = 1, 2, \ldots, n$, with inner loop $j = 0, 1, \ldots, i$). The final answer is given by $L(n,k)$.

Correctness and Running Time: The recursion is correct as we can get j heads in i coin tosses either by obtaining $j-1$ heads in the first $i-1$ coin tosses and throwing a head on the last coin, which takes place with probability $p_i L(i-1,j-1)$, or by having already j heads after $i-1$ tosses and throwing a tail last, which has probability $(1-p_i)L(i-1,j)$. Besides, these two events are disjoint, so the sum of their probabilities equals $L(i,j)$. Finally, computing each subproblem takes constant time, so the algorithm runs in $O(n^2)$ time.

Another approach is to observe that the probability is exactly the coefficient of x^k in the polynomial $g(x) = \prod_{i=1}^{n}(p_i x + (1-p_i))$. We can compute $g(x)$ using divide-and-conquer in $O(n \log^2 n)$ time by recursively computing $g_1(x) = \prod_{i=1}^{n/2}(p_i x + (1-p_i))$, $g_2(x) = \prod_{i=n/2+1}^{n}(p_i x + (1-p_i))$ and then using FFT to calculate $g(x) = g_1(x) g_2(x)$.

6.11. *Subproblems:* Let us the define the following subproblems for all i, j such that $1 \leq i \leq n$, $1 \leq j \leq m$:

$$L(i,j) = \text{length of longest common subsequence between } x[1, \cdots, i] \text{ and } y[1, \cdots, j]$$

Then, the solution to the original problem will be given by $L(n,m)$.

Algorithm and Recursion: Initialize the dynamic program by setting $L(i,0) = 0$ and $L(0,j) = 0$ for all i, j. The recursion to compute the values $L(i,j)$ correctly is the following:

$$L(i,j) = \max\{L(i-1,j), L(i,j-1), L(i-1,j-1) + \text{equal}(x_i, y_j)\}$$

where $\text{equal}(x,y)$ is 1 if x and y are the same character and is 0 otherwise.

Correctness and Running Time: The recursion is correct as the alignment producing the longest common subsequence for $L(i, j)$ will have in its last position either a deletion, two characters matched (either equal or different characters) or an insertion. The running time is $O(mn)$ as there are mn subproblems, each of which takes $O(1)$ time.

6.12. Let $A(i, j)$ be the minimum cost triangulation of the polygon P_{ij} spanned by vertices $i, i+1, \ldots, j$. In the optimum triangulation of P_{ij}, the edge (i, j) is covered by some triangle, say (i, k, j). Then the optimum triangulation of P_{ij} splits it into the triangle (i, k, j) and the optimum triangulations of the polygons P_{ik} and P_{kj}. That is, for $i < j$,

$$A(i, j) = \min_{i<k<j} \{A(i, k) + A(k, j) + d(i, k) + d(k, j)\},$$

where $d(\cdot, \cdot)$ are the distances between vertices. The base cases are $A(i, i+1) = 0$, for all i. Since each computation $A(i, j)$ can be done in $O(n)$ time and there are $O(n^2)$ pairs (i, j), the running time of the algorithm is $O(n^3)$. To recover the best triangulation, it is sufficient to keep track of a choice of k which is maximizing for each $A(i, j)$ above. Then, starting at $A(1, n)$, we can backtrack to identify the diagonals included in the triangulation.

6.13. a) Consider the sequence $(2, 9, 1, 1)$. Then, the best available card at the start has value 2, but this leads only to a score of 3, as the opponent picks the card of value 9 in the next turn. Instead, choosing the 1 at the start yields a final score of 10.

b) *Subproblems:* Define subproblems $A(i, j)$ for $i \leq j$, where $A(i, j)$ is the difference between the largest total the first player can obtain and the corresponding score of the second player when playing on sequence $s_i, s_{i+1}, \cdots, s_j$. This will be positive if and only if the first player has the larger total.

Algorithms and Recursion: We can solve all the subproblem by initializing $A(i, i) = s_i$ for all i and using the update rule for $i < j$:

$$A(i, j) = \max\{s_i - A(i+1, j), s_j - A(i, j-1)\}$$

We can also keep track of the optimal move at each stage by simultaneously maintaining a matrix $M(i, j)$, where $M(i, j)$ will be set to `first` if $A(i, j)$ takes the value of the first element in the maximization, and to `last` otherwise.

Correctness and Running Time: The recursion is correct, as at any stage of the game there are two possible moves for the first player: either choose the first card, in which case he will gain s_i and score $-A(i+1, j)$ in the rest of the game, or the last card, gaining s_j and $-A(i, j-1)$ from the remaining cards. The algorithm computes all the subproblems and the entries of M in time $O(n^2)$ as each update takes time $O(1)$. Moreover, the player can then just read off the matrix M to learn the best strategy at any point of the game.

6.14. *Subproblems:* Define XY subproblems. For $1 \leq i \leq X$ and $1 \leq j \leq Y$, let $C(i, j)$ be the best return that can be obtained from a cloth of shape $i \times j$. Define also a function `rect` as follows:

$$\texttt{rect}(i, j) = \begin{cases} \max_k c_k \text{ for all products } k \text{ with } a_k = i \text{ and } b_k = j \\ 0 \text{ if no such product exists} \end{cases}$$

Algorithm and Recursion: Then the recursion is:

$$C(i, j) = \max\{\max_{1 \leq k < i} \{C(k, j) + C(i-k, j)\}, \max_{1 \leq h < j} \{C(i, h) + C(i, j-h)\}, \texttt{rect}(i, j)\}$$

It remains to initialize the smallest subproblems correctly:

$$C(1, j) = \max\{0, \texttt{rect}(1, j)\}$$
$$C(i, 1) = \max\{0, \texttt{rect}(i, 1)\}$$

The final solution is then the value of $C(X, Y)$.

Correctness and Running Time: For proving correctness, notice that $C(i, j)$ trivially has the intended meaning for the base cases with $i = 1$ or $j = 1$. Inductively, $C(i, j)$ is solved correctly, as a rectangle $i \times j$ can only be cut in the $(i-1)+(j-1)$ ways considered by the recursion or be occupied completely by a product, which is accounted for by the $\texttt{rect}(i, j)$ term. The running time is $O(XY(X+Y+n))$ as there are XY subproblems and each takes $O(X+Y+n)$ to evaluate.

6.15. One way to solve this problem is by dynamic programming.
Subproblems: We define subproblem $A(i, j)$ for $1 \leq i, j \leq n$ to represent the probability that A is the first to win n games, given that after $i+j$ games A has won i.

Algorithm and Recursion: We can initialize A by setting $A(n, j) = 1$ for $j \neq n$ and $A(i, n) = 0$ for all i. The other subproblems can be solved incrementally in decreasing order of $i+j$ using the recursion:

$$A(i, j) = \frac{1}{2}\left(A(i, j+1) + A(i+1, j)\right)$$

Correctness and Running Time: The recursion is correct as we can compute $A(i, j)$ by conditioning on the outcome of the $(i+j+1)$th game. Both outcomes take place with probability $\frac{1}{2}$. If A wins, then it wins the game with probability $A(i+1, j)$. If B wins, A has a probability of winning of $A(i, j+1)$. If we are interested in finding solutions for all subproblems, we need to solve $O(n^2)$ subproblems, each taking $O(1)$ for a total running time of $O(n^2)$. If we are only after the $A(i, j)$ subproblem, we can stop once we solved $O((n-(i+j))^2)$ subproblems.

6.16. For a subset of garage sales $S \subseteq \{g_1, \cdots, g_n\}$ and $g_j \in S$, let $C(S, j)$ be the profit of the most profitable path starting at home, visiting only garage sales in S and ending at g_j. To express $C(S, j)$ in terms of smaller subproblems, consider the laste garage sale $g_i \in S - \{g_j\}$ visited before g_j. The path profit up to g_i is $C(S - \{g_j\}, i)$, whilst the marginal profit of visiting g_j is $p_j - d_{ij}$. We must pick i as to maximize the profit, so we have:

$$C(S, j) = \max_{g_i \in S : i \neq j} \{C(S - \{g_j\}, i) + p_j - d_{ij}\}$$

We can then compute solutions by initializing $C(\{g_i\}, i)$ to be $p_i + d_{0i}$ for all $1 \leq i \leq n$ and solving the subproblems in increasing order of $|S|$. The final solution is the minimum over all S,$g_j \in S$ of $C(S^*, j^*) + d_{j0}$, where we also include the cost of returning home. Moreover, we can also keep record, for every $C(S, j)$ of which garage g_i yielded a maximum in the recursion; this g_i is the last garage visited before getting to g_j in a best path for $C(S, j)$. We can then use this information to backtrack from $C(S*, j*)$ to reconstruct the entire optimal path. This algorithm solves $O(n2^n)$ subproblems, each in $O(n)$ time, so it has total running time $O(n^2 2^n)$.

6.17. This problem reduces to Knapsack with repetitions. The total capacity is v and there is an item i of value x_i and weight x_i for each coin denomination. It is possible to make change for value v if and only if the maximum value we can fit in v is v. The running time is $O(nv)$.

6.18. Use the same reduction as in 6.17, but reduce to Knapsack without repetition. The running time is $O(nv)$.

6.19. This is similar to 6.17 and 6.18. The problem reduces to Knapsack without repetition with a capacity of v, but this time we have k items of value x_i and weight x_i for each coin denomination x_i, i.e. a total of kn items. The running time is $O(nkv)$.

6.20. Let $S(i, j)$ be the cost of cheapest tree formed by words i to j, for $1 \leq i, j \leq n$. Also, initialize $S(i, j)$ to 0 if $i > j$. Then $S(i, j)$ will be the minimum cost of the tree over all choices of word k, $i \leq k \leq j$, to

place at the root. If word k is at the root, the cost of the left subtree will be $S(i,k-1)$ and the cost of right subtree will be $S(k+1,j)$. Moreover, all words will need to pay one comparison at the root node, so the total cost of the tree will be $\sum_{t=i}^{j} p_t + S(i,k-1) + S(k+1,j)$. Hence:

$$S(i,j) = \min_{i \le k \le j} \{\sum_{t=i}^{j} p_t + S(i,k-1) + S(k+1,j)\}$$

Finally, the cost of the optimal tree will be $S(1,n)$. To reconstruct the tree, it suffices to keep track of which root k minimized the expression in the recursion for each subproblem and backtrack from $S(1,n)$.

6.21. The subproblem $V(u)$ will be defined to be the size of the minimum vertex cover for the subtree rooted at node u. We have $V(u) = 0$ if u is a leaf, as the subtree rooted at u has no edges to cover. The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes. Hence, for any internal node i

$$V(i) = \min \left\{ \sum_{j:(i,j)\in E} \left(1 + \sum_{k:(j,k)\in E} V(k) \right), 1 + \sum_{j:(i,j)\in E} V(j) \right\}$$

The algorithm can then solve all the subproblems in order of decreasing depth in the tree and output $V(n)$. The running time is linear in n because while calculating $V(i)$ for all i we look at most at $2 * |E| = O(n)$ edges in total.

6.22. This problem reduces to Knapsack without repetition. The knapsack will have capacity t and each number a_i will be an item of value a_i and weight a_i. A subset adding up to t will exist if and only if a total value of t can fin in the knapsack. The dynamic programming algorithm then solves the problem in time $O(n \sum_i a_i)$.

6.23. We define $P(i,b)$ to be the maximum probability that the system works correctly up to its ith stage when adding redundancy to stages 1 to i and using budget b. We can initialize $P(0,b) = 1$ for all b. To compute $P(i,b)$, we consider the number of machines m_i we want to add at stage i. If we are adding m_i machines, our success probability will be the product of $P(i-1, b-m_i c_i)$, the best we can do on the previous stages with the remaining budget, and $(1-(1-r_i)^{m_i})$, the probability of stage i working correctly. More formally, we can update $P(i,b)$ for $1 \le i \le n$ and $0 \le b \le B$ using the recursion:

$$P(i,b) = \max_{0 \le m_i \le \left\lfloor \frac{b}{c_i} \right\rfloor} \{P(i-1, b-m_i c_i) \times (1-(1-r_i)^{m_i})\}$$

We can then solve all subproblems in ascending order of i by looping over all b's before proceeding to the next i. $P(n,B)$ will be success probability associated with the best allocation of redundancy. To recover the actual number of machines added at each stage for this optimal allocation, we need to keep track for every $P(i,b)$ of which value of m_i yielded the maximum value. We can then start at $P(n,B)$ and backtrack to find the optimal allocation. We have nB subproblems, each taking time at most $O(\max_i \frac{B}{c_i})$ for a total running time of $O(nB^2 \max_i \frac{1}{c_i})$.

6.24. Assume $m = O(n)$.

a) Consider the usual matrix of subproblems $E(i,j)$. If we update the values column by column, at every point we only need the current column and the previous column to perform all calculations. Hence, if we are just interested in the final value $E(m,n)$ we may keep only two columns at every time, using space $O(n)$. Note that we are not able now to have a pointer structure to recover the optimal alignment, as we would need pointers for all subproblems, which would take space mn.

b) Together with the subproblem solution $L(i,j)$, for each $j \geq m/2$, in each of the active two columns, maintain a pointer to the index k at which the optimal path leading to (i,j) crossed the $m/2$ column. Such pointer can be easily updated at every recursion by copying the pointer of the subproblem from which the optimal solution is derived.

c) Consider the space requirement first. At any time during the running of this scheme, a single dynamic programming data structure is active, taking up space $O(n)$. All that is left to store is the values k for all the subproblems on which we recurse. These are at most m as every values corresponds to an index of x matched to one of y in the minimum edit distance alignment.. Hence, the total space required is $O(n)$. For the running time analyisis, notice that level $i+1$ of the recursion takes half the time of level i. Hence, the total running time will be bounded above by $O(mn)(1+\frac{1}{2}+\frac{1}{4}+\cdots) = 2O(mn)$.

6.25. Let $M[i,s_1,s_2]$ be 1 if there are two disjoint subsets $I,J \subseteq \{1,\dots i\}$ such that $\sum_{j\in I} a_j = s_1$ and $\sum_{j\in J} a_j = s_2$. An instance of 3-Partition has positive solution if and only if $M[n,S/3,S/3] = 1$. We can use dynamic programming to fill out the three dimensional table specified by $M[i,s_1,s_2]$ for $1 \leq i \leq n$, $0 \leq s_1, s_2 \leq S/3$. An arbitrary entry in the table can be calculated from three previous entries:

$$M[i,s_1,s_2] = M[i-1,s_1-a_i,s_2] \vee M[i-1,s_1,s_2-a_i] \vee M[i-1,s_1,s_2].$$

In words, this definition says that if there exist two disjoint subsets $I,J \subseteq \{1,\dots,i\}$ such that $\sum_{j\in I} a_j = s_1$ and $\sum_{j\in J} a_j = s_2$, then either $i \in I, i \notin J$ or $i \in J, i \notin I$ or $i \notin I,J$. In the case that $i \in I$, $\sum_{j\in I, j\neq i} = s_1 - a_i$ and $\sum_{j\in J} = s_2$ (since I and J are disjoint), and so $M[i-1,s_1-a_i,s_2] = 1$. The case of $i \in J$ is symmetric. In the case that $i \notin I,J$, then $I,J \subseteq \{1,\dots,i-1\}$, and so $M[i-1,s_1,s_2] = 1$.

The base cases for the recursion are $M[i,0,0] = 1$ for all $i \geq 0$ and $M[i,s_1,s_2] = 0$ if $i < 0$, $s_1 < 0$ or $s_2 < 0$. The table size is $n \times S/3 \times S/3$. Filling in a single entry takes constant time, so the total running time for the algorithm is $O(nS^2)$. (One could also notice that the table is symmetric because $M[i,s_1,s_2] = M[i,s_2,s_1]$ but this symmetry only gives a constant factor improvement in the running time.)

6.26. This is a simple variant of the edit distance algorithm defined in class. The recursion is modified to:

$$E(i,j) = \max\{E(i-1,j) + \delta(x_i,-), E(i-1,j-1) + \delta(x_i,y_j), E(i,j-1) + \delta(-,y_j)\}$$

The initialization has also to be modified to deal properly with the new scoring for gaps. We have, for $i,j > 0$:

$$\begin{aligned} E(0,0) &= 0 \\ E(i,0) &= E(i-1,0) + \delta(x_i,-) \\ E(0,j) &= E(0,j-1) + \delta(-,y_j) \end{aligned}$$

The correctness follows by the same argument as for the edit distance algorithm. The running time is again $O(mn)$.

6.27. This is another modification of the standard edit distance algorithm, only more advanced. You should notice that there is no simple way to modify the existing recursion to incorporate the new kind of gap penalties. In particular, it would be necessary at every point to know whether the previous subproblem solutions were given by alignments with a deletion or insertion in their last position. Moreover, it might be that an optimal alignment for $E(i,j)$ is not an extension of the optimal for $E(i-1,j), E(i-1,j-1), E(i,j-1)$ as an alignment with smaller previous score but terminating with a gap might have a smaller gap penalty and beat all extensions of optimal alignments. This suggests that we should not keep a single matrix of subproblems, but 3. E will be the matrix of subproblems

where the alignments are constrained to be a substitution or a match in their last position. E_x will be the matrix of subproblems over the alignments which have a gap in the last position of string x. E_y is defined similarly for y. Given these definitions, we just need to work out the recursion correctly case by case.

$$\begin{aligned}
E(i,j) &= \max\{E(i-1,j-1), E_x(i-1,j-1), E_y(i-1,j-1)\} + \delta(x_i, y_j) \\
E_x(i,j) &= \max\{E(i-1,j) - c_0 - c_1, E_x(i-1,j) - c_1, E_y(i-1,j) - c_0 - c_1\} \\
E_y(x,y) &= \min\{E(i,j-1) - c_0 - c_1, E_x(i,j-1) - c_0 - c_1, E_y(i,j-1) - c_1\}
\end{aligned}$$

The output will just be the maximum of $E(m,n), E_x(m,n)$ and $E_y(m,n)$. This takes $O(mn)$ as we still have $O(mn)$ subproblems, each evaluated in constant time.

6.28. This only requires a simple modification to 6.26. $E(i,j)$ becomes the score of the best scoring substring of $x[1,\cdots,i]$ and $y[1,\cdots,j]$. The recursion is then modified to take into account the possibility that the best local alignment be empty:

$$E(i,j) = \max\{E(i-1,j) + \delta(x_i,-), E(i-1,j-1) + \delta(x_i,y_j), E(i,j-1) + \delta(-,y_j), 0\}$$

6.29. For $i \in \{1,\cdots,n\}$, let $W(i)$ be the the weight of the best subset of consistent partial matches in $x[1,\cdots,i]$. To compute $W(i)$, we consider the two following cases:

- the best subset of partial matches contains a match j with $r_j = i$,
- the best subset of partial matches does not contain such j

In the first case, $W(i)$ will be the sum of w_j and the weight of the best match on the remaining of the string, i.e. $W(l_j - 1)$. In the second case, we will just have $W(i) = W(i-1)$. This shows that the following recursion is correct:

$$W(i) = \max\{W(i-1), \max_{j:r_j=i}\{W(l_j - 1) + w_j\}\}$$

The algorithm will then proceed computing $W(i)$ in ascending order of i and will output $W(n)$ as best total weight achievable. To reconstruct the actual sequence of partial matches, it suffices to keep track, for all $W(i)$ of which j maximizes the expression in the recursion, when the second maximum is the larger. We can then follow these pointers from $W(n)$ backwards to identify the optimal alignment. The running time is $O(n+m)$, where m is the number of partial matches, as we have n subproblems and each partial match is considered once in the maximizations.

6.30. a) See Figure 1.

b) It suffices to give an algorithm that minimizes the score for length 1 sequences, i.e. single characters. We can then apply this algorithm to all positions. The final sequence assignment will have score equal to the sum of the scores of the trees for each position and, as there are all minimum, it will also be minimum score.

The dynamic programming algorithm works by solving the problem in each subtree. For each internal node u, we define $S(u)$ to be the set of the labels $s(u)$ of u over all optimal assignments to u's subtree.

Claim: Let p be an internal node and let c_1 and c_2 be its children. If $S(c_1) \cap S(c_2)$ is non-empty, then $S(p) = S(c_1) \cap S(c_2)$. Otherwise, $S(p) = S(c_1) \cup S(c_2)$.
Proof: Let $C(u)$ be the minimum cost of a labeling of the subtree rooted at u. Consider the two cases:

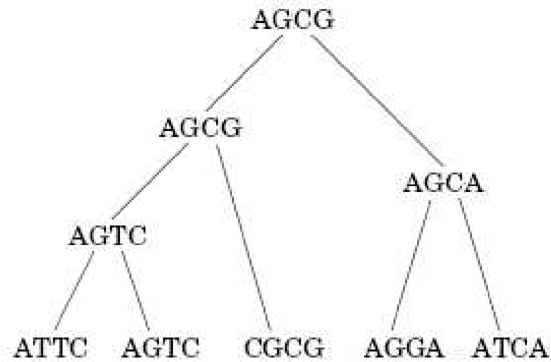

Figure 1: Parsimony tree for 6.30 a)

1) $I = S(c_1) \cap S(c_2)$ is non-empty. Then, any assignment with $s(p) \in I$ will have cost $C(c_1) + C(c_2)$, which is minimum for the subtree rooted at p. Moreover, any assignment with $s(p) \notin I$ will pay at least $C(c_1) + C(C_2) + 1$ and will not be optimal. Hence, $S(p) = I$.
2) If $I = S(c_1) \cap S(c_2)$ is empty, consider the union $U = S(c_1) \cup S(c_2)$. For any $s(p) \in U$, it is possible to construct an assignment of cost $C(c_1) + C(c_2) + 1$ by choosing a optimal assignment corresponding to $s(p)$ for one child's subtree and any assignment for the other. For $s(p) \notin U$, we pay $1 + C(c_i)$ for each child i for whose subtree we use an optimal labeling and at least the same quantity $1 + C(c_i)$ for each child i for which we do not use an optimal labeling. Hence, we pay at least $2 + C(c_1) + C(c_2)$. This shows that the optimal assignments for the subtree rooted at p are those consisting of $s(p) \in U$ and optimal assignments for the children's subtrees.

We can then initialize $S(v) = \{s(v)\}$ for the leaves and proceed up the tree, updating $S(p)$ to be the intersection of $S(c_1)$ and $S(c_2)$, if this is non-empty, and their union otherwise. Once, we have reached the root and calculated $S(r)$ we can produce an actual labeling by setting $s(r)$ to any label in $S(r)$. The label for all other internal nodes u can then be calculated from the label $s(p)$ of u's parent by taking $s(u) = s(p)$ if $s(p) \in S(u)$ and $s(u)$ to be any label in $S(u)$ otherwise. For each position of the string, we have $O(n)$ subproblems $S(u)$, each of which takes time $O(|\Sigma|)$ to update. The total running time is then $O(|\Sigma|nk)$.