CS 473 ✧ Spring 2016

☙ Homework 0 ❧

Due Tuesday, January 26, 2016 at 5pm

- **This homework tests your familiarity with prerequisite material:** designing, describing, and analyzing elementary algorithms (at the level of CS 225); fundamental graph problems and algorithms (again, at the level of CS 225); and especially facility with recursion and induction. Notes on most of this prerequisite material are available on the course web page.
- **Each student must submit individual solutions for this homework.** For all future homeworks, groups of up to three students will be allowed to submit joint solutions.
- **Submit your solutions electronically on the course Moodle site as PDF files.**
  - Submit a separate file for each numbered problem.
  - You can find a LaTeX solution template on the course web site; please use it if you plan to typeset your homework.
  - If you must submit scanned handwritten solutions, use a black pen (not pencil) on blank white printer paper (not notebook or graph paper), use a high-quality scanner (not a phone camera), and print the resulting PDF file on a black-and-white printer to verify readability before you submit.

☞ Some important course policies ☜

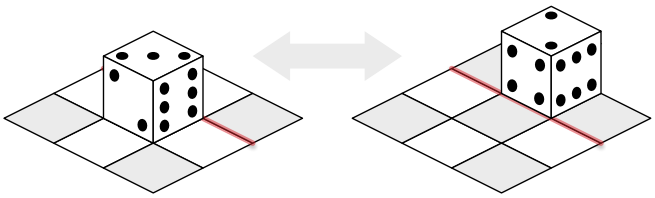
- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- The answer ***"I don't know"*** (and *nothing* else) is worth 25% partial credit on any problem or subproblem, on any homework or exam, except for extra-credit problems. We will accept synonyms like "No idea" or "WTF" or "¯\(•_•)/¯", but you must write *something*.
- **Avoid the Three Deadly Sins!** There are a few dangerous writing (and thinking) habits that will trigger an ***automatic zero*** on any homework or exam problem, unless your solution is nearly perfect otherwise. Yes, we are completely serious.
  - Always give complete solutions, not just examples.
  - Always declare all your variables, in English.
  - Never use weak induction.

See the course web site for more information.

If you have any questions about these policies,
please don't hesitate to ask in class, in office hours, or on Piazza.

---

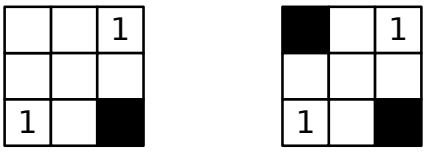
1. A ***rolling die maze*** is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on top of a table; the die always rests on and exactly covers one square. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.



Rolling a die.

Some squares in the grid may be *blocked*; the die must never be rolled onto a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number of pips on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

For example, here are two rolling die mazes. Black squares are blocked; empty white squares are free. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is solvable.

Describe and analyze an efficient algorithm to determine whether a given rolling die maze is solvable. Your input is a two-dimensional array $Label[1..n, 1..n]$, where each entry $Label[i, j]$ stores the label of the square in the ith row and jth column, where the label 0 means the square is free, and the label -1 means the square is blocked.

[Hint: Build a graph. What are the vertices? What are the edges? Is the graph directed or undirected? Do the vertices or edges have weights? If so, what are they? What textbook problem do you need to solve on this graph? What textbook algorithm should you use to solve that problem? What is the running time of that algorithm as a function of n? What does the number 24 have to do with anything?]

2. Describe and analyze fast algorithms for the following problems. The input for each problem is an unsorted array $A[1..n]$ of n arbitrary numbers, which may be positive, negative, or zero, and which are not necessarily distinct.

   (a) Are there two distinct indices $i < j$ such that $A[i] + A[j] = 0$?

   (b) Are there three distinct indices $i < j < k$ such that $A[i] + A[j] + A[k] = 0$?

   For example, if the input array is $[2, -1, 0, 4, 0, -1]$, both algorithms should return TRUE, but if the input array is $[4, -1, 2, 0]$, both algorithms should return FALSE. You do ***not*** need to prove that your algorithms are correct. *[Hint: The devil is in the details.]*

3. A ***binomial tree of order k*** is defined recursively as follows:

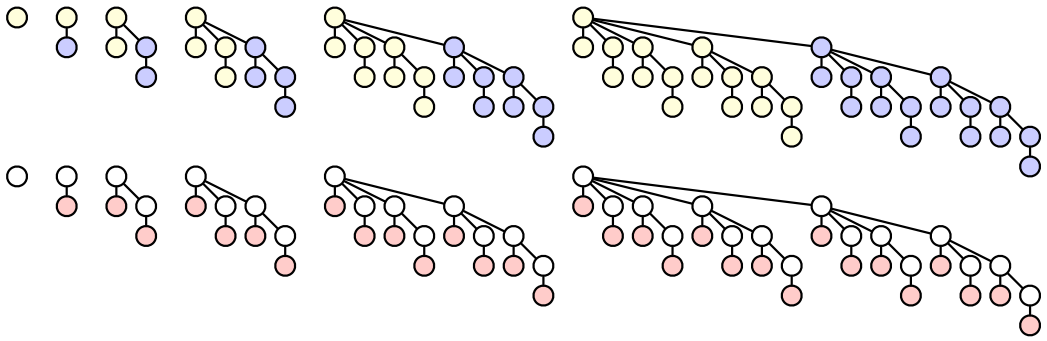
   - A binomial tree of order 0 is a single node.
   - For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k-1$, with the root of one tree connected as a new child of the root of the other. (See the figure below.)

   Prove the following claims:

   (a) For all non-negative integers k, a binomial tree of order k has exactly 2^k nodes.

   (b) For all positive integers k, attaching a new leaf to every node in a binomial tree of order $k-1$ results in a binomial tree of order k.

   (c) For all non-negative integers k and d, a binomial tree of order k has exactly $\binom{k}{d}$ nodes with depth d. (Hence the name!)

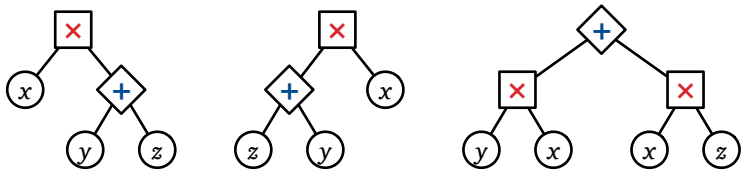


Binomial trees of order 0 through 5.
Top row: The recursive definition. Bottom row: The property claimed in part (b).

*4. ***[Extra credit]*** An ***arithmetic expression tree*** is a binary tree where every leaf is labeled with a variable, every internal node is labeled with an arithmetic operation, and every internal node has exactly two children. For this problem, assume that the only allowed operations are + and ×. Different leaves may or may not represent different variables.

Every arithmetic expression tree represents a function, transforming input values for the leaf variables into an output value for the root, by following two simple rules: (1) The value of any +-node is the sum of the values of its children. (2) The value of any ×-node is the product of the values of its children.

Two arithmetic expression trees are ***equivalent*** if they represent the same function; that is, the same input values for the leaf variables always leads to the same output value at both roots. An arithmetic expression tree is in ***normal form*** if the parent of every +-node (if any) is another +-node.



Three equivalent expression trees. Only the third expression tree is in normal form.

Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form. *[Hint: This is harder than it looks.]*

CS 473 ✧ Spring 2016

Homework 1

Due Tuesday, February 2, 2016, at **8pm**

- Starting with this homework, groups of up to three students may submit joint solutions. **Group solutions must represent an honest collaborative effort by all members of the group.** Please see the academic integrity policies for more information.
- You are responsible for forming your own groups. Groups can change from homework to homework, or even from (numbered) problem to problem.
- Please make sure the names and NetIDs of *all* group members appear prominently at the top of the first page of each submission.
- Please only upload one submission per group for each problem. In the Online Text box on the problem submission page, you must type in the NetIDs of ***all*** group members, including the person submitting. See the Homework Policies for examples. Failure to enter all group NetIDs will delay (if not prevent) giving all group members the grades they deserve.

- For dynamic programming problems, a full-credit solution must include the following:
  - A clear English specification of the underlying recursive function. (For example: "Let $Edit(i, j)$ denote the edit distance between $A[1..i]$ and $B[1..j]$.") Omitting the English description is a Deadly Sin, which will result in an automatic zero.
  - ***One*** of the following:
    * A correct recursive function or algorithm that computes the specified function, a clear description of the memoization structure, and a clear description of the iterative evaluation order.
    * Pseudocode for the final iterative dynamic programming algorithm.
  - The running time.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, subsequence, partition, coloring, tree, or path—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions will provide target time bounds for full credit. Correct algorithms that are faster than the official solution will receive extra credit points; correct algorithms that are slower than the official solution will get partial credit. We rarely include these target time bounds in the actual questions, because when we do, more students submit fast but incorrect algorithms (worth 0/10 on exams) instead of correct but slow algorithms (worth 8/10 on exams).

---

1. Let's define a *summary* of two strings A and B to be a concatenation of substrings of the following form:

   - ▲SNA indicates a substring SNA of only the first string A.
   - ♦FOO indicates a common substring FOO of both strings.
   - ▼BAR indicates a substring BAR of only the second string B.

   A summary is *valid* if we can recover the original strings A and B by concatenating the appropriate substrings of the summary in order and discarding the delimiters ▲, ♦, and ▼. Each regular character has length 1, and each delimiter ▲, ♦, or ▼ has some fixed non-negative length Δ. The *length* of a summary is the sum of the lengths of its symbols.

   For example, each of the following strings is a valid summary of the strings KITTEN and KNITTING:

   - ♦K▼N♦ITT▲E▼I♦N▼G has length $9 + 7\Delta$.
   - ♦K▼N♦ITT▲EN▼ING has length $10 + 5\Delta$.
   - ♦K▲ITTEN▼NITTING has length $13 + 3\Delta$.
   - ▲KITTEN▼KNITTING has length $14 + 2\Delta$.

   Describe and analyze an algorithm that computes the length of the shortest summary of two given strings $A[1..m]$ and $B[1..n]$. The delimiter length Δ is also part of the input to your algorithm. For example:

   - Given strings KITTEN and KNITTING and $\Delta = 0$, your algorithm should return 9.
   - Given strings KITTEN and KNITTING and $\Delta = 1$, your algorithm should return 15.
   - Given strings KITTEN and KNITTING and $\Delta = 2$, your algorithm should return 18.

2. Suppose you are given a sequence of positive integers separated by plus (+) and minus (−) signs; for example:

$$1 + 3 - 2 - 5 + 1 - 6 + 7$$

   You can change the value of this expression by adding parentheses in different places. For example:

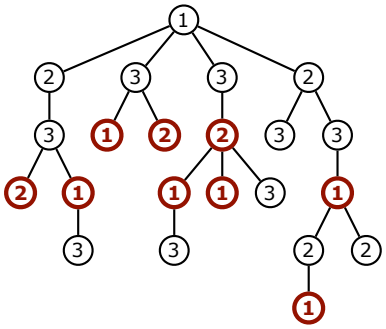
$$1 + 3 - 2 - 5 + 1 - 6 + 7 = -1$$
$$(1 + 3 - (2 - 5)) + (1 - 6) + 7 = 9$$
$$(1 + (3 - 2)) - (5 + 1) - (6 + 7) = -17$$

   Describe and analyze an algorithm to compute the maximum possible value the expression can take by adding parentheses.

   You may only use parentheses to group additions and subtractions; in particular, you are not allowed to create implicit multiplication as in $1 + 3(-2)(-5) + 1 - 6 + 7 = 33$.

3. The president of the Punxsutawney office of Giggle, Inc. has decided to give every employee a present to celebrate Groundhog Day! Each employee must receive one of three gifts: (1) an all-expenses-paid six-week vacation with Bill Murray,[1] (2) an all-the-Punxsutawney-pancakes-you-can-eat breakfast for two at Punxy Phil's Family Restaurant, or (3) a burning paper bag of groundhog poop. Corporate regulations prohibit any employee from receiving exactly the same gift as their direct supervisor. Unfortunately, any employee who receives a better gift than their direct supervisor will almost certainly be fired in a fit of jealousy.

   As Giggle-Punxsutawney's official gift czar, it's *your* job to decide which gift each employee receives. Describe an algorithm to distribute gifts so that the minimum number of people are fired. Yes, you can give the president groundhog poop.



A tree labeling with cost 9. The nine bold nodes have smaller labels than their parents.
The president got a vacation with Bill Murray. This is *not* the optimal labeling for this tree.

   More formally, you are given a rooted tree T, representing the company hierarchy, and you want to label each node in T with an integer 1, 2, or 3, so that every node has a different label from its parent. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T.

[1] The details of scheduling n distinct six-week vacations with Bill Murray, all in a single year, are left as an exercise for the reader.

CS 473 ✧ Spring 2016

Homework 2

Due Tuesday, February 9, 2016, at 8pm

---

1. *[Insert amusing story about distributing polling stations or cell towers or Starbucks or something on a long straight road in rural Iowa. Ha ha ha, how droll.]*

   More formally, you are given a sorted array $X[1..n]$ of distinct numbers and a positive integer k. A set of k intervals ***covers*** X if every element of X lies inside one of the k intervals. Your aim is to find k intervals $[a_1, z_1], [a_2, z_2], \ldots, [a_k, z_k]$ that cover X where the function $\sum_{i=1}^{k}(z_i - a_i)^2$ is as small as possible. Intuitively, you are trying to cover the points with k intervals whose lengths are as close to equal as possible.

   (a) Describe an algorithm that finds k intervals with minimum total squared length that cover X. The running time of your algorithm should be a simple function of n and k.

   (b) Consider the two-dimensional matrix $M[1..n, 1..n]$ defined as follows:

   $$M[i,j] = \begin{cases} (X[j]-X[i])^2 & \text{if } i \le j \\ \infty & \text{otherwise} \end{cases}$$

   Prove that M satisfies the ***Monge property***: $M[i,j] + M[i',j'] \le M[i,j'] + M[i',j]$ for all indices $i < i'$ and $j < j'$.

   (c) ***[Extra credit]*** Describe an algorithm that finds k intervals with minimum total squared length that cover X ***in $O(nk)$ time***. *[Hint: Solve part (a) first, then use part (b).]*

   We strongly recommend submitting your solution to part (a) separately, and only describing your changes to that solution for part (c).

2. The Doctor and River Song decide to play a game on a directed acyclic graph G, which has one source s and one sink t.□

   Each player has a token on one of the vertices of G. At the start of the game, The Doctor's token is on the source vertex s, and River's token is on the sink vertex t. The players alternate turns, with The Doctor moving first. On each of his turns, the Doctor moves his token forward along a directed edge; on each of her turns, River moves her token *backward* along a directed edge.

   If the two tokens ever meet on the same vertex, River wins the game. ("Hello, Sweetie!") If the Doctor's token reaches t or River's token reaches s before the two tokens meet, then the Doctor wins the game.

   Describe and analyze an algorithm to determine who wins this game, assuming both players play perfectly. That is, if the Doctor can win *no matter how River moves*, then your algorithm should output "Doctor", and if River can win *no matter how the Doctor moves*, your algorithm should output "River". (Why are these the only two possibilities?) The input to your algorithm is the graph G.

□possibly short for the Untempered **S**chism and the **T**ime Vortex, or the Shining World of the Seven Systems (otherwise known as Gallifrey) and Trenzalore, or Skaro and Telos, or something timey-wimey.

CS 473 ✧ Spring 2016

Homework 3

Due Tuesday, February 9, 2016, at 8pm

Unless a problem specifically states otherwise, you may assume a function RANDOM that takes a positive integer k as input and returns an integer chosen uniformly and independently at random from $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, to flip a fair coin, you could call RANDOM(2).

---

1. Suppose we want to write an efficient function RANDOMPERMUTATION(n) that returns a permutation of the set $\{1, 2, \ldots, n\}$ chosen uniformly at random.

   (a) Prove that the following algorithm is ***not*** correct. *[Hint: There is a one-line proof!]*

```
RandomPermutation(n):
  for i ← 1 to n
      π[i] ← i
  for i ← 1 to n
      swap π[i] ↔ π[Random(n)]
```

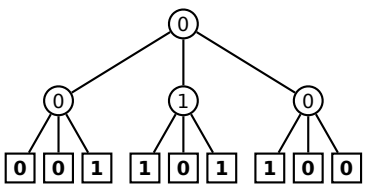
   (b) Consider the following implementation of RANDOMPERMUTATION.

```
RandomPermutation(n):
  for i ← 1 to n
      π[i] ← NULL
  for i ← 1 to n
      j ← Random(n)
      while (π[j] != NULL)
          j ← Random(n)
      π[j] ← i
  return π
```

   Prove that this algorithm is correct and analyze its expected running time.

   (c) Describe and analyze an implementation of RANDOMPERMUTATION that runs in expected worst-case time $O(n)$.

2. A ***majority tree*** is a complete ternary tree in which every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. For example, if the tree has depth 2 and its leaves are labeled $1, 0, 0, 0, 1, 0, 1, 1, 1$, the root has value 0.



A majority tree with depth 2.

It is easy to compute value of the root of a majority tree of depth n in $O(3^n)$ time, given the sequence of 3^n leaf labels as input, using a simple post-order traversal of the tree. Prove that this simple algorithm is optimal, and then describe a better algorithm. More formally:

(a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. *[Hint: Consider the special case $n = 1$. Recurse.]*

(b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some explicit constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.]*

3. A ***meldable priority queue*** stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- MakeQueue: Return a new priority queue containing the empty set.
- FindMin(Q): Return the smallest element of Q (if any).
- DeleteMin(Q): Remove the smallest element in Q (if any).
- Insert(Q, x): Insert element x into Q, if it is not already there.
- DecreaseKey(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
- Delete(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
- Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. Meld can be implemented using the following randomized algorithm:

```
Meld(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1

  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2

  with probability 1/2
      left(Q1) ← Meld(left(Q1), Q2)
  else
      right(Q1) ← Meld(right(Q1), Q2)

  return Q1
```

(a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of Meld(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: What is the expected length of a random root-to-leaf path in an n-node binary tree, where each left/right choice is made with equal probability?]*

(b) Prove that Meld(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

(c) Show that each of the other meldable priority queue operations can be implemented with at most one call to Meld and $O(1)$ additional time. (It follows that each operation takes only $O(\log n)$ time with high probability.)

CS 473 ✧ Spring 2016

Homework 4

Due Tuesday, March 1, 2016, at 8pm

Unless a problem specifically states otherwise, you may assume a function RANDOM that takes a positive integer k as input and returns an integer chosen uniformly and independently at random from $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, to flip a fair coin, you could call RANDOM(2).

---

1. Suppose we are given a two-dimensional array $M[1..n, 1..n]$ in which every row and every column is sorted in increasing order and no two elements are equal.

   (a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M larger than $M[i, j]$ and smaller than $M[i', j']$.

   (b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements larger than $M[i, j]$ and smaller than $M[i', j']$. Assume the requested range is always non-empty.

   (c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

2. ***Tabulated hashing*** uses tables of random numbers to compute hash values. Suppose $|\mathcal{U}| = 2^w \times 2^w$ and $m = 2^\ell$, so the items being hashed are pairs of w-bit strings (or $2w$-bit strings broken in half) and hash values are ℓ-bit strings.

   Let $A[0..2^w - 1]$ and $B[0..2^w - 1]$ be arrays of independent random ℓ-bit strings, and define the hash function $h_{A,B} : \mathcal{U} \to [m]$ by setting

   $$h_{A,B}(x, y) := A[x] \oplus B[y]$$

   where $\oplus$ denotes bit-wise exclusive-or. Let $\mathcal{H}$ denote the set of all possible functions $h_{A,B}$. Filling the arrays A and B with independent random bits is equivalent to choosing a hash function $h_{A,B} \in \mathcal{H}$ uniformly at random.

   (a) Prove that $\mathcal{H}$ is 2-uniform.

   (b) Prove that $\mathcal{H}$ is 3-uniform. *[Hint: Solve part (a) first.]*

   (c) Prove that $\mathcal{H}$ is ***not*** 4-uniform.

   Yes, "see part (b)" is worth full credit for part (a), but only if your solution to part (b) is correct.

CS 473 ✧ Spring 2016

Homework 5

Due Tuesday, March 1, 2016, at 8pm

Unless a problem specifically states otherwise, you may assume a function RANDOM that takes a positive integer k as input and returns an integer chosen uniformly and independently at random from $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, to flip a fair coin, you could call RANDOM(2).

---

1. ***Reservoir sampling*** is a method for choosing an item uniformly at random from an arbitrarily long stream of data.

```
GETONESAMPLE(stream S):
  ℓ ← 0
  while S is not done
      x ← next item in S
      ℓ ← ℓ + 1
      if RANDOM(ℓ) = 1
          sample ← x        (⋆)
  return sample
```

At the end of the algorithm, the variable ℓ stores the length of the input stream S; this number is *not* known to the algorithm in advance. If S is empty, the output of the algorithm is (correctly!) undefined. In the following, consider an arbitrary non-empty input stream S, and let n denote the (unknown) length of S.

(a) Prove that the item returned by GETONESAMPLE(S) is chosen uniformly at random from S.

(b) Describe and analyze an algorithm that returns a subset of k distinct items chosen uniformly at random from a data stream of length at least k. The integer k is given as part of the input to your algorithm. Prove that your algorithm is correct.

For example, if $k = 2$ and the stream contains the sequence $\langle ♠, ♥, ♦, ♣ \rangle$, the algorithm should return the subset $\{♦, ♠\}$ with probability $1/6$.

2. In this problem, we will derive a streaming algorithm that computes an accurate estimate $\widetilde{n}$ of the number of distinct items in a data stream S. Suppose S contains n unique items (but possible several copies of each item); the algorithm does *not* know n in advance. Given an accuracy parameter $0 < \varepsilon < 1$ and a confidence parameter $0 < \delta < 1$ as part of the input, our final algorithm will guarantee that $\Pr[|\widetilde{n} - n| > \varepsilon n] < \delta$.

   As a first step, fix a positive integer m that is large enough that we don't have to worry about round-off errors in the analysis. Our first algorithm chooses a hash function $h : \mathcal{U} \to [m]$ at random from a ***2-uniform*** family, computes the minimum hash value $\hbar = \min\{h(x) \mid x \in S\}$, and finally returns the estimate $\widetilde{n} = m/\hbar$.

   (a) Prove that $\Pr[\widetilde{n} > (1+\varepsilon)n] \le 1/(1+\varepsilon)$. *[Hint: Markov's inequality]*

   (b) Prove that $\Pr[\widetilde{n} < (1-\varepsilon)n] \le 1-\varepsilon$. *[Hint: Chebyshev's inequality]*

   (c) We can improve this estimator by maintaining the k smallest hash values, for some integer $k > 1$. Let $\widetilde{n}_k = k \cdot m/\hbar_k$, where $\hbar_k$ is the kth smallest element of $\{h(x) \mid x \in S\}$.

   Estimate the smallest value of k (as a function of the accuracy parameter ε) such that $\Pr[|\widetilde{n}_k - n| > \varepsilon n] \le 1/4$.

   (d) Now suppose we run d copies of the previous estimator in parallel to generate d independent estimates $\widetilde{n}_{k,1}, \widetilde{n}_{k,2}, \ldots, \widetilde{n}_{k,d}$, for some integer $d > 1$. Each copy uses its own independently chosen hash function, but they all use the same value of k that you derived in part (c). Let $\widetilde{N}$ be the *median* of these d estimates.

   Estimate the smallest value of d (as a function of the confidence parameter δ) such that $\Pr[|\widetilde{N} - n| > \varepsilon n] \le \delta$.

CS 473 ✧ Spring 2016

Homework 6

Due Tuesday, March 15, 2016, at 8pm

For problems that use maximum flows as a black box, a full-credit solution requires the following.

- A complete description of the relevant flow network, specifying the set of vertices, the set of edges (being careful about direction), the source and target vertices s and t, and the capacity of every edge. (If the flow network is part of the original input, just say that.)
- A description of the algorithm to construct this flow network from the stated input. This could be as simple as "We can construct the flow network in $O(n^3)$ time by brute force."
- A description of the algorithm to extract the answer to the stated problem from the maximum flow. This could be as simple as "Return TRUE if the maximum flow value is at least 42 and False otherwise."
- A proof that your reduction is correct. This proof will almost always have two components. For example, if your algorithm returns a boolean, you should prove that its TRUE answers are correct and that its FALSE answers are correct. If your algorithm returns a number, you should prove that number is neither too large nor too small.
- The running time of the overall algorithm, expressed as a function of the original input parameters, not just the number of vertices and edges in your flow network.
- You may assume that maximum flows can be computed in $O(VE)$ time. Do *not* regurgitate the maximum flow algorithm itself.

Reductions to other flow-based algorithms described in class or in the notes (for example: edge-disjoint paths, maximum bipartite matching, minimum-cost circulation) or to other standard graph problems (for example: reachability, minimum spanning tree, shortest paths) have similar requirements.

---

1. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t in V, a capacity function $c : E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Describe an algorithm to determine whether f is a maximum (s, t)-flow in G. Do not assume ***anything*** about the function f.

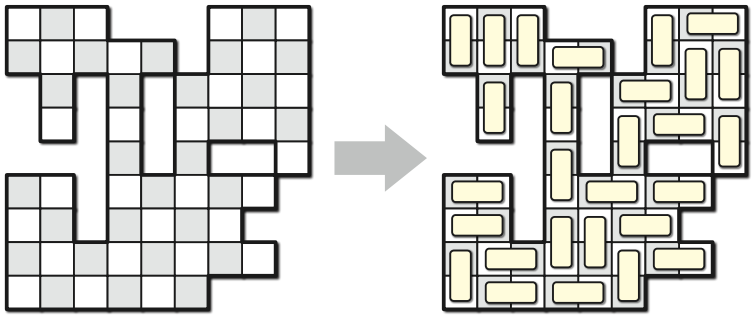
2. Suppose you have already computed a maximum flow f^* in a flow network G with ***integer*** edge capacities.

   (a) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is increased by 1.

   (b) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is decreased by 1.

   Both algorithms should be significantly faster than recomputing the maximum flow from scratch.

3. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether it is possible to tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.



Your input is a two-dimensional array $Deleted[1..n, 1..n]$ of bits, where $Deleted[i, j] = \text{TRUE}$ if and only if the square in row i and column j has been deleted. Your output is a single bit; you do ***not*** have to compute the actual placement of dominos. For example, for the board shown above, your algorithm should return TRUE.

CS 473 ✧ Spring 2016

Homework 7

Due Tuesday, March 29, 2016, at 8pm

This is the last homework before Midterm 2.

---

1. Suppose we are given a two-dimensional array $A[1..m, 1..n]$ of non-negative real numbers. We would like to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding exists.

2. You're organizing the Third Annual UIUC Computer Science 72-Hour Dance Exchange, to be held all day Friday, Saturday, and Sunday in Siebel Center.[1] Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints.

   - Exactly k sets of music must be played each day, and thus $3k$ sets altogether.
   - Each set must be played by a single DJ in a consistent musical genre (ambient, bubblegum, dancehall, horrorcore, trip-hop, Nashville country, Chicago blues, axé, laïkó, skiffle, shape note, Nitzhonot, J-pop, K-pop, C-pop, T-pop, 8-bit, Tesla coil, ...).
   - Each genre must be played at most once per day.
   - Each DJ has given you a list of genres they are willing to play.
   - No DJ can play more than five sets during the entire event.

   Suppose there are n candidate DJs and g different musical genres available. Describe and analyze an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.

3. Describe and analyze an algorithm to determine, given an undirected[2] graph $G = (V, E)$ and three vertices $u, v, w \in V$ as input, whether G contains a simple path from u to w that passes through v.

---

[1] Efforts to secure overflow space in ECEB were sadly unsuccessful.

[2] This adjective is important; if the input graph were directed, this problem would be NP-hard.

CS 473 ✧ Spring 2016
Homework 8

Due Tuesday, April 12, 2016, at 8pm

You may assume the following results in your solutions:

- Maximum flows and minimum cuts can be computed in $O(VE)$ time.
- Minimum-cost flows can be computed in $O(E^2 \log^2 V)$ time.
- Linear programming problems with integer coefficients can be solved in polynomial time.

For problems that ask for a linear-programming formulation of some problem, a full credit solution requires the following components:

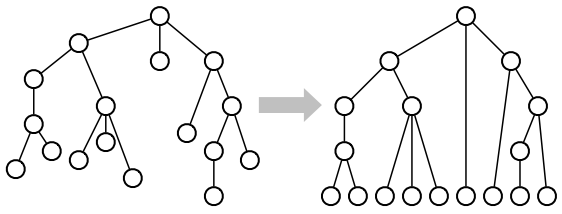
- A list of variables, along with a brief English description of each variable. (Omitting these English descriptions is a Deadly Sin.)
- A linear objective function (expressed either as minimization or maximization, whichever is more convenient), along with a brief English description of its meaning.
- A sequence of linear inequalities (expressed using $\leq$, $=$, or $\geq$, whichever is more appropriate or convenient), along with a brief English description of each constraint.
- A proof that your linear programming formulation is correct, meaning that the optimal solution to the original problem can always be obtained from the optimal solution to the linear program. This may be very short.

It is ***not*** necessary to express the linear program in canonical form, or even in matrix form. Clarity is much more important than formality.

---

1. Suppose your are given a rooted tree T, where every edge e has two associated values: a non-negative *length* $\ell(e)$, and a *cost* $\$(e)$ (which could be positive, negative, or zero). Your goal is to add a non-negative *stretch* $s(e) \geq 0$ to the length of every edge e in T, subject to the following conditions:

   - Every root-to-leaf path π in T has the same total stretched length $\sum_{e \in \pi}(\ell(e) + s(e))$
   - The total *weighted* stretch $\sum_e s(e) \cdot \$(e)$ is as small as possible.



   (a) Describe an instance of this problem with no optimal solution.

   (b) Give a concise linear programming formulation of this problem. (For the instance described in part (a), your linear program will be unbounded.)

   (c) Suppose that for the given tree T and the given lengths and costs, the optimal solution to this problem is unique. Prove that in this optimal solution, we have $s(e) = 0$ for every edge on some longest root-to-leaf path in T. In other words, prove that the optimally stretched tree with the same depth as the input tree. *[Hint: What is a basis in your linear program? What is a **feasible** basis?]*

Problem 1(c) originally omitted the uniqueness assumption and asked for a proof that ***every*** optimal solution has an unstretched root-to-leaf path, but that more general claim is false. For example, if every edge has cost zero, there are optimal solutions in which every edge has positive stretch.

2. Describe and analyze an efficient algorithm for the following problem, first posed and solved by the German mathematician Carl Jacobi in the early 1800s.[1]

> *Disponantur nn quantitates $h_k^{(i)}$ quaecunque in schema Quadrati, ita ut k habeantur n series horizontales et n series verticales, quarum quaeque est n terminorum. Ex illis quantitatibus eligantur n transversales, i.e. in seriebus horizontalibus simul atque verticalibus diversis positae, quod fieri potest $1.2\ldots n$ modis; ex omnibus illis modis quaerendum est is, qui summam n numerorum electorum suppeditet maximam.*

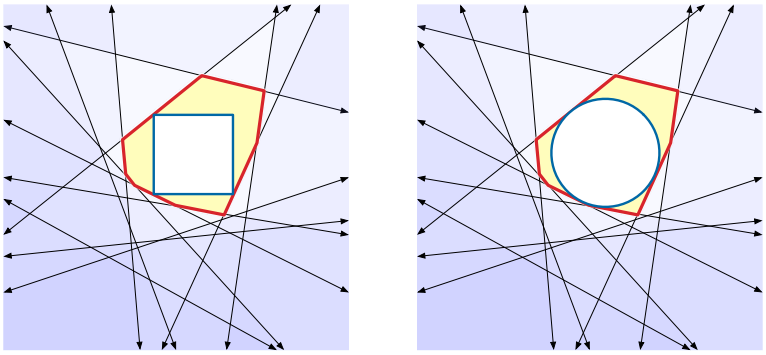
For those few students who are not fluent in mid-19th century academic Latin, here is a modern English translation of Jacobi's problem. Suppose we are given an $n \times n$ matrix M. Describe and analyze an algorithm that computes a permutation σ that maximizes the sum $\sum_{i=1}^{n} M_{i,\sigma(i)}$, or equivalently, permutes the columns of M so that the sum of the elements along the diagonal is as large as possible.

Please do not submit your solution in mid-19th century academic Latin.

3. Suppose we are given a sequence of n linear inequalities of the form $a_i x + b_i y \le c_i$. Collectively, these n inequalities describe a convex polygon P in the plane.

   (a) Describe a linear program whose solution describes the largest square with horizontal and vertical sides that lies entirely inside P.

   (b) Describe a linear program whose solution describes the largest circle that lies entirely inside P.



[1] Carl Gustav Jacob Jacobi. De investigando ordine systematis aequationum differentialum vulgarium cujuscunque. *J. Reine Angew. Math.* 64(4):297–320, 1865. Posthumously published by Carl Borchardt.

CS 473 ✧ Spring 2016

☙ Homework 9 ❧

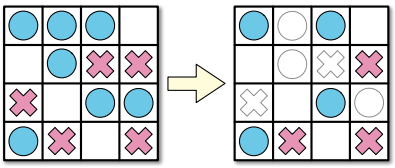
Due Tuesday, April 19, 2016, at 8pm

For problems that ask to prove that a given problem X is NP-hard, a full-credit solution requires the following components:

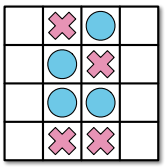
- Specify a known NP-hard problem Y, taken from the problems listed in the notes.
- Describe a polynomial-time algorithm for Y, using a black-box polynomial-time algorithm for X as a subroutine. Most NP-hardness reductions have the following form: Given an arbitrary instance of Y, describe how to transform it into an instance of X, pass this instance to a black-box algorithm for X, and finally, describe how to transform the output of the black-box subroutine to the final output. A cartoon with boxes may be helpful.
- Prove that your reduction is correct. As usual, correctness proofs for NP-hardness reductions usually have two components ("one for each f").

---

1. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

2. Everyone's having a wonderful time at the party you're throwing, but now it's time to line up for ***The Algorithm March (***アルゴリズムこうしん)! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから) for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat".□ Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake during lead to horrible embarrassment between strangers.

   Suppose you are given a complete list of which people at your party know each other. Prove that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March. You may assume without loss of generality that there are no ninjas at your party.

□そろそろおわりかな。そろそろおわりかな。そろそろおわりかな。

CS 473 ✧ Spring 2016

Homework 10

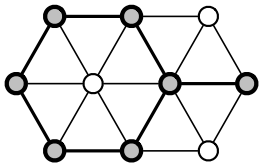
Due Tuesday, April 26, 2016, at 8pm

This is the last graded homework of the semester.

---

1. A *double-Hamiltonian circuit* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Prove that determining whether whether a given undirected graph contains a double-Hamiltonian circuit is NP-hard.

2. A subset S of vertices in an undirected graph G is called ***triangle-free*** if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.



A triangle-free subset of 7 vertices.
This is ***not*** the largest triangle-free subset in this graph.

3. Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.]*

CS 473 ✧ Spring 2016

Homework 11

Solutions will be released on Tuesday, May 3, 2016.

This homework will not be graded.
However, material covered by this homework *may* appear on the final exam.

---

1. The ***linear arrangement*** problem asks, given an n-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of forward edges: directed edges $v_i \to v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem. (Solving this problem exactly is NP-hard.)

2. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

   (a) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we independently assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.

   (b) Prove that with high probability, the expected number of interesting edges is at least $\frac{1}{2}wow(G)$. *[Hint: Use Chebyshev's inequality. But wait... How do we know that we **can** use Chebyshev's inequality?]*

   (c) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

3. Suppose we want to schedule a give set of n jobs on on a machine containing a row of p identical processors. Our input consists of two arrays $duration[1..n]$ and $width[1..n]$. A valid schedule consists of two arrays $start[1..n]$ and $first[1..n]$ that satisfy the following constraints:

   - $start[j] \geq 0$ for all j.
   - The jth job runs on processors $first[j]$ through $first[j] + width[j] - 1$, starting at time $start[j]$ and ending at time $start[j] + duration[j]$.
   - No processor can run more than one job simultaneously.

   The *makespan* of a schedule is the largest finishing time: $\max_j(start[j] + duration[j])$. Our goal is to compute a valid schedule with the smallest possible makespan.

   (a) Prove that this scheduling problem is NP-hard.

(b) Describe a polynomial-time algorithm that computes a 3-approximation of the minimum makespan of the given set of jobs. That is, if the minimum makespan is M, your algorithm should compute a schedule with makespan at most $3M$. You may assume that p is a power of 2. *[Hint: Assume that p is a power of 2.]*

(c) Describe an algorithm that computes a 3-approximation of the minimum makespan of the given set of jobs ***in $O(n \log n)$ time***. Again, you may assume that p is a power of 2.

These are the standard 10-point rubrics that we will use for certain types of exam questions. When these problems appear in the homework, a score of x on this 10-point scale corresponds to a score of $\lceil x/3 \rceil$ on the 4-point homework scale.

Proof by Induction

- 2 points for stating a valid **strong** induction hypothesis.
  - The inductive hypothesis need not be stated explicitly if it is a mechanical translation of the theorem (that is, "Assume $P(k)$ for all $k < n$" when the theorem is "$P(n)$ for all n") *and* it is applied correctly. However, if the proof requires a stronger induction hypothesis ("Assume $P(k)$ and $Q(k)$ for all $k < n$") then it must be stated explicitly.
  - By course policy, ***stating a weak inductive hypothesis triggers an automatic zero***, unless the proof is otherwise *perfect*.
  - ***Ambiguous*** induction hypotheses like "Assume the statement is true for all $k < n$." are not valid. *What* statement? The theorem you're trying to prove doesn't use the variable k, so that can't possibly be the statement you mean.
  - ***Meaningless*** induction hypotheses like "Assume that k is true for all $k < n$" are not valid. Only propositions can be true or false; k is an integer, not a proposition.
  - ***False*** induction hypotheses like "Assume that $k < n$ for all k" are not valid. The inequality $k < n$ does *not* hold for all k, because it does not hold when $k = n + 5$.
- 1 point for explicit and clearly exhaustive case analysis.
  - No penalty for overlapping or redundant cases. However, mistakes in redundant cases are still penalized.
- 2 points for the base case(s).
- 2 point for correctly applying the *stated* inductive hypothesis.
  - It is not possible to correctly apply an invalid inductive hypothesis.
  - No credit for correctly applying a different induction hypothesis than the one stated.
- 3 points for other details of the inductive case(s).

Dynamic Programming

- **6 points for a correct recurrence**, described either using functional notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) ***Automatic zero if the English description is missing.***
  - + 1 point for stating how to call your recursive function to get the final answer.
  - + 1 point for the base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for the recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- **4 points for iterative details**
  - + 1 point for describing the memoization data structure; a clear picture may be sufficient.
  - + 2 points for describing a correct evaluation order; a clear picture may be sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for running time
- Proofs of correctness are not required for full credit on exams, unless the problem specifically asks for one.
- Do not analyze (or optimize) space.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative psuedocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. However, you ***must*** give an English description of the underlying recursive function.
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is ***scaled*** to the new maximum score. All points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

Graph Reductions

For problems solved by reducing them to a standard graph algorithm covered either in class or in a prerequisite class (for example: shortest paths, topological sort, minimum spanning trees, maximum flows, bipartite maximum matching, vertex-disjoint paths, . . .):

- **1 point for listing the vertices of the graph.** (If the original input is a graph, describing how to modify that graph is fine.)
- **1 point for listing the edges of the graph**, including whether the edges are directed or undirected. (If the original input is a graph, describing how to modify that graph is fine.)
- **1 point for describing appropriate weights** and/or lengths and/or capacities and/or costs and/or demands and/or whatever for the vertices and edges.
- **2 points for an explicit description of the problem being solved on that graph.** (For example: "We compute the maximum number of vertex-disjoint paths in G from v to z.")
- **3 points for other algorithmic details**, assuming the rest of the reduction is correct.
  - + 1 point for describing how to build the graph from the original input (for example: "by brute force")
  - + 1 point for describing the algorithm you use to solve the graph problem (for example: "Orlin's algorithm" or "as described in class")
  - + 1 point for describing how to extract the output for the original problem from the output of the graph algorithm.
- **2 points for the running time**, expressed in terms of the original input parameters, not just V and E.
- **If the problem explicitly asks for a proof of correctness,** divide all previous points in half and add **5 points for proof of correctness.** These proofs almost always have two parts; for example, for algorithms that return TRUE or FALSE:
  - – 2½ points for proving that if your algorithm returns TRUE, then the correct answer is TRUE.
  - – 2½ points for proving that if your algorithm returns FALSE, then the correct answer is FALSE.

  These proofs do not need to be as detailed as in the homeworks; we are really just looking for compelling evidence that *you* understand why your reduction is correct.
- It is still possible to get partial credit for an incorrect algorithm. For example, if you describe an algorithm that sometimes reports false positives, but you prove that all FALSE answers are correct, you would still get 2½ points for half of the correctness proof.

NP-Hardness Reductions

For problems that ask "***Prove*** that X is NP-hard":

- **4 points for the polynomial-time reduction:**
  - 1 point for explicitly naming the NP-hard problem Y to reduce from. You may use any of the problems listed in the lecture notes; a list of NP-hard problems will appear on the back page of the exam.
  - 2 points for describing the polynomial-time algorithm to transform arbitrary instances of Y into inputs to the black-box algorithm for X
  - 1 point for describing the polynomial-time algorithm to transform the output of the black-box algorithm for X into the output for Y.
  - Reductions that call the black-box algorithm for X more than once are perfectly acceptable. You do *not* need to explicitly analyze the running time of your resulting algorithm for Y, but it must be polynomial in the size of the input instance of Y.
- **6 points for the proof of correctness.** **This is the entire point of the problem.** These proofs always have two parts; for example, if X and Y are both decision problems:
  - 3 points for proving that your reduction transforms positive instances of Y into positive instances of X.
  - 3 points for proving that your reduction transforms negative instances of Y into negative instances of X.

  These proofs do not need to be as detailed as in the homeworks; however, it must be clear that you have at least considered all possible cases. We are really just looking for compelling evidence that *you* understand why your reduction is correct.
- It is still possible to get partial credit for an incorrect reduction. For example, if you describe a reduction that sometimes reports false positives, but you prove that all FALSE answers are correct, you would still get 3 points for half of the correctness proof.
- Zero points for reducing X ***to*** some NP-hard problem Y.
- Zero points for attempting to solve X.

Approximation Algorithms

For problems that ask you to describe a polynomial-time approximation algorithm for some NP-hard problem X, analyze its approximation ratio, and prove that your approximation analysis is correct:

- **4 points for the actual approximation algorithm.** You do not need to analyze the running time of your algorithm (unless we explicitly ask for the running time), but it must clearly run in polynomial time. If we give you the algorithm, ignore this part and scale the rest of the rubric up to 10 points.
- **2 points for stating the correct approximation ratio.** If we give you the approximation ratio, ignore this part and scale the rest of the rubric up to 10 points.
- **4 points for proving that the stated approximation ratio is correct.** If we do not *explicitly* ask for a proof, ignore this part and scale the rest of the rubric up to 10 points.

For example, suppose we give you an algorithm and ask for its approximation ratio, but we do not explicitly ask for a proof. If the given algorithm is a 3-approximation algorithm, then you would get full credit for writing "3".

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

---

1. For any positive integer n, the nth ***Fibonacci string*** F_n is defined recursively as follows, where $x \bullet y$ denotes the concatenation of strings x and y:

$$F_1 := \texttt{0}$$
$$F_2 := \texttt{1}$$
$$F_n := F_{n-1} \bullet F_{n-2} \quad \text{for all } n \geq 3$$

   For example, $F_3 = \texttt{10}$ and $F_4 = \texttt{101}$.

   (a) What is F_8?

   (b) ***Prove*** that every Fibonacci string except F_1 starts with `1`.

   (c) ***Prove*** that no Fibonacci string contains the substring `00`.

2. You have reached the inevitable point in the semester where it is no longer possible to finish all of your assigned work without pulling at least a few all-nighters. The problem is that pulling successive all-nighters will burn you out, so you need to pace yourself (or something).

   Let's model the situation as follows. There are n days left in the semester. For simplicity, let's say you are taking one class, there are no weekends, there is an assignment due every single day until the end of the semester, and you will only work on an assignment the day before it is due. For each day i, you know two positive integers:

   - $Score[i]$ is the score you will earn on the ith assignment if you do *not* pull an all-nighter the night before.
   - $Bonus[i]$ is the number of additional points you could potentially earn if you *do* pull an all-nighter the night before.

   However, pulling multiple all-nighters in a row has a price. If you turn in the ith assignment immediately after pulling k consecutive all-nighters, your actual score for that assignment will be $(Score[i] + Bonus[i])/2^{k-1}$.

   Design and analyze an algorithm that computes the maximum total score you can achieve, given the arrays $Score[1..n]$ and $Bonus[1..n]$ as input.

3. The following algorithm finds the smallest element in an unsorted array. The subroutine SHUFFLE randomly permutes the input array A; every permutation of A is equally likely.

```
RANDOMMIN(A[1..n]):
  min ← ∞
  SHUFFLE(A)
  for i ← 1 to n
      if A[i] < min
          min ← A[i]          (⋆)
  return min
```

In the following questions, assume all elements in the input array $A[\,]$ are distinct.

(a) In the worst case, how many times does RANDOMMIN execute line $(\star)$?

(b) For each index i, let $X_i = 1$ if line $(\star)$ is executed in the ith iteration of the for loop, and let $X_i = 0$ otherwise. What is $\Pr[X_i = 1]$? *[Hint: First consider $i = 1$ and $i = n$.]*

(c) What is the *exact* expected number of executions of line $(\star)$?

(d) ***Prove*** that line $(\star)$ is executed $O(\log n)$ times with high probability, *assuming* the variables X_i are mutually independent.

(e) **[Extra credit]** ***Prove*** that the variables X_i are mutually independent.
[Hint: Finish the rest of the exam first!]

4. Your eight-year-old cousin Elmo decides to teach his favorite new card game to his baby sister Daisy. At the beginning of the game, n cards are dealt face up in a long row. Each card is worth some number of points, which may be positive, negative, or zero. Then Elmo and Daisy take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, each player can decide which of the two cards to take. When the game ends, the player that has collected the most points wins.

Daisy isn't old enough to get this whole "strategy" thing; she's just happy to play with her big brother. When it's her turn, she takes the either leftmost card or the rightmost card, each with probability $1/2$.

Elmo, on the other hand, *really* wants to win. Having never taken an algorithms class, he follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value.

Describe and analyze an algorithm to determine Elmo's expected score, given the initial sequence of n cards as input. Assume Elmo moves first, and that no two cards have the same value.

For example, suppose the initial cards have values $1, 4, 8, 2$. Elmo takes the 2, because it's larger than 1. Then Daisy takes either 1 or 8 with equal probability. If Daisy takes the 1, then Elmo takes the 8; if Daisy takes the 8, then Elmo takes the 4. Thus, Elmo's expected score is $2 + (8 + 4)/2 = 8$.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.
Red text reflects corrections or clarifications given during the actual exam.

---

1. Suppose we insert n distinct items into an initially empty hash table of size $m \gg n$, using an ***ideal random*** hash function h. Recall that a collision is a set of two distinct items $\{x, y\}$ in the table such that $h(x) = h(y)$.

   (a) What is the exact expected number of collisions?

   (b) Estimate the probability that there are no collisions. *[Hint: Use Markov's inequality.]*

   (c) Estimate the largest value of n such that the probability of having no collisions is at least $1 - 1/n$. Your answer should have the form $n = O(f(m))$ for some simple function f.

   (d) Fix an integer $k > 1$. A ***k-way collision*** is a set of k distinct items $\{x_1, \ldots, x_k\}$ that all have the same hash value: $h(x_1) = h(x_2) = \cdots = h(x_k)$. Estimate the largest value of n such that the probability of having no k-way collisions is at least $1 - 1/n$. Your answer should have the form $n = O(f(m, k))$ for some simple function f. *[Hint: You may want to repeat parts (a) and (b).]*

2. Quentin, Alice, and the other Brakebills Physical Kids are planning an excursion through the Neitherlands to Fillory. The Neitherlands is a vast, deserted city composed of several plazas, each containing a single fountain that can magically transport people to a different world. Adjacent plazas are connected by gates, which have been cursed by the Beast. The gates open for only five minutes every hour, all at the same time. During those five minutes, if more than one person passes through any single gate, the Beast will detect their presence.[1] However, people can safely pass through *different* gates at the same time. Moreover, anyone attempting to pass through more than one gate in the same five-minute period will turn into a niffin.[2]

   You are given a map of the Neitherlands, which is a graph G with a vertex for each fountain and an edge for each gate, with the fountains to Earth and Fillory clearly marked; you are also given a positive integer h. Describe and analyze an algorithm to compute the maximum number of people that can walk from the Earth fountain to the Fillory fountain in h hours, without anyone alerting the Beast or turning into a niffin.

[1] This is very bad.
[2] This is very bad.

3. Recall that a ***Bloom filter*** is an array $B[1..m]$ of bits, together with a collection of k independent ideal random hash functions $h_1, h_2, \ldots, h_k$. To insert an item x into a Bloom filter, we set $B[h_i(x)] \leftarrow 1$ for every index i. To test whether an item x belongs to a set represented by a Bloom filter, we check whether $B[h_i(x)] = 1$ for every index i. This algorithm always returns TRUE if x is in the set, but may return either TRUE or FALSE when x is not in the set. Thus, there may be false positives, but no false negatives.

   If there are n distinct items stored in the Bloom filter, then the probability of a false positive is $(1-p)^k$, where $p \approx e^{-kn/m}$ is the probability that $B[j] = 0$ for any particular index j. In particular, if we set $k = (m/n)\ln 2$, then $p = 1/2$, and the probability of a false positive is $(1/2)^{(m/n)\ln 2} \approx (0.61850)^{m/n}$.

   After months spent lovingly crafting a Bloom filter of size m for a set S of n items, using exactly $k = (m/n)\ln 2$ hash functions (so $p = 1/2$), your boss tells you that you must reduce the size of your Bloom filter from m bits down to $m/2$ bits. Unfortunately, you no longer have the original set S, and your company's product ships tomorrow; you have to do something quick and dirty. Fortunately, your boss has a couple of ideas.

   (a) First your boss suggests simply discarding half of the Bloom filter, keeping only the subarray $B[1..m/2]$. Describe an algorithm to check whether a given item x is an element of the original set S, using only this smaller Bloom filter. As usual, if $x \in S$, your algorithm ***must*** return TRUE.

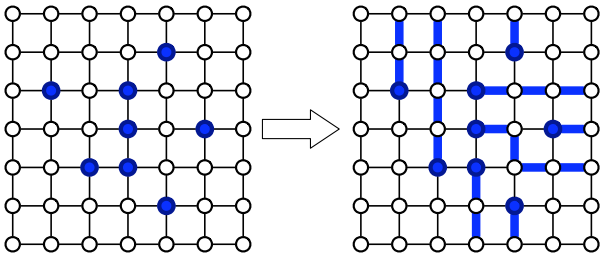
   (b) What is the probability that your algorithm returns TRUE when $x \notin S$?

   (c) Next your boss suggests merging the two halves of your old Bloom filter, defining a new array $B'[1..m/2]$ by setting $B'[i] \leftarrow B[i] \vee B[i + m/2]$ for all i. Describe an algorithm to check whether a given item x is an element of the original set S, using only this smaller Bloom filter B'. As usual, if $x \in S$, your algorithm ***must*** return TRUE.

   (d) What is the probability that your algorithm returns TRUE when $x \notin S$?

4. An $n \times n$ grid is an undirected graph with n^2 vertices organized into n rows and n columns. We denote the vertex in the ith row and the jth column by (i, j). Every vertex (i, j) has exactly four neighbors $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$, except the *boundary* vertices, for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

   Let $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ be distinct vertices, called *terminals*, in the $n \times n$ grid. The ***escape problem*** is to determine whether there are m vertex-disjoint paths in the grid that connect these terminals to any m distinct boundary vertices. Describe and analyze an efficient algorithm to solve the escape problem.



A positive instance of the escape problem, and its solution.

Write your answers in the separate answer booklet.
Please return this question handout and your cheat sheets with your answers.

---

1. Let $G = (V, E)$ be an arbitrary undirected graph. A ***triple-Hamiltonian circuit*** in G is a closed walk in G that visits every vertex of G exactly *three* times. ***Prove*** that it is NP-hard to determine whether a given undirected graph has a triple-Hamiltonian circuit. *[Hint: Modify your reduction for double-Hamiltonian circuits from Homework 10.]*

2. Marie-Joseph Paul Yves Roch Gilbert du Motier, Marquis de Lafayette, colonial America's favorite fighting Frenchman, needs to choose a subset of his ragtag volunteer army of m soldiers to complete a set of n important tasks, like "go to France for more funds" or "come back with more guns". Each task requires a specific set of skills, such as "knows what to do in a trench" or "ingenuitive and fluent in French". For each task, exactly k soldiers are qualified to complete that task.

   Unfortunately, Lafayette's soldiers are extremely lazy. For each task, if Lafayette chooses more than one soldier qualified for that task, each of them will assume that someone else will take on that task, and so the task will never be completed. A task will be completed if and only if exactly one of the chosen soldiers has the necessary skills for that task.

   So Lafayette needs to choose a subset S of soldiers that maximizes the number of tasks for which *exactly one* soldier in S is qualified. Not surprisingly, Lafayette's problem is NP-hard.

   (a) Suppose Lafayette chooses each soldier independently with probability p. What is the *exact* expected number of tasks that will be completed, in terms of p and k?

   (b) What value of p maximizes this expected value?

   (c) Describe a randomized polynomial-time $O(1)$-approximation algorithm for Lafayette's problem. What is the expected approximation ratio for your algorithm?

3. Suppose we are given a set of n rectangular boxes, each specified by their height, width, and depth in centimeters. All three dimensions of each box lie strictly between 10cm and 20cm, and all $3n$ dimensions are distinct. As you might expect, one box can be nested inside another if the first box can be rotated so that is is smaller in every dimension than the second box. Boxes can be nested recursively, but two boxes cannot be nested side-by-side inside a third box. A box is *visible* if it is not nested inside another box.

   Describe and analyze an algorithm to nest the boxes, so that the number of visible boxes is as small as possible.

4. Hercules Mulligan, a tailor spyin' on the British government, has determined a set of routes and towns that the British army plans to use to move their troops from Charleston, South Carolina to Yorktown, Virginia. (He took their measurements, information, and then he smuggled it.) The American revolutionary army wants to set up ambush points in some of these towns, so that every unit of the British army will face at least one ambush before reaching Yorktown. On the other hand, General Washington wants to leave as many troops available as possible to help defend Yorktown when the British army inevitably arrives.

   Describe an efficient algorithm that computes the smallest number of towns where the revolutionary army should set up ambush points. The input to your algorithm is Mulligan's graph of towns (vertices) and routes (edges), with Charleston and Yorktown clearly marked.

5. Consider the following randomized algorithm to approximate the smallest vertex cover in an undirected graph $G = (V, E)$. For each vertex $v \in V$, define the *priority* of v to be a real number between 0 and 1, chosen independently and uniformly at random. Finally, let S be the subset of vertices with higher priority than at least one of their neighbors:

$$S := \left\{ v \in V \;\middle|\; \mathit{priority}(v) > \min_{uv \in E} \mathit{priority}(u) \right\}$$

   (a) What is the probability that the set S is a vertex cover of G? ***Prove*** your answer is correct. (Your proof should be *short*.)

   (b) Suppose the input graph G is a cycle of length n. What is the *exact* expected size of S?

   (c) Suppose the input graph G is a ***star***: a tree with one vertex of degree $n-1$ and $n-1$ vertices of degree 1. What is the *exact* probability that S is the *smallest* vertex cover of G?

   (d) Again, suppose G is a star. Suppose we run the randomized algorithm N times, generating a sequence of subsets $S_1, S_2, \ldots, S_N$. How large must N be to guarantee with high probability that some S_i is the minimum vertex cover of G?

6. After the Revolutionary War, Alexander Hamilton's biggest rival as a lawyer was Aaron Burr. (Sir!) In fact, the two worked next door to each other. Unlike Hamilton, Burr cannot work non-stop; every case he tries exhausts him. The bigger the case, the longer he must rest before he is well enough to take the next case. (Of course, he is willing to wait for it.) If a case arrives while Burr is resting, Hamilton snatches it up instead.

   Burr has been asked to consider a sequence of n upcoming cases. He quickly computes two arrays $\mathit{profit}[1..n]$ and $\mathit{skip}[1..n]$, where for each index i,

   - $\mathit{profit}[i]$ is the amount of money Burr would make by taking the ith case, and
   - $\mathit{skip}[i]$ is the number of consecutive cases Burr must skip if he accepts the ith case. That is, if Burr accepts the ith case, he cannot accept cases $i+1$ through $i+\mathit{skip}[i]$.

   Design and analyze an algorithm that determines the maximum total profit Burr can secure from these n cases, using his two arrays as input.