COMS20010 — Problem sheet 10

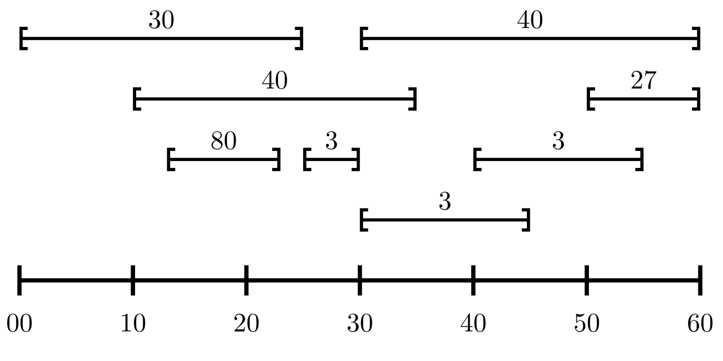
You don't have to finish the problem sheets before the class — focus on understanding the material the problem sheet is based on. If working on the problem sheet is the best way of doing that, for you, then that's what you should do, but don't be afraid to spend the time going back over the quiz and videos instead. (Then come back to the sheet when you need to revise for the exam!) I'll make solutions available shortly after the problem class. As with the Blackboard quizzes, question difficulty is denoted as follows:

- ⋆ You'll need to understand facts from the lecture notes.
- ⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in unfamiliar situations.
- ⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes and also move a bit beyond them, e.g. by seeing how to modify an algorithm.
- ⋆⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in a way that requires significant creativity. You should expect to spend at least 5–10 minutes thinking about the question before you see how to answer it, and maybe far longer. Only 20% of marks in the exam will be from questions set at this level.
- ⋆⋆⋆⋆⋆ These questions are harder than anything that will appear on the exam, and are intended for the strongest students to test themselves. It's worth trying them if you're interested in doing an algorithms-based project next year — whether you manage them or not, if you enjoy thinking about them then it would be a good fit.

If you think there is an error in the sheet, please post to the unit Team — if you're right then it's much better for everyone to know about the problem, and if you're wrong then there are probably other people confused about the same thing.

This problem sheet covers week 11, focusing on dynamic programming.

1. This question will walk through the weighted interval scheduling problem. Suppose we have the following set of weighted intervals.



We will go through creating a relevant recurrence relation, and then building a dynamic programming algorithm from it.

(a) [⋆⋆] Given a set S of intervals, let $\mathrm{WIS}(S)$ be the highest weight of any compatible subset of S. Write down a recurrence relation for $\mathrm{WIS}(S)$ based on choosing $I \in S$, then dividing compatible subsets of S into those containing I and those not containing I.

Solution: Let S be the set of all intervals. Then if $S = \emptyset$ we have $\text{WIS}(S) = 0$. Otherwise, given $I \in S$, let $X_I \subseteq S$ be the set of intervals compatible with I (i.e. those which don't intersect it). Then we have

$$\text{WIS}(S) = \max\{w(I) + \text{WIS}(X_I), \text{WIS}(S \setminus \{I\})\}.$$

The first term is the maximum weight of all compatible sets containing I, and the second term is the maximum weight of all compatible sets not containing I.

(b) [$\star\star$] Using part (a), write down a naïve exponential time algorithm which uses recursion to solve the weighted interval scheduling problem. Given a set S of intervals, it should return a maximum-weight compatible subset of S (rather than just the weight of that set as in part (a)).

Solution:

```
Algorithm: Naïve Weighted Interval Scheduling: NWIS(S)
Input  : A set of weighted intervals, $S$
Output: A subset of weighted intervals, $S' \subseteq S$, of maximum weight
begin
    if $S = \emptyset$ then
        return $\emptyset$.
    let $I \in S$.
    let $X_I = \{Y \in S\colon Y \cap I \neq \emptyset\}$.
    if $w(I) + \text{WIS}(X_I) \geq \text{WIS}(S \setminus \{I\})$ then
        return $\{I\} \cup \text{NWIS}(X_I)$.
    else
        return $\text{NWIS}(S \setminus \{I\})$.
```

Note the way this algorithm is *slightly* different to the naïve algorithm presented in lectures. How can we show that these two algorithms are actually equivalent?

(c) [$\star\star$] How can we re-arrange our recursive calls so that we can re-use the values?

Solution: We want to keep track of the values of all our recursive calls so that we can re-use them. We will do this in two parts.

First, we create a cache to store these values ("memoising"). Then, we will change our algorithm to take a sorted list of intervals as input (in increasing order of finish time), and take I to be the last interval in the list. Now, if our original input is $S = \{I_1, \ldots, I_n\}$ where I_1 finishes first, I_2 finishes second and so on, then X_{I_n} will be the set of all intervals which end before I_n starts. This set will be of the form $\{I_1, \ldots, I_j\}$ for some j. Meanwhile, $S \setminus \{I_n\}$ is just $\{I_1, \ldots, I_{n-1}\}$. So our only recursive calls over the entire algorithm will be to sets of the form $\{I_1, \ldots, I_j\}$ for some $j \leq n$, and our cache will only need to hold n values.

(d) [$\star\star$] Use this to write down a iterative polynomial time algorithm for the weighted interval scheduling problem.

Solution:

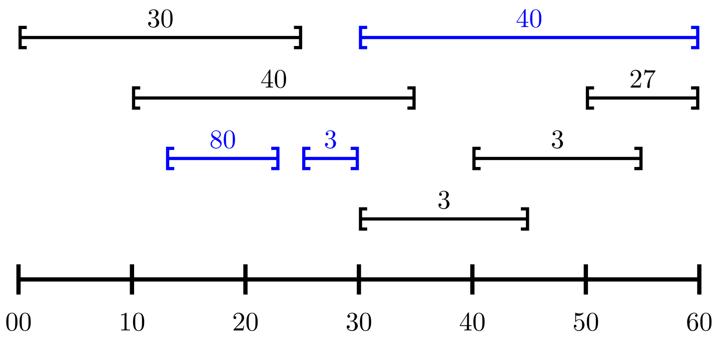
We turn the memoised recursive algorithm into an iterative algorithm by building our cache iteratively rather than recursively, starting from the base case of $S = \emptyset$ and working out way up the recursion tree.

```
Input  : An unsorted array $S$ of $n$ requests and a weight function $w$.
Output: A maximum-weight compatible subset of $\mathcal{R}$.
1 begin
2     Sort $S$ in increasing order of finish time and write $S = \{I_1, \dots, I_n\}$ as above.
3     let cache$[0] = \emptyset$ and cache$[1], \dots,$ cache$[n] =$ Null.
4     for $i = 1$ to $n$ do
5         Use binary search to find $p(i)$ such that $\{I_1, \dots, I_{p(i)}\} = X_{I_i}$, so that $I_{p(i)}$ is the
           latest-finishing interval that is compatible with $I_i$.
6         if $w(I_i) + w($cache$[p(i)]) \geq w($cache$[i-1])$ then
7             let cache$[i] = \{I_i\} \cup$ cache$[p(i)]$.
8         else
9             let cache$[i] =$ cache$[i-1]$.
10    Return cache$[n]$.
```

Here the loop invariant is that at the start of iteration i, for all $j \leq i-1$, `cache`$[j]$ contains a maximum-weight compatible set for $\{I_1, \dots, I_j\}$ (which we consider to be $\emptyset$ when $j = 0$).

(e) [$\star\star$] Run this algorithm on the above problem instance. What is the optimal set of intervals?

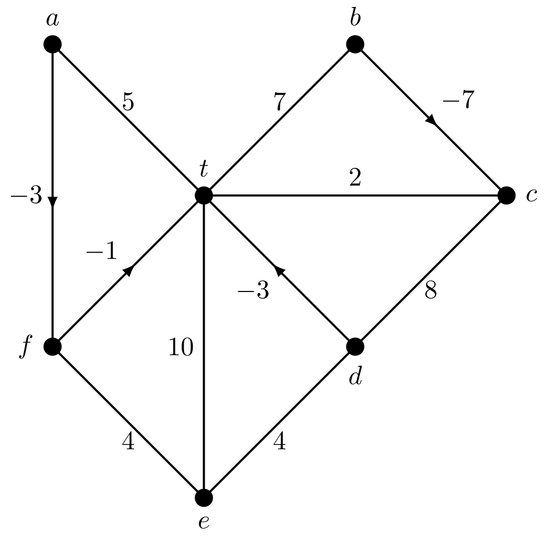
Solution:



This set of intervals has weight $80 + 3 + 40 = 123$.

2. This question will go through the Bellman-Ford algorithm to find shortest paths between vertices on a graph with no negative-weight cycles.

   Consider the following graph.



(a) [⋆⋆] Writing $d_H(x,y)$ for the distance from x to y in a graph H, write down a simple recurrence relation for $d_G(v,t)$ in terms of the out-neighbourhood $N^+(v)$ and the weight function w of the graph.

Solution:

$$d_G(v,t) = \begin{cases} 0 & \text{if } v = t, \\ \min_{x \in N^+(v)} w(v,x) + d_{G-v}(x,t) & \text{otherwise.} \end{cases}$$

If $v = t$, then since G has no negative-weight cycles we have $d(v,t) = 0$. Otherwise, any shortest path from v to t has to start by picking some edge (v,x) out of v, and is then followed by a shortest path from x to t in $G - v$.

(b) [⋆⋆] Recall that we are trying to find the shortest paths between vertices in a graph.
The recurrence relation of part (a) doesn't immediately lead to a polynomial time algorithm. What condition do we need to add to our general problem statement to achieve this?

Solution: The problem is that there are $2^{|V(G)|}$ possible sets of vertices to delete from G, so we can't evaluate $d_{G-X}(x,y)$ for all vertex pairs x,y and all vertex sets $X \subseteq V(G)$. We need to extend our problem to ask for the shortest paths between vertices, **using at most *k* edges** (where k is part of the input). This helps us out by allowing us to avoid deleting vertices. Writing $d_H(x,y,k)$ for the length of a shortest path from x to y in H with at most k edges, the recurrence of part (a) becomes

$$d_G(v,t,k) = \begin{cases} 0 & \text{if } v = t, \\ \infty & \text{if } v \neq t \text{ and } k = 0, \\ \min_{x \in N^+(v)} w(v,x) + d_{G-v}(x,t,k-1) & \text{otherwise.} \end{cases}$$

The key insight, as discussed in lectures, is that since G has no negative-weight cycles we can also write this as

$$d_G(s,v,k) = \begin{cases} 0 & \text{if } v = t, \\ \infty & \text{if } v \neq t \text{ and } k = 0, \\ \min_{x \in N^+(v)} w(v,x) + d_G(x,t,k-1) & \text{otherwise.} \end{cases}$$

i.e. taking all our distances in G rather than having to delete v. So now instead of having to calculate $d_{G-X}(x,y)$ for exponentially many vertex sets X, we only need to calculate $d_G(x,y,k)$ for $|V(G)|^3$ possible values of x, y and k.

(c) [⋆⋆] Use the polynomial time version of Bellman-Ford to find $d_G(v,t)$ for each vertex $v \in V$.

Solution: $d(a,t) = -4$, $d(b,t) = -5$, $d(c,t) = 2$, $d(d,t) = -3$, $d(e,t) = 1$, $d(f,t) = -1$, and $d(t,t) = 0$.

3. [⋆⋆] The form of the Bellman-Ford algorithm discussed in lectures will, given a weighted digraph G with no negative-weight cycles and a vertex t, return shortest paths from each $s \in V(G)$ to t in $O(|V||E|)$ time. Suppose you are instead given G and a vertex s, and you wish to find shortest paths from s to each $t \in V(G)$ in $O(|V||E|)$ time (i.e. the single-source shortest path problem). Explain how to use Bellman-Ford to do this.

Solution: Form a new digraph H by reversing every edge of G, so that $(x,y) \in E(H)$ if and only if $(y,x) \in E(G)$. Apply Bellman-Ford to H with sink s, and reverse the resulting paths so that $x_1 \dots x_t$ becomes $x_t \dots x_1$. Any length-ℓ path from a to b in G becomes a length-ℓ path from b to a in H when reversed and vice versa, so when Bellman-Ford finds a shortest path from t to s in H, reversing it yields a shortest path from s to t in G.

4. The *edit distance* between two strings is the minimum number total of character insertions, deletions and substitutions to move from one to the other. For example, the edit distances between "fad" and "fade", between "fade" and "face", and between "face" and "fae" are all 1, as is the edit distance from "fad" to "fae". The edit distance from "kitten" to "sitting" is 3, e.g. via the path "kitten" to "sitten" to "sittin" to "sitting". You wish to come up with an algorithm to find the edit distance between two strings of lengths m and n.

(a) [⋆⋆] Explain how to reduce this problem to finding a shortest path in a (large) graph, and explain why this does **not** allow you to apply e.g. breadth-first search to get an algorithm with running time polynomial in m and n.

Solution: Let S be the set of all strings on at most $\max\{m,n\}$ characters. Form a graph G on the set of strings by joining string s to string t by an edge if they are edit distance 1 apart. Then the edit distance between two strings is precisely the distance between them in G. Unfortunately, G contains more than $26^{\max\{m,n\}}$ vertices, so breadth-first search will run in time exponential in m and n.

(b) [⋆⋆⋆] Using dynamic programming or otherwise, give an algorithm with running time $O(mn)$.

Solution: First, some notation. Write $D(a,b)$ for the edit distance between a and b. Suppose our input is (w_1, w_2), where w_1 has length m and w_2 has length n. Write x_1 for the last character of w_1 and write w_1^- for the length-$(m-1)$ substring formed by removing x_1 from the end of w_1. Likewise, write x_2 for the last character of w_2 and write w_2^- for the length-$(n-1)$ substring formed by removing x_2 from the end of w_2.

We form a recursive algorithm by breaking the problem down into choices, taking our choice to be: if w_1 and w_2 are both non-empty, what operations do we perform to make the last character of w_1 match the last character of w_2? If they already match, i.e. $x_1 = x_2$, then it remains to turn w_1^- into w_2^-. If they don't match, then we must either delete x_1 (in which case it remains to turn w_1^- into w_2), delete x_2 (in which case it remains to turn w_1 into w_2^-), or turn x_1 into

x_2 via a substitution (in which case it remains to turn w_1^- into w_2^-). In any case, if $x_1 \neq x_2$, we have used an operation. It follows that

$$D(w_1, w_2) = \begin{cases} \text{length}(w_1) & \text{if } w_2 \text{ is empty,} \\ \text{length}(w_2) & \text{if } w_1 \text{ is empty,} \\ D(w_1^-, w_2^-) & \text{if } w_1, w_2 \text{ aren't empty and } x_1 = x_2, \\ 1 + \min\{D(w_1^-, w_2^-), D(w_1, w_2^-), D(w_1^-, w_2)\} & \text{otherwise.} \end{cases}$$

Observe that in evaluating this recurrence relation, we only call $D(x, y)$ where x is an initial substring of w_1 and y is an initial substring of w_2, for a total of $O(mn)$ calls. The base case takes time $O(m + n) \subseteq O(mn)$, and each other call takes time $O(1)$. Thus memoisation yields an $O(mn)$-time recursive algorithm. **Optionally**, we could make this into an iterative algorithm as follows:

```
Input  : Two words $w_1$ and $w_2$ (indexed from zero).
Output: $D(w_1, w_2)$.
begin
    let $m \leftarrow \text{length}(w_1)$ and $n \leftarrow \text{length}(w_2)$.
    let cache be an empty $(m+1) \times (n+1)$ array.
    let cache$[i][0] \leftarrow i$ for all $i \in [m]$.
    let cache$[0][j] \leftarrow j$ for all $j \in [n]$.
    for $i = 1$ to $m$ do
        for $j = 1$ to $n$ do
            if $w_1[i-1] == w_2[j-1]$ then
                Let cache$[i][j]$ = cache$[i-1][j-1]$.
            else
                Let cache$[i][j] = 1 +$
                  $\min\{$cache$[i-1][j-1]$, cache$[i][j-1]$, cache$[i-1][j]\}$.
```

This is known as the Wagner-Fischer algorithm.

5. [★★★] You are writing a word processor. When the user is writing left-aligned text, you wish your word wrap feature to make the right margins of each paragraph as even as possible. Thus for each paragraph you are given as input an ordered list $w_1, \ldots, w_n$ of words of given lengths $\ell(w_1), \ldots, \ell(w_n)$, and a maximum line length L. You may assume $\ell(w_i) \leq L$ for all $i \in [n]$. You wish to divide the words into lines $L_1, \ldots, L_t$ such that:

   - Every word appears on at most one line (i.e. you are not allowed to insert hyphens).
   - The lines preserve the order of the words, so that $L_1 = w_1 w_2 \ldots w_{i_1}$, $L_2 = w_{i_1+1} w_{i_1+2} \ldots w_{i_2}$, and so on up to $L_t = w_{i_{t-1}+1} w_{i_{t-1}+2} \ldots w_n$ for some $0 < i_1 \leq i_2 \leq \cdots \leq i_{t-1} < n$.
   - Every line has length at most L including spaces between words (which have length 1 each). Thus the length of line j is given by $\ell(L_j) = \sum_{w \in L_j} (\ell(w) + 1) - 1$, and we require $\ell(L_j) \leq L$ for all j.
   - The *raggedness* of your paragraph is given by the sum of the **squares** of the right margins for all but the last line, that is, $\sum_{i=1}^{t-1} (L - \ell(L_j))^2$. This should be as small as possible. (The reason we square the $L - \ell(L_j)$ terms here is to heavily penalise lines which are short by more than a few characters.)

   Using dynamic programming or otherwise, give an efficient algorithm to accomplish this.

Solution: We form a recursive algorithm by breaking the problem down into choices, taking our choice to be: where do we put the first line break? Let I be the largest value of i such that $w_1 \dots w_i$ can all fit on one line, i.e.

$$I = \max\Big\{i \colon i - 1 + \sum_{j=1}^{i} \ell(w_j) \le L\Big\}.$$

The lowest raggedness we can achieve by inserting a line break after w_i will be equal to the raggedness contributed by that first line plus the lowest raggedness we can achieve in $w_{i+1} \dots w_n$, so writing $R(w_1, \dots, w_n)$ for the lowest raggedness we can achieve on $w_1, \dots, w_n$ we have

$$R(w_1, \dots, w_n) = \begin{cases} 0 & \text{if } I = n, \\ \min\Big\{\Big(L - \sum_{j=1}^{k} \ell(w_j) - k + 1\Big)^2 + R(w_{k+1}, \dots, w_n) \colon 1 \le k \le I\Big\} & \text{otherwise.} \end{cases}$$

There are $O(n)$ possible calls, one for each set of arguments $w_i, \dots, w_n$, and the non-recursive parts of each call take $O(n)$ time to resolve. Thus with memoisation, we achieve an $O(n^2)$-time algorithm, choosing the first line break at whichever location gives the lowest raggedness. **Optionally**, we could make this into an iterative algorithm as follows:

```
Input  : Words $w_1, \dots, w_n$, lengths $\ell(w_1), \dots, \ell(w_n)$, and a maximum line length
         $L \ge \max\{w_i \colon i \in [n]\}$.
Output: A list of line break locations which minimises raggedness.
1  begin
2      let cache and next be empty arrays of length $n + 1$.
3      let cache[$n$] $\leftarrow 0$.
4      let next[$n$] $\leftarrow -1$.
5      let $I[j] \leftarrow \max\{i \colon i - j + \sum_{k=j}^{i} \ell(w_k) \le L\}$.
6      for $i = n - 1$ to $1$ do
7          if $I[j] == n$ then
8              let next[$i$] $\leftarrow -1$.
9              let cache[$i$] $\leftarrow 0$.
10         else
11             let next[$i$] $\leftarrow$ a value of $k$ in $\{i + 1, \dots, I[i] + 1\}$ which minimises
                 $(L - \sum_{j=i}^{k-1} \ell(w_j) - (k - 1) + i)^2 +$ cache[$k$].
12             let cache[$i$] $\leftarrow$ cache[next[$i$]] $+ (L - \sum_{j=i}^{\texttt{next}[i]-1} \ell(w_j) - (\texttt{next}[i] - 1) + i)^2$.
13     let place $\leftarrow 1$. while next[place] $\ne -1$ do
14         output line break before $w_{\texttt{next[place]}}$.
15         let place $\leftarrow$ next[place].
```

6. (a) [⋆⋆] Give an exponential-time recursive algorithm to check whether a given graph contains an independent set of a given size k. (You do not need to find the set.)

Solution: We form a recursive algorithm by breaking the problem down into choices, taking our choice to be: do we include a given vertex in an independent set, or not? Let v be an arbitrary vertex of G. Then the independent sets in G not containing v are precisely the independent sets of $G - v$, and the independent sets in G containing v are precisely the sets

$\{v\} \cup X$ where X is an independent set of $G - v - N(v)$. Thus G contains an independent set of size at least k if and only if either $G - v$ contains an independent set of size at least k or $G - v - N(v)$ contains an independent set of size at least $k - 1$, and we are left with the following recurrence relation:

$$\mathrm{MaxIS}(G) = \begin{cases} \texttt{Yes} & \text{if } k = 0, \\ \texttt{No} & \text{if } k \neq 0 \text{ and } V(G) = \emptyset, \\ \max\{\mathrm{MaxIS}(G - v), \mathrm{MaxIS}(G - v - N(v))\} & \text{otherwise, where } v \in V(G) \text{ is arbitrary.} \end{cases}$$

Evaluating this recurrence relation, with or without memoisation, yields an exponential-time algorithm.

(b) [★★] Using your answer to part (a) or otherwise, give a polynomial-time algorithm to check whether a **tree** contains an independent set of size k. **Hint:** What happens when one or more vertices are removed from the input graph, disconnecting it?

Solution: For any graph H, write $\mathcal{C}(H)$ for the set of all components of H. Observe that if G is a disconnected graph with components $C_1, \dots, C_r$, then the independent sets of G are precisely the unions of independent sets in $C_1, \dots, C_r$, so we have

$$\mathrm{MaxIS}(G) = \sum_{C \in \mathcal{C}(G)} \mathrm{MaxIS}(C).$$

Thus when G is a tree, we have

$$\mathrm{MaxIS}(G) = \begin{cases} \texttt{Yes} & \text{if } k = 0, \\ \texttt{No} & \text{if } k \neq 0 \text{ and } V(G) = \emptyset, \\ \max\{\sum_{C \in \mathcal{C}(G-v)} \mathrm{MaxIS}(C), & \text{otherwise, where} \\ \qquad \sum_{C \in \mathcal{C}(G-v-N(v))} \mathrm{MaxIS}(C)\} & v \in V(G) \text{ is arbitrary.} \end{cases}$$

We now use the trick of imposing an order. Pick an arbitrary root for the input tree, and direct all edges from the root to the leaves; thus every subtree has a unique root. Now evaluate the recurrence relation above, always taking v to be the root of the tree. Every component we recurse into will be a subtree of the original, so we have only $O(n)$ unique calls. Memoising therefore yields a polynomial-time algorithm.

(c) [★★★] A *bandwidth-b ordering* of a graph $G = (V, E)$ is an ordering $v_1, \dots, v_n$ of V such that for all edges $\{v_i, v_j\} \in E$, we have $|i - j| \leq b$. For example, if G is the path $v_1 \dots v_n$, then $v_1, \dots, v_n$ and $v_n, \dots, v_1$ are both bandwidth-1 orderings. Give a $2^b\mathrm{poly}(n)$-time algorithm for IS given a bandwidth-b ordering of the input graph.

Solution: We apply the same algorithm as in part (a), but always taking v to be the first vertex in the bandwidth-b ordering. Then at any point in the recursion, if $v = v_i$, then the only vertices we have removed from the original graph are $v_1, \dots, v_{i-1}$ and some subset of $N(v_1) \cup \dots \cup N(v_{i-1})$. Since the ordering has bandwidth b, we have

$$N(v_1) \cup \dots \cup N(v_{i-1}) \subseteq \{v_1, \dots, v_{i+b-1}\};$$

hence there are only $O(2^b)$ possibilities for the input graph, and only $O(2^b n)$ distinct recursive calls overall. We can therefore obtain a $2^b\mathrm{poly}(n)$-time algorithm by memoising.

7. [⋆⋆⋆] Using dynamic programming or otherwise, give an algorithm to determine whether or not an n-vertex graph contains a Hamilton cycle in $O(2^n \text{poly}(n))$ time.

Solution: To express this problem recursively, we'll reformulate it and add an argument. A *Hamilton path* in a graph is a path containing every vertex of the graph; thus a Hamilton cycle with an edge removed is a Hamilton path, and a Hamilton path whose endpoints are joined is a Hamilton cycle. We will ask the question $\text{HP}(G, v, X)$: Does the input graph G contain a Hamilton path from a given vertex v to some vertex in a given set $X \subseteq N(V(G))$? Observe that G contains a Hamilton cycle if and only if $\text{HP}(G, v, N(v)) = \texttt{Yes}$ for all $v \in V(G)$, so it suffices to give an $O(n2^n)$-time algorithm for HP.

We now form a recursive algorithm by breaking the problem down into choices, taking our choice to be: which edge out of v do we choose? The Hamilton paths from v to X in G that start with the edge $\{v, w\}$ correspond exactly to Hamilton paths from w to $X \setminus \{v\}$ in $G - v$, so we have

$$\text{HP}(G, v, X) = \begin{cases} \texttt{Yes} & \text{if } V(G) = \{v\}, \\ \texttt{No} & \text{if } |V(G)| > 1 \text{ and } d(v) = 0, \\ \bigvee_{w \in N(v)} \text{HP}(G - v, w, X \setminus \{v\}) & \text{otherwise.} \end{cases}$$

There are at most n possible values of v, at most 2^n possible values of G, and X is determined by G, so memoising this algorithm yields running time $O(2^n \text{poly}(n))$. This can be improved to $O(n2^n)$ with some clever optimisations.

8. (a) [⋆⋆⋆] For entirely legitimate reasons, you are inside a bank vault with a large sack that you are trying to fill with as much wealth as possible. The vault contains many different types of items, from banknotes to gold bars to mysterious treasure maps. Each item i has a price $P_i \geq 0$ and an integer volume $V_i \geq 0$, and your sack can hold total volume V. You want to find the maximum value of any set of items that you can fit in your sack; this is known as the 0-1 Knapsack problem. Give a dynamic programming algorithm which solves an n-item instance in time $O(nV)$.

Solution: Say we are given items $I_1, \dots, I_k$ and sack volume U, and write $\text{KnapsackPrice}(I_1, \dots, I_k, U)$ for the corresponding maximum price. Every possible set of items either contains I_k or it doesn't. The price of the most valuable set of items with volume at most U which *doesn't* contain I_k is $\text{KnapsackPrice}(I_1, \dots, I_{k-1}, U)$, by definition. The price of the most valuable set of items with volume at most U which *does* contain I_k is going to be the price of the most valuable set of items in $I_1, \dots, I_{k-1}$ with volume at most $U - V_k$ plus the price of I_k itself, i.e. $P_k + \text{KnapsackPrice}(I_1, \dots, I_{k-1}, U - V_k)$. Of course, we can only take I_k at all if we have enough space for it. Thus if for brevity we write

$$\begin{aligned} \text{Price}_{\text{out}} &= \text{KnapsackPrice}(I_1, \dots, I_{k-1}, U), \\ \text{Price}_{\text{in}} &= P_k + \text{KnapsackPrice}(I_1, \dots, I_{k-1}, U - V_k), \end{aligned}$$

then we have

$$\text{KnapsackPrice}(I_1, \dots, I_k, U) = \begin{cases} 0 & \text{if } k = 0, \\ \text{Price}_{\text{out}} & \text{if } k \geq 1 \text{ and } V_k > U \text{ or } \text{Price}_{\text{out}} \geq \text{Price}_{\text{in}}, \\ \text{Price}_{\text{in}} & \text{otherwise.} \end{cases}$$

If our original instance has n items and volume V, we can see that there are only V possible values of U and n possible values of k. Thus on memoising the above recurrence we will only have $O(nV)$ instances with a running time of $O(1)$ per instance, for at total of $O(nV)$.

(b) [★★★] Explain how to adjust your dynamic programming algorithm to return a collection of items which attains this value.

Solution: Again, say we are given items $I_1, \ldots, I_k$ and sack volume U, and now write Knapsack$(I_1, \ldots, I_k, U)$ for an optimal set of items (as opposed to the price). For brevity, we write

$$\begin{aligned}\text{Set}_{\text{out}} &= \text{Knapsack}(I_1, \ldots, I_{k-1}, U),\\ \text{Set}_{\text{in}} &= \{I_k\} \cup \text{Knapsack}(I_1, \ldots, I_{k-1}, U - V_k).\end{aligned}$$

Then by essentially the same argument as above we have a recurrence relation of

$$\text{Knapsack}(I_1, \ldots, I_k, U) = \begin{cases} \emptyset & \text{if } k = 0,\\ \text{Set}_{\text{out}} & \text{if } n \geq 1 \text{ and } V_k > U \text{ or } \text{Price}_{\text{out}} \geq \text{Price}_{\text{in}},\\ \text{Set}_{\text{in}} & \text{otherwise.}\end{cases}$$

Thus we add I_k to the knapsack if it leads to a higher optimum price by our old algorithm. This still runs in time $O(nV)$.

(c) [★★] In the Subset-Sum problem, you are given a set S of non-negative integers and a non-negative integer x, and you wish to decide whether there is a subset of S whose elements sum to x. Give a Cook reduction from Subset-Sum to 0-1 Knapsack.

Solution: Let (S, x) be an instance of Subset-Sum. Form an instance of 0-1 Knapsack with one item for each element of S, such that the item corresponding to element z has both price and volume equal to z, and the total sack volume is x. Then the maximum value we can fit in the knapsack will be precisely the largest number $y \leq k$ such that some subset of S sums to y; in particular, (S, x) is a Yes instance of Subset-Sum if and only if the optimal price of this 0-1 Knapsack instance is x. Thus we call our 0-1 Knapsack oracle on this instance and return Yes if it returns x and No otherwise. This process took polynomial time, and the instance has polynomial size, so we have a Cook reduction as required.

(d) [★★★★] Prove that 0-1 Knapsack is NP-hard by giving a Karp reduction from VC (a.k.a. Vertex Cover) to Subset-Sum. (**Hint:** The easiest way of doing this involves creating a Subset-Sum instance which has one number associated with each vertex *and edge* of the VC instance. You can then encode information in the digits of these numbers.)

Solution: Let (G, k) be an instance of VC, and form an instance of Subset-Sum as follows. Order the edges of G arbitrarily, writing $E(G) = \{e_1, \ldots, e_m\}$. For each $e_i \in E(G)$, let $y_{e_i} = 10^i$. Let $N = 10^{m+1}$, and for each $v \in V(G)$, let $y_v = N + \sum_{e_i \colon v \in e_i} y_{e_i}$. Let the set of numbers for the instance be $S = \{y_v \colon v \in V(G)\} \cup \{y_e \colon e \in E(G)\}$, and let $x = kN + 2\sum_{e \in E(G)} y_e$, so that the digits of x are k followed by a string of m twos. Note that forming this instance (S, x) of Subset-Sum takes polynomial time as required.

Suppose G has a vertex cover C of size k. Then we can form a corresponding set $X \subseteq S$ by including y_v for every vertex $v \in C$ and y_e for every edge $e \in E(G)$ which has *exactly one* endpoint in C, i.e.

$$X = \{y_v \colon v \in S\} \cup \{y_e \colon |e \cap C| = 1\}.$$

Then for all $i \leq |E(G)|$, the i'th least significant digit of the sum of X will be 2 (matching x) no matter whether one endpoint of e is covered by C or two. Moreover, the remaining digits of the sum of X will be k, again matching x. Thus X sums to x, and so if (G, k) is a Yes instance of VC then (S, x) is a Yes instance of Subset-Sum.

Conversely, suppose there is a subset $X \subseteq S$ which sums to x, and let $C = \{v\colon y_v \in X\}$; we claim that C is a vertex cover of G with size k. First observe that no matter what X is, there are no carries in evaluating the m least significant digits of the sum, as each such digit is 1 in exactly three elements of S (each element corresponding to an endpoint plus the element corresponding to the edge itself). Since the i'th least significant digit of this sum matches x, it is 2; writing $e_i = \{u, v\}$, it follows that exactly two out of three of y_{e_i}, y_u and y_v are in X. In particular, this implies that at least one of u or v is in C for each edge $e = \{u, v\}$ of G, so C is a vertex cover. Similarly, since the $(m+1)$'st least significant digit and up are only non-zero in elements y_v (where they are 1), the remaining digits of the sum are precisely $|\{v\colon y_v \in X\}|$; again by our choice of x, this implies $|C| = k$. Thus if (S, x) is a Yes instance of Subset-Sum, then (G, k) is a Yes instance of VC.

Putting the previous two paragraphs together, we've shown that (G, k) is a Yes instance of VC if and only if (S, x) is a Yes instance of Subset-Sum, so we have a valid Karp reduction from VC to Subset-Sum. Together with part (c), we've shown that VC $\leq_K$ SubsetSum $\leq_C$ 0-1 Knapsack, so since VC is NP-complete it follows that every problem in NP Cook-reduces to 0-1 Knapsack as required for NP-hardness.

(e) [$\star\star$] Given parts (a) and (d), why have we **not** just proved P = NP?

Solution: Part a) doesn't actually imply that 0-1 Knapsack $\in$ P. The problem is that while the algorithm is polynomial in n and V, in order for 0-1 Knapsack to be in P it would need to be polynomial in the input size — and the input size is $\Theta(n + \log V)$, since V is represented by $\log_2 V$ bits. So despite being polynomial in n and V, and despite being reasonable for many applications where V is relatively small, it actually runs in time exponential in the input size! (Notice that we relied on this in part (d), too — if we tried to solve VC by reducing to 0-1 Knapsack, then the instance we got would have an exponentially large value of V.)