# CS 473G: Graduate Algorithms, Spring 2007
# Homework 0

Due **in class** at 11:00am, Tuesday, January 30, 2007

| Name: | |
|---|---|
| Net ID: | Alias: |

☐ I understand the Course Policies.

---

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and staple this page to your solution to problem 1. We will list homework and exam grades on the course web site by alias. **By providing an alias, you agree to let us list your grades; 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 use the same alias for every homework and exam.

- Read the Course Policies on the course web site, and then check the box above. Among other things, this page describes what we expect in your homework solutions, as well as policies on grading standards, regrading, extra credit, and plagiarism. In particular:

  - Submit each numbered problem separately, on its own piece(s) of paper. If you need more than one page for a problem, staple just *those* pages together, but keep different problems separate. **Do not staple your entire homework together.**
  - You may use *any* source at your disposal—paper, electronic, or human—but you *must* write your answers in your own words, and you *must* cite every source that you use.
  - Algorithms or proofs containing phrases like "and so on" or "repeat this for all $n$", instead of an explicit loop, recursion, or induction, are worth zero points.
  - Answering "I don't know" to any homework or exam problem is worth 25% partial credit.

  If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup.

- 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 knowledge. **You are responsible for filling those gaps on your own.** The early chapters of Kleinberg and Tardos (or any algorithms textbook) should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks.

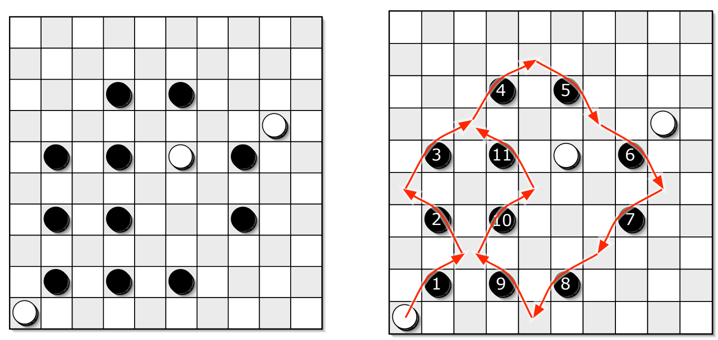
- Every homework will have five problems, each worth 10 points. Stars indicate more challenging problems. Many homeworks will also include an extra-credit problem.

---

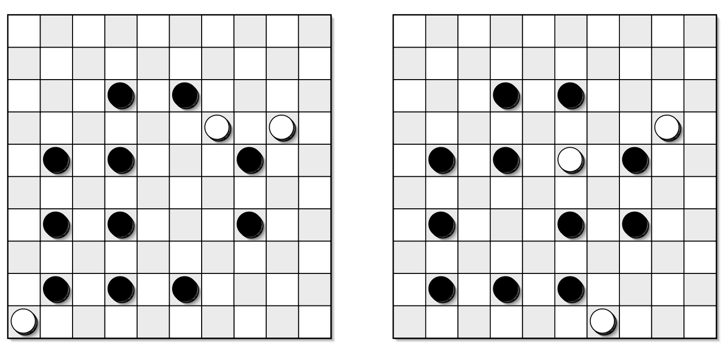
*1. Draughts/checkers is a game played on an $m \times m$ grid of squares, alternately colored light and dark. (The game is usually played on an $8 \times 8$ or $10 \times 10$ board, but the rules easily generalize to any board size.) Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ("White") moves the white pieces; the other ("Black") moves the black pieces.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[1] Pieces can be moved in any of the four diagonal directions, either one or two steps at a time. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her checkers. In a single jump, a piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. Multiple jumps are allowed in a single turn as long as they are made by the same piece. A player wins if her opponent has no pieces left on the board.

Describe an algorithm[2] that correctly determines whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board ($m$), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where $n$ is the total number of pieces, but any algorithm that runs in time polynomial in $n$ and $m$ is worth significant partial credit.




White wins in one turn.



White cannot win in one turn from either of these positions.

*[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does **not** always find a winning sequence of jumps even when one exists.]*

[1]Most variants of draughts have 'flying kings', which behave *very* differently than what's described here.
[2]Since you've read the Course Policies, you know what this phrase means.

2. (a) Prove that any positive integer can be written as the sum of distinct powers of 2. *[Hint: "Write the number in binary" is **not** a proof; it just restates the problem.]* For example:

$$\begin{aligned} 16+1 = \quad & 17 & = 2^4 + 2^0 \\ 16+4+2+1 = \quad & 23 & = 2^4 + 2^2 + 2^1 + 2^0 \\ 32+8+1 = \quad & 42 & = 2^5 + 2^3 + 2^1 \end{aligned}$$

   (b) Prove that *any* integer (positive, negative, or zero) can be written as the sum of distinct powers of $-2$. For example:

$$\begin{aligned} -32+16-2+1 = \quad & -17 & = (-2)^5 + (-2)^4 + (-2)^1 + (-2)^0 \\ 64-32-8-2+1 = \quad & 23 & = (-2)^6 + (-2)^5 + (-2)^3 + (-2)^1 + (-2)^0 \\ 64-32+16-8+4-2 = \quad & 42 & = (-2)^6 + (-2)^5 + (-2)^4 + (-2)^3 + (-2)^2 + (-2)^1 \end{aligned}$$

3. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

   Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left.

4. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

   Rosencrantz: H H T
   Guildenstern: H T H H

   Rosencrantz: T
   Guildenstern: (no flips)

   Rosencrantz: H H H T
   Guildenstern: T H H T H H T H T T H H H

   (a) What is the expected number of flips in one of Rosencrantz's turns?

   (b) Suppose Rosencrantz flips $k$ heads in a row on his turn. What is the expected number of flips in Guildenstern's next turn?

   (c) What is the expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

***Prove*** that your answers are correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! You must give *exact* answers for full credit, but a correct asymptotic bound for part (b) is worth significant credit.

5. (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.

$$A(n) = 3A(n/9) + \sqrt{n}$$

$$B(n) = 4B(n-1) - 4B(n-2)$$

$$C(n) = \frac{\pi\, C(n-1)}{\sqrt{2}\, C(n-2)}$$ *[Hint: This is easy!]*

$$D(n) = \max_{n/4<k<3n/4} \big(D(k) + D(n-k) + n\big)$$

$$E(n) = 2E(n/2) + 4E(n/3) + 2E(n/6) + n^2$$

**Do *not* turn in proofs**—just a list of five functions—but you should do them anyway, just for practice. *[Hint: On the course web page, you can find a handout describing several techniques for solving recurrences.]*

(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.

To simplify your answer, write $f(n) \ll g(n)$ to indicate that $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

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

Recall that $\lg n = \log_2 n$.

# CS 473G: Graduate Algorithms, Spring 2007
# Homework 1

Due February 6, 2007

---

Remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire to students which lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of "strongly favor", "mostly neutral", or "strongly oppose". Each student may respond with "strongly favor" or "strongly oppose" to at most five questions. Because Jeff's students are very understanding, each student is happy if he or she prevails in just one of his or her strong policy preferences. Either describe a polynomial time algorithm for setting course policy to maximize the number of happy students or show that the problem is NP-hard.

2. Consider a variant 3SAT′ of 3SAT which asks, given a formula $\phi$ in conjunctive normal form in which each clause contains at most 3 literals and each variable appears in at most 3 clauses, is $\phi$ satisfiable? Prove that 3SAT′ is NP-complete.

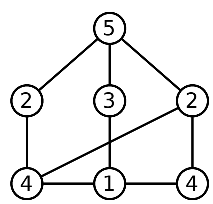
3. For each problem below, either describe a polynomial-time algorithm to solve the problem or prove that the problem is NP-complete.

   (a) A *double-Eulerian* circuit in an undirected graph $G$ is a closed walk that traverses every edge in $G$ exactly twice. Given a graph $G$, does $G$ have a double-Eulerian circuit?

   (b) A *double-Hamiltonian* circuit in an undirected graph $G$ is a closed walk that visits every vertex in $G$ exactly twice. Given a graph $G$, does $G$ have a double-Hamiltonian circuit?

4. Suppose you have access to a magic black box; if you give it a graph $G$ as input, the black box will tell you, in constant time, if there is a proper 3-coloring of $G$. Describe a polynomial time algorithm which, given a graph $G$ that is 3-colorable, uses the black box to compute a 3-coloring of $G$.

5. Let $C_5$ be the graph which is a cycle on five vertices. A $(5, 2)$-coloring of a graph $G$ is a function $f : V(G) \to \{1, 2, 3, 4, 5\}$ such that every pair $\{u, v\}$ of adjacent vertices in $G$ is mapped to a pair $\{f(u), f(v)\}$ of vertices in $C_5$ which are at distance two from each other.




A $(5, 2)$-coloring of a graph.

Using a reduction from 5COLOR, prove that the problem of deciding whether a given graph $G$ has a $(5, 2)$-coloring is NP-complete.

# CS 473G: Graduate Algorithms, Spring 2007
# Homework 2

Due Tuesday, February 20, 2007

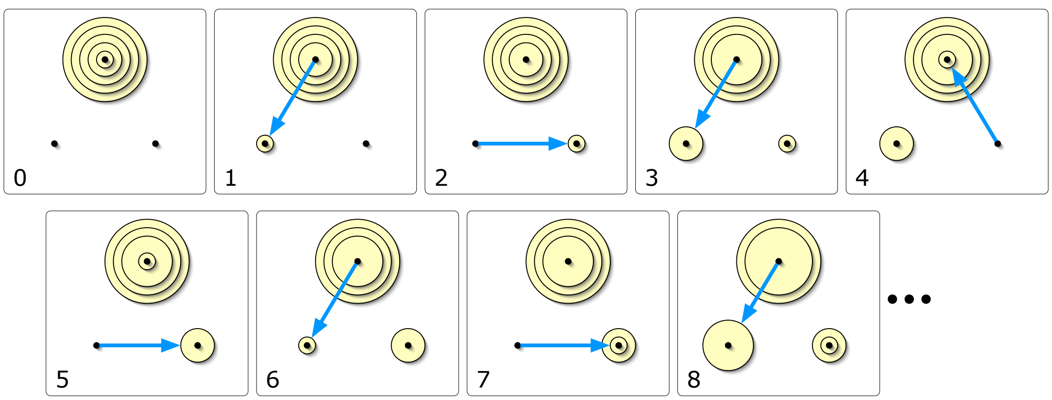
---

Remember to submit **separate, individually stapled** solutions to each problem.

As a general rule, a complete full-credit solution to any homework problem should fit into two typeset pages (or five hand-written pages). If your solution is significantly longer than this, you may be including too much detail.

---

1. Consider a restricted variant of the Tower of Hanoi puzzle, where the three needles are arranged in a triangle, and you are required to move each disk *counterclockwise*. Describe an algorithm to move a stack of $n$ disks from one needle to another. *Exactly* how many moves does your algorithm perform? To receive full credit, your algorithm must perform the minimum possible number of moves. *[Hint: Your answer will depend on whether you are moving the stack clockwise or counterclockwise.]*




A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

*2. You find yourself working for The Negation Company ("We Contradict Everything... Not!"), the world's largest producer of multi-bit Boolean inverters. Thanks to a recent mining discovery, the market prices for amphigen and opoterium, the key elements used in AND and OR gates, have plummeted to almost nothing. Unfortunately, the market price of inverton, the essential element required to build NOT gates, has recently risen sharply as natural supplies are almost exhausted. Your boss is counting on you to radically redesign the company's only product in response to these radically new market prices.

   Design a Boolean circuit that inverts $n = 2^k - 1$ bits, using only $k$ NOT gates but *any* number of AND and OR gates. The input to your circuit consists of $n$ bits $x_1, x_2, \ldots, x_n$, and the output consists of $n$ bits $y_1, y_2, \ldots, y_n$, where each output bit $y_i$ is the inverse of the corresponding input bit $x_i$. *[Hint: Solve the case $k = 2$ first.]*

3. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays. A *common supersequence* of $X$ and $Y$ is another sequence that contains both $X$ and $Y$ as subsequences. Give a simple recursive definition for the function $scs(X, Y)$, which gives the length of the *shortest* common supersequence of $X$ and $Y$.

   (b) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even $i$, and $X[i] > X[i+1]$ for all odd $i$. Give a simple recursive definition for 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 integers *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all $i$. Give a simple recursive definition for the function $lxs(X)$, which gives the length of the longest accelerating subsequence of an arbitrary array $X$ of integers.

   Each recursive definition should translate directly into a recursive algorithm, *but you do **not** need to analyze these algorithms.* We are looking for correctness and *simplicity*, not algorithmic efficiency. Not yet, anyway.

4. Describe an algorithm to solve 3SAT in time $O(\phi^n \operatorname{poly}(n))$, 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?]*

5. (a) Describe an algorithm that determines whether a given set of $n$ integers contains two distinct elements that sum to zero, in $O(n \log n)$ time.

   (b) Describe an algorithm that determines whether a given set of $n$ integers contains *three* distinct elements that sum to zero, in $O(n^2)$ time.

   (c) Now suppose the input set $X$ contains $n$ integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether $X$ contains three ***distinct*** elements that sum to zero, in $O(n \log n)$ time.

   For example, if the input set is $\{-10, -9, -7, -3, 1, 3, 5, 11\}$, your algorithm for part (a) should return TRUE, because $(-3) + 3 = 0$, and your algorithms for parts (b) and (c) should return FALSE, even though $(-10) + 5 + 5 = 0$.

# CS 473G: Graduate Algorithms, Spring 2007
# Homework 3

Due Friday, March 9, 2007

---

Remember to submit **separate, individually stapled** solutions to each problem.

As a general rule, a complete, full-credit solution to any homework problem should fit into two typeset pages (or five hand-written pages). If your solution is significantly longer than this, you may be including too much detail.

---

1. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays. 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 $X[1..n]$ *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 integers *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.

   *[Hint: Use the recurrences you found in Homework 2. You do not need to prove* ***again*** *that these recurrences are correct.]*

2. 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 $\infty$ if $G$ has no Hamiltonian cycles).

3. Let $G$ be an arbitrary undirected graph. A set of cycles $\{c_1, \ldots, c_k\}$ in $G$ is *redundant* if it is non-empty and every edge in $G$ appears in an even number of $c_i$'s. A set of cycles is *independent* if it contains no redundant subsets. (In particular, the empty set is independent.) A maximal independent set of cycles is called a *cycle basis* for $G$.

   (a) Let $C$ be any cycle basis for $G$. Prove that for any cycle $\gamma$ in $G$ **that is not an element of $C$**, there is a subset $A \subseteq C$ such that $A \cup \{\gamma\}$ is redundant. In other words, prove that $\gamma$ is the 'exclusive or' of some subset of basis cycles.

   **Solution:** The claim follows directly from the definitions. A cycle basis is a *maximal* independent set, so if $C$ is a cycle basis, then for any cycle $\gamma \notin C$, the larger set $C \cup \{\gamma\}$ cannot be an independent set, so it must contain a redundant subset. On the other hand, if $C$ is a basis, then $C$ is independent, so $C$ contains no redundant subsets. Thus, $C \cup \{\gamma\}$ must have a redundant subset $B$ that contains $\gamma$. Let $A = B \setminus \{\gamma\}$. ■

(b) Prove that the set of independent cycle sets form a matroid.

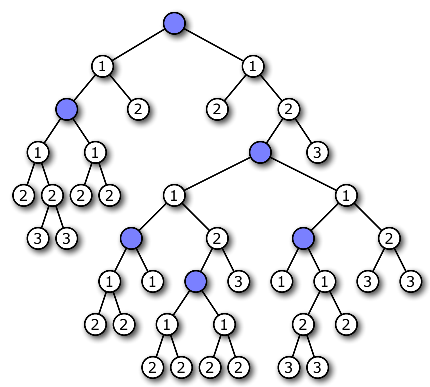
(c) Now suppose each edge of $G$ has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a *set* of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis of $G$.

4. Let $T$ be a rooted binary tree with $n$ vertices, and let $k \le n$ be a positive integer. We would like to mark $k$ vertices in $T$ so that every vertex has a nearby marked ancestor. More formally, we define the *clustering cost* of a clustering of any subset $K$ of vertices as

$$\mathit{cost}(K) = \max_v \mathit{cost}(v, K),$$

where the maximum is taken over all vertices $v$ in the tree, and

$$\mathit{cost}(v, K) = \begin{cases} 0 & \text{if } v \in K \\ \infty & \text{if } v \text{ is the root of } T \text{ and } v \notin K \\ 1 + \mathit{cost}(\mathit{parent}(v)) & \text{otherwise} \end{cases}$$

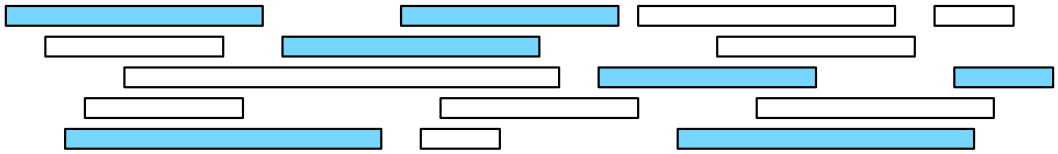


A subset of 5 vertices with clustering cost 3

Describe and analyze a dynamic-programming algorithm to compute the minimum clustering cost of any subset of $k$ vertices in $T$. For full credit, your algorithm should run in $O(n^2k^2)$ time.

5. 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 cover 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.

# CS 473G: Graduate Algorithms, Spring 2007
# Homework 4

Due March 29, 2007

---

Please remember to submit **separate, individually stapled** solutions to each problem.

---

1. Given a graph $G$ with edge weights and an integer $k$, suppose we wish to partition the 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.*, have endpoints in different subsets) is as large as possible.

   (a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

   (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 the new problem? Justify your answer.

2. In class, we saw a $(3/2)$-approximation algorithm for the metric traveling salesman problem. Here, we consider computing minimum cost Hamiltonian *paths*. Our input consists of a graph $G$ whose edges have weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.

   (a) If our input includes zero endpoints, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path.

   (b) If our input includes one endpoint $u$, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at $u$.

   (c) If our input includes two endpoints $u$ and $v$, describe a $(5/3)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at $u$ and ends at $v$.

3. Consider the greedy algorithm for metric TSP: start at an arbitrary vertex $u$, and at each step, travel to the closest unvisited vertex.

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

   $^\star$(b) **[Extra Credit]** Show that the greedy algorithm for metric TSP is no better than an $O(\log n)$-approximation.

4. In class, we saw that the greedy algorithm gives an $O(\log n)$-approximation for vertex cover. Show that our analysis of the greedy algorithm is asymptotically tight by describing, for any positive integer $n$, an $n$-vertex graph for which the greedy algorithm produces a vertex cover of size $\Omega(\log n) \cdot \mathrm{OPT}$.

5. Recall the minimum makespan scheduling problem: Given an array $T[1..n]$ of processing times for $n$ jobs, we wish to schedule the jobs on $m$ machines to minimize the time at which the last job terminates. In class, we proved that the greedy scheduling algorithm has an approximation ratio of at most $2$.

   (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2-1/m)$ times the makespan of the optimal assignment.

   (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2-1/m)$ times the makespan of the optimal assignment.

   (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

# CS 473G: Graduate Algorithms, Spring 2007
# Homework 5

Due Thursday, April 17, 2007

---

Please remember to submit **separate, individually stapled** solutions to each problem.

---

Unless a problem specifically states otherwise, you can assume the function RANDOM($k$), which returns an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$, in $O(1)$ time. For example, to perform a fair coin flip, you would call RANDOM($2$).

---

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

   (a) What is the expected running time of the following RANDOMPERMUTATION algorithm?

   ```
   RANDOMPERMUTATION(n):
     for i ← 1 to n
         π[i] ← EMPTY
     for i ← 1 to n
         j ← RANDOM(n)
         while (π[j] ≠ EMPTY)
             j ← RANDOM(n)
         π[j] ← i
     return π
   ```

   (b) Consider the following partial implementation of RANDOMPERMUTATION.

   ```
   RANDOMPERMUTATION(n):
     for i ← 1 to n
         A[i] ← RANDOM(n)
     π ← SOMEFUNCTION(A)
     return π
   ```

   Prove that if the subroutine SOMEFUNCTION is deterministic, then this algorithm cannot be correct. *[Hint: There is a one-line proof.]*

   (c) Describe and analyze an RANDOMPERMUTATION algorithm whose expected worst-case running time is $O(n)$.

   $^\star$(d) **[Extra Credit]** Describe and analyze an RANDOMPERMUTATION algorithm that uses only fair coin flips; that is, your algorithm can't call RANDOM($k$) with $k > 2$. Your algorithm should run in $O(n \log n)$ time with high probability.

2. 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 element $y$. (If $y > x$, the operation fails.) The input is a pointer directly to the node in $Q$ that contains $x$.
   - DELETE($Q, x$): Delete the element $x \in Q$. The input is a pointer directly to the node in $Q$ that contains $x$.
   - 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: 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.

   (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.)

3. Prove that GUESSMINCUT returns the *second* smallest cut in its input graph with probability $\Omega(1/n^2)$. (The second smallest cut could be significantly larger than the minimum cut.)

4. A *heater* is a sort of dual treap, in which the priorities of the nodes are given by the user, but their search keys are random (specifically, independently and uniformly distributed in the unit interval $[0, 1]$).

   (a) Prove that for any $r$, the node with the $r$th smallest *priority* has expected depth $O(\log r)$.

   (b) Prove that an $n$-node heater has depth $O(\log n)$ with high probability.

   (c) Describe algorithms to perform the operations INSERT and DELETEMIN in a heater. What are the expected worst-case running times of your algorithms?

   You may assume all priorities and keys are distinct. *[Hint: Cite the relevant parts (but only the relevant parts!) of the treap analysis instead of repeating them.]*

5. Let $n$ be an arbitrary positive integer. Describe a set $\mathcal{T}$ of binary search trees with the following properties:

   - Every tree in $\mathcal{T}$ has $n$ nodes, which store the search keys $1, 2, 3, \ldots, n$.
   - For any integer $k$, if we choose a tree uniformly at random from $\mathcal{T}$, the expected depth of node $k$ in that tree is $O(\log n)$.
   - Every tree in $\mathcal{T}$ has depth $\Omega(\sqrt{n})$.

   (This is why we had to prove via Chernoff bounds that the maximum depth of an $n$-node treap is $O(\log n)$ with high probability.)

★6. **[Extra Credit]** Recall that $F_k$ denotes the $k$th Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for all $k \geq 2$. Suppose we are building a hash table of size $m = F_k$ using the hash function

$$h(x) = (F_{k-1} \cdot x) \bmod F_k$$

Prove that if the consecutive integers $0, 1, 2, \ldots, F_k - 1$ are inserted in order into an initially empty table, each integer is hashed into one of the largest contiguous empty intervals in the table. Among other things, this implies that there are no collisions.

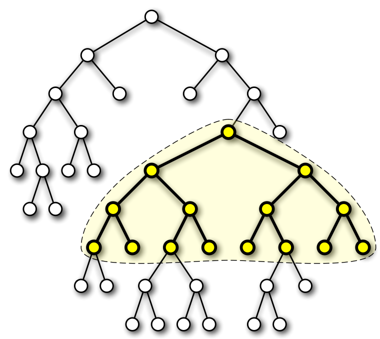
For example, when $m = 13$, the hash table is filled as follows.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | | | | | |
| 0 | | | | | | | | 1 | | | | |
| 0 | | | 2 | | | | | 1 | | | | |
| 0 | | | 2 | | | | | 1 | | | 3 | |
| 0 | | | 2 | | | 4 | | 1 | | | 3 | |
| 0 | 5 | | 2 | | | 4 | | 1 | | | 3 | |
| 0 | 5 | | 2 | | | 4 | | 1 | 6 | | 3 | |
| 0 | 5 | | 2 | 7 | | 4 | | 1 | 6 | | 3 | |
| 0 | 5 | | 2 | 7 | | 4 | | 1 | 6 | | 3 | 8 |
| 0 | 5 | | 2 | 7 | | 4 | 9 | 1 | 6 | | 3 | 8 |
| 0 | 5 | 10 | 2 | 7 | | 4 | 9 | 1 | 6 | | 3 | 8 |
| 0 | 5 | 10 | 2 | 7 | | 4 | 9 | 1 | 6 | 11 | 3 | 8 |
| 0 | 5 | 10 | 2 | 7 | 12 | 4 | 9 | 1 | 6 | 11 | 3 | 8 |

You have 90 minutes to answer four of these questions.
**Write your answers in the separate answer booklet.**
You may take the question sheet with you when you leave.

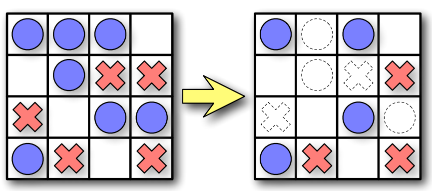
1. Recall that a binary tree is *complete* if every internal node has two children and every leaf has the same depth. An *internal subtree* of a binary tree is a connected subgraph, consisting of a node and some (possibly all or none) of its descendants.

   Describe and analyze an algorithm that computes the depth of the *largest complete internal subtree* of a given $n$-node binary tree. For full credit, your algorithm should run in $O(n)$ time.

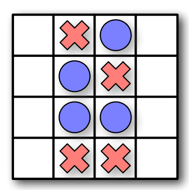


The largest complete internal subtree in this binary tree has depth 3.

2. 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.

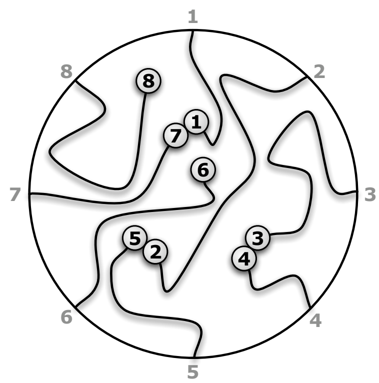
   ***Prove*** that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

3. Suppose you are given two sorted arrays $A[1\,..\,n]$ and $B[1\,..\,n]$ and an integer $k$. Describe and analyze an algorithm to find the $k$th largest element in the union of $A$ and $B$ in $O(\log n)$ time. For example, given the input

$$A[1\,..\,8] = [0, 1, 6, 9, 12, 13, 18, 21], \qquad B[1\,..\,8] = [2, 4, 5, 8, 14, 17, 19, 20], \qquad k = 10,$$

   your algorithm should return 13. You can assume that the arrays contain no duplicates. *[Hint: What can you learn from comparing one element of $A$ to one element of $B$?]*

4. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from $1$ to $n$. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.




The end of a typical Antarctican SLUG race. Snails $6$ and $8$ never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1\,..\,n, 1\,..\,n]$ posted on the wall behind the Round Table, where $M[i,j] = M[j,i]$ is the reward to be paid if snails $i$ and $j$ meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array $M$ as input.

5. SUBSETSUM and PARTITION are two closely-related NP-hard problems.

- SUBSETSUM: Given a set $X$ of positive integers and an integer $t$, determine whether there is a subset of $X$ whose elements sum to $t$.
- PARTITION: Given a set $X$ of positive integers, determine whether $X$ can be partitioned into two subsets whose elements sum to the same value.

(a) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.

(b) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

Don't forget to ***prove*** that your reductions are correct.

You have 120 minutes to answer four of these questions.
**Write your answers in the separate answer booklet.**
You may take the question sheet with you when you leave.

1. Consider the following algorithm for finding the smallest element in an unsorted array:

```
RandomMin(A[1..n]):
    min ← ∞
    for i ← 1 to n in random order
        if A[i] < min
            min ← A[i]    (⋆)
    return min
```

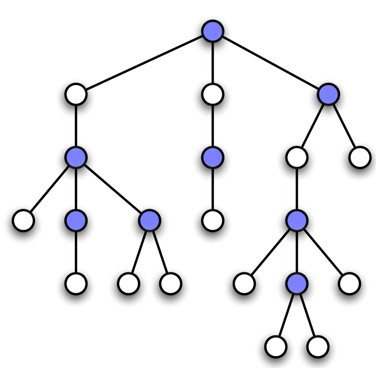
   (a) **[1 pt]** In the worst case, how many times does RandomMin execute line $(\star)$?

   (b) **[3 pts]** What is the probability that line $(\star)$ is executed during the last iteration of the for loop?

   (c) **[6 pts]** What is the *exact* expected number of executions of line $(\star)$? (A correct $\Theta(\,)$ bound is worth $4$ points.)

2. Describe and analyze an efficient algorithm to find the size of the smallest vertex cover of a given tree. That is, given a tree $T$, your algorithm should find the size of the smallest subset $C$ of the vertices, such that every edge in $T$ has at least one endpoint in $C$.

   The following hint may be helpful. Suppose $C$ is a vertex cover that contains a leaf $\ell$. If we remove $\ell$ from the cover and insert its parent, we get another vertex cover of the same size as $C$. Thus, there is a minimum vertex cover that includes none of the leaves of $T$ (except when the tree has only one or two vertices).



A tree whose smallest vertex cover has size $8$.

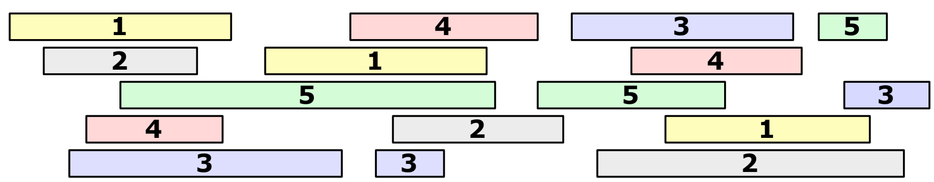
3. A *dominating set* for a graph $G$ is a subset $D$ of the vertices, such that every vertex in $G$ is either in $D$ or has a neighbor in $D$. The MINDOMINATINGSET problem asks for the size of the smallest dominating set for a given graph.

   Recall the MINSETCOVER problem from lecture. The input consists of a *ground set* $X$ and a collection of subsets $S_1, S_2, \ldots, S_k \subseteq X$. The problem is to find the minimum number of subsets $S_i$ that completely cover $X$. This problem is NP-hard, because it is a generalization of the vertex cover problem.

   (a) **[7 pts]** Describe a polynomial-time reduction from MINDOMINATINGSET to MINSETCOVER.

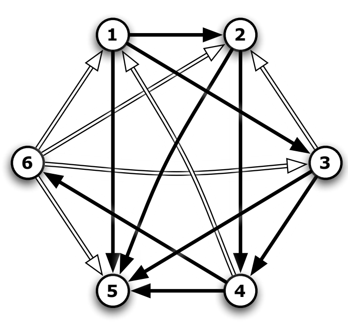
   (b) **[3 pts]** Describe a polynomial-time $O(\log n)$-approximation algorithm for MINDOMINATINGSET. *[Hint: There is a two-line solution.]*

4. Let $X$ be a set of $n$ intervals on the real line. A *proper coloring* of $X$ assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color $X$. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, where $L[i]$ and $R[i]$ are the left and right endpoints of the $i$th interval. As usual, if you use a greedy algorithm, you must prove that it is correct.




A proper coloring of a set of intervals using five colors.

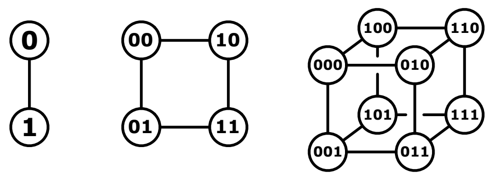
5. The *linear arrangement problem* asks, given an $n$-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of *forward edges*: directed edges $v_i \to v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for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




A directed graph with six vertices with nine forward edges (black) and six backward edges (white)

You have 180 minutes to answer six of these questions.
**Write your answers in the separate answer booklet.**

1. 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.




The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

   (a) *[8 pts]* Recall that a Hamiltonian cycle is a closed walk that visits each vertex in a graph exactly once. ***Prove*** that for all $d \geq 2$, the $d$-dimensional hypercube has a Hamiltonian cycle.

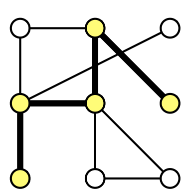
   (b) *[2 pts]* Recall that an Eulerian circuit is a closed walk that traverses each edge in a graph exactly once. Which hypercubes have an Eulerian circuit? *[Hint: This is very easy.]*

2. The University of Southern North Dakota at Hoople has hired you to write an algorithm to schedule their final exams. Each semester, USNDH offers $n$ different classes. There are $r$ different rooms on campus and $t$ different time slots in which exams can be offered. You are given two arrays $E[1..n]$ and $S[1..r]$, where $E[i]$ is the number of students enrolled in the $i$th class, and $S[j]$ is the number of seats in the $j$th room. At most one final exam can be held in each room during each time slot. Class $i$ can hold its final exam in room $j$ only if $E[i] < S[j]$.

   Describe and analyze an efficient algorithm to assign a room and a time slot to each class (or report correctly that no such assignment is possible).

3. What is the *exact* expected number of leaves in an $n$-node treap? (The answer is obviously at most $n$, so no partial credit for writing "$O(n)$".) *[Hint: What is the probably that the node with the $k$th largest key is a leaf?]*

4. A *tonian path* in a graph $G$ is a simple path in $G$ that visits more than half of the vertices of $G$. (Intuitively, a tonian path is "most of a Hamiltonian path".) ***Prove*** that it is NP-hard to determine whether or not a given graph contains a tonian path.



A tonian path.

5. 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,

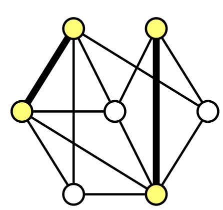
   bub + baseesab + anana
   b + u + bb + a + sees + aba + nan + a
   b + u + bb + a + sees + a + b + anana
   b + u + b + b + a + s + e + e + s + a + b + a + n + a + n + a

   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.

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

```
APPROXMINMAXMATCHING(G):
  M ← ∅
  while G has at least one edge
      (u, v) ← any edge in G
      G ← G \ {u, v}
      M ← M ∪ {(u, v)}
  return M
```

   (a) *[2 pts]* ***Prove*** that the output graph $M$ is a *matching*—no pair of edges in $M$ share a common vertex.
   (b) *[2 pts]* ***Prove*** that $M$ is a *maximal* matching—$M$ is not a proper subgraph of another matching in $G$.
   (c) *[6 pts]* ***Prove*** that $M$ contains at most twice as many edges as the *smallest* maximal matching in $G$.



The smallest maximal matching in a graph.

7. Recall that in the standard maximum-flow problem, the flow through an edge is limited by the capacity of that edge, but there is no limit on how much flow can pass through a vertex. Suppose each vertex $v$ in our input graph has a capacity $c(v)$ that limits the total flow through $v$, in addition to the usual edge capacities. Describe and analyze an efficient algorithm to compute the maximum $(s, t)$-flow with these additional constraints. *[Hint: Reduce to the standard max-flow problem.]*