Homework 1: Algorithm Design Basics

Handed out Thu, Sep 10. Due at the start of class Tue, Sep 22.

Problem 1. Arrange the following functions in increasing order of asymptotic growth rate. If two or more functions have the same asymptotic growth rates, then indicate this. No explanation is required, but they are encouraged, since we can give partial credit if your explanation shows some understanding of the concepts. (Remember that lg is shorthand for $\log_2$)

$$\begin{aligned}
f_0(n) &= n \lg n + n^{3/2} \\
f_1(n) &= n! \\
f_2(n) &= 2^n \\
f_3(n) &= n^5 \lg n + 3^n \\
f_4(n) &= \lg \lg n \\
f_5(n) &= \log_{10} n \\
f_6(n) &= 3^{\lg n} \\
f_7(n) &= \sqrt{\log_3 n} \\
f_8(n) &= n\sqrt{n} \\
f_9(n) &= n \lg n
\end{aligned}$$

Problem 2. In the stable-marriage problem, suppose that all the men share the same preference ranking for the women. For example, suppose that they all list woman 1 first, woman 2 second, and so on. Further, suppose that the women obtain a copy of this shared rank ordering.

The women meet and decide the final pairing that they desire. By using their knowledge of the men's preference ordering, can the women force the Gale-Shapley algorithm to produce their desired pairing? (The women cannot change the algorithm, but they can coordinate their preference lists.)

If possible, present the women's solution and explain why it works. If not, give an counterexample in the form of a pairing that women desire, but that no matter how the women present their preferences, they cannot cause Gale-Shapley to produce this pairing.

Problem 3. Consider the following (extremely gender-biased) algorithm for the stable marriage problem. As in the standard problem, there are n men, and n women. Each man and each woman has an n-element preference list that rank orders all the members of the opposite sex. This algorithm *ignores* the preferences of the women and simply pairs each man with the first available woman on his list.

```
for (i = 1 to n) {
  j = first woman on man m[i]'s preference list that is not already engaged
  create the engagement between man i and woman j
  mark woman j as engaged
}
```

(a) Is this algorithm guaranteed to produce a perfect matching (that is, is every man paired exactly one woman and vice versa)? If so, give a proof, and if not, give a counterexample and explain your counterexample.

(b) If your answer to (a) was "no", skip this part. Otherwise, is the matching produced by this algorithm guaranteed to be stable? If so, give a proof, and if not, present a counterexample and explain your counterexample.

(To make the grader's life easier, strive to make any counterexamples as short as possible. The grader reserves the right to deduct some points if your counterexample, even if correct, is much more complicated than it needs to be.)

Problem 4. Consider the following alternative approach to the previous-larger problem. (This was motivated by a suggestion from one of the students in the class.)

Rather than scan the list from left to right, we scan from right to left. We let i run from n down to 1. When processing element a_i, some of the elements a_j for $j \geq i$ will have no prior element that is larger than they are. (This will certainly be true for the largest element among $\{a_{i+1}, \ldots, a_n\}$.) We will store the indices of these elements on a stack S, with the property that the top of the stack has the smallest a-value and the values increase as we descend deeper into the stack. In order to process the element a_i, we will remove all items from the stack whose a-value is not greater than a_i, and set their p-value to point to a_i. After this, we push a_i onto the stack.

After processing all the elements of a, if there are any other elements remaining on the stack, we set their p-value to zero. The pseudo-code is given below.

```
// Input: An array of numeric values a[1..n]
// Returns: An array p[1..n] where p[i] contains the index of the previous
//   larger element to a[i], or 0 if no such element exists.
prevLarger2(a, n) {
    S = an empty stack
    for (i = n downto 1) {
        while (S is nonempty and a[top(S)] <= a[i]) {
            p[top(S)] = i
            pop(S)
        }
        push i onto S
    }
    while (S is nonempty) {
        j = pop(S)
        p[j] = 0
    }
    return p
}
```

(a) Show that for any fixed i, where $1 \leq i \leq n$, there exists an input to this algorithm such that when the for loop runs on index i, exactly $n - i$ elements are popped from the stack.

(b) Show that if the observation from (a) applied to *every* iteration of the for loop, then the above algorithm would have a worst-case running time of $\Theta(n^2)$. (In particular, express the running time as a sum, and show that this sum is $\Theta(n^2)$.)

(c) Show that the worst-case scenario hypothesized in part (b) *cannot occur*. In particular, prove that the worst-case running time of the above algorithm is $\Theta(n)$.

Challenge Problem. Challenge problems count for extra credit points. These additional points are factored in only after the final cutoffs have been set, and can only increase your final grade.

The *distance* between two nodes in a graph is defined to be the length of the shortest path between them. Assuming that there are no edge weights, the length of a path is defined to be the number of edges on the path. The *diameter* of a graph is defined to be the maximum distance between any two nodes of the graph. Two nodes u and v whose distance equals the diameter is said to be a *diametrical pair*. A graph may generally have many diametrical pairs of nodes. The diameter is an important graph statistic. For example, many large networks have very small diameters[1]

Computing the diameter of a graph generally involves computing the distances between all pairs of nodes, but there are special cases where it is possible to do better. In particular, suppose that you are given a *free tree*, that is, a connected, acyclic, undirected graph (see Fig. 1(a)).

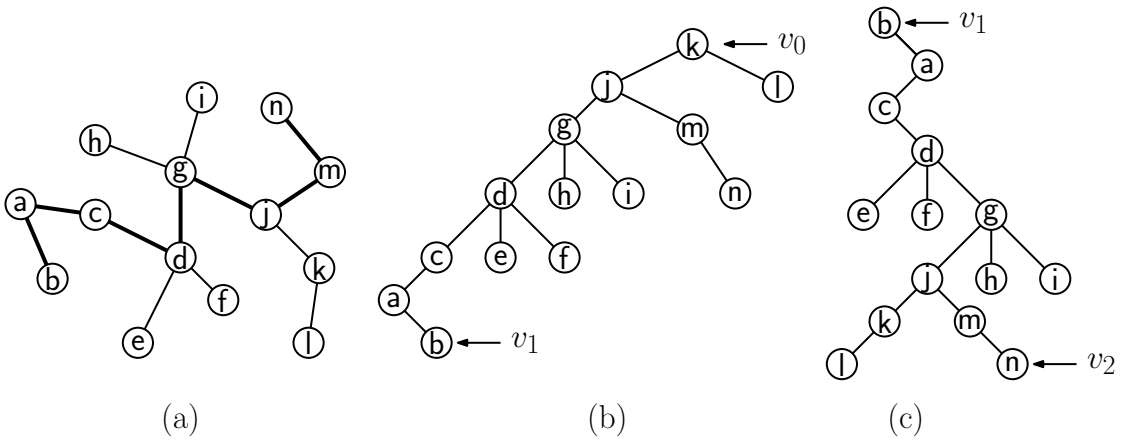


Figure 1: Challenge Problem.

There are a number of algorithms for efficiently computing the diameter of a free tree with n nodes. Here is a clever and rather surprising solution. First, start at any node v_0, and find the node v_1 that is farthest from v_0. (This can be done in $O(n)$ time by any tree traversal algorithm starting with v_0 as the root and maintaining the level of each node in the resulting rooted tree. BFS can also be used.) Next, compute the node v_2 that is farthest from v_1. The claim is that the distance between v_1 and v_2 is the diameter of the tree.

[1]This is evidenced by the famous "6-degrees of separation" hypothesis, which states that every two people on earth are within six friendship links of each other. According to Wikipedia, the Facebook friend graph has a diameter of 12.

For example, suppose we start this algorithm with the tree of Fig. 1(a) letting $v_0 \leftarrow \mathsf{k}$. Its farthest node is b (see Fig. 1(b)), which plays the role of v_1 in the algorithm. We then compute the farthest node from v_1, which is either n or l (see Fig. 1(c)). Thus, $v_2 \leftarrow \mathsf{n}$. The final diametrical pair is (b, n), which we can see is the correct answer in this case. (We could have also output (b, l), which is also correct.)

To prove the general correctness of the above algorithm, answer each of the following questions. (You will receive partial credit if you answer any subset.)

(a) A node of a free tree that has degree 1 is called a *leaf*. Prove that if v_0 is any node and v_1 is a node of maximum distance from v_0, then v_1 is a leaf.

(b) Suppose that (u_1, u_2) is a diametrical pair. Prove that both u_1 and u_2 are leaves. (Hint: The proof is very short, given (a).)

(c) Let v_0 be an arbitrary node. Prove that if (u_1, u_2) is a diametrical pair, then one of these two nodes is the farthest node from v_0. (Hint: Let v_1 be the node that is farthest from v_0. It may help to first prove that the path from v_0 to v_1 and the path from u_1 to u_2 must share at least one vertex in common. The proof is complicated by the fact that there may be multiple longest paths of the same length. If it makes your life easier, you may assume that each time you compute a longest path it is unique.)

(d) Using (c), prove that the above algorithm correctly computes the diameter of a free tree.

(e) (Optional) Show that the algorithm may fail to produce a correct result if the graph has cycles. (Specify both the graph and initial node v_0.)

Homework 2: BFS, DFS, and Greedy

Handed out Tue, Sep 29. Due at the start of class Thu, Oct 8.

Problem 1. Consider the graph shown in Fig. 1 below.

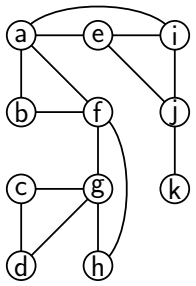


Figure 1: Problem 1.

(a) Show the result of running BFS on this graph using "a" as the source vertex. For the sake of uniformity, whenever you have a choice of vertex to visit/process next, chose the one that is lowest in alphabetical order. Label each vertex with its d-value (distance from the source). Indicate tree edges with solid lines and cross edges with dashed lines.

(b) Show the result of running DFS on this graph using the algorithm given on page 2 of Lecture 5 (Same alphabetical rule applies.) Label each node u with its discovery and finish times ($d[u]/f[u]$). (You need only show the final DFS tree, not the intermediate results.) As in the lecture notes, show tree edges with solid lines and back edges using dashed lines.

(c) Show the result of running the findCutVertices algorithm from Lecture 6 on this graph. (Same alphabetical rule applies.) As in Fig. 8 of the lecture notes, label each node u with the values $d[u]$/Low$[u]$. Also, indicate which vertices are cut vertices.

Problem 2. Suppose that you have an undirected graph $G = (V, E)$ where each edge either has weight 1 or weight 2. The *length* of a path in this graph is the sum of the edge weights along the path. The *distance* between two vertices is the minimum length of any path joining them. (You may assume that you have access to a function $w(u, v)$ that returns the weight of edge (u, v) in constant time.)

Given a source node $s \in V$, describe an $O(n+m)$-time algorithm that computes the distance between s and every other node of G. Recall that $n = |V|$ and $m = |E|$. (Hint: This can be done by means of a simple modification to BFS, but I will accept any algorithm. Your algorithm may create new labels associated with the vertices of the graph, but it should not make any structural modifications to the graph.)

As always, justify your algorithm's correctness and derive its running time. (If your algorithm is a simple modification of BFS, you can rely on the BFS analysis given in class.)

Problem 3. You are given a directed acyclic graph $G = (V, E)$. Each node u of this graph represents a task to be performed, and each edge (u, v) is a precedence constraint, which means that task u must be completed before task v is started. Each node u is associated with a numeric value $\mathsf{time}[u]$, which indicates the time needed to perform this task. The *cost* of a path is defined to be the sum of the time values of the nodes on the path. As shown in class, if we attempt to run all the jobs in parallel, except where order is imposed by the precedence constraints, the path of highest cost in the DAG yields the total time needed to complete all the tasks.

In Lecture 6, we presented a DFS algorithm for computing the cost of the maximum cost path in a DAG. Here we will consider some additional related problems.

(a) Present an algorithm to compute not just the cost of the maximum cost path, but the path itself. (If there are multiple paths achieving the maximum cost, you may output any one of them.) You may present a complete algorithm, or explain how to modify the algorithm given in Lecture 6.

(b) Suppose that time starts at $t = 0$. For each node u, define its *earliest start time*, denoted $\mathsf{EST}[u]$, to be earliest time at which the task associated with this node can be started, subject to the precedence constraints. Present an $O(n+m)$-time algorithm that computes $\mathsf{EST}[u]$ for all $u \in V$.

(Hint: I know of a DFS algorithm for this problem, but it requires either modifying the DAG or constructing a modified copy of the DAG. I also know of a non-DFS algorithm that has the desired running time, but does not need to copy or modify the DAG. Either approach is fine.)

(c) For some tasks (for example for tasks that are not on any maximum-cost path) we can start this task later than its earliest start time without affecting the total time needed to complete all the tasks. We would like to know how late we can push each task back until this happens. For each node u, define its *latest start time*, denoted $\mathsf{LST}[u]$, to be the latest time at which task u can be started such that (assuming no further tasks are delayed) the total completion time of all the tasks is unaffected. Present an $O(n + m)$-time algorithm to compute this value for each node.

In all instances, justify the correctness of your algorithm and (if there are nontrivial changes to DFS) derive the running time.

Problem 4. You are a working for a game development company. In your company's latest game, called "Duke Nukem vs. My Little Ponies," pits our hero Duke Nukem against a horde of super-cute colorful ponies. There are n ponies, and each pony makes an appearance on screen for a single time interval. Let $[s_i, t_i]$ denote the start time and end time for the ith pony. Duke's only defense is a bomb, which when activated destroys *all* the ponies that are currently on screen.

Sadly, Duke has a limited supply of bombs. Given that Duke has exactly B bombs, the question is whether he can judiciously choose when to activate them in order to blow up all n of the ponies, thus saving the world from the horror of their excessive cuteness.

Write an efficient algorithm that, given the value of B and the n time intervals $[s_i, t_i]$, determines whether Duke can blow up all the ponies. If so, your program should also output

the exact times at which the bombs should be exploded. Derive the running time of your algorithm and prove that it is correct. (Hint: Apply a greedy approach.)

Challenge Problem. Challenge problems count for extra credit points. These additional points are factored in only after the final cutoffs have been set, and can only increase your final grade.

This problem involves the question of how the number of back edges in a graph affects the number of simple cycles. Recall that a cycle is *simple* if it no vertex appears twice in the cycle. For the purposes of counting, two cycles containing the same set of vertices and the same set of edges are the same cycle (whether you use a different starting node or traverse the cycle backwards).

(a) Suppose that G is an undirected graph, and after running DFS you find that there is exactly one back edge. What is the maximum number of simple cycles that G can have? Explain.

(b) Answer (a) again, but now suppose that there two back edges in the DFS. Explain.

(c) Answer (a) again, but now suppose that there are k back edges. Express your answer as a function of k. Explain. (Hint: I don't know the exact number. Try to derive as large a value as you can. You can express your answer asymptotically (e.g., $O(k)$, $O(k^2)$, $2^{O(k)}$).

(d) Suppose now that G is a directed graph, and after running DFS you find that there is exactly one back edge, but there are potentially an arbitrary number of forward and cross edges. As a function of the number of vertices n, what is the maximum number of simple cycles that G can have? (If you prefer, you can express your answer in terms of some other parameter, such as the total number of edges or the numbers of forward/cross edges.)

Homework 3: Greedy Algorithms and Dynamic Programming

Handed out Thu, Oct 15. Due at the start of class Thu, Oct 22.

Problem 1. The input to this problem consists of an ordered list of n words. The length of the ith word is w_i, that is the ith word takes up w_i spaces. (For simplicity assume that there are no spaces between words.) The goal is to break this ordered list of words into lines, this is called a *layout*. Note that you are *not* allowed to reorder the words. The length of a line is the sum of the lengths of the words on that line. The ideal line length is L. No line may be longer than L, but it may be shorter. The penalty for having a line of length k is $L - k$. The *total penalty* is defined to be –**fill in later**– (see below). The problem is to find a layout that minimizes the total penalty. Prove of disprove that the following greedy algorithm correctly solves this problem.

```
for (i = 1 to n) {
    Place the ith word on the current line if it fits
    else {
       start a new line and place the ith word on this line
    }
}
```

(a) Suppose that we set "–fill in later–" to "the **sum** of the individual penalties". Is the greedy algorithm optimal? Either give a proof or present a (short) counterexample.

(b) Suppose that we set "–fill in later–" to "the **maximum** of the individual penalties". Is the greedy algorithm optimal? Either give a proof or present a (short) counterexample.

(Hint: For one of the above alternatives, the greedy algorithm is optimal and for the other it is not.)

Problem 2. You are given an integer n and two sequences of nonnegative integers $R = \langle r_1, \ldots, r_n \rangle$ and $C = \langle c_1, \ldots, c_n \rangle$, such that $0 \le r_i, c_j \le n$, and $\sum_i r_i = \sum_j c_i$.

Given these sequences, you are asked to determine whether it is possible to place pawns on an $n \times n$ chess board such for $1 \le i, j \le n$, row i has exactly r_i pawns and column j has exactly c_j pawns (see Fig. 1). If so, specify which squares of the board contain pawns. (There may be many valid solutions, and your algorithm can generate any one of them.)

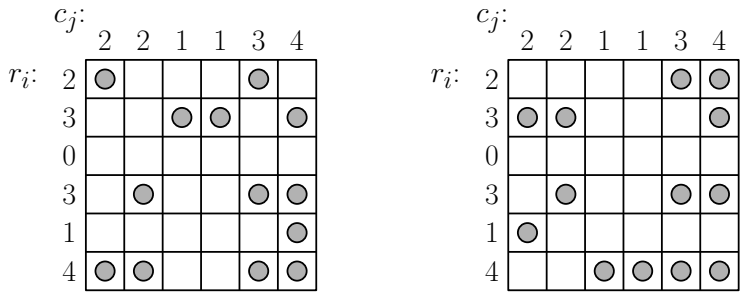


Figure 1: Two possible solutions to Challenge Problem 2 for the inputs $R = \langle 2, 3, 0, 3, 1, 4 \rangle$ and $C = \langle 2, 2, 1, 1, 3, 4 \rangle$.

Show that there exists an algorithm that solves this problem in $O(n^2)$ time. Prove that your algorithm is correct. (Hint: Greedy.)

Problem 3. You are working for a private space corporation that wants to configure its next space mission. The job is to fill a rocket with a set of scientific experiments to be run in space. There are n experiments that have been proposed as candidates to be on the mission, but their total weight is more than the rocket can lift. You have been asked to determine the best subset of experiments to launch on the mission.

For $1 \leq i \leq n$, let w_i denote the weight of the ith experiment. Let W denote the total weight that can be carried by the rocket. The objective is to determine the subset of experiments whose total weight comes as close to W without exceeding this value.

More formally, given the weights $\langle w_1, \ldots, w_n \rangle$ and you want to compute the subset $E \subset \{1, \ldots, n\}$, to maximize $\sum_{i \in E} w_i$ subject to the constraint $\sum_{i \in E} w_i \leq W$.

(a) Suggest a greedy approach for solving this problem. That is, you will sort the items according to some statistic, and then take as many items as possible (according to your ordering) as long as the total weight does not exceed W. How would you order the experiments? (No explanation is required. But please read (b) before trying to prove that your algorithm is optimal!)

(b) Show that your greedy algorithm is *not* optimal by showing that there is a set of of weights such that your algorithm fails to achieve the optimum.

(c) Let us add the assumption that the weights w_i and W are all integers. Present a dynamic programming algorithm for this problem that achieves the optimum. (Hint: For $0 \leq i \leq n$ and $0 \leq w \leq W$, define $P[i, w]$ to be a boolean array where $P[i, w] = \mathsf{true}$ if and only if there is a subset of $\{w_1, \ldots, w_i\}$ whose total weight sums to w.) Derive the running time of your algorithm and justify its correctness. Note that the running time will depend on W.

(d) Show that your greedy algorithm is not that bad after all, by proving that if the optimum algorithm achieves a total weight of W_O, your greedy algorithm will achieve a total weight of $W_G \geq W_O/2$.

Challenge Problem. Challenge problems count for extra credit points. These additional points are factored in only after the final cutoffs have been set, and can only increase your final grade.

Consider the following hiking-path problem. You are given n people that start at one end of a hiking path and want to walk to the other end. The ith person walks at speed $s_i > 0$. The goal is to get all the people from one end of the path to the other in the minimum time subject to the following constraints:

- It is dark, and any party that crosses the path must carry a flashlight
- The path is narrow, and maximum of two people can walk along the path any one time
- The group has only have one flashlight, which must be shared by everyone
- The flashlight must be walked back and forth along the path—it cannot be thrown
- When a pair walks together, they move at the speed of the slower person

Present an efficient algorithm to compute the minimum time needed to get the group of people across the path. You *must* provide a proof of correctness.

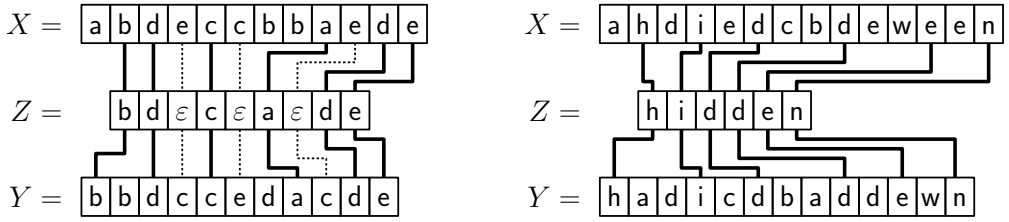
Homework 4: Dynamic Programming and Network Flows

Handed out Thu, Nov 5. Due at the start of class Tue, Nov 17. Late homeworks are not accepted, but you may drop your lowest homework score.

Problem 1. This problem involves some simple variants of the LCS problem. In each case you are given two sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ as input.

(a) It is sometimes useful to allow mismatches in the LCS, but at a penalty. Suppose, for example, that the alphabet is $\Sigma = \{a, b, c, d, e\}$. Your scanning equipment sometimes confuses c's and e's. When evaluating the length of a common subsequence, we allow c's to be matched with e's (and vice versa), but we penalize each such match by counting it as just *half* a character.

In the LCS, we show these special half-character matches with the symbol "ε". This character can match either a c or an e. An example is shown in Fig. 1(a). The standard LCS is $\langle bdccade \rangle$ of length 7. By allowing matches between c's and e's, we have a *generalized LCS* $Z = \langle bd\varepsilon c\varepsilon a\varepsilon de \rangle$ of length $6 + 3/2 = 7.5$.



(a) (b)

Figure 1: Variants on the LCS problem

Present a DP algorithm for computing the length of the generalized LCS of two input strings X and Y. (It is sufficient to present just the recursive rule, not a complete algorithm.) Briefly explain. What is the running time of your algorithm (if you had implemented it)?

(b) The FBI is developing a new document analyzer that is looking for hidden messages in documents. The criminals conceal the same message in multiple documents, but they hide the secret message by embedding it as a subsequence. To make these concealed messages harder to find, whenever they place a hidden character, they make sure that two consecutive characters of the hidden message are never consecutive in the document.

The FBI asks you to design an algorithm to find the LCS within a pair of documents, but subject to the condition that consecutive characters of the LCS do not appear consecutively either in X or in Y. An example is given in Fig. 1(b). The standard LCS is (I believe) $\langle adicbdewn \rangle$ (of length 9), but this involves many consecutive characters of X and Y. The longest LCS without consecutive characters is (I believe) $Z = \langle hidden \rangle$ (of length 6).

Present a DP algorithm for computing the length of the nonconsecutive LCS of two input strings X and Y. (It is sufficient to present just the recursive rule, not a complete algorithm.) Briefly explain. What is the running time of your algorithm (if you had implemented it)?

Problem 2. The following problem arises in image processing and compression. You are given a black and white digitized picture P in the form a of a two dimensional $n \times n$ matrix P. For $1 \leq i, j \leq n$, $P[i, j]$ is 0 if the pixel on row i and column j is black and 1 if it is white. We want to decompose the picture into a minimum number of *monochromatic rectangles*, which means that each rectangle is either all white or all black.

The decomposition must be performed in the following hierarchical manner. Starting with the full image as the starting rectangle, we can split it into two rectangles either by a vertical line or a horizontal line that cuts through the entire rectangle. After this, we can split each of these rectangles again either by a vertical or horizontal line that cuts through the entire rectangles, and so on. The process stops when a rectangle is either all white or all black. An example of such a decomposition is shown in Fig. 2(c). The question is where to place these cuts so that the final number of rectangles is minimized.

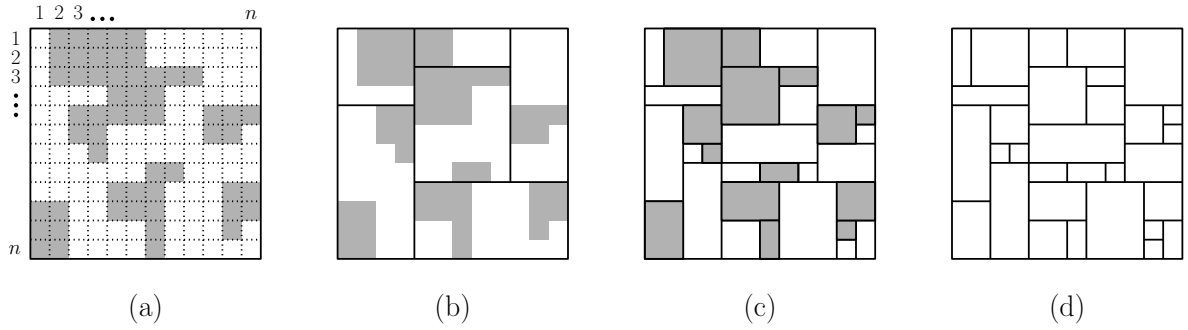


Figure 2: (a) the image rectangle, (b) a partial decomposition after five cuts, (c) the final decomposition into 31 monochromatic rectangles, (d) the cuts of the final decomposition.

(a) Derive a (recursive) dynamic programming rule, which given an image P, determines the minimum number of rectangles in a hierarchical monochromatic decomposition. (Briefly justify your algorithm's correctness.)

(Hint: The subproblems are associated with rectangles of the original image. For $1 \leq i_0 \leq i_1 \leq n$ and $1 \leq j_0 \leq j_1 \leq n$, let $R[i_0, i_1, j_0, j_1]$ be the minimum number of monochromatic rectangles in a hierarchical decomposition of the rectangular portion of the image of rows $\langle i_0, \ldots, i_1 \rangle$ and columns $\langle j_0, \ldots, j_1 \rangle$.)

(b) Present an implementation of recursive rule of part (a). You may assume that you have access to a function $\mathsf{monochrome}(i_0, i_1, j_0, j_1)$ that returns true if the image rectangle $P[i, j]$, for $i_0 \leq i \leq i_1$ and $j_0 \leq j \leq j_1$ is monochromatic (all white or all black) in $O(1)$ time. (Hint: Memoization is probably simpler than a bottom-up computation in this case, but either is acceptable.)

(c) Derive the running time of your algorithm.

Problem 3. Let's return to the typesetting problem from Homework 3. Recall that we are given a line of length L and a paragraph consisting of a sequence of words whose lengths are $W = \langle w_1, \ldots, w_n \rangle$. (We assume that $w_i \leq L$ for all i.) We are to place words in order along each line subject to the condition that the sum of word lengths on any line does not exceed L. The *penalty* for each line is defined to be the difference between the sum of word lengths on this line and L. The objective is to place the words to minimize the *maximum penalty* over all the lines (see Fig. 3(a)).

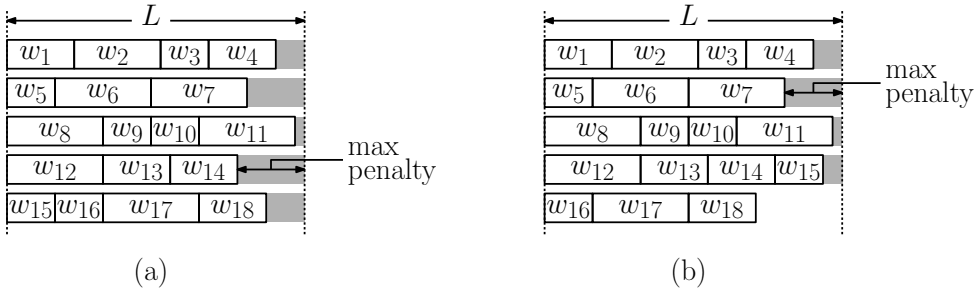


Figure 3: Optimal typesetting of words to minimize the maximum penalty.

In Homework 3 we showed that a greedy strategy is not optimal. In this problem we will show that this problem can be solved optimally by dynamic programming.

(a) Derive a (recursive) dynamic programming rule, which given L and the word sequence W, determines the layout that minimizes the maximum penalty (see Fig. 3(a)). Actually, I don't care about the layout, just the final *value* of the maximum penalty, and I don't need a full algorithm, just the recursive DP formulation.

Briefly justify the correctness of your algorithm and derive its running time (if it were implemented). It may help to imagine that you have access to a function $W(i, j)$ that returns the sum of word lengths from w_i up to w_j (assuming that $1 \leq i \leq j \leq n$) that runs in constant time.

(b) In practice, when laying out a paragraph we do not care whether the last line is "ragged." Modify your solution from part (a) to compute the layout that minimizes the maximum layout *excluding* the last line. (For example, by this metric the layout shown in Fig. 3(b) has a lower cost than the layout from Fig. 3(a).)

As in part (a), briefly justify the correctness of your algorithm and derive its running time.

Problem 4. The Ford-Fulkerson algorithm operates by finding an augmenting path in the residual network. The effect of pushing flow along this path may result in flows increasing on some edges and decreasing on others. Consider instead an algorithm that *only increases* flows along the edges of some path from s to t. (There may generally be many such paths, and the algorithm is free to chose any of them.) We call this the *nondecreasing flow algorithm*.

Show that the nondecreasing flow algorithm can be *arbitrarily bad*. In particular, given any positive integer $b > 1$, give an example of an s-t network G such that the ratio between the optimum flow in G and the flow generated by the nondecreasing algorithm is at least as large

as b. (The structure of the network will depend of course on b. You may give your example for a specific value of b, but it should be easy to see how to generalize it to arbitrary values of b. Remember that you may select the sequence of paths along which augmentations are to be performed.)

In either case, explain how your counterexample works.

Challenge problems count for extra credit points. These additional points are factored in only after the final cutoffs have been set, and can only increase your final grade.

Challenge Problem 1. In Problem 3, we suggested using a function $W(i, j)$ that return the sum of word lengths w_i through w_j in constant time. Show that, given the word lengths $\langle w_1, \ldots, w_n \rangle$, after $O(n)$ preprocessing time it is possible to build a data structure from which $W(i, j)$ can be computed in $O(1)$ time. (If you do not see how to do this, you might try for a solution in which the preprocessing time is increased to $O(n^2)$ and/or the access time is increased to $O(\log n)$.)

Challenge Problem 2. In Problem 2, we suggested using a function $\mathsf{monochrome}(i_0, i_1, j_0, j_1)$ that returns true if the given rectangle is monochromatic. Show that, given the $n \times n$ input image, after $O(n^2)$ preprocessing time it is possible to build a data structure from which $\mathsf{monochrome}(i_0, i_1, j_0, j_1)$ can be computed in $O(1)$ time. (If you do not see how to do this, you might try for a solution in which the preprocessing time is increased to $O(n^4)$ and/or the access time is increased to $O(\log n)$.)

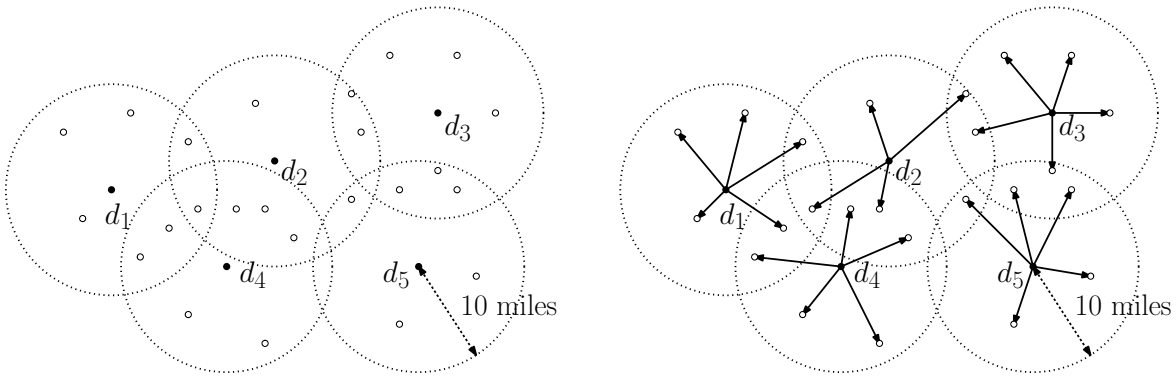
(Hint: Try to generalize the solution to Challenge Problem 1 to a 2-dimensional setting.)

Homework 5: Network Flow and NP-Completeness (Part 1)

Handed out Tue, Nov 24. Due at the start of class Thu, Dec 3. (Part 2 will be handed out on Tue, Dec 1 and will be due Thu, Dec 10.) Late homeworks are not accepted, but you may drop your lowest homework score.

Problem 1. Your friend has a new drone delivery startup, and he has asked you to help him by designing software to assist with scheduling deliveries.

- There are m drone stations throughout the city. For $1 \leq i \leq m$, let $d_i = (d_{i,x}, d_{i,y})$ denote the (x, y) coordinates of the ith drone station (see Fig. 1(a)). Due to FAA regulations, each drone station can launch no more than *5 drones* each day.
- There are n customers expecting to receive a package this day. For $1 \leq j \leq n$, let $c_i = (c_{j,x}, c_{j,y})$ denote the (x, y) coordinates of the jth customer. (You may assume that no two customers occupy the same location, and each customer is expected exactly one delivery.)
- Each drone station is attached to complete warehouse, so in theory a drone from any station can deliver the desired package to any customer. However, because of fuel limitations, each drone can make a delivery only within a 10 mile radius of the station (see Fig. 1(a)). That is, station i can only deliver packages to those customers j such that $\text{dist}(d_i, c_j) \leq 10$.



(a) (b)

Figure 1: Drone delivery service. Black points are drone stations and hollow points are customers: (a) Input and (b) Possible solution.

Your algorithm is given the coordinates of the m drone stations and the coordinates of the n customers. The problem is to determine the maximum number of deliveries that can be made (ideally all n of them), subject to the constraints given above (see Fig. 1(b)).

(Hint: Reduce to network flow or some variant like circulations with flow demands. Give both the reduction and a proof that your reduction is correct. Given the flow output, explain how to determine the set of customers that station i will ship to.)

Problem 2. Most NP-complete problems are expressed as *decision problems*, where the answer is simply "yes" or "no," but in practice a user wants to know *why* the answer is "yes" or "no." In this problem, we will show that if we are given access to an algorithm for a decision problem, it is often possible to use this to obtain the entire answer.

(a) **Hamiltonian Cycle:** Given an undirected graph $G = (V, E)$, does there exist a cycle that visits every vertex of graph exactly once?

Suppose that we had a function Hamiltonian_Cycle(G), which (by some miracle) ran in polynomial time and returns true if G has a Hamiltonian cycle and false otherwise. Show that if G has a Hamiltonian cycle, then it is possible to use this function (as a black box) to compute the sequence of vertices on the Hamiltonian cycle in polynomial time.

(Let $n = |V|$ and $m = |E|$. For full credit, solve this problem using $O(m)$ calls to the function. For partial credit, any polynomial number of calls is allowed.)

(b) **3-Colorable:** Given an undirected graph $G = (V, E)$ can the vertices of G be labeled with three colors (say, 1, 2, and 3) such that no edge is incident to two vertices of the same color?

Suppose that we had a function Three_Color(G), which (by some miracle) ran in polynomial time and returns true if G is 3-colorable and false otherwise. Show that if G is 3-colorable, then it is possible to use this function (as a black box) to determine the assignment of colors to the vertices.

(Let $n = |V|$ and $m = |E|$. For full credit, solve this problem using $O(n)$ calls to the function. For partial credit, any polynomial number of calls is allowed.)

CMSC 451:Fall 2015 Dave Mount

Homework 5: Network Flow and NP-Completeness (Part 2)

Handed out Tue, Dec 1. Due at the start of class Thu, Dec 10. (Remember that Part 1 is due on Thu, Dec 3.) Late homeworks are not accepted, but you may drop your lowest homework score.

Problem 3. A graph is said to be *k-weird* if it has both a clique of size k and an independent set of size k. Given a graph $G = (V, E)$ and an integer k, the k-weird problem (kWP) is that of determining whether G is k-weird. (For example, the graph in Fig. 1(a) is k-weird.)

(a) Show that kWP is in NP. (Either give a polynomial time verification procedure or present a nondeterministic polynomial time algorithm.)

(b) Prove that kWP is NP-hard. (Hint: Reduction from either the clique problem or the independent set problem.)

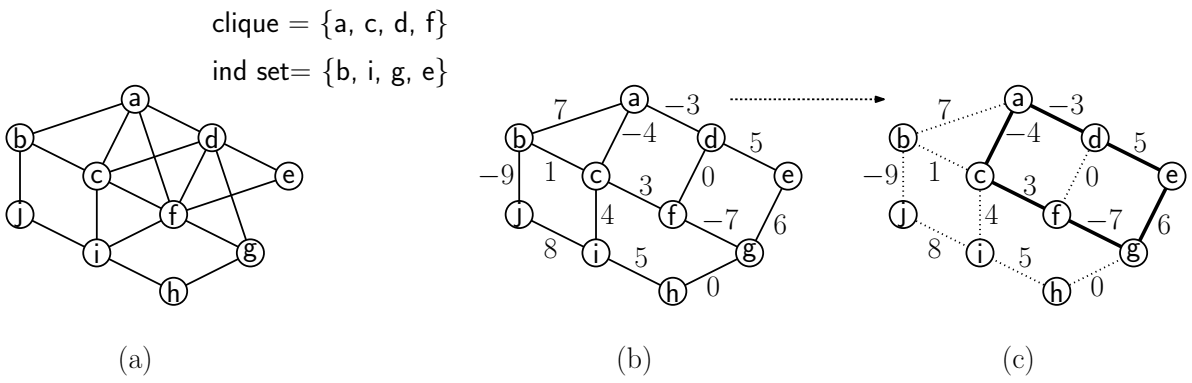


Figure 1: Problems 3 and 4.

Problem 4. Consider the following problem, called the *zero cycle problem* (ZC). You are given an undirected graph $G = (V, E)$ with integer weights on its edges (which may be positive, negative or zero). The question is whether there exists a simple cycle consisting of at least three edges whose total weight is zero? (For example, the graph shown in Fig. 1(b) has the zero cycle shown in Fig. 1(c).)

(a) Show that ZC is in NP. (Either give a polynomial time verification procedure or present a nondeterministic polynomial time algorithm.)

(b) Prove that ZC is NP-hard. (Hint: Reduction from Hamiltonian cycle.)

Challenge Problem 1. Back in the old days, there were things called "video stores". A local video store owner received a shipment of a large number of video tapes. Among these is one cursed tape. Exactly seven days after viewing this tape, the viewer dies a horrible death without warning. (If you don't believe this, see the 2002 movie "The Ring".)

The video store manager wants to determine which of the tapes is the cursed one before the upcoming big sale in eight days from now. He has managed to find a number of foolish people,

who are willing to test the tapes. Each tester will be given some subset of the tapes to view on the first day, and will then wait nervously for seven days for the final results to develop. (There is no limit on the number of tapes that may be assigned to one tester, and the same tape may be viewed by many different testers.)

Suppose that there are n tapes total. The manager realizes that he can determine the cursed tape by arranging for n testers, each of whom will view exactly one tape. A smart clerk informs him that he can achieve the same result with fewer testers, if he cleverly arranges which testers see which tapes.

(a) What is the minimum number of tape testers needed to determine which tape is cursed, and how is the test conducted? (The greedy store manager does not care how many of the testers dies in the process; he just wants to minimize the number of testers that he needs to pay.)

(b) If the deadly tape is random among the set of n tapes, then what is the expected number of testers that survive the process?

Challenge Problem 2. (This puzzle came from fellow classmate Joe Brosnihan.)

After a wild night of merry-making, you and nine of your friends wind up in jail. The jailer decides to play a game with all of you.

Here is how the game is played. You are each assigned an integer 0-9 by the jailer, but no one is told what their number is. Note that the assigned numbers are completely arbitrary, and there may be duplicates. Each person is led into a separate room where they are shown the nine numbers of their other friends. (There is no particular order in which these numbers are shown, and so it is not possible to infer which number is associated with which friend.)

While still in these separate rooms, each person is then asked to guess the value of their own number. Each person stays in their separate room until all the guesses are finished. There is no way to communicate with your friends between the time you enter the room and the time you guess your number. If at least one of you guesses their number correctly, then you all go free. If not, you will be in jail for a long time.

You have all been informed of how the game will be played. The objective is for you and your friends to devise a strategy beforehand, so that *at least one* person is guaranteed to guess their number correctly. (Hint: There is no trick. The solution involves simple logic.)

Practice Problems for the Midterm

The midterm will be on **Tue, Oct 27**. The exam will be closed-book and closed-notes, but you will be allowed one cheat-sheet (front and back).

Disclaimer: These are practice problems, which have been taken from old homeworks and exams. They **do not** reflect the actual length, difficulty, or coverage for the exam.

Problem 0. You should expect one problem in which you will be asked to work an example of one of the algorithms we have presented in class on a short example.

Problem 1. Short answer questions.

(a) Suppose that in the Gale-Shapley algorithm, a *man's* proposal has just been accepted. **True or false:** He is guaranteed to remain engaged (to this person or someone else) for the remainder of the algorithm's execution.

(b) As a function of n, what is the asymptotic running time of the following function? (Express your running time using Θ notation.)

```
void f(int n) {
  i = n;
  while (i > 0) {
    for (j = 1 to i) print("hello!\n");
    i = i/2;
  }
}
```

(c) List the following functions in increasing asymptotic order. If two functions are asymptotically equivalent, then indicate this.

$$\text{(a)} n^{\lg 4} \quad \text{(b)} 2^{\lg n} \quad \text{(c)} 2^{(2 \lg n)} \quad \text{(d)} \min(2^{20} n^2, n^3)$$

Remember that "lg" means logarithm base 2.

(d) Let G be a undirected connected graph. Recall that a *cut edge* is an edge whose removal causes G to be disconnected. For each of the following, is it True or False?

(i) There is a graph G that has a cut vertex but no cut edges.

(ii) There is a graph G that has a cut edge but no cut vertices.

(e) What is the maximum number of edges in an undirected graph with n vertices, in which each vertex has degree at most k?

(f) You are given a DAG G with n vertices and m edges. As a function of n and m, what is the maximum number of each of the following that can arise in a DFS of G? (No explanation is required.)

(i) tree edges

(ii) back edges

(iii) forward edges

(iv) cross edges

Remember, the digraph is a DAG.

Problem 2. Recall the following problem, called the *Interval Scheduling Problem*. We are given a set $R = \{x_1, \ldots, x_n\}$ of n *activity requests*, each of which has a given start and finish time, $[s_i, f_i]$. The objective is to compute the maximum number of activities whose corresponding intervals do not overlap. In class we presented an greedy algorithm that solves this problem. We will consider some alternatives here.

(a) **Earliest Activity First** (EAF): Schedule the activity with the earliest start time. Remove all activities that overlap it. Repeat until no more activities remain.

Give an example to show that EAF is not optimal. Your example should show not only that it is not optimal, but its approximation ratio can be *arbitrarily high*.

(b) **Shortest Activity First** (SAF): Schedule the activity with the smallest duration $(f_i - s_i)$. Remove all activities that overlap it. Repeat until no more activities remain.

Give an example to show that SAF is not optimal.

(c) Prove that SAF has an approximation ratio of 2, that is, it schedules at least half as many activities as the optimal algorithm.

Problem 3. Professor Farnsworth drives from College Park to Miami Florida along I-95. He starts with a full tank and can go for 100 miles on a full tank. Let $x_1 < \ldots < x_n$ denote the locations of the various gas stations along the way, measured in miles from College Park (see Fig. 1). Present an algorithm that determines the *fewest number* of gas stations he needs to stop at to make it to Miami without running out of gas along the way. Give a short *proof of the correctness*.

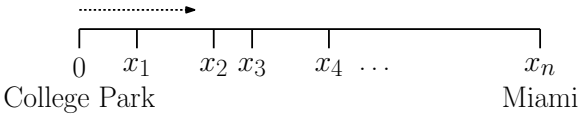


Figure 1: Example for Problem 3.

Problem 4. A pharmacist has W pills and n empty bottles. Let b_i denote the capacity of bottle i, that is, the number of pills it can hold. Let v_i denote the cost of purchasing bottle i. The objective is find the least expensive combination of bottles into which to place all W pills.

Describe a greedy algorithm, which given the number of pills W, the bottle capacities b_i, and the bottle costs v_i, determines the most inexpensive set of bottles needed to store all the pills. Assume that you pay only for the *fraction* of the bottle that is used. For example, if the ith bottle is half filled with pills, you pay only $v_i/2$. (This assumption is very important.) Prove the correctness of your solution.

Problem 5. Given a DAG $G = (V, E)$, a path of G is said to be *tail-maximal* if it ends at a vertex with outdegree zero. (If u is a vertex of outdegree zero then the path consisting of just u itself is a tail-maximal path.) Describe an $O(n + m)$ algorithm which, given a DAG, $G = (V, E)$,

computes for each vertex u the number of distinct tail-maximal paths that originate at u. (For example, your algorithm can compute an array $P[1..n]$ such that $P[u]$ contains the desired number of paths for vertex u.)

(Hint: Use DFS.)

Problem 6. Let $G = (V, E)$ be an undirected graph. Write an $O(n + m)$ time algorithm to determine whether it is possible to direct the edges of G such that the indegree of every vertex is at least one. If it is possible, then your algorithm should show a way to do this. (Hint: Use DFS.)

Problem 7.

(a) Describe a greedy algorithm for making change consisting of quarters, dimes, nickels, and pennies. Assume that the input is given as the number of cents. Prove that your algorithm yields the minimum number of coins. (Hint: For partial credit, prove it for the simpler binary sequence of denominations: $1, 2, 4, 8, 16$.)

(b) The greedy algorithm is not optimum for all choices of coin denominations. Give a set of coin denominations (including a 1 cent coin) for which the greedy algorithm does not always give the minimum number of coins. Explain briefly.

Problem 8. You operate a business that has two offices, one in Washington DC and one in Los Angeles. Each week, you need to decide whether you want to work in the DC office or the LA office. Depending on the week, your business makes more profit by having you at one office or the other. You are given a table of weekly profits, based on your location. Here is an example:

Week	1	2	3	4	5
DC	\$400	\$100	\$200	\$50	\$1100
LA	\$210	\$900	\$100	\$1500	\$20

Clearly, you would prefer to work at the location where you get the greater profit, but here is the catch. It costs \$1000 to fly from one office to the other. (For example, if you do the first job in DC, the next three in LA, and the last in DC, the total profit will be $\$400 - \$1000 + (\$900 + \$100 + \$1500) - \$1000 + \$1100 = \$2000$.

You are given two lists of length n, DC$[1..n]$ and LA$[1..n]$, where DC$[i]$ is the profit for spending week i in DC, and LA$[i]$ is the profit for spending week i in LA. Present an efficient algorithm, which given these two arrays, determines your maximum overall profit. You must *start* and *end* in DC, but you may travel back and forth any number of times. Briefly justify your algorithm's correctness and derive its running time.

Hint: $O(n)$ time is possible using dynamic programming. It suffices to give just the recursive rule. You will need to find a way to keep track of where you were the previous week.

Problem 9. The objective of this problem is to write a dynamic programming algorithm to play a game. Two players, called Jen and Ben alternate in taking moves, with Jen always going first. Initially the board consists of three piles of diamonds, which we denote (A, B, C), meaning

that there are A diamonds in the first pile, B in the second pile, and C in the third. The board always consists of three numbers that are nonnegative. During a move a player can do any one of the following:

(1) Remove 1 diamond from pile 1.
(2) Remove either 1 or 2 diamonds from pile 2.
(3) Remove either 2 or 3 diamonds from pile 3.

The first player who cannot make a move loses. (And the winner gets all the diamonds.) That is, if it is a player's turn to move and the board is either $(0, 0, 0)$ or $(0, 0, 1)$ then he/she loses.

Given the initial configuration, (A, B, C), and with the assumptions that Jen plays first and both players play as well as possible, determine which of the two players can force a win. (Since there is no element of chance, and the game is finite in length, one of the two can always force a win.)

For example, suppose that the initial board is $(0, 1, 4)$. In this case Jen can force a win. She uses rule (3) to remove 2 diamonds from pile 3, resulting in the board $(0, 1, 2)$. Ben's only choices are to remove 1 from pile 2 or 2 from pile 3, resulting in the possible boards $(0, 0, 2)$ and $(0, 1, 0)$. In either case, Jen can remove the final diamonds (using either rules (3) or (2), respectively) leaving Ben without a move.

(a) Derive a (recursive) dynamic programming rule to determine the winner, given the initial board (A, B, C). (Be sure to include a description of the basis cases.) Justify the correctness of your formulation. (For this part I do not want a full algorithm, just the recursive rule.) You are *not* allowed to use mathematical results from the game of Nim to find a short-cut solution.

(b) Present an implementation of recursive rule of part (a). (You may use memoization or the bottom-up method.) Express your running time as a function of A, B, and C.

Problem 10. A thief is robbing a store. There are n items in the store. The ith item has a weight of w_i and a dollar value v_i. The thief has a knapsack that can hold a total of W units of weights before ripping open. All weights and values are positive integers. The problem is to determine the greatest value of goods that the thief can carry away in his knapsack. The thief may either leave an object or take the entire object. (So, he cannot steal a fraction of an object for a fraction of the value and weight.)

(a) Give a recursive dynamic programming rule for this problem. (Hint: For $0 \leq i \leq n$ and $0 \leq u \leq W$, let $V[i, u]$ be the maximum value that the thief could steal assuming that he may select only among the first i objects and that he has a knapsack of capacity u.)

(b) Give the pseudo-code for a dynamic programming algorithm to solve this problem. Your algorithm need not determine the actual items to be stolen, just the maximum value. Your algorithm should run in $O(nW)$ time and $O(nW)$ space.

Problem 11. Given a graph $G = (V, E)$, a subset of vertices $V' \subseteq V$ is called a *dominating set* if every vertex of G is either in the set V' or is a neighbor of a vertex in V'. In the *dominating*

set problem you are given a graph G and the objective is to compute a dominating set of minmum cardinality. (For example, in the graph shown in Fig. 2 there is a dominating set of size two, as indicated by the shaded nodes.)

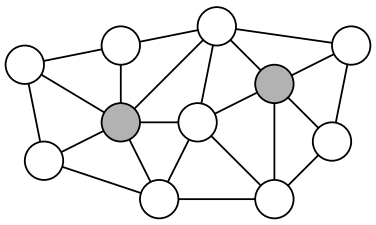


Figure 2: Example for Problem 11.

(a) Describe a greedy algorithm for computing a dominating set of minimum size. (Note, please read part (b) before trying to prove that your algorithm is optimal.) Your algorithm should run in time that is polynomial in n, where $n = |V|$.

(b) Present an example to show that your greedy algorithm is not optimal.

(c) Show that your greedy algorithm achieves an approximation factor of $\ln n$, where $n = |V|$. That is, if there exists a dominating set of size k, then your greedy algorithm will find a dominating set of size at most $k \cdot \ln n$.

Problem 12. Suppose you have a sequence of points $X = \langle x_1, \ldots, x_n \rangle$ sorted from left to right along a line. The distance between two points x_i and x_j is just their absolute difference $|x_j - x_i|$. The *bottleneck TSP problem* is the following: Find a cycle that visits each point exactly once, such that *maximum* distance traveled between any two consecutive points is as small as possible.

Consider the following *alternating heuristic* for this problem: Travel from x_1 to x_n, skipping every other point. Then return from x_n to x_1 visiting the skipped points. (An example is shown in Fig. 3. The final cost is the longest segment traversed, which is the segment of length 7 between the points at positions 9 and 16.)

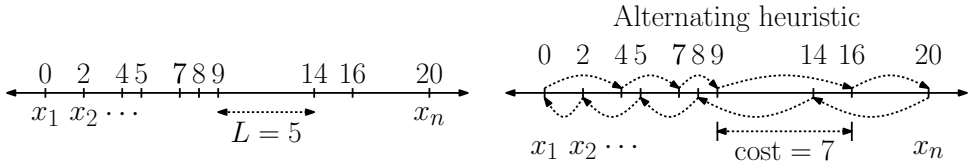


Figure 3: Example for Problem 12.

Prove that this heuristic provides a factor-2 approximation to the bottleneck TSP problem (for points on a line). Hint: Let L be the maximum distance between any two consecutive points. Relate the costs of the optimum path and the heuristic path to L.

(Note: I believe that the alternating heuristic is actually optimal, but it is much easier to prove the factor-2 approximation bound.)

Just for Fun. (If you get bored studying for the exam, you can waste your time thinking about this puzzle.)

You and your roommate are contestants in a game of wits. A guy dressed up like the devil gives each of you a card with a positive integer written on it. Each of you cannot see the other person's card, but he tells you that the difference in the two numbers is 1. For example, if your number is "53", then you know that your roommate may have the number "52"or "54", but you don't know which. Otherwise, all you know about the possible numbers is that they must be 1 or larger.

This devilish fellow tells you that if either of you can guess the number on your roommate's card, you will receive a "shiney fiddle made of gold" (proving that the devil listened to country rock music from the 1970's). Otherwise, you and your roommate will have to pay the devil's $2.50 parking bill. You and your roommate are pretty smart, so you take the devil's challenge. The devil starts his game:

- The devil asks you whether you know the number on your roommate's card. After thinking, you answer "no".
- The devil then asks your roommate whether he/she knows your number. After thinking, your roommate answers "no".
- The devil is nice enough to give you another chance. After thinking, you again say "no".
- The devil gives your roommate another chance. After thinking, your roommate answers "no".

At this point, the devil gives up in disgust, and asks you two to cough up the $2.50. Suddenly, you exclaim, "I know the number on my roommate's card!" Your roommate says the same, and you both get your golden fiddles.

Explain how each of you determined the number on your card. (Hint: There is no trick. Just simple logic, but this works only because the devil was foolish enough to give you a particularly nice pair of numbers.)

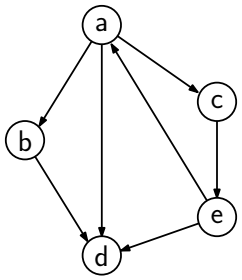
CMSC 451:Fall 2015 Dave Mount

Midterm Exam

This exam is closed-book and closed-notes. You may use one sheet of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (10 points) Show the result of running DFS on the digraph shown below using the algorithm given in class. (Whenever you have a choice which vertex to visit next, take the lowest vertex in alphabetical order.)

Label each node u with its discovery and finish times ($d[u]/f[u]$). As in the lecture notes, show tree edges with solid lines and the other edges with dashed lines. Label these other edges as *forward*, *cross*, or *back* edges. (Show only the **final tree**, not the intermediate results.)



Problem 2. (25 points; 3–6 points each.) Short answer questions. Explanations are not required, but may be given for partial credit.

(a) Suppose that in the Gale-Shapley algorithm, a *woman* has just accepted a proposal.

True or false: She is guaranteed to remain engaged (to this person or someone else) for the remainder of the algorithm's execution.

(b) List the following functions in increasing asymptotic order. If two functions are asymptotically equivalent, then indicate this.

$$\text{(a) } n^{3/2} + n^{2/3} \quad \text{(b) } n(\lg n)^2 \quad \text{(c) } 4^{\lg n} \quad \text{(d) } \max(2000 \cdot n^2, n^3)$$

(Remember that "lg" means logarithm base 2.)

(c) Let G be a free tree (a connected, acyclic, undirected graph) with n vertices.

(i) As a function of n, what is the *minimum* number of cut vertices that G can have?

(ii) As a function of n, what is the *maximum* number of cut vertices that G can have?

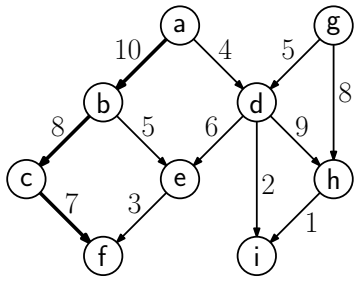
(d) Let P be a set of n points in space, and let p and q be the two *closest* points in P. Suppose that Gonzalez's algorithm (that is, the greedy k-center algorithm) is given p as the initial point and runs for n iterations. Where in the sequence will the point q be added? (Second? Near the median? Last? We have no way of knowing?)

(e) Dynamic programming solutions make use of the *Principle of Optimality*, which states that for the global problem to be solved optimally, the individual subproblems should be solved optimally as well.

Give an example of an optimization problem that *does not* satisfy the principle of optimality.

Problem 3. (20 points) You are given a directed graph $G = (V, E)$. Each edge (u, v) of this graph is associated with a positive numeric weight $w(u, v)$. The *cost* of a path is defined to be the sum of the weights of the edges along the path.

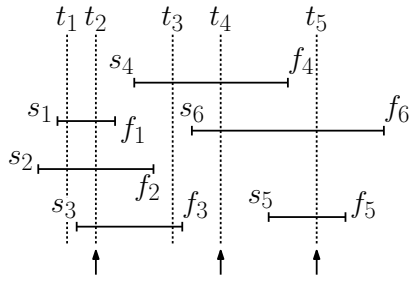
Suppose that G is a DAG. Present an efficient algorithm that computes the cost of the maximum cost path in G. For example, in the figure below the maximum cost path is $\langle a, b, c, f\rangle$ of total cost $10 + 8 + 7 = 25$. (You need only the compute the cost, **not** the actual path.)



Briefly justify your algorithm's correctness and derive its running time. (Hint: Use DFS.)

Problem 4. (25 points) This problem involves a variant of the pony-bombing problem from the homework. You are given a set I of m time intervals $[s_i, f_i]$, where $1 \le i \le m$. You are also given a set of n possible bomb times $T = \{t_1, \ldots, t_n\}$. We say that bomb j *hits* interval i if this bomb time lies within the interval, that is, $t_j \in [s_i, f_i]$. The objective is to determine the *minimum* number of bombs from T to hit every one of the intervals.

(The principal difference from the homework problem is that you cannot detonate a bomb whenever you like. It must come from a time in T. For example, in the figure below, if we could explode bombs whenever we wanted, we could hit all the intervals with bombs at f_1 and f_4. However, for this problem three bombs are needed, for example, at times t_2, t_4, and t_5.)



Present an efficient algorithm to determine a minimum set of bombs to hit all the intervals. (Hint: Modify the greedy solution from the homework. The coding can be tricky, so first *explain* your idea in English, then give the details.) You may assume that every interval is hit by at least one bomb, therefore a solution always exists. Running time is not a big issue; I will accept any (correct!) algorithm that runs in polynomial time.

Justify your algorithm's correctness. (If you like, you can explain how to modify the correctness proof from the homework.)

Problem 5. (20 points) Given a sequence $X = \langle x_1, \ldots, x_n \rangle$ an *increasing subsequence* is any subsequence of elements of X that are in strictly increasing order. The *longest increasing subsequence* (LIS) is the increasing subsequence of maximum length. For example, the LIS of $X = \langle 10, 22, 9, 33, 21, 50, 41, 60, 80 \rangle$ is $\langle 10, 22, 33, 50, 60, 80 \rangle$, and its length is 6. Present an efficient algorithm, which given an n-element sequence computes the length of its LIS. (It suffices to compute just the length, **not** the actual subsequence.)

Hint: Use Dynamic Programming. For $1 \leq i \leq n$, let $L[i]$ denote the length of the LIS of $\langle x_1, \ldots, x_i \rangle$ with the constraint that the LIS *must end* with x_i. Show how to compute this array in $O(n^2)$ time. Given the array, what is the final solution to the LIS problem? (I do not need a complete algorithm. It is sufficient to present the recursive formula for computing $L[i]$.)

CMSC 451:Fall 2015 Dave Mount

Practice Problems for the Final Exam

The final will be **Tue, Dec 15, 8:00-10:00am** in our usual classroom. The exam will be closed-book and closed-notes, but you will be allowed two sheets of notes (front and back of each sheet). **Disclaimer:** These are practice problems, which have been taken from old homeworks and exams. They do not necessarily reflect the actual length, difficulty, or coverage for the exam. For example, we have covered some topics this year that were not covered in previous semesters. So, just because a topic is not covered here, do not assume it will not be on the exam.

Problem 0. You should expect one problem in which you will be asked to work an example of one of the algorithms or NP-complete reductions that we have presented in class.

Problem 1. Short answer questions.

(a) Suppose you are given an undirected graph which has n vertices, and each vertex has exactly six incident edges. As a function of n, what is the total number edges in this graph? (Give an exact answer for full credit, asymptotic answer for partial credit.)

(b) What is the longest common subsequence of the strings $X = \langle ababa \rangle$ and $Y = \langle babab \rangle$? (If there are multiple, list any one. I don't need to see a trace of the algorithm, just the final answer.)

(c) Given a flow network G, let (X, Y) denote a cut. Let c denote the sum of the edge capacities of all edges (x, y), where $x \in X$ and $y \in Y$. What can be said about the maximum flow in G?

(i) The value of the maximum flow in G is at most c (but may be smaller)

(ii) The value of the maximum flow in G is at least c (but may be larger)

(iii) The value of the maximum flow in G is equal to c.

(iv) We cannot say anything about the maximum flow, because we failed to consider the capacities of the edges going from Y to X when defining c.

(d) True or False: It is possible to determine in polynomial time whether a graph G has an independent set of size 100.

Problem 2. Recall that in the longest common subsequence (LCS) problem the input consists of two strings $X = \langle x_1, \ldots x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ and the the objective is to compute the longest string that is a subsequence of both X and Y.

(a) (LCS with wild cards) Each of the strings X and Y may contain a special character "?", which is allowed to match *any single character* of the other string, *except* another wild-card character (see Fig 1(a)).

(b) (LCS with swaps) Any two consecutive characters of either string are allowed to be swapped before matching in the LCS (see Fig 1(b)).

In all cases, your revised rule should admit an $O(mn)$ time solution.

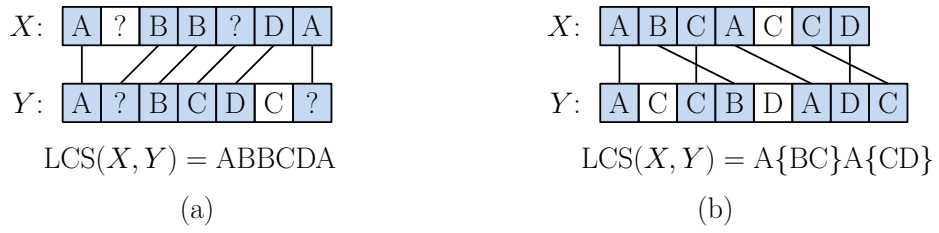


Figure 1: LCS variants.

Problem 3. A shipping company wants to ship n objects of weights $\{w_1, \ldots, w_n\}$. Each weight is a positive integer. The company wants to partition these objects between two different ships, so that the total weight of the two ships is as similar as possible. In particular, if W_1 is the total weight of objects on Ship 1, and W_2 is the total weight on Ship 2, then the objective is to minimize the *weight ratio*,

$$\frac{\max(W_1, W_2)}{\min(W_1, W_2)}.$$

Observe that this ratio is never smaller than 1, and it equals 1 if and only if the two ships are carrying identical total weights.

For example, suppose the inputs are $w_1 = 40$, $w_2 = 70$, $w_3 = 20$, $w_4 = 30$, $w_5 = 60$, and $w_6 = 50$. If we partition the elements as Ship-1 $= \{2, 5\}$ and Ship-2 $= \{1, 3, 4, 6\}$, then the total weights are $70 + 60 = 130$ and $40 + 20 + 30 + 50 = 140$. The final weight ratio is $140/130 \approx 1.077$.

This is called the *Partition Problem*. Present an efficient algorithm, which given the set of weights $\{w_1, \ldots, w_n\}$, computes the optimum weight ratio. You can express your running time as a function of both n and the total weight $W = \sum_{i=1}^{n} w_i$.

(Hints: Use Dynamic Programming. You are not required to give the entire DP algorithm, just a recursive formulation. You need only compute the optimum weight ratio, not the actual partition. Justify your algorithm's correctness and derive its running time. Note that $O(n \cdot W)$ time is possible. It suffices to focus on computing the total weight carried by just one of the ships, since the other must carry all the remaining weight.)

By the way, the Partition Problem is NP-hard. The above algorithm is only *pseudo-polynomial time*, because the running time depends on the magnitude of the numbers, not on the logarithm of their magnitude.

Problem 4. You are given a directed network $G = (V, E)$ with a root node r and a set of terminal nodes $T = \{t_1, \ldots, t_k\}$. Present a polynomial time algorithm to determine the minimum number of edges to remove so that there is no path from r to any of the terminals (see Fig 2). (Hint: Use network flow.) Prove that your algorithm is correct.

Problem 5. (Bucket redistribution) You are given a collection of n blue buckets, and n red buckets. These are denoted B_i and R_i for $0 \le i \le n - 1$. Initially each of the blue buckets contains some number of balls and each red bucket is empty. The objective is to transfer all the balls from the blue buckets into the red buckets, subject to some rules.

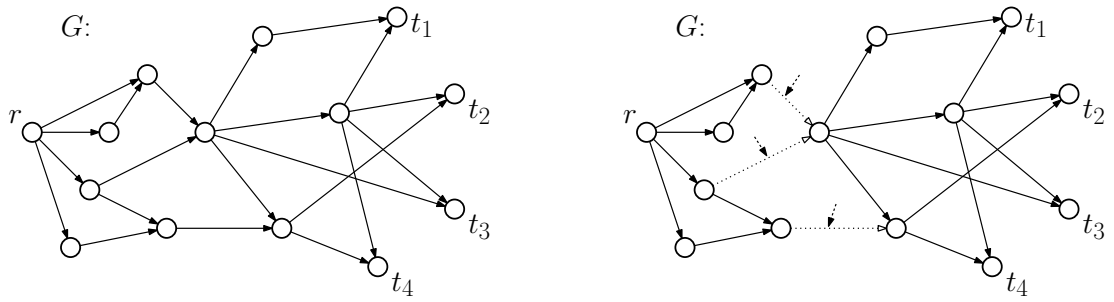


Figure 2: Eliminating edges to separated r from terminals.

More formally, the input consists of two sequences of integers, $\langle b_0, b_1, \ldots, b_{n-1}\rangle$ and $\langle r_0, r_1, \ldots, r_{n-1}\rangle$. Blue bucket B_i holds b_i balls initially, and at the end, red bucket R_i should hold exactly r_i balls. The balls from blue bucket B_i may be redistributed only among the red buckets R_{i-1}, R_i, and R_{i+1} (indices taken modulo n). You may assume that $\sum_i b_i = \sum_i r_i$.

Design a polynomial time algorithm which given the lists $\langle b_0, b_1, \ldots, b_{n-1}\rangle$ and $\langle r_0, r_1, \ldots, r_{n-1}\rangle$, determines whether it is possible to redistribute the balls from the blue buckets into the red buckets according to these restrictions. (Hint: Reduce to either network flow or circulation.)

Problem 6. You are given a collection of n points $U = \{u_1, u_2, \ldots, u_n\}$ in the plane, each of which is the location of a cell-phone user. You are also given the locations of m cell-phone towers, $C = \{c_1, c_2, \ldots, c_m\}$. A cell-phone user can connect to a tower if it is within distance Δ of the tower. For the sake of fault-tolerance each cell-phone user must be connected to at least three different towers. For each tower c_i you are given the maximum number of users, m_i, that can connect to this tower.

Give a polynomial time algorithm, which determines whether it is possible to assign all the cell-phone users to towers, subject to these constraints. Prove its correctness. (You may assume you have a function that returns the distance between any two points in $O(1)$ time.)

Problem 7. In the High-Degree Independent Set (HDIS) problem, you are given a graph $G = (V, E)$ and an integer k, and you want to know whether there exists an independent set V' in G of size k such that each vertex of V' is of degree at least k. (For example, the graph in Fig. 3 has an HDIS for $k = 3$, shown as the shaded vertices. Note that it does *not* have an HDIS for $k = 4$. Although adding the topmost vertex would still yield an independent set, this vertex does not have degree at least four.) In this problem, we will show that the following variant of the independent-set problem is NP-complete.

(a) Show that HDIS is in NP.

(b) Show that HDIS is NP-hard. (Hint: Use standard independent set (IS).)

Problem 8. Prove that the following problem, called the *acyclic subgraph problem* (AS) is NP-complete. Given a directed graph $G = (V, E)$ and an integer k, determine whether G contains a subset V' of k vertices such that the induced subgraph on V' is acyclic. Recall that the *induced subgraph* on V' is the subgraph $G' = (V', E')$ whose vertex set is V', and for which

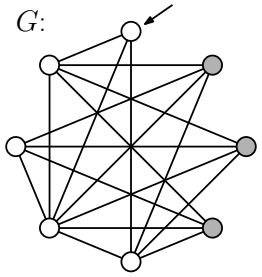


Figure 3: High-degree independent set.

$(u, v) \in E'$ if $u, v \in V'$ and $(u, v) \in E$. (Hint: Reduction from Independent Set. Think of a reduction that maps undirected edges to directed cycles.)

Problem 9. Show that the following problem is NP-complete.

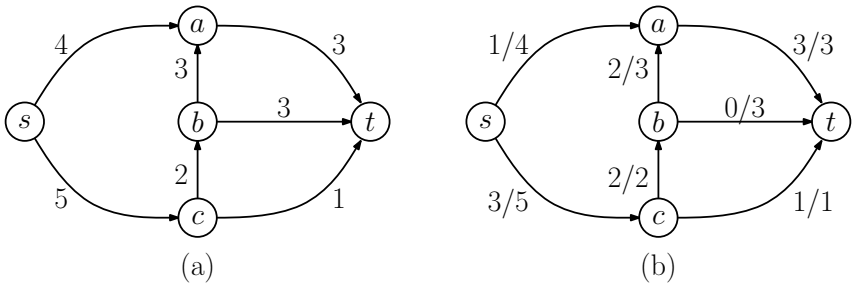
Balanced 3-coloring (B3C): Given a graph $G = (V, E)$, where $|V|$ is a multiple of 3, can G can be 3-colored such that the sizes of the 3 color groups are all equal to $|V|/3$. That is, can we assign an integer from $\{1, 2, 3\}$ to each vertex of G such that no two adjacent vertices have the same color, and such that all the colors are used equally often.

Hint: Reduction from the standard 3-coloring problem (3COL).

Final Exam

This exam is closed-book and closed-notes. You may use two sheets of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (15 points) Consider the s-t network G shown in the figure below (a). Suppose that we have already computed a partial flow f, shown in part (b) of the figure. (Each edge e is labeled with $f(e)/c(e)$, where $f(e)$ is the flow on this edge and $c(e)$ is the edge's capacity.)



(a) (8 points) Show the *residual graph* G_f corresponding to this flow.

(b) (3 points) Show any *augmenting path* from s to t in G_f.

(c) (4 points) Show the new flow f' that results by pushing as much flow as possible through this augmenting path.

Problem 2. (30 points: 3–8 points each) Short answer questions. (Unless otherwise specified, explanations are not required but may be given for partial credit.)

(a) Consider the following summation, $f(n) = \sum_{i=1}^{n} i^3$. Which of the following three assertions are true? (Select any/all that apply.)

$$\text{(i): } f(n) = O(n^3) \qquad \text{(ii): } f(n) = O(n^4) \qquad \text{(iii): } f(n) = O(n^5)$$

(b) Suppose that you perform a DFS on an undirected graph $G = (V, E)$, and for each vertex $u \in V$, you compute the discovery time $d[u]$ and finish time $f[u]$. Let u be a descendant of v in the DFS tree. What can be said about the relative order of $d[u]$, $f[u]$, $d[v]$, and $f[v]$?

(c) Give a definition of the k-center problem. (What is the input and what is the output?) What does it mean to say that an algorithm yields a factor-2 approximation to this problem?

(d) True or False: Suppose that the capacities of the edges of an s-t network are integers that are all evenly divisible by 3. (E.g., 3, 6, 9, 12, etc.) Then there exists a maximum flow, such that the flow on each edge is also evenly divisible by 3.

(e) It is known that determining whether a graph has a cut-vertex is in P. It is also known that determining whether a graph has a Hamiltonian cycle in NP-hard. Answer the following two questions under the assumption that P $\neq$ NP. (*No explanation required.*)

(i) Is it possible to determine in polynomial time whether a graph has *both* a cut vertex and a Hamiltonian cycle?

(ii) Is it possible to determine in polynomial time whether a graph has *either* a cut vertex or a Hamiltonian cycle?

(f) Hyper-intelligent aliens from another world come to Earth and tell us (1) that the 3SAT problem is solvable in $O(n^9)$ time and (2) that no algorithm for the 3SAT problem exists that runs faster than $\Omega(n^9)$ time in the worst case.

Which of the following statements follow as a consequence? (List all that are true. No explanations are required.)

(i) All NP-complete problems are solvable in polynomial time.

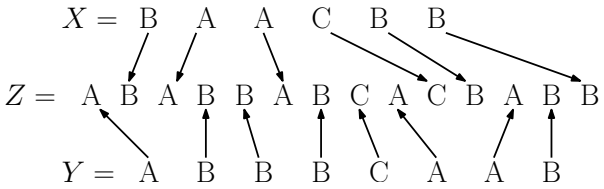
(ii) All NP-complete problems are solvable in $O(n^9)$ time.

(iii) All problems in NP, even those that are not NP-complete, are solvable in polynomial time.

(iv) No NP-complete problem can be solved faster than $O(n^9)$ time in the worst case.

Problem 3. (15 points) You are given three strings $X = \langle x_1, \ldots, x_m \rangle$, $Y = \langle y_1, \ldots, y_n \rangle$, and $Z = \langle z_1, \ldots, z_p \rangle$, where $p = m + n$. We say that Z is a *shuffle* of X and Y if it is possible to interleave the individual symbols of X and Y (without changing their relative order within each string) to obtain Z. (The name "shuffle" comes from the obvious analogy with shuffling two decks of cards X and Y together to form a single deck Z.)

For example, the figure below shows that $Z = \langle$ABABBABCACBABB$\rangle$ can be formed by shuffling $X = \langle$BAACBB$\rangle$ and $Y = \langle$ABBBCAAB$\rangle$.



Present an efficient algorithm which, given strings X, Y, and Z, determines whether Z is a shuffle of X and Y. (Hint: Use dynamic programming. It suffices to present just the recursive formulation. It is possible to do this in $O(mn)$ time.) Present a short proof of your algorithm's correctness and derive its running time.

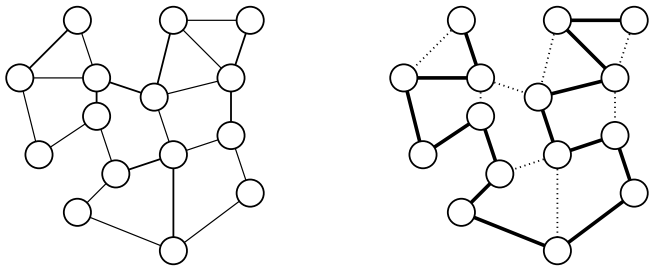
Problem 4. (15 points) Your local pharmacy has asked you to help set up the work schedule for the next month. There are n pharmacists on the staff and m days in the month. Each pharmacist gives a list of the days of the month that he/she is available to work. For the ith pharmacist, this is given as a list A_i, where each number in the list is in the range from 1 to m. For example, if $A_i = \langle 3, 5, 9, 15, 23 \rangle$, then the ith pharmacist is available to work on days 3, 5, 9, 15, and 23 of the month. Let $d_i = |A_i|$ denote the number of days that pharmacist i is available to work. Then he/she should be scheduled to work at least $\lceil d_i/2 \rceil$ of these days.

Each day there must be exactly 3 pharmacists on duty. (An example is shown in the figure below. There are 5 pharmacists and 4 days in the month.)

Availability Lists		(Possible) Final Schedule:	
$A_1 = \langle 1, 2, 4 \rangle$	⇢	Day	Pharmacists working
$A_2 = \langle 1, 2, 3 \rangle$		1	1, 2, 3
$A_3 = \langle 1, 2, 3, 4 \rangle$		2	1, 4, 5
$A_4 = \langle 2, 4 \rangle$		3	2, 3, 5
$A_5 = \langle 2, 3, 4 \rangle$		4	3, 4, 5

Present an efficient algorithm that is given the values of n, m, and the availability lists $A_1, \ldots, A_n$, and determines whether there exists a schedule that satisfies all the above requirements. (Hint: Reduce to either Max-Flow or Circulation. You may give a figure for the above example, but your description should work for any valid input.) Present a *brief* proof that your algorithm is correct.

Problem 5. (25 points) Given an undirected graph $G = (V, E)$, a *Hamiltonian path* is a simple path (not a cycle) that visits every vertex in the graph. (The graph shown in the figure below has a Hamiltonian path.) The *Hamiltonian Path problem* (HP) is the problem of determining whether a given graph has a Hamiltonian path.



(a) (5 points) Show that HP is in NP.

(b) (2 points) Professor Mount observes that if a graph has a Hamiltonian Cycle, then it also has a Hamiltonian Path. He suggests the following trivial reduction in order to prove that HP is NP-hard. Given a graph G for the Hamiltonian Cycle problem, simply output a copy of this graph. Explain why Professor Mount's reduction is *incorrect*.

(c) (18 points) Give a (correct) proof that HP is NP-hard. (Hint: The reduction is from the Hamiltonian Cycle problem, HC.)