1 Study Group
List the names and SIDs of the members in your study group.

2 2-SAT
Please provide solutions to parts (d), (e) and (f) of Question 3.28 from http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf.

Solution:

(d) Suppose there is a SCC containing both \( x \) and \( \overline{x} \). Notice that the edges of the graph are necessary implications. Thus, if some \( x \) and \( \overline{x} \) are in the same component, there is a chain of implications which is equivalent to \( x \rightarrow \overline{x} \) and a different chain which is equivalent to \( \overline{x} \rightarrow x \), i.e. there is a contradiction in the set of clauses.

(e) Take any sink component, and assign variables so all the literals in this component are True. Because of how we define the graph, there is a corresponding source component which has the negations of all literals in this component. Remove this source/sink component pair, and repeat the process until the graph is empty. Since we set components to true in reverse topological order, there is no implication from a true literal to a false literal. Since no literal and its negation are in the same SCC, we never try to set a variable to be both true and false. So this produces an assignment satisfying all clauses.

(f) Let \( \varphi \) be a formula acting on \( n \) literals \( x_1, \ldots, x_n \). Construct a graph with \( 2n \) vertices representing the set of literals and their negations. For each clause \((a \lor b)\) of \( \varphi \) add the edges \( \overline{a} \Rightarrow b \) and \( b \Rightarrow a \). Use the strongly connected components algorithm and for each \( i \), check if there is a SCC containing both \( x_i \) and \( \overline{x_i} \). If any such component is found, report unsatisfiable. Otherwise, report satisfiable.

(Note: A common mistake is to report unsatisfiable if there is a path from \( x_i \) to \( \overline{x_i} \) in this graph, even if there is no path from \( \overline{x_i} \) to \( x_j \). Even if there is a series of implications which combined give \( x_i \rightarrow \overline{x_i} \), unless we also know \( \overline{x_i} \rightarrow x_i \) we could set \( x_i \) to False and still possibly satisfy the clauses. For example, consider the 2-SAT formula \((\overline{a} \lor b) \land (\overline{a} \lor \overline{b})\). These clauses are equivalent to \( a \rightarrow b, b \rightarrow \overline{a} \), which implies \( a \rightarrow \overline{a} \), but this 2-SAT formula is still easily satisfiable.)

3 Minimum Spanning \( k \)-Forest
Given a graph \( G(V, E) \) with nonnegative weights, a spanning \( k \)-forest is a cycle-free collection of edges \( F \subseteq E \) such that the graph with the same vertices as \( G \) but only the edges in \( F \) has \( k \) connected components. For example, consider the graph \( G(V, E) \) with vertices \( V = \{A, B, C, D, E\} \) and all possible edges. One spanning 2-forest of this graph is \( F = \)
CS 170, Fall 2019

HW 6

P. Raghavendra & S. Rao

{(A, C), (B, D), (D, E)}, because the graph with vertices \( V \) and edges \( F \) has components \{A, C\}, \{B, D, E\}.

The minimum spanning \( k \)-forest is defined as the spanning \( k \)-forest with the minimum total edge weight. (Note that when \( k = 1 \), this is equivalent to the minimum spanning tree).

In this problem, you will design an algorithm to find the minimum spanning \( k \)-forest. For simplicity, you may assume that all edges in \( G \) have distinct weights.

(a) Define a \( j \)-partition of a graph \( G \) to be a partition of the vertices \( V \) into \( j \) (non-empty) sets. That is, a \( j \)-partition is a list of \( j \) sets of vertices \( \Pi = \{S_1, S_2, \ldots, S_j\} \) such that every \( S_i \) includes at least one vertex, and every vertex in \( G \) appears in exactly one \( S_i \).

For example, if the vertices of the graph are \{A, B, C, D, E\}, one 3-partition is to split the vertices into the sets \( \Pi = \{\{A, B\}, \{C\}, \{D, E\}\} \).

Define an edge \((u, v)\) to be crossing a \( j \)-partition \( \Pi = \{S_1, S_2, \ldots, S_j\} \) if the set in \( \Pi \) containing \( u \) and the set in \( \Pi \) containing \( v \) are different sets. For example, for the 3-partition \( \Pi = \{\{A, B\}, \{C\}, \{D, E\}\} \), an edge from \( A \) to \( C \) would cross \( \Pi \).

Show that for any \( j \)-partition \( \Pi \) of a graph \( G \), if \( j > k \) then the lightest edge crossing \( \Pi \) must be in the minimum spanning \( k \)-forest of \( G \).

(b) Give an efficient algorithm for finding the minimum spanning \( k \)-forest.

Please give a 3-part solution.

Solution:

(a) It helps to note that when \( j = 2, k = 1 \) this is exactly the cut property. A similar argument lets us prove this claim.

For some \( j \)-partition \( \Pi \) where \( j > k \), suppose that \( e \) is the lightest edge crossing \( \Pi \) but \( e \) is not in the minimum spanning \( k \)-forest. Let \( F \) be the minimum spanning \( k \)-forest.

Now, consider adding \( e \) to \( F \). One of two cases occurs:

- \( F + e \) contains a cycle. In this case, some edge \( e' \) in this cycle besides \( e \) must cross \( \Pi \). This means \( F + e - e' \) is a spanning \( k \)-forest, since deleting an edge in a cycle cannot increase the number of components. Since \( e \) by definition is cheaper than \( e' \), the forest \( F + e - e' \) is cheaper than \( F \), which contradicts \( F \) being a minimum spanning \( k \)-forest.

- \( F + e \) does not contain a cycle. In this case, the endpoints of \( e \) are in two different components of \( F \), so \( F + e \) has \( k + 1 \) components. Since \( \Pi \) is a \( j \)-partition and \( j > k \), some edge \( e' \) in \( F \) must cross \( \Pi \). Deleting an edge from a forest increases the number of components in the forest by only 1, so \( F + e - e' \) has \( k \) components, i.e. is a \( k \)-forest. Since \( e \) by definition is cheaper than \( e' \), the forest \( F + e - e' \) is cheaper than \( F \), which contradicts \( F \) being a minimum spanning \( k \)-forest.

In either case, we arrive at a contradiction and have thus proven the claim.

(b) There are multiple solutions, we recommend the following one because its proof of correctness follows immediately from part a:
**Main Idea:** The algorithm is to run Kruskal’s, but stop when \( n - k \) edges are bought, i.e. the solution is a spanning \( k \)-forest.

**Correctness:** Any time the algorithm adds an edge \( e \), let \( S_1 \ldots S_j \) be the components defined by the solution Kruskal’s arrived at prior to adding \( e \). \( S_1 \ldots S_j \) form a \( j \)-partition and by definition of the algorithm, \( j > k \). \( e \) is the cheapest edge crossing this \( j \)-partition, so by part a \( e \) must be in the (unique) minimum spanning \( k \)-forest. Since every edge we add is in the minimum spanning \( k \)-forest, our final solution must be the minimum spanning \( k \)-forest.

**Runtime Analysis:** This is just modified Kruskal’s so the runtime is \( O(|E| \log |V|) \) (Kruskal’s runtime is dominated by the edge sorting, so the fact that we may make less calls to the disjoint sets data structure because the algorithm terminates early does not affect our asymptotic runtime).

### 4 Steel Beams

You’re a construction engineer tasked with building a new transit center for a large city. The design for the center calls for a \( T \)-foot-long steel beam for integer \( T > 0 \). Your supplier can provide you with an unlimited number of steel beams of integer lengths \( 0 < c_1 < \ldots < c_k \) feet. You can weld as many beams as you like together; if you weld together an \( a \)-foot beam and a \( b \)-foot beam you’ll have an \((a + b)\)-foot beam. Unfortunately, every weld increases the chance that the beam might break, so you want as few as possible.

Your task is to design an algorithm which outputs how many beams of each length you need to obtain a \( T \)-foot beam with the minimum number of welds, or ‘not possible’ if there’s no way to make a \( T \)-foot beam from the lengths you’re given. (If there are multiple optimal solutions, your algorithm may return any of them.)

(a) Consider the following greedy strategy. Start with zero beams of each type. While the total length of all the beams you have is less than \( T \), add the longest beam you can without the total length going over \( T \).

(i) Suppose that we have 1-foot, 2-foot and 5-foot beams. Show that the greedy strategy always finds the optimum.

(ii) Find a (short) list of beam sizes \( c_1, \ldots, c_k \) and target \( T \) such that the greedy strategy fails to find the optimum. Briefly justify your choice.

(b) Give a dynamic programming algorithm which always finds the optimum.

(i) State your recurrence relation.

(ii) Prove correctness of your algorithm by induction.

i. Show that the base case is correct.

ii. Assuming that your recurrence relation is correct for previous subproblems, show that it gives the correct value for the current subproblem.

iii. Give the order where you can solve the subproblems, and show that for this order, evaluating the recurrence relation will use only subproblems that have already been computed.
(iii) Find the running time and space requirement of your algorithm.

**Solution:** Formal statement of problem: Given a list of integers $C = (c_1, \ldots, c_k)$ with $0 < c_1 < \ldots < c_k$ and a target $T > 0$, the algorithm should output nonnegative integers $(a_1, \ldots, a_k)$ such that $\sum_{i=1}^{k} a_i c_i = T$ where $\sum_{i=1}^{k} a_i$ is as small as possible, or return ‘not possible’ if no such integers exist.

(a) (i) Let $a_1, a_2, a_3$ be some optimum solution. We know that $a_1 < 2$ since if $a_1 \geq 2$ we can improve the solution by taking $a_1 - 2, a_2 + 1, a_3$. If $a_1 = 1$ then $a_2 < 2$ because otherwise we could improve by taking $0, a_2 - 2, a_3 + 1$. So the possible values of the optimum are $0, 1, j, 0, 2, j, 1, 1, j$ for some $j \geq 0$. In each case the greedy algorithm would give the same answer.

(ii) $C = (4, 5), T = 8$ is one possibility.

(b) (i) We create a dynamic programming algorithm where, for each $n \leq A$, we will find the minimum integer combination that sums to $n$. The recurrence is $f(n) = \min_{k=1}^{k} f(n - c_i) + 1$, with $f(0) = 0$.

(ii) i. The base case is $f(0) = 0$.

ii. Fix $n > 0$. Suppose that $f(n')$ is optimal for all $n' < n$. Firstly note that if $a_1', \ldots, a_k'$ is a minimum integer combination summing to $n - c_i$, then $a_1', \ldots, a_i' + 1, \ldots, a_k'$ is an integer combination summing to $n$ (not necessarily minimum).

Let $a_1, \ldots, a_k$ be a minimum integer combination summing to $n$. Then for every $i$ with $a_i > 0$, $a_1, \ldots, a_i - 1, \ldots, a_k$ is a minimum integer combination summing to $n - c_i$ (otherwise $a_1, \ldots, a_k$ wouldn’t be optimal). We know that some $a_i > 0$ (we don’t know which), so we take the minimum over all $i$.

iii. We solve the subproblems in the order of smallest to largest $n$, since to compute $f(n)$, we only use $f(n')$ for $n' < n$.

(iii) We compute $T$ subproblems, each one being a minimum of $k$ values, so the running time is $O(Tk)$. The space requirement is $O(T)$.

5 Non-Prefix Code

As we have learned in lecture, the Huffman code satisfies the Prefix Property, which states that the bit string representing each symbol is not a prefix of the bit string representing any other symbol. One nice property of such codes is that, given a bit string, there is at most one way to decode it back to a sequence of symbols. However, this is not true anymore once we are working with codes that do not satisfy the Prefix Property. For example, consider the code that maps $A$ to 1, $B$ to 01 and $C$ to 101. A bit string 101 can be interpreted in two ways: as $C$ or as $AB$.

Your task is to, given a bit string $s$, determine how many ways one can interpret $s$. The mapping from symbols to bit strings of the code will be given to you as a dictionary $d$ (e.g., in the example, $d = \{A : 1, B : 01, C : 101\}$); you may assume that you can access each symbol in the dictionary in constant time. Your algorithm should run in time at most $O(nm\ell)$ where $n$ is the length of the input bit string $s$, $m$ is the number of symbols, and $\ell$ is an upper bound
on the length of the bit strings representing symbols.

Please give a 3-part solution.

Solution:
Main Idea: We define our subproblems as follows: let $A[i]$ be the number of ways of interpreting the string $s[:i]$. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$A[i] = \sum_{\text{symbol } a \text{ in } d \text{ such that } s[i - \text{length}(d[a]) + 1 : i] = d[a]} A[i - \text{length}(d[a])]$$

Note here that we set $A[0] = 1$. Our algorithm simply computes the above formula in a trivial manner.

Pseudocode:

```
procedure Translate(s):
    Create an array $A$ of length $n + 1$ and initialize all entries with zeros.
    Let $A[0] = 1$
    for $i := 1$ to $n$ do
        for each symbol $a$ in $d$ do
            if $i \geq \text{length}(d[a])$ and $d[a] = s[i - \text{length}(d[a]) + 1 : i]$ then
                $A[i] += A[i - \text{length}(d[a])]$
        return $A[n]$
```

Proof of Correctness: We can show this via a simple induction argument.

Base Case. When $i = 0$, there is only one way to interpret $s[:0]$ (the empty string). Hence, $A[0] = 1$.

Inductive Step. Suppose that $A[0], \ldots, A[i-1]$ contains the right value. We will show that the above recurrence relation gives the right value for $A[i]$. To do this, we partition interpretations of $s[:i]$ as a sequence of symbols $a_1 \ldots a_k$ based on the ending symbol $a_k$. For $a_k = a$, if the suffix of $s[:i]$ coincides with $d[a]$, every interpretation $a_1 \ldots a_{k-1}$ of $s[:i - \text{length}(d[a])]$ from our inductive hypothesis, there are exactly $A[i - \text{length}(d[a])]$ of the latter. On the other hand, if the suffix of $s[:i]$ differs from $d[a]$, then there is no interpretation of $s[:i]$ ending with symbol $a$. Summing this up over all symbols $a$’s implies that our recurrence relation yields the right value for $A[i]$. Finally, note that our program below implements this recurrence in a straightforward way, so the output of our program is indeed $A[n]$, the number of ways to interpret $s$.

Runtime Analysis: There are $n$ iterations of the outer for loop and $m$ iterations of the inner for loop. Inside each of these loops, checking that the two strings are equal takes $O(\text{length}(d[a])) \leq O(\ell)$ time. Hence, the total running time is $O(nm\ell)$.

Note that it is possible to speed up the algorithm running time to $O((n + m)\ell)$ using a trie instead of reconstructing the string every time, but this is not required to receive full credit.
6 Breaking Chocolate

There is a chocolate bar consisting of an $m \times n$ rectangular grid of squares. Some of the squares have raisins in them, and you hate raisins. You would like to break the chocolate bar into pieces so as to separate all the squares with raisins, from all the squares with no raisins. For example, shown below is a $6 \times 4$ chocolate bar with raisins in squares marked $R$. As shown in the picture, one can separate the raisins out in exactly four breaks.

\[
\begin{array}{cccc}
R & R & R & \Rightarrow \ \ R & R & R & \Rightarrow \ \ R & R & R \Rightarrow \ \ R & R & R
\end{array}
\]

(At any point in time, a break is a cut either horizontally or vertically of one of the pieces at the time.)

Design a DP based algorithm to find the smallest number of breaks needed to separate all the raisins out. Formally, the problem is as follows:

**Input:** Dimensions of the chocolate bar $m \times n$ and an array $A[i, j]$ such that $A[i, j] = 1$ if and only if the $ij$th square has a raisin.

**Goal:** Find the minimum number of breaks needed to separate the raisins out.

(a) Define your subproblem.

(b) Write down the recurrence relation for your subproblems.

(c) What is the time complexity of solving the above mentioned recurrence? Provide a justification.

**Solution:**

(a) We define $B[i_1, j_1, i_2, j_2]$ to be the minimum number of breaks needed to separate the sub-matrix $A[i_1 \leq i \leq i_2, j_1 \leq j \leq j_2]$ into pieces consisting either entirely of raisin pieces or entirely of non-raisin pieces.
(b) \[
B[i_1, j_1, i_2, j_2] = \min \begin{cases} 
0, & \text{if all entries of } A[i_1 \ldots i_2, j_1 \ldots j_2] \text{ are equal} \\
1 + B[i_1, j_1, i_1 + k, j_2] + B[i_1 + k + 1, j_1, i_2, j_2] & \text{for any } k \in \{1, \ldots, i_2 - i_1\} \\
1 + B[i_1, j_1, i_2, j_1 + k] + B[i_1 + 1, j_1, i_2, j_2] & \text{for any } k \in \{1, \ldots, j_2 - j_1\}
\end{cases}
\] (1)

Alternatively, you could have also encapsulated the 0 base case in all single-square pieces, and determined if a piece was pure via the merging, see below.

(c) Two answers are acceptable: \(O((m + n)m^2n^2)\) and \(O(m^3n^3)\)

We have \(O(m^2n^2)\) total subproblems: \(O(mn)\) possibilities for \((i_1, j_1)\), and \(O(mn)\) possibilities for \((i_2, j_2)\). For each subproblem, we examine up to \(m\) possible choices for horizontal splits, and \(n\) possible choices for vertical splits. A single split consideration will result in two smaller subproblems, which we can assume have already been solved, so we just need to find the best split, which takes \(O(n + m)\) time.

In addition, for a subproblem, we also want to check the base case for if the piece is “pure” (contains only raisins, or contains only non-raisins). Brute force checking this takes \(O(mn)\) time, for a total subproblem time of \(O(mn + (m + n)) \rightarrow O(mn)\).

However, this \(O(mn)\) factor can be reduced to \(O(m + n)\) (this is not required to receive full points). We can precompute the purities of every single possible subrectangle and store it in a table. Technically, for us to pre-compute faster than \(O(m^3n^3)\), we’ll need to compute the purities more intelligently than by brute-force. It is possible to do this in as fast as \(O(\max\{m, n\}^2)\) time, although the details of this are complicated and won’t be explained here. So to solve our recurrence relation, if we can determine purity/impurity in \(O(1)\) time, then we can reach an overall time of \(O((m + n)m^2n^2)\).

Alternatively, we can initialize all min-break values of single square pieces to be 0. Then, if it is possible to have some break such that both resulting pieces have min-break values of 0, and both resulting pieces are of the same type (raisin-only or non-raisin-only, and we can take any sample of either and compare them), then we ourself are a pure piece. This would allow you to avoid the entire pre-computation business as mentioned before, and still achieve a runtime of \(O((m + n)m^2n^2)\).