1. The *simple knapsack problem* is:

   Input: Positive integers $w_1, \ldots, w_n, W$.

   Output: A subset $S \subseteq \{1, \ldots, n\}$ such that $K = \sum_{i \in S} w_i$ is as large as possible subject to the constraint $K \leqslant W$.

   We are to give a greedy algorithm which solves the simple knapsack problem for the case that each weight w_i is a power of 2, and prove that the algorithm is correct.

   Algorithm:

   ```
   KNAPSACK(w_1,...,w_n, W):
       Sort the weights in non-increasing order (so w_1 ⩾ w_2 ⩾ ... ⩾ w_n).
       S ← ∅    # S is the current set of weight indices in the knapsack
       K ← 0    # K is the current total weight in the knapsack
       for i = 1,...,n:   # loop invariant: S can be extended to an optimum solution
           if w_i + K ⩽ W:
               S ← S ∪ {i}
               K ← K + w_i
       return (K, S)
   ```

   Running Time: Sorting takes time $\Theta(n \log n)$ in the worst-case; the main loop takes time $\Theta(n)$ in the worst-case; total worst-case time is $\Theta(n \log n)$.

   Correctness: To prove the algorithm is correct, it suffices to show that the final value of K is the largest possible total weight that can be placed in the knapsack with $K \leqslant W$, and $K = \sum_{i \in S} w_i$ for the final value of S.

   Let $S_0 = \varnothing$ and $K_0 = 0$ (the values of S and K before the first execution of the for loop), and for $1 \leqslant i \leqslant n$ let S_i be the value of S and K_i be the value of K after i iterations of the for loop.

   Thus $K_n = K$ and $S_n = S$, where K and S are the outputs of the algorithm.

   By an easy induction on i we have $K_i \leqslant W$ and $K_i = \sum_{j \in S_i} w_j$. Let (K^*, S^*) be an optimum solution. Thus $K^* \leqslant W$, and $\sum_{i \in S^*} w_i = K^*$ and for all $T \subseteq \{1, \ldots, n\}$, if $\sum_{i \in T} w_i \leqslant W$ then $\sum_{i \in T} w_i \leqslant K^*$. We must show $K^* = K_n$, where K_n is the final value of K in the algorithm.

   Claim: For $0 \leqslant i \leqslant n$, there exists $OPT_i \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in OPT_i} w_j = K^*$ and $OPT_i \cap \{1, \ldots, i\} = S_i$.

   In other words, the Claim says that each set S_i can be extended to an optimum solution OPT_i by adding some elements from $\{i+1, \ldots, n\}$.

   Note that the correctness of the algorithm follows from the Claim when $i = n$, since $OPT_n = S_n \cap \{1, \ldots, n\} = S_n = S$, and $K^* = K_n = K$.

   To prove the Claim we need the following Lemma.

   Lemma: Let $w_1, \ldots, w_m$ be powers of 2, and let $k \in \mathbb{N}$ be such that $w_i \leqslant 2^k$ for $1 \leqslant i \leqslant m$. Suppose $\sum_i w_i \geqslant 2^k$. Then there is a subset $S \subseteq \{1, \ldots, m\}$ such that $\sum_{i \in S} w_i = 2^k$.

   We prove the Lemma below.

   But first, we prove the Claim by induction on i, using the Lemma.

   The base case is $i = 0$. Let $OPT_0 = S^*$, where (K^*, S^*) is the optimum solution mentioned above. Then $OPT_0 \cap \{\} = \varnothing = S_0$.

   For the induction step $i \to i+1$, let OPT_i be an optimum solution that extends S_i (i.e., $\sum_{j \in OPT_i} w_j = K^*$ and $OPT_i \cap \{1, \ldots, i\} = S_i$). There are several cases to consider.

   Case 1: $i + 1 \notin S_{i+1}$.

Then by the algorithm we see that $w_{i+1} + K_i > W$, so $i+1 \notin OPT_i$, since otherwise (by the induction hypothesis) we would have:

$$\sum_{j \in OPT_i} w_j \geqslant \sum_{j \in (OPT_i \cap \{1,\ldots,i,i+1\})} w_j = \left(\sum_{j \in S_i} w_j \right) + w_{i+1} = K_i + w_{i+1} > W.$$

Thus we can let $OPT_{i+1} = OPT_i$.
Case 2: $i+1 \in S_{i+1}$.
Subcase 2A: $i+1 \in OPT_i$.
Then let $OPT_{i+1} = OPT_i$.
Subcase 2B: $i+1 \notin OPT_i$.
Then $w_{i+1} = 2^k$ for some k. Apply the Lemma for this k, and the set of weights $\{w_{i+2},\ldots,w_n\}$. Note that $w_{i+2} + \cdots + w_n \geqslant w_{i+1} = 2^k$, since otherwise the output solution K_n of the algorithm would exceed $\sum_{j \in OPT_i} w_j$, which contradicts our assumption that OPT_i is optimum. Therefore by the Lemma, there is $S \subseteq \{i+2,\ldots,n\}$ such that $\sum_{j \in S} = 2^k$.
Let $OPT_{i+1} = OPT_i \cup \{i+1\} - S$. Then $\sum_{j \in OPT_{i+1}} w_j = \sum_{j \in OPT_i} w_j = K^*$, and $OPT_{i+1} \cap \{1,\ldots,i+1\} = (OPT_i \cap \{1,\ldots,i\}) \cup \{i+1\} = S_{i+1}$.
This completes the proof of the Claim.
Proof of the Lemma: We use induction on k. The base case is $k=0$: then each $w_i = 2^0 = 1$ and $\sum_i w_i \geqslant 1$. Let $S = \{1\}$ so that $\sum_{i \in S} w_i = w_1 = 1$.
The induction step is $k \to k+1$. Assume that whenever $\sum_i w_i \geqslant 2^k$ for $w_1,\ldots,w_m \leqslant 2^k$, there is some $S \subseteq \{1,\ldots,m\}$ such that $\sum_{i \in S} w_i = 2^k$.
Assume $w_i \leqslant 2^{k+1}$ for $1 \leqslant i \leqslant m$, and

$$\sum_{i=1}^{m} w_i \geqslant 2^{k+1}. \tag{1}$$

Case I: If there exists $j \leqslant m$ such that $w_j = 2^{k+1}$, let $S = \{j\}$. Then $\sum_{i \in S} w_i = w_j = 2^{k+1}$.
Case II: If $w_i \leqslant 2^k$ for $1 \leqslant i \leqslant m$, then by the Induction Hypothesis there exists $S_1 \subseteq \{1,\ldots,m\}$ such that

$$\sum_{i \in S_1} w_i = 2^k. \tag{2}$$

Let $S_2 = \{1,\ldots,m\} - S_1$. Then

$$\sum_{i \in S_2} w_i = \sum_{i=1}^{m} w_i - \sum_{i \in S_1} w_i \geqslant 2^{k+1} - 2^k = 2^k$$

so by the Induction Hypothesis and (1) and (2) and the Case II assumption, there exists $S_3 \subseteq S_2$ such that

$$\sum_{i \in S_3} w_i = 2^k \tag{3}$$

Let $S = S_1 \cup S_3$ and note that this is a union of disjoint sets. Hence by (2) and (3) we conclude

$$\sum_{i \in S} w_i = \sum_{i \in S_1} w_i + \sum_{i \in S_3} w_i = 2^k + 2^k = 2^{k+1}.$$

This completes the proof of the Lemma and hence the correctness of the algorithm.

2. The minimum heavyweight spanning tree problem is:

Input: A connected undirected graph $G = (V, E)$ and a weight function $w : E \to \mathbb{N}$.

Output: A spanning tree T_ℓ for G with the property that *every* spanning tree T for G has some edge e' with $w(e') \geqslant w(e)$, for every edge $e \in T_\ell$.

(a) We are to give an efficient greedy algorithm solving this problem, analyze its running time, and prove that our algorithm is correct.

We will show the following:

Claim: Every minimum spanning tree is also a minimum heavyweight spanning tree (the latter is usually called a *minimum bottleneck spanning tree*).

It follows from the Claim that every algorithm that we've studied for finding minimum spanning trees also finds a minimum heavyweight spanning tree. Thus it suffices to prove the above Claim.

We will prove the contrapositive (which is equivalent):

> (*) If (V, T) is not a minimum heavyweight spanning tree for $G = (V, E)$ then it is not a minimum spanning tree.

Suppose that (V, T_h) is a spanning tree for the graph $G = (V, E)$ with weight function $w : E \to \mathbb{N}$. Let e_h be a maximum weight edge in T_h, so $w(e_h) \geqslant w(e)$ for every edge $e \in T_h$.

Assume that T_h is not a minimum heavyweight spanning tree for G. Then there is a spanning tree T' for G such that $w(e') < w(e_h)$ for every $e' \in T'$. We will use the following variation on the Exchange Lemma to show that T_h is not a minimum spanning tree for G, thus completing the proof of (*) and hence of this question.

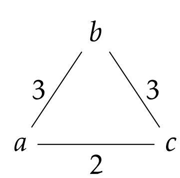
Lemma: If T and T' are spanning trees for G, then for every edge e in $T - T'$ there is an edge e' in $T' - T$ such that $T'' = T \cup \{e'\} - \{e\}$ is a spanning tree for G.

We apply the Lemma with $T = T_h$ and $e = e_h$ and T' as in the paragraph preceding the lemma. Let $e' \in (T' - T_h)$ be the edge stated to exist in the lemma. Then $w(e') < w(e_h)$ so $w(T'') < w(T)$, so T is not a minimum spanning tree for G, as required.

Proof of the Lemma:

Let $e = (u, v)$. Then the graph $T_e = (V, T - e)$ is a "cut" for T: the vertices V of T_e are partitioned into two connected components V_u and V_v, where V_u is the set of nodes connected to u in $T - e$ and V_v is the set of nodes connected to v in $T - e$. Since (V, T') is connected, T' must contain an edge e' which connects T_u and T_v. By assumption $e \notin T'$, so $e' \neq e$. Thus $T \cup \{e'\} - \{e\}$ is connected, and has the same number of edges as T, and hence it is a spanning tree for G.

(b) The answer is **no**: there are minimum heavyweight spanning trees which are not minimum spanning trees. For example,



Let $T = \{(a, b), (b, c)\}$ and $T' = \{(a, b), (a, c)\}$. Then T is a minimum heavyweight spanning tree, with heaviest weight 3 and total weight 6, and T' is also a minimum heavyweight spanning tree with heaviest weight 3, but it has total weight 5. Hence T is a minimum heavyweight spanning tree but not a minimum spanning tree.

3. We are to write an efficient algorithm that takes as inputs two strings $x, y \in \{A, C, G, T\}^*$ along with a $[5 \times 5]$ scoring matrix δ (with $\delta(-,-) = -\infty$), and that returns the highest-scoring alignment between x and y.

Step 0: Recursive structure.

Let $x = x_1 \cdots x_m$ and $y = y_1 \cdots y_n$ where each x_i and each y_i is in $\{A, C, G, T\}$ and $m \geqslant 0$ and $n \geqslant 0$ (but not both m and n are 0).

Consider an optimum alignment of x and y. Either x_m is aligned with y_n, or x_m is aligned with a gap, or y_n is aligned with a gap. In every case, the alignment for the rest of x and y must have the maximum possible score—else it would be possible to increase the total score of the current optimum alignment.

Step 1: Array definition.

Let $C[k, \ell]$ be the score of the highest scoring alignment between the initial segments $x_1 \cdots x_k$ and $y_1 \cdots y_\ell$, where $0 \leqslant k \leqslant m$ and $0 \leqslant \ell \leqslant n$.

The highest score among all possible alignments of x and y is given by $C[m, n]$.

Step 2: Recurrence relation.

$$C[k,\ell] = \begin{cases} 0 & \text{if } k = 0 \text{ and } \ell = 0, \\ \delta(x_k, -) + C[k-1, \ell] & \text{if } k > 0 \text{ and } \ell = 0, \\ \delta(-, y_\ell) + C[k, \ell-1] & \text{if } k = 0 \text{ and } \ell > 0, \\ \max\Big(\delta(x_k, -) + C[k-1, \ell], & \\ \qquad \delta(-, y_\ell) + C[k, \ell-1], & \\ \qquad \delta(x_k, y_\ell) + C[k-1, \ell-1]\Big) & \text{if } k > 0 \text{ and } \ell > 0, \end{cases}$$

for all $0 \leqslant k \leqslant m$ and $0 \leqslant \ell \leqslant n$, based on the recursive structure of optimum solutions discussed above.

Step 3: Iterative algorithm.

$\text{SCORE}(x = x_1 \cdots x_m, y = y_1 \cdots y_n, \delta)$:
$\quad C[0,0] \leftarrow 0$
$\quad$ **for** $k = 1, \ldots, m$:
$\qquad C[k,0] \leftarrow \delta(x_k, -) + C[k-1, 0]$
$\quad$ **for** $\ell = 1, \ldots, n$:
$\qquad C[0,\ell] \leftarrow \delta(-, y_\ell) + C[0, \ell-1]$
$\qquad$ **for** $k = 1, \ldots, m$:
$\qquad\quad C[k,\ell] \leftarrow \max\Big(\delta(x_k, -) + C[k-1, \ell],$
$\qquad\qquad\qquad\qquad \delta(-, y_\ell) + C[k, \ell-1],$
$\qquad\qquad\qquad\qquad \delta(x_k, y_\ell) + C[k-1, \ell-1]\Big)$

This computes the values of $C[k, \ell]$ directly from the recurrence above and runs in worst-case time $\Theta((n+1)(m+1)) = \Theta(nm)$.

Step 4: Optimum solution.

Once we have computed the array values $C[k, \ell]$, we can use them to print an optimum alignment as follows: starting at $C[m, n]$, test the current value against all three possibilities in the recurrence relation to determine the best alignment for the last character(s) of x and y, until both sequences are completely aligned.

```
ALIGN(x = x_1 ··· x_m, y = y_1 ··· y_n, δ, C):
    A = []   # current alignment, stored as a list of pairs
    (k, ℓ) ← (m, n)
    while k > 0 and ℓ > 0:
        if C[k, ℓ] = δ(x_k, −) + C[k − 1, ℓ]:
            A ← [(x_k, −)] + A
            k ← k − 1
        elif C[k, ℓ] = δ(−, y_ℓ) + C[k, ℓ − 1]:
            A ← [(−, y_ℓ)] + A
            ℓ ← ℓ − 1
        else:
            A ← [(x_k, y_ℓ)] + A
            k ← k − 1
            ℓ ← ℓ − 1
    while k > 0:
        A ← [(x_k, −)] + A
        k ← k − 1
    while ℓ > 0:
        A ← [(−, y_ℓ)] + A
        ℓ ← ℓ − 1
    return A
```

This requires additional time $\Theta(m+n)$ in the worst case.

4. An edge in a flow network is called *critical* if decreasing the capacity of this edge reduces the maximum possible flow in the network.

   We are to give an efficient algorithm that finds a critical edge in a network, argue its correctness, and analyse its running time.

   Observation: First we note that if (S,T) is a minimum cut in a flow network G then any edge $e=(u,v)$ with $u \in S$ and $v \in T$ is critical. This is because for every cut (S',T') in a flow network and for every flow f in the network, we have $|f| \leqslant c(S',T')$, where $|f|$ is the value of the flow and $c(S',T')$ is the capacity of the cut. By the Max-Flow Min-Cut Theorem, if f is a maximum flow and and (S,T) is a minimum cut, then $|f| = c(S,T)$. Since decreasing the capacity of any edge e crossing the cut reduces the capacity of the cut, it follows that no flow of value $|f|$ or more is possible in the network G modified by reducing the capacity of e.

   So it suffices to give an algorithm that, given a flow network $G=(V,E)$, finds an edge e crossing some minimum cut (S,T) in G.

   Algorithm: (a) Compute a maximum flow f in G.
   (b) Compute the residual graph G_f.
   (c) Compute the minimum cut (S,T) constructed as part of the proof of Theorem 26.6 (Max-Flow Min-Cut Theorem).
   (d) Output any edge $e_{crit} = (u,v)$ with $u \in S$ and $v \in T$.

   Correctness: To prove the correctness of the algorithm it suffices to show that an edge e_{crit} in the last step exists.

   Note that every cut (S,T) has at least one edge crossing it, since by definition $s \in S$ and $t \in T$, and for every node $u \in V$ there is a path in G from s to t which includes u. Now follow this path starting with s (which is in S) until it reaches an edge (u,v) with $u \in S$ and $v \in T$.

This completes the proof that the algorithm is correct. It remains to explain how the four steps in the algorithm are implemented, and to estimate their run times.

Running Time: For steps (a) and (b) we use the Edmonds-Karp algorithm which runs in time $O(|V||E|^2)$ (p. 730 in the text).

For step (c) we use breadth first search in the residual graph G_f to make a Boolean array showing which nodes are reachable from s by paths in E_f. This defines the set S for the minimum cut and can be done in time $O(|E|)$.

Now we find the edge e_{crit} using the following algorithm:

for each edge $e = (u, v) \in E$:
    if $u \in S$ and $v \notin S$:
        return e

This takes time $O(|E|)$ (assuming E is given by adjacency lists).

Hence the entire algorithm runs in time $O(|V||E|^2)$.