CS 573: Graduate Algorithms, Fall 2008
Homework 0

Due **in class** at 12:30pm, Wednesday, September 3, 2008

Name:	
Net ID:	Alias:

☐ I understand the course policies.

- Each student must submit their own solutions for this homework. For all future homeworks, groups of up to three students may submit a single, common solution.

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and staple this page to the front of your homework solutions. We will list homework and exam grades on the course web site by alias.

  Federal privacy law and university policy forbid us from publishing your grades, even anonymously, without your explicit written permission. ***By providing an alias, you grant us permission to list your grades on the course web site. If you do not provide an alias, your grades will not be listed.*** For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid) your Social Security number.

- Please carefully read the course policies linked from the course web site. If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup. Once you understand the policies, please check the box at the top of this page. In particular:

  - You may use any source at your disposal—paper, electronic, or human—but you ***must*** write your solutions in your own words, and you ***must*** cite every source that you use.
  - Unless explicitly stated otherwise, ***every*** homework problem requires a proof.
  - Answering "I don't know" to any homework or exam problem is worth 25% partial credit.
  - Algorithms or proofs containing phrases like "and so on" or "repeat this for all n", instead of an explicit loop, recursion, or induction, will receive 0 points.

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, discrete probability, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The early chapters of any algorithms textbook should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks. If you need help, please ask in office hours and/or on the course newsgroup.

---

1. (a) **[5 pts]** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases. If your solution requires a particular base case, say so.

$$A(n) = 4A(n/8) + \sqrt{n}$$

$$B(n) = B(n/3) + 2B(n/4) + B(n/6) + n$$

$$C(n) = 6C(n-1) - 9C(n-2)$$

$$D(n) = \max_{n/3<k<2n/3} \big(D(k) + D(n-k) + n\big)$$

$$E(n) = \big(E(\sqrt{n})\big)^2 \cdot n$$

   (b) **[5 pts]** Sort the functions in the box from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do *not* turn in proofs**—just a sorted list of 16 functions—but you should do them anyway, just for practice. We use the notation $\lg n = \log_2 n$.

n	$\lg n$	$\sqrt{n}$	3^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$3^{\sqrt{n}}$	$\sqrt{3^n}$
$3^{\lg n}$	$\lg(3^n)$	$3^{\lg \sqrt{n}}$	$3\sqrt{\lg n}$
$\sqrt{3^{\lg n}}$	$\lg(3^{\sqrt{n}})$	$\lg \sqrt{3^n}$	$\sqrt{\lg(3^n)}$

2. Describe and analyze a data structure that stores set of n records, each with a numerical *key* and a numerical *priority*, such that the following operation can be performed quickly:

   - RANGETOP(a, z) : return the highest-priority record whose key is between a and z.

   For example, if the (key, priority) pairs are

$$(3,1),\ (4,9),\ (9,2),\ (6,3),\ (5,8),\ (7,5),\ (1,10),\ (0,7),$$

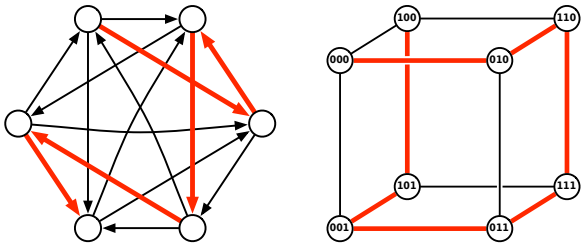
   then RANGETOP$(2,8)$ would return the record with key 4 and priority 9 (the second in the list).

   Analyze both the size of your data structure and the running time of your RANGETOP algorithm. For full credit, your space and time bounds must both be as small as possible. You may assume that no two records have equal keys or equal priorities, and that no record has a or z as its key. *[Hint: How would you compute the number of keys between a and z? How would you solve the problem if you knew that a is always $-\infty$?]*

3. A *Hamiltonian path* in G is a path that visits every vertex of G exactly once. In this problem, you are asked to prove that two classes of graphs always contain a Hamiltonian path.

   (a) **[5 pts]** A *tournament* is a directed graph with exactly one edge between each pair of vertices. (Think of the nodes in a round-robin tournament, where edges represent games, and each edge points from the loser to the winner.) Prove that every tournament contains a *directed* Hamiltonian path.

   (b) **[5 pts]** Let d be an arbitrary non-negative integer. The d-dimensional *hypercube* is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if and only if their labels differ in exactly one bit. Prove that the d-dimensional hypercube contains a Hamiltonian path.



Hamiltonian paths in a 6-node tournament and a 3-dimensional hypercube.

4. Penn and Teller agree to play the following game. Penn shuffles a standard deck[1] of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

   The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[2] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

   (a) What is the expected number of cards that Teller draws?

   (b) What is the expected *maximum* value among the cards Teller gives to Penn?

   (c) What is the expected *minimum* value among the cards Teller gives to Penn?

   (d) What is the expected number of cards that Teller gives to Penn?

   Full credit will be given only for *exact* answers (with correct proofs, of course). *[Hint: Let* $13 = n$*.]*

[1] In a standard deck of playing cards, each card has a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$ and a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$; each of the 52 possible suit-value pairs appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

[2] Specifically, he hurls them from the opposite side of the stage directly into the back of Penn's right hand. Ouch!

5. (a) The ***Fibonacci numbers F_n*** are defined by the recurrence $F_n = F_{n-1} + F_{n-2}$, with base cases $F_0 = 0$ and $F_1 = 1$. Here are the first several Fibonacci numbers:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Prove that any non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers. That is, if the Fibonacci number F_i appears in the sum, it appears exactly once, and its neighbors F_{i-1} and F_{i+1} do not appear at all. For example:

$$17 = F_7 + F_4 + F_2, \qquad 42 = F_9 + F_6, \qquad 54 = F_9 + F_7 + F_5 + F_3 + F_1.$$

(b) The Fibonacci sequence can be extended backward to negative indices by rearranging the defining recurrence: $F_n = F_{n+2} - F_{n+1}$. Here are the first several negative-index Fibonacci numbers:

F_{-10}	F_{-9}	F_{-8}	F_{-7}	F_{-6}	F_{-5}	F_{-4}	F_{-3}	F_{-2}	F_{-1}
−55	34	−21	13	−8	5	−3	2	−1	1

Prove that $F_{-n} = -F_n$ if and only if n is even.

(c) Prove that *any* integer—positive, negative, or zero—can be written as the sum of distinct, non-consecutive Fibonacci numbers *with negative indices*. For example:

$$17 = F_{-7} + F_{-5} + F_{-2}, \qquad -42 = F_{-10} + F_{-7}, \qquad 54 = F_{-9} + F_{-7} + F_{-5} + F_{-3} + F_{-1}.$$

[Hint: Zero is both non-negative and even. Don't use weak induction!]

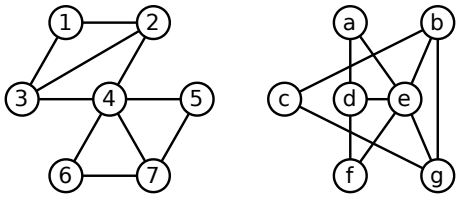
CS 573: Graduate Algorithms, Fall 2008
Homework 1

Due at 11:59:59pm, Wednesday, September 17, 2008

For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and alias (if any) of every group member on the first page of your submission.

---

1. Two graphs are said to be ***isomorphic*** if one can be transformed into the other just by relabeling the vertices. For example, the graphs shown below are isomorphic; the left graph can be transformed into the right graph by the relabeling $(1, 2, 3, 4, 5, 6, 7) \mapsto (c, g, b, e, a, f, d)$.



Two isomorphic graphs.

Consider the following related decision problems:

- GRAPHISOMORPHISM: Given two graphs G and H, determine whether G and H are isomorphic.
- EVENGRAPHISOMORPHISM: Given two graphs G and H, such that every vertex in G and H has even degree, determine whether G and H are isomorphic.
- SUBGRAPHISOMORPHISM: Given two graphs G and H, determine whether G is isomorphic to a subgraph of H.

(a) Describe a polynomial-time reduction from EVENGRAPHISOMORPHISM to GRAPHISOMORPHISM.

(b) Describe a polynomial-time reduction from GRAPHISOMORPHISM to EVENGRAPHISOMORPHISM.

(c) Describe a polynomial-time reduction from GRAPHISOMORPHISM to SUBGRAPHISOMORPHISM.

(d) Prove that SUBGRAPHISOMORPHISM is NP-complete.

(e) What can you conclude about the NP-hardness of GRAPHISOMORPHISM? Justify your answer.

[Hint: These are all easy!]

2. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-complete.

(b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-complete. *[Hint: Use part (a).]*

3. The following variant of 3SAT is called either EXACT3SAT or 1IN3SAT, depending on who you ask.

> Given a boolean formula in conjunctive normal form with 3 literals per clause, is there an assignment that makes ***exactly one*** literal in each clause TRUE?

Prove that this problem is NP-complete.

4. Suppose you are given a magic black box that can solve the MAXCLIQUE problem ***in polynomial time***. That is, given an arbitrary graph G as input, the magic black box computes the number of vertices in the largest complete subgraph of G. Describe and analyze a ***polynomial-time*** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

5. A boolean formula in ***exclusive-or conjunctive normal form*** (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive*-or of several literals. The XCNF-SAT problem asks whether a given XCNF boolean formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-complete.

*6. ***[Extra credit]*** Describe and analyze an algorithm to solve 3SAT in $O(\phi^n \operatorname{poly}(n))$ time, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*

[0]In class, I asserted that Gaussian elimination was probably discovered by Gauss, in violation of Stigler's Law of Eponymy. In fact, a method very similar to Gaussian elimination appears in the Chinese treatise *Nine Chapters on the Mathematical Art*, believed to have been finalized before 100AD, although some material may predate emperor Qin Shi Huang's infamous 'burning of the books and burial of the scholars' in 213BC. The great Chinese mathematician Liu Hui, in his 3rd-century commentary on *Nine Chapters*, compares two variants of the method and counts the number of arithmetic operations used by each, with the explicit goal of find the more efficient method. This is arguably the earliest recorded *analysis* of any algorithm.

CS 573: Graduate Algorithms, Fall 2008
Homework 2

Due at 11:59:59pm, Wednesday, October 1, 2008

- For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and alias (if any) of every group member on the first page of your submission.
- We will use the following point breakdown to grade dynamic programming algorithms: 60% for a correct recurrence (including base cases), 20% for correct running time analysis of the memoized recurrence, 10% for correctly transforming the memoized recursion into an iterative algorithm.
- A greedy algorithm *must* be accompanied by a proof of correctness in order to receive *any* credit.

---

1. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays of numbers. A ***common supersequence*** of X and Y is another sequence that contains both X and Y as subsequences. Describe and analyze an efficient algorithm to compute the function $scs(X, Y)$, which gives the length of the *shortest* common supersequence of X and Y.

   (b) Call a sequence $X[1..n]$ of numbers ***oscillating*** if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe and analyze an efficient algorithm to compute the function $los(X)$, which gives the length of the longest oscillating subsequence of an arbitrary array X of integers.

   (c) Call a sequence $X[1..n]$ of numbers ***accelerating*** if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Describe and analyze an efficient algorithm to compute the function $lxs(X)$, which gives the length of the longest accelerating subsequence of an arbitrary array X of integers.

2. A *palindrome* is a string that reads the same forwards and backwards, like `x`, `pop`, `noon`, `redivider`, or `amanaplanacatahamayakayamahatacanalpanama`. Any string can be broken into sequence of palindromes. For example, the string `bubbaseesabanana` ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example:

   `bub + baseesab + anana`
   `b + u + bb + a + sees + aba + nan + a`
   `b + u + bb + a + sees + a + b + anana`
   `b + u + b + b + a + s + e + e + s + a + b + a + n + a + n + a`

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string `bubbaseesabanana`, your algorithm would return the integer 3.

3. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \operatorname{poly}(n))$ time. Given an undirected n-vertex graph G with weighted edges, your algorithm should return the weight of the lightest Hamiltonian cycle in G, or ∞ if G has no Hamiltonian cycles. *[Hint: The obvious recursive algorithm takes $O(n!)$ time.]*

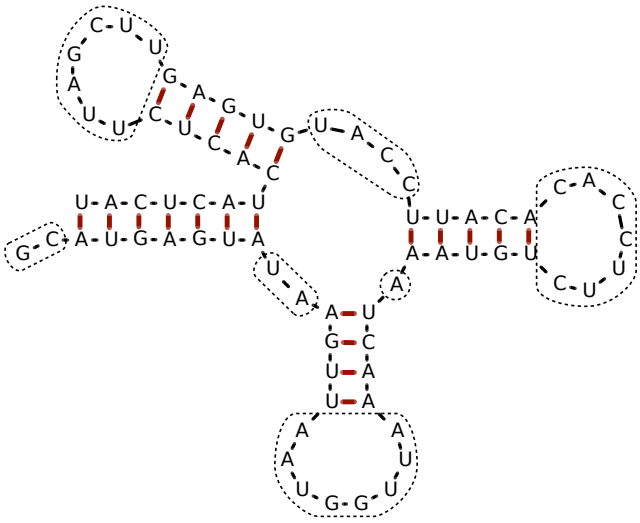
4. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{A, C, G, U\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule.

   We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ ***overlap*** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

   Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

   - Each base can be paired with at most one other base.
   - Only A-U pairs and C-G pairs can bond.
   - Pairs of the form $(i, i+1)$ and $(i, i+2)$ cannot bond.
   - Overlapping base pairs cannot bond.

   The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.



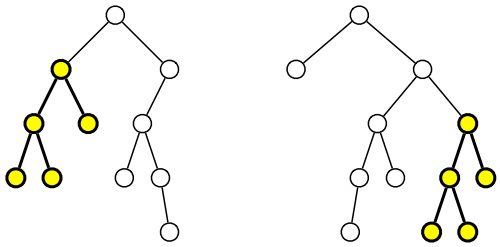
Example RNA secondary structure with 21 base pairs, indicated by heavy red lines.
Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$

   (a) Describe and analyze an algorithm that computes the maximum possible *number* of base pairs in a secondary structure for a given RNA sequence.

   (b) A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths.[1] Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

[1] This score function has absolutely no connection to reality; I just made it up. Real RNA structure prediction requires much more complicated scoring functions.

5. A *subtree* of a (rooted, ordered) binary tree T consists of a node and all its descendants. Design and analyze an efficient algorithm to compute the ***largest common subtree*** of two given binary trees T_1 and T_2; this is the largest subtree of T_1 that is isomorphic to a subtree in T_2. The contents of the nodes are irrelevant; we are only interested in matching the underlying combinatorial structure.



Two binary trees, with their largest common subtree emphasized

*6. ***[Extra credit]*** Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. A ***digital subsequence*** of D is an sequence of positive integers composed in the usual way from disjoint substrings of D. For example, $3, 4, 5, 6, 23, 38, 62, 64, 83, 279$ is an increasing digital subsequence of the first several digits of π:

$$\boxed{3}, 1, \boxed{4}, 1, \boxed{5}, 9, \boxed{6}, \boxed{2,3}, 4, \boxed{3,8}, 4, \boxed{6,2}, \boxed{6,4}, 3, 3, \boxed{8,3}, \boxed{2,7,9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the previous example has length 10.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: Be careful about your computational assumptions. How long does it take to compare two k-digit numbers?]*

CS 573: Graduate Algorithms, Fall 2008
Homework 3

Due at 11:59:59pm, Wednesday, October 22, 2008

- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.

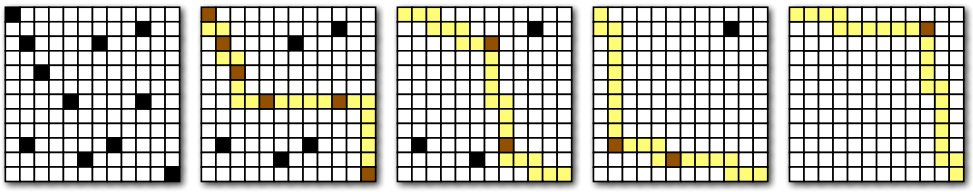
---

1. Consider an $n \times n$ grid, some of whose cells are marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. We want to compute the minimum number of monotone paths that cover all marked cells. The input to our problem is an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \textsc{True}$ if and only if cell (i, j) is marked.

   One of your friends suggests the following greedy strategy:

   - Find (somehow) one "good" path π that covers the maximum number of marked cells.
   - Unmark the cells covered by π.
   - If any cells are still marked, recursively cover them.

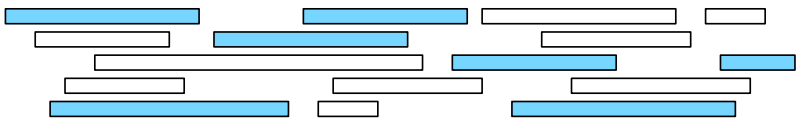
   Does this greedy strategy always compute an optimal solution? If yes, give a proof. If no, give a counterexample.



Greedily covering the marked cells in a grid with four monotone paths.

2. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a ***tiling path*** if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling path is just the number of intervals.

   Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.



A set of intervals. The seven shaded intervals form a tiling path.

3. Given a graph G with edge weights and an integer k, suppose we wish to partition the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (*i.e.*, that have endpoints in different subsets) is as large as possible.

   (a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem. *[Hint: Solve the special case $k = 2$ first.]*

   (b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for this new problem? Justify your answer.

4. Consider the following heuristic for constructing a vertex cover of a connected graph G: ***Return the set of all non-leaf nodes of any depth-first spanning tree.*** (Recall that a depth-first spanning tree is a *rooted* tree; the root is not considered a leaf, even if it has only one neighbor in the tree.)

   (a) Prove that this heuristic returns a vertex cover of G.

   (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.

   (c) Prove that for any $\varepsilon > 0$, there is a graph for which this heuristic returns a vertex cover of size at least $(2 - \varepsilon) \cdot OPT$.

5. Consider the following greedy approximation algorithm to find a vertex cover in a graph:

```
GreedyVertexCover(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

   In class we proved that the approximation ratio of this algorithm is $O(\log n)$; your task is to prove a matching lower bound. Specifically, for any positive integer n, describe an n-vertex graph G such that GreedyVertexCover(G) returns a vertex cover that is $\Omega(\log n)$ times larger than optimal. *[Hint: $H_n = \Omega(\log n)$.]*

*6. ***[Extra credit]*** Consider the greedy algorithm for metric TSP: Start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

   (a) Prove that this greedy algorithm is an $O(\log n)$-approximation algorithm, where n is the number of vertices. *[Hint: Show that the kth least expensive edge in the tour output by the greedy algorithm has weight at most $OPT/(n - k + 1)$; try $k = 1$ and $k = 2$ first.]*

   *(b) Prove that the greedy algorithm for metric TSP is no better than an $O(\log n)$-approximation. That is, describe an infinite family of weighted graphs that satisfy the triangle inequality, such that the greedy algorithm returns a cycle whose length is $\Omega(\log n)$ times the optimal TSP tour.

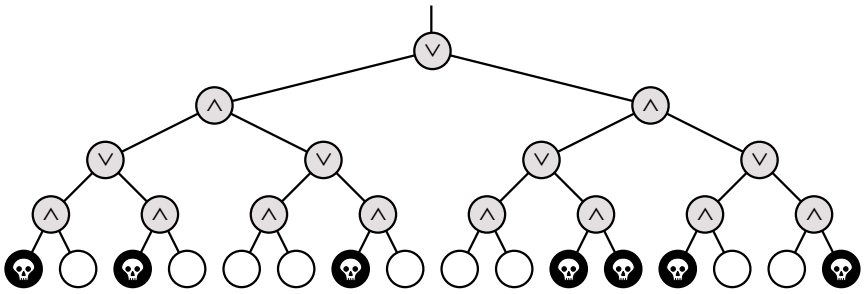
CS 573: Graduate Algorithms, Fall 2008
Homework 4

Due at 11:59:59pm, Wednesday, October 31, 2008

- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.
- Unless a problem explicitly states otherwise, you can assume the existence of a function RANDOM(k), which returns an integer uniformly distributed in the range $\{1, 2, \ldots, k\}$ in $O(1)$ time; the argument k must be a positive integer. For example, RANDOM(2) simulates a fair coin flip, and RANDOM(1) *always* returns 1.

---

1. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black leaves stand represent TRUE and FALSE inputs, respectively. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

   (a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

   (b) Unfortunately, Death won't let you even look at every node in the tree. Describe and analyze a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

   (c) ***[Extra credit]*** Describe and analyze a *randomized* algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm at all.]*

2. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

   What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. *[Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]*

3. (a) Prove that the expected number of proper descendants of any node in a treap is *exactly* equal to the expected depth of that node.

   (b) Why doesn't the Chernoff-bound argument for depth imply that, with high probability, *every* node in a treap has $O(\log n)$ descendants? The conclusion is obviously bogus—every n-node treap has one node with exactly n descendants!—but what is the flaw in the argument?

   (c) What is the expected number of leaves in an n-node treap? *[Hint: What is the probability that in an n-node treap, the node with kth smallest search key is a leaf?]*

4. The following randomized algorithm, sometimes called "one-armed quicksort", selects the rth smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call RANDOMSELECT$(A, 1)$; to find the median element, you would call RANDOMSELECT$(A, \lfloor n/2 \rfloor)$. The subroutine PARTITION$(A[1..n], p)$ splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n-1$ comparisons altogether, and returns the new index of the pivot element.

```
RandomSelect(A[1..n], r):
  k ← Partition(A[1..n], Random(n))
  if r < k
      return RandomSelect(A[1..k-1], r)
  else if r > k
      return RandomSelect(A[k+1..n], r-k)
  else
      return A[k]
```

   (a) State a recurrence for the expected running time of RANDOMSELECT, as a function of n and r.

   (b) What is the *exact* probability that RANDOMSELECT compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and r. *[Hint: Check your answer by trying a few small examples.]*

   (c) Show that for any n and r, the expected running time of RANDOMSELECT is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b).

   (d)* ***[Extra Credit]*** Find the *exact* expected number of comparisons executed by RANDOMSELECT, as a function of n and r.

5. 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 is the total number of nodes in both trees. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if 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. *[Hint: You don't need Chernoff bounds, but you might use the identity $\binom{ck}{k} \le (ce)^k$.]*

   (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. (This implies that every operation takes $O(\log n)$ time with high probability.)

*6. ***[Extra credit]*** In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0,1]$, but in practice, computers only have access to random *bits*. This problem asks you to analyze a modification of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1..\infty]$ of random bits, which is interpreted as the rational number

$$\mathit{priority}(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or 'reveal') new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LargerPriority(v, w):
  for i ← 1 to ∞
      if i > ℓ_v
          ℓ_v ← i;  π_v[i] ← RandomBit
      if i > ℓ_w
          ℓ_w ← i;  π_w[i] ← RandomBit
      if π_v[i] > π_w[i]
          return v
      else if π_v[i] < π_w[i]
          return w
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LargerPriority during these insertions.

(a) Prove that $E[L] = \Theta(n)$.

(b) Prove that $E[\ell_v] = \Theta(1)$ for any node v. *[Hint: This is equivalent to part (a). Why?]*

(c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. *[Hint: Why doesn't this contradict part (b)?]*

CS 573: Graduate Algorithms, Fall 2008
Homework 5

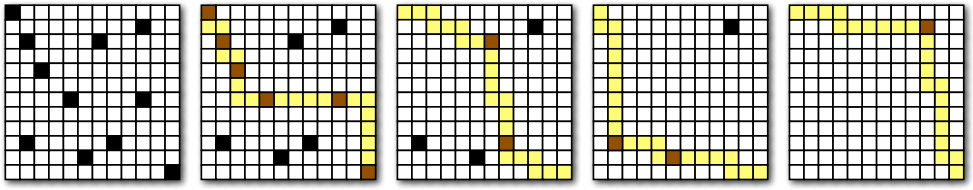
Due at 11:59:59pm, Wednesday, November 19, 2008

- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.

---

1. Recall the following problem from Homework 3: You are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{TRUE}$ if and only if cell (i, j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell.

   Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.



Greedily covering the marked cells in a grid with four monotone paths.

2. Suppose we are given a directed graph $G = (V, E)$, two vertices s an t, and a capacity function $c : V \to \mathbb{R}^+$. A flow f is *feasible* if the total flow into every vertex v is at most $c(v)$:

$$\sum_u f(u \to v) \le c(v) \quad \text{for every vertex } v.$$

   Describe and analyze an efficient algorithm to compute a feasible flow of maximum value.

3. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want 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 an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

4. *Ad-hoc networks* are made up of cheap, low-powered wireless devices. In principle[1], these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other situations where people might want to monitor conditions in hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be dropped into the area from an airplane (for instance), and then they would somehow automatically configure themselves into an efficiently functioning wireless network.

   The devices can communication only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

   We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit the information it has to some other *backup* device within its communication range. To improve reliability, we require each device x to have k potential backup devices, all within distance D of x; we call these k devices the ***backup set*** of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

   So suppose we are given the communication radius D, parameters b and k, and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe an algorithm that either computes a backup set of size k for each of the n devices, such that that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

5. Let $G = (V, E)$ be a directed graph where for each vertex v, the in-degree and out-degree of v are equal. Let u and v be two vertices G, and suppose G contains k edge-disjoint paths from u to v. Under these conditions, must G also contain k edge-disjoint paths from v to u? Give a proof or a counterexample with explanation.

*6. ***[Extra credit]*** A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe a polynomial-time algorithm to compute, given two rooted trees A and B, the largest common rooted subtree of A and B.

   [Hint: Let $LCS(u, v)$ denote the largest common subtree whose root in A is u and whose root in B is v. Your algorithm should compute $LCS(u, v)$ for all vertices u and v using dynamic programming. This would be easy if every vertex had $O(1)$ children, and still straightforward if the children of each node were ordered from left to right and the common subtree had to respect that ordering. But for unordered trees with large degree, you need another trick to combine recursive subproblems efficiently. Don't waste your time trying to reduce the polynomial running time.]

[1] but not really in practice

CS 573: Graduate Algorithms, Fall 2008
Homework 6

Practice only

- This homework is only for practice; do not submit solutions. At least one (sub)problem (or something *very* similar) will appear on the final exam.

---

1. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.

   (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.

   (b) Prove that finding the optimal solution to an integer program is NP-hard.

   [Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]

2. Describe precisely how to dualize a linear program written in general form:

$$\begin{aligned}
\text{maximize } & \sum_{j=1}^{d} c_j x_j \\
\text{subject to } & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1..p \\
& \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for each } i = p+1..p+q \\
& \sum_{j=1}^{d} a_{ij} x_j \ge b_i \quad \text{for each } i = p+q+1..n
\end{aligned}$$

   Keep the number of dual variables as small as possible. The dual of the dual of any linear program should be *syntactically identical* to the original linear program.

3. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time, using this subroutine as a black box. Your algorithm should return an optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one constraint to guarantee boundedness; add one variable to guarantee feasibility.]*

4. Suppose you are given a set P of n points in some high-dimensional space $\mathbb{R}^d$, each labeled either *black* or *white*. A *linear classifier* is a d-dimensional vector c with the following properties:

   - If p is a black point, then $p \cdot c > 0$.
   - If p is a while point, then $p \cdot c < 0$.

   Describe an efficient algorithm to find a linear classifier for the given data points, or correctly report that none exists. *[Hint: This is almost trivial, but not quite.]*

Lots more linear programming problems can be found at http://www.ee.ucla.edu/ee236a/homework/problems.pdf. Enjoy!

You have 120 minutes to answer all five questions.
Write your answers in the separate answer booklet.
Please turn in your question sheet and your cheat sheet with your answers.

1. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, several cards are dealt face up in a long row. Then you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. Each card is worth a different number of points. The player that collects the most points wins the game.

   Like most eight-year-olds who haven't studied algorithms, Elmo follows the obvious greedy strategy every time he plays: ***Elmo always takes the card with the higher point value.*** Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

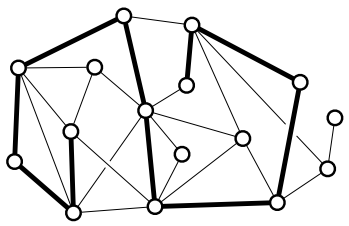
   (a) Describe an initial sequence of cards that allows you to win against Elmo, no matter who moves first, but *only* if you do *not* follow Elmo's greedy strategy.

   (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

   Here is a sample game, where both you and Elmo are using the greedy strategy. Elmo wins 8–7. You cannot win this particular game, no matter what strategy you use.

Initial cards	2	4	5	1	3
Elmo takes the 3	2	4	5	1	~~3~~
You take the 2	~~2~~	4	5	1	
Elmo takes the 4		~~4~~	5	1	
You take the 5			~~5~~	1	
Elmo takes the 1				~~1~~	

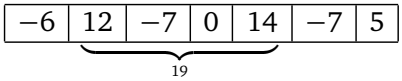
2. ***Prove*** that the following problem is NP-hard: Given an undirected graph G, find the longest path in G whose length is a multiple of 5.



This graph has a path of length 10, but no path of length 15.

3. Suppose you are given an array $A[1..n]$ of integers. Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i..j]$.

   For example, if the array A contains the numbers $[-6, 12, -7, 0, 14, -7, 5]$, your algorithm should return the number 19:



4. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, 'bananaaananas' is a shuffle of 'banana' and 'ananas' in several different ways:

$$\texttt{banana}_{\texttt{ananas}} \qquad \texttt{ban}_{\texttt{ana}}\texttt{ana}_{\texttt{nas}} \qquad \texttt{b}_{\texttt{an}}\texttt{an}_{\texttt{a}}\texttt{a}_{\texttt{na}}\texttt{na}_{\texttt{s}}$$

   The strings 'prodgyrnamammiincg' and 'dyprongarmammicing' are both shuffles of 'dynamic' and 'programming':

$$\texttt{pro}^{\texttt{d}}_{\texttt{g}}{}^{\texttt{y}}_{\texttt{r}}\texttt{nam}_{\texttt{ammi}}{}^{\texttt{i}}_{\texttt{n}}{}^{\texttt{c}}_{\texttt{g}} \qquad \texttt{dy}_{\texttt{pro}}{}^{\texttt{n}}_{\texttt{g}}{}^{\texttt{a}}_{\texttt{r}}{}^{\texttt{m}}_{\texttt{amm}}{}^{\texttt{ic}}_{\texttt{ing}}$$

   Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

5. Suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in $\Theta(\log(m+n))$ time. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..5] = [2, 5, 8, 17, 19] \qquad k = 6$$

   your algorithm should return 8. You can assume that the arrays contain no duplicates. An algorithm that works only in the special case $n = m = k$ is worth 7 points.

   [Hint: What can you learn from comparing one element of A to one element of B?]

You have 120 minutes to answer all five questions.
Write your answers in the separate answer booklet.
Please turn in your question sheet and your cheat sheet with your answers.

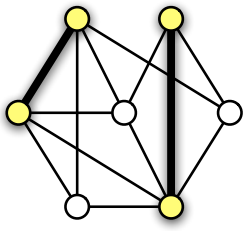
1. Consider the following modification of the 'dumb' 2-approximation algorithm for minimum vertex cover that we saw in class. The only change is that we output a set of edges instead of a set of vertices.

```
APPROXMINMAXMATCHING(G):
  M ← ∅
  while G has at least one edge
      let (u, v) be any edge in G
      remove u and v (and their incident edges) from G
      add (u, v) to M
  return M
```

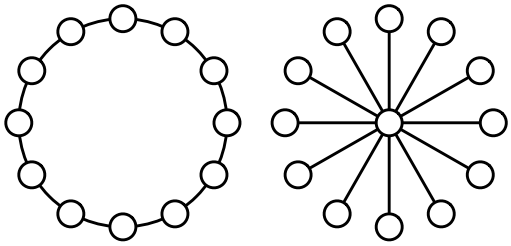
   (a) ***Prove*** that this algorithm computes a *matching*—no two edges in M share a common vertex.

   (b) ***Prove*** that M is a *maximal* matching—M is not a proper subgraph of another matching in G.

   (c) ***Prove*** that M contains at most twice as many edges as the *smallest* maximal matching in G.



The smallest maximal matching in a graph.



A cycle and a star.

2. Consider the following heuristic for computing a small vertex cover of a graph.

   - Assign a random *priority* to each vertex, chosen independently and uniformly from the real interval $[0, 1]$ (just like treaps).
   - Mark every vertex that does *not* have larger priority than *all* of its neighbors.

   For any graph G, let $OPT(G)$ denote the size of the smallest vertex cover of G, and let $M(G)$ denote the number of nodes marked by this algorithm.

   (a) ***Prove*** that the set of vertices marked by this heuristic is *always* a vertex cover.

   (b) Suppose the input graph G is a *cycle*, that is, a connected graph where every vertex has degree 2. What is the expected value of $M(G)/OPT(G)$? ***Prove*** your answer is correct.

   (c) Suppose the input graph G is a *star*, that is, a tree with one central vertex of degree $n-1$. What is the expected value of $M(G)/OPT(G)$? ***Prove*** your answer is correct.

3. Suppose we want to write an efficient function SHUFFLE($A[1..n]$) that randomly permutes the array A, so that each of the $n!$ permutations is equally likely.

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

   ```
   SHUFFLE(A[1..n]):
     for i = 1 to n
         swap A[i] ↔ A[RANDOM(n)]
   ```

   (b) Describe and analyze a correct SHUFFLE algorithm whose expected running time is $O(n)$. Your algorithm may call the function RANDOM(k), which returns an integer uniformly distributed in the range $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, RANDOM(2) simulates a fair coin flip, and RANDOM(1) *always* returns 1.

4. Let Φ be a legal input for 3SAT—a boolean formula in conjunctive normal form, with exactly three literals in each clause. Recall that an assignment of boolean values to the variables in Φ ***satisfies*** a clause if at least one of its literals is TRUE. The ***maximum satisfiability problem***, sometimes called MAX3SAT, asks for the maximum number of clauses that can be simultaneously satisfied by a single assignment. Solving MAXSAT exactly is clearly also NP-hard; this problem asks about approximation algorithms.

   (a) Let $MaxSat(\Phi)$ denote the maximum number of clauses that can be simultaneously satisfied by one variable assignment. Suppose we randomly assign each variable in Φ to be TRUE or FALSE, each with equal probability. ***Prove*** that the expected number of satisfied clauses is at least $\frac{7}{8}MaxSat(\Phi)$.

   (b) Let $MinUnsat(\Phi)$ denote the *minimum* number of clauses that can be simultaneously *unsatisfied* by a single assignment. ***Prove*** that it is NP-hard to approximate $MinUnsat(\Phi)$ within a factor of $10^{10^{100}}$.

5. Consider the following randomized algorithm for generating biased random bits. The subroutine FAIRCOIN returns either 0 or 1 with equal probability; the random bits returned by FAIRCOIN are mutually independent.

   ```
   ONEINTHREE:
     if FAIRCOIN = 0
         return 0
     else
         return 1 − ONEINTHREE
   ```

   (a) ***Prove*** that ONEINTHREE returns 1 with probability 1/3.

   (b) What is the *exact* expected number of times that this algorithm calls FAIRCOIN? ***Prove*** your answer is correct.

   (c) Now suppose you are *given* a subroutine ONEINTHREE that generates a random bit that is equal to 1 with probability 1/3. Describe a FAIRCOIN algorithm that returns either 0 or 1 with equal probability, using ONEINTHREE as a subroutine. **Your *only* source of randomness is ONEINTHREE; in particular, you may not use the RANDOM function from problem 3.**

   (d) What is the *exact* expected number of times that your FAIRCOIN algorithm calls ONEINTHREE? ***Prove*** your answer is correct.

You have 180 minutes to answer all seven questions.
Write your answers in the separate answer booklet.
You can keep everything except your answer booklet when you leave.

1. An *integer program* is a linear program with the additional constraint that the variables must take only integer values. ***Prove*** that deciding whether an integer program has a feasible solution is NP-complete. *[Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]*

2. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A *heater* is a priority search tree in which the priorities are given by the user, and the search keys are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is the 'opposite' of a treap.

   The following problems consider an n-node heater T whose node priorities are the integers from 1 to n. We identify nodes in T by their priorities; thus, 'node 5' means the node in T with priority 5. The min-heap property implies that node 1 is the root of T. Finally, let i and j be integers with $1 \leq i < j \leq n$.

   (a) ***Prove*** that in a random permutation of the $(i+1)$-element set $\{1, 2, \ldots, i, j\}$, elements i and j are adjacent with probability $2/(i+1)$.

   (b) ***Prove*** that node i is an ancestor of node j with probability $2/(i+1)$. *[Hint: Use part (a)!]*

   (c) What is the probability that node i is a *descendant* of node j? *[Hint: **Don't** use part (a)!]*

   (d) What is the *exact* expected depth of node j?

3. The UIUC Faculty Senate has decided to convene a committee to determine whether Chief Illiniwek should become the official ~~mascot~~ *symbol* of the University of Illinois Global Campus. Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member will represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that any committee on virtual ~~mascots~~ *symbols* must contain the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.

   Describe an efficient algorithm to select the membership of the Global Illiniwek Committee. Your input is a list of all UIUC faculty members, their ranks (assistant, associate, or full), and their departmental affiliation(s). There are n faculty members and $3k$ departments.

4. Let $\alpha(G)$ denote the number of vertices in the largest independent set in a graph G. ***Prove*** that the following problem is NP-hard: Given a graph G, return *any* integer between $\alpha(G) - 31337$ and $\alpha(G) + 31337$.

5. Let $G = (V, E)$ be a directed graph with capacities $c : E \to \mathbb{R}^+$, a source vertex s, and a target vertex t. Suppose someone hands you a function $f : E \to \mathbb{R}$. Describe and analyze a fast algorithm to determine whether f is a maximum (s, t)-flow in G.

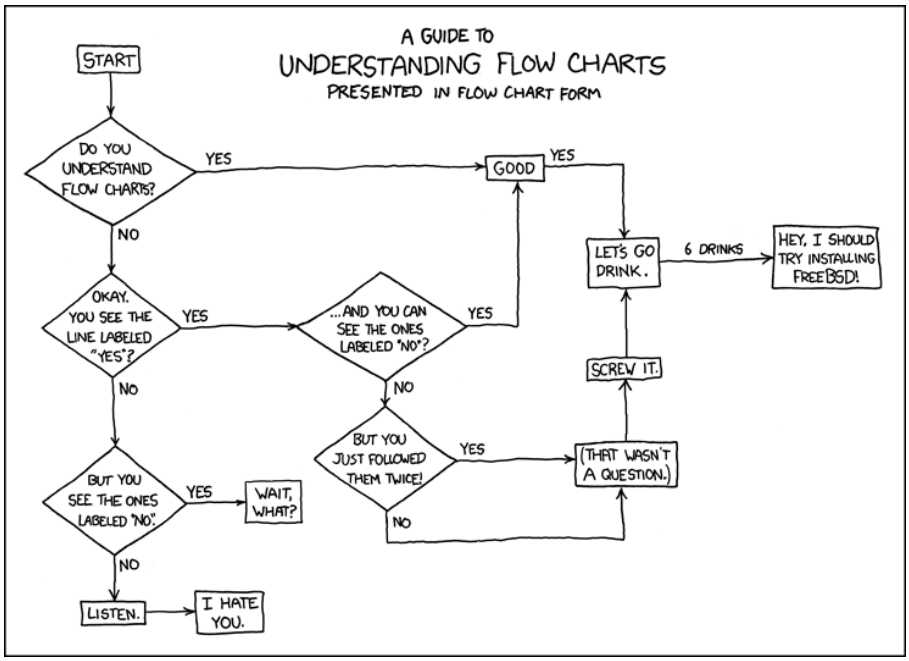
6. For some strange reason, you decide to ride your bicycle 3688 miles from Urbana to Wasilla, Alaska, to join in the annual Wasilla Mining Festival and Helicopter Wolf Hunt. The festival starts exactly 32 days from now, so you need to bike an average of 109 miles each day. Because you are a poor starving student, you can only afford to sleep at campgrounds, which are unfortunately *not* spaced exactly 109 miles apart. So some days you will have to ride more than average, and other days less, but you would like to keep the variation as small as possible. You settle on a formal scoring system to help decide where to sleep; if you ride x miles in one day, your score for that day is $(109 - x)^2$. What is the minimum possible total score for all 32 days?

   More generally, suppose you have D days to travel DP miles, there are n campgrounds along your route, and your score for traveling x miles in one day is $(x - P)^2$. You are given a sorted array $dist[1..n]$ of real numbers, where $dist[i]$ is the distance from your starting location to the ith campground; it may help to also set $dist[0] = 0$ and $dist[n + 1] = DP$. Describe and analyze a fast algorithm to compute the minimum possible score for your trip. The running time of your algorithm should depend on the integers D and n, but not on the real number P.

7. Describe and analyze efficient algorithms for the following problems.

   (a) Given a set of n integers, does it contain elements a and b such that $a + b = 0$?

   (b) Given a set of n integers, does it contain elements a, b, and c such that $a + b = c$?



— Randall Munroe, *xkcd*, December 17, 2008 (http://xkcd.com/518/)