COMS20010 — Problem sheet 10

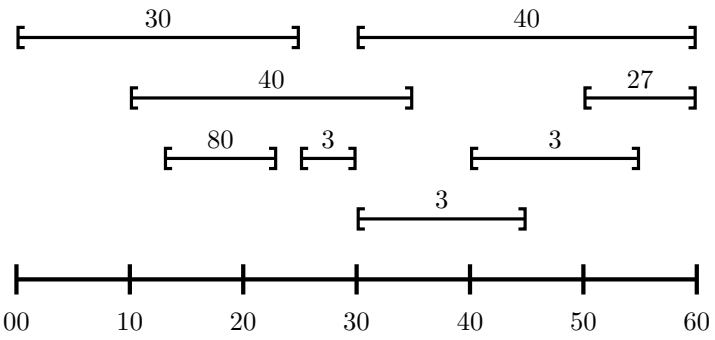
You don't have to finish the problem sheets before the class — focus on understanding the material the problem sheet is based on. If working on the problem sheet is the best way of doing that, for you, then that's what you should do, but don't be afraid to spend the time going back over the quiz and 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.

(b) [⋆⋆] 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)).

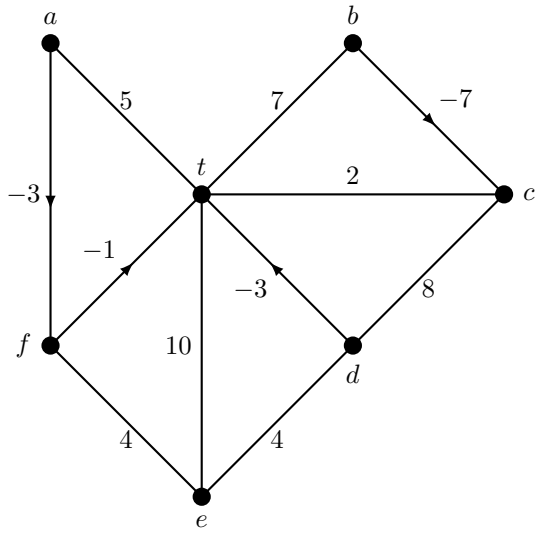
(c) [⋆⋆] How can we re-arrange our recursive calls so that we can re-use the values?

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

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

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.

(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?

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

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.

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.

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

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.

6. (a) [★★] Give an exponential-time recursive algorithm to find the maximum **size** of an independent set in a graph.

   (b) [★★] Using your answer to part (a) or otherwise, give a polynomial-time algorithm to find the maximum size of an independent set in a **tree**. **Hint:** What happens when the input graph is disconnected?

   (c) [★★★] A *bandwidth-b ordering* of a graph $G = (V, E)$ is an ordering $v_1, \ldots, v_n$ 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 $v_1 \ldots v_n$, then $v_1, \ldots, v_n$ and $v_n, \ldots, v_1$ are both bandwidth-1 orderings. Give a $2^b \text{poly}(n)$-time algorithm for IS given a bandwidth-b ordering of the input graph.

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.

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 $O(nV)$.

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

   (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.

   (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.)

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