CS 473 ✧ Spring 2020

☙ Homework 0 ❧

Due Wednesday, January 29, 2020 at 9pm

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

☞ Some important course policies ☜

- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details.
- **Avoid the Deadly Sins!** There are a few common writing (and thinking) practices that will be automatically penalized on every homework or exam problem. We're not just trying to be scary control freaks; history strongly suggests people who commit these sins are more likely to make other serious mistakes as well. So we're trying to break bad habits that seriously impede mastery of the course material.
  - Always give complete solutions, not just examples.
  - Every algorithm requires an English specification.
  - Greedy algorithms require formal correctness proofs.
  - Never use weak induction. Weak induction should die in a fire.

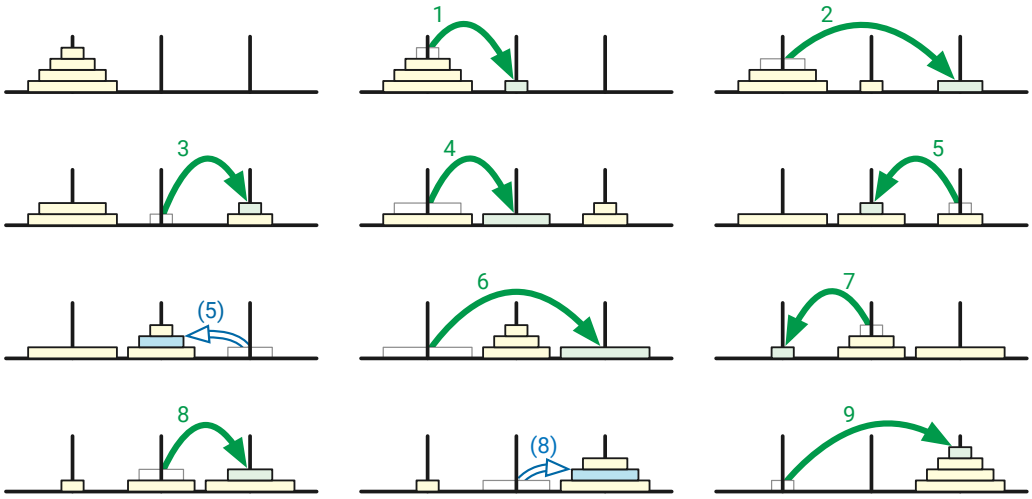
See the course web site for more information.

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

---

1. The standard Tower of Hanoi puzzle consists of n circular disks of different sizes, each with a hole in the center, and three pegs. The disks are labeled $1, 2, \ldots, n$ in increasing order of size. Initially all n disks are on one peg, sorted by size, with disk n at the bottom and disk 1 at the top. The goal is to move all n disks to a different peg, by repeatedly moving individual disks. At each step, we are allowed to move the highest disk on any peg to any other peg, subject to the constraint that a larger disk is never placed above a smaller disk. A recursive strategy that solves this problem using exactly $2^n - 1$ moves is well known (and described in the textbook).

   This question concerns a variant of the Tower of Hanoi that I'll call the *Tower of Fibonacci*. In this variant, whenever any two disks $i-1$ and $i+1$ are adjacent, the intermediate disk i immediately teleports between them; otherwise, the setup, rules, and goal of the puzzle are unchanged. This teleport does *not* count as a move; on the other hand, the teleport is *not* optional. For example, the four-disk Tower of Fibonacci can be solved in nine moves (and two teleports, after moves 5 and 8) as follows:



   (a) Describe a recursive algorithm to solve the Tower of Fibonacci puzzle. Briefly justify why your algorithm is correct. (We don't need a complete formal proof of correctness; just convince us that you know why it works.)

   (b) How many *moves* does your algorithm perform, as a function of the number of disks? Prove that your answer is correct.

   (c) How many *teleports* does your algorithm induce, as a function of the number of disks? Prove that your answer is correct.

   For full credit, give *exact* answers to parts (b) and (c), not just $O(\,)$ bounds. Express your answers to parts (b) and (c) in terms of the Fibonacci numbers, defined as follows:

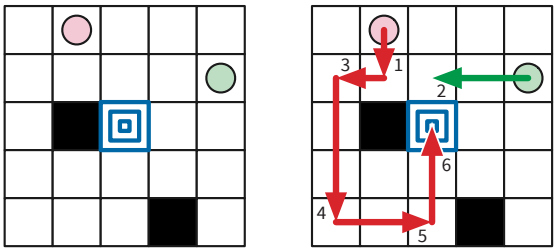
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

2. Prove that every integer (positive, negative, or zero) can be written as the sum of distinct powers of −2. For example:

$$\begin{aligned}
17 &= (-2)^4 + (-2)^0 && = 16 + 1 \\
-23 &= (-2)^5 + (-2)^4 + (-2)^3 + (-2)^0 && = -32 + 16 - 8 + 1 \\
42 &= (-2)^6 + (-2)^5 + (-2)^4 + (-2)^3 + (-2)^2 + (-2)^1 && = 64 - 32 + 16 - 8 + 4 - 2 \\
473 &= (-2)^{10} + (-2)^9 + (-2)^5 + (-2)^3 + (-2)^0 && = 1024 - 512 - 32 - 8 + 1
\end{aligned}$$

[Hint: The empty set is a set, and weak induction should die in a fire.]

3. The famous puzzle-maker Kaniel the Dane invented a solitaire game played with two tokens on an $n \times n$ square grid. Some squares of the grid are marked as *obstacles*, and one grid square is marked as the *target*. In each turn, the player must move one of the tokens from its current position *as far as possible* upward, downward, right, or left, stopping just before the token hits (1) the edge of the board, (2) an obstacle square, or (3) the other token. The goal is to move either of the tokens onto the target square.



An instance of Kaniel the Dane's puzzle that can be solved in six moves.
Circles indicate initial token positions; black squares are obstacles; the center square is the target.

For example, we can solve the puzzle shown above by moving the red token down until it hits the obstacle, then moving the green token left until it hits the red token, and then moving the red token left, down, right, and up. The red token stops at the target on the 6th move *because* the green token is just above the target square.

Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consists of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise.

[Hint: Construct and search a graph. What are the vertices? What are the edges? Is the graph directed or undirected? Do the vertices or edges have weights? How long does it take to construct the graph? What problem do you need to solve on this graph? What textbook algorithm can you use to solve that problem? (Don't regurgitate the textbook algorithm; just point to the textbook!) What is the running time of that algorithm as a function of n?]

CS 473 ✧ Spring 2020

Homework 1

Due Wednesday, February 5, 2020 at 9pm

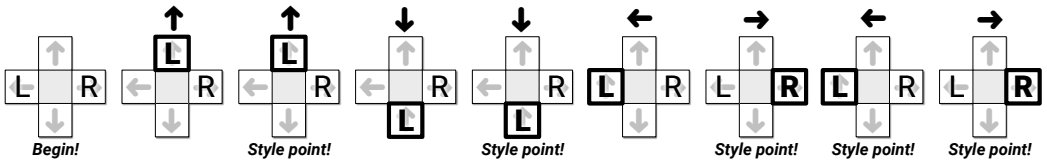
- Starting with this homework, groups of up to three students can submit joint solutions for each problem. For each numbered problem, exactly one member of each homework group should submit a solution and identify the other group members (if any).
- If you use a greedy algorithm, you ***must*** prove that it is correct, or you will get zero points ***even if your algorithm is correct***.

---

1. ***Dance Dance Revolution*** (ダンスダンスレボリューション) is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (←, ↑, ↓, or →) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you'll step with the beat of the song.)

   You are playing a variant of this game called "Vogue Vogue Revolution" where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) foot from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, all your style points are taken away and you lose the game.

   How should you move your feet to maximize your total number of style points? Assume you start the game with your left foot on ← and your right foot on →, and that you've memorized the entire sequence of arrows. For example, if the sequence is ↑↑↓↓←→←→, you can earn 5 style points by moving your feet as shown below:



   (a) Prove that for *any* sequence of n arrows, it is possible to earn at least $n/4 - 1$ style points.

   (b) Describe and analyze an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. The input to your algorithm is an array $Arrow[1..n]$ containing the sequence of arrows.

2. You've been hired to store a sequence of n books on shelves in a library, using as little *vertical* space as possible. The order of the books is fixed by the cataloging system and cannot be changed; each shelf must store a contiguous interval of the given sequence of books. You can adjust the height of each shelf to match the tallest book on that shelf; in particular, you can change the height of any empty shelf to zero.

   You are given two arrays $H[1..n]$ and $W[1..n]$, where $H[i]$ and $W[i]$ are respectively the height and width of the ith book. Each shelf has the same fixed length L. Each book as width at most L, and the total width of all books on each shelf cannot exceed L. Your task is to shelve the books so that the *sum of the heights* of the shelves is as small as possible.

   (a) There is a natural greedy algorithm, which actually yields an optimal solution when all books have the same height: If $n > 0$, pack as many books as possible onto the first shelf, and then recursively shelve the remaining books.

   Show that this greedy algorithm does *not* yield an optimal solution if the books can have different heights. *[Hint: There is a small counterexample.]*

   (b) Describe and analyze an efficient algorithm to assign books to shelves to minimize the total height of the shelves.

3. (a) Any string can be decomposed into a sequence of palindromes. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (and 65 others):

BUB • BASEESAB • ANANA

B • U • BB • ASEESA • B • ANANA

BUB • B • A • SEES • ABA • N • ANA

B • U • BB • A • S • EE • S • A • B • A • NAN • A

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 should return 3.

   (b) A ***metapalindrome*** is a decomposition of a string into a sequence of palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example, the string BOBSMAMASEESAUKULELE ("Bob's mama sees a ukulele") has the following metapalindromes (among others):

BOB • S • MAM • ASEESA • UKU • L • ELE

B • O • B • S • M • A • M • A • S • E • E • S • A • U • K • U • L • E • L • E

   The length sequences of these metapalindromes are $(3, 1, 3, 6, 3, 1, 3)$ and $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$; notice that both of these sequences are themselves palindromes.

   Describe and analyze an efficient algorithm to find the smallest number of palindromes in any metapalindrome for a given string. For example, given the input string BOBSMAMASEESAUKULELE, your algorithm should return 7.

CS 473 ✧ Spring 2020

Homework 2

Due Wednesday, February 12, 2020 at 9pm

---

1. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. A ***digital subsequence*** of D is a sequence of positive integers composed in the usual way from disjoint intervals of D. For example, the sequence $3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279$ is a digital subsequence of the first several digits of π:

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

   The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the preceding example has length 12. As usual, a digital subsequence is ***increasing*** if each number is larger than its predecessor.

   Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: First consider the special case where none of the digits is* 0*. Be careful about your computational assumptions. How long does it take to compare two k-digit integers?]*

2. The (Eleventh) Doctor and River Song decide to play a game on a directed acyclic graph G, which has one source vertex s and one sink vertex t.[1]

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

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

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

[1] The labels s and t are abbreviations for the Untempered **S**chism and the **T**ime Vortex, or the Shining World of the Seven Systems (also known as Gallifrey) and Trenzalore, or Skaro and Telos, or Something else Timey-wimey. It's all very complicated, never mind.

3. ***[Extra credit[2]]*** After seeing your expertise in solving problem 2 in Homework 1, the Order of the Library of the Neitherlands immediately recruited you to tackle a significantly larger shelving project. Their problem is exactly the same as the one you already solved—they need an algorithm to put a sequence of n books on shelves using as little *vertical* space as possible—but the Neitherlands Library contains *far* too many books for an $O(n^2)$-time algorithm to be useful.

   Describe and analyze an algorithm to compute the minimum total height required to shelve a sequence of n books ***in $O(n \log n)$ time***. As in Homework 1 problem 2, the input consists of two arrays $H[1..n]$ and $W[1..n]$, specifying the height and width of each book, and a number L, which is the common length of every shelf. Heights, widths, and lengths are not necessarily integers.

   *[I have no reason to believe that $O(n \log n)$ is the best possible running time. For the special case where heights, widths, and lengths **are** integers, $O(n \log n)$ is definitely **not** the best possible running time.]*

[2]Extra credit problems are ignored when we compute the curve at the end of the semester. In particular, they do not count toward the top 24 homework scores we will use to compute your raw homework average.

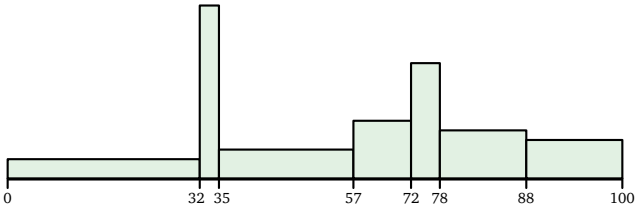
CS 473 ✧ Spring 2020

Homework 3

Due Wednesday, February 19, 2020 at 9pm

---

1. ***[Extra credit]*** Suppose we want to visualize a large set S of values—for example, grade-point averages for every student who ever attended UIUC—using a variable-width histogram. To construct a histogram, we choose a sorted sequence of ***breakpoints*** $b_0 < b_1 < \cdots < b_k$, such that every element of S lies between b_0 and b_k. Each interval $[b_{i-1}, b_i)$ between two consecutive buckets is called a *bin*. Any histogram includes a rectangle for each bin, whose height is the number of elements of S that lie inside that bin.



A variable-width histogram with seven bins.

Unlike a standard histogram, which requires the intervals to have equal width, we are free to choose the breakpoints arbitrarily. For visualization purposes, it is useful for the *areas* of the rectangles to be as close to equal as possible, so we want the sum of the squares of the areas to be as small as possible.[1] To simplify computation, we require that every breakpoint is an element of the dataset S.

More precisely, suppose we are given a sorted array $S[1..n]$ of distinct real numbers and an integer k. For any indices $i < j$, let

$$area(i, j) = (j - i) \cdot (S[j] - S[i])$$

denote the area of a single histogram rectangle representing the $j - i$ items in the interval $S[i..j-1]$. A histogram for S is determined by a sorted array $B[0..k]$ of distinct *breakpoint* indices, such that $B[0] = 1$ and $B[k] = n$. We need to choose these breakpoints to minimize the sum of the squared areas of its rectangles:

$$Cost(B) = \sum_{i=1}^{k} \big(area(B[i-1], B[i])\big)^2.$$

(a) Define an upper-triangular array $A[1..n, 1..n]$ by setting $A[i, j] = (area(i, j))^2$ if $i < j$ and leaving $A[i, j]$ undefined otherwise. Prove that this array has the Monge property.

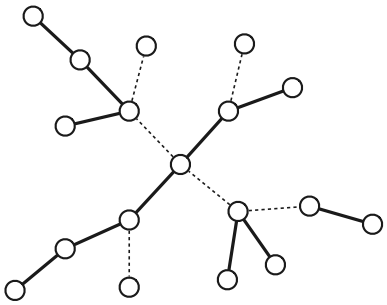
A partial array has the Monge property if, for all indices $i < i'$ and $j < j'$ *such that* $A[i, j]$ *and* $A[i', j]$ *and* $A[i, j']$ *and* $A[i', j']$ *are defined*, we have $A[i, j] + A[i', j'] \leq A[i, j'] + A[i', j]$. If A is upper-triangular, it suffices to check the Monge condition where $i' = i + 1$ and $j' = j + 1$.

[1] As far as I know, this objective has no statistical justification, but it makes the pictures look nice.

(b) Describe an algorithm to find the minimum element in every row of an $n \times n$ *upper triangular* Monge array in $O(n \log n)$ time. (The original SMAWK algorithm requires a full rectangular array.) *[Hint: Use SMAWK as a subroutine.]*

(c) Describe and analyze an algorithm to compute a variable-width histogram with minimum cost for a given set S of data values and a given number k of bins. For full credit, your algorithm should take advantage of parts (a) and (b). *[Hint: You can assume parts (a) and (b) even without a proof.]*

2. Let T be an arbitrary tree—a connected undirected graph with no cycles. Describe and analyze an algorithm to cover the vertices of T with as few disjoint paths as possible. Each vertex of T must lie on exactly one of the paths. (The figure below shows a tree covered by seven disjoint paths, three of which have length zero.)



3. Consider the following non-standard algorithm for shuffling a deck of n cards, initially numbered in order from 1 on the top to n on the bottom. At each step, we remove the top card from the deck and ***insert*** it randomly back into in the deck, choosing one of the n possible positions uniformly at random. The algorithm ends immediately after we pick up card $n-1$ and insert it randomly into the deck.

(a) Prove that this algorithm uniformly shuffles the deck, meaning each permutation of the deck has equal probability. *[Hint: Prove that at all times, the cards below card $n-1$ are uniformly shuffled.]*

(b) What is the *exact* expected number of steps executed by the algorithm? *[Hint: Split the algorithm into phases that end when card $n-1$ changes position.]*

π. [Warmup only; do not submit solutions]

After sending his loyal friends Rosencrantz and Guildenstern off to Norway, Hamlet decides to amuse himself by repeatedly flipping a fair coin until the sequence of flips satisfies some condition. For each of the following conditions, compute the *exact* expected number of flips until that condition is met.

(a) Hamlet flips heads.

(b) Hamlet flips both heads and tails (in different flips, of course).

(c) Hamlet flips heads twice.

(d) Hamlet flips heads twice in a row.

(e) Hamlet flips heads followed immediately by tails.

(f) Hamlet flips more heads than tails.

(g) Hamlet flips the same number of heads and tails.

(h) Hamlet flips the same positive number of heads and tails.

(i) Hamlet flips more than twice as many heads as tails.

[Hint: Be careful! If you're relying on intuition instead of a proof, you're probably wrong.]

CS 473 ✧ Spring 2020

Homework 4

Due Wednesday, March 4, 2020 at 9pm

---

1. Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n.

   (a) Prove that if we start at vertex 1, the probability that the walk ends by falling off the *right* end of the path is exactly $1/(n+1)$.

   (b) Prove that if we start at vertex k, the probability that the walk ends by falling off the *right* end of the path is exactly $k/(n+1)$.

   (c) Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n.

   (d) What is the *exact* expected length of the random walk if we start at vertex k, as a function of n and k? Prove your result is correct. (For partial credit, give a tight Θ-bound for the case $k = (n+1)/2$, assuming n is odd.)

   [Hint: Trust the recursion fairy. Yes, (b) implies (a) and (d) implies (c).]

2. The following randomized variant of "one-armed quicksort" selects the kth smallest element in an unsorted array $A[1..n]$. As usual, PARTITION$(A[1..n], p)$ partitions the array A into three parts by comparing the pivot element $A[p]$ to every other element, using $n-1$ comparisons, and returns the new index of the pivot element.

```
QUICKSELECT(A[1..n], k):
  r ← PARTITION(A[1..n], RANDOM(n))
  if k < r
      return QUICKSELECT(A[1..r−1], k)
  else if k > r
      return QUICKSELECT(A[r+1..n], k−r)
  else
      return A[k]
```

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

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

   (c) What is the *exact* probability that in some recursive call to QUICKSELECT, the first argument is the subarray $A[i..j]$? The correct answer is a simple function of i, j, and k (with more cases). *[Hint: Check your answer by trying a few small examples.]*

   (d) Show that for any n and k, QUICKSELECT runs in $\Theta(n)$ expected time. You can use either the recurrence from part (a) or the probabilities from part (b) or (c).

3. A ***meldable priority queue*** stores a set of priorities 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.
   - DECREASEPRIORITY(Q, x, y): Replace an element $x \in Q$ with a new element $y < x$. (If $y \geq x$, the operation fails.) The input includes 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. The elements of Q_1 and Q_2 could be arbitrarily intermixed; we do *not* assume, for example, that every element of Q_1 is smaller than every element of Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a priority, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm. The input consists of pointers to the roots of the two trees.

```
MELD(Q1, Q2):
  if Q1 = NULL then return Q2
  if Q2 = NULL then return Q1

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

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

  return Q1
```

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

   (b) Prove that MELD(Q_1, Q_2) runs in $O(\log n)$ time with high probability. *[Hint: You can use Chernoff bounds, but the simpler identity $\binom{c}{k} \leq (ce)^k$ is actually sufficient.]*

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

CS 473 ✧ Spring 2020

Homework 5

Due Wednesday, March 11, 2020 at 9pm

---

1. In this problem we consider yet another method for universal hashing. Suppose we are hashing from the universe $\mathcal{U} = \{0, 1, \ldots, 2^w - 1\}$ of w-bit strings to a hash table of size $m = 2^\ell$; that is, we are hashing w-bit *words* into ℓ-bit *labels*. To define our universal family of hash functions, we think of words and labels as *boolean vectors* of length w and ℓ, respectively, and we specify our hash function by choosing a random *boolean matrix*.

   For any $\ell \times w$ matrix M of 0s and 1s, define the hash function $h_M : \{0,1\}^w \to \{0,1\}^\ell$ by the boolean matrix-vector product

$$h_M(x) = Mx \bmod 2 = \bigoplus_{i=1}^{w} M_i x_i = \bigoplus_{i\,:\,x_i=1} M_i.$$

   where $\oplus$ denotes bitwise exclusive-or (that is, addition mod 2), M_i denotes the ith column of M, and x_i denotes the ith bit of x. Let $\mathcal{M} = \{h_m \mid M \in \{0,1\}^{w\times\ell}\}$ denote the set of all such random-matrix hash functions.

   For example, suppose $w = 8$ and $\ell = 4$. Let M be the $w \times \ell$ matrix

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

   Then we can compute $h_M(173) = 12$ as follows:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

   (a) Prove that $\mathcal{M}$ is a universal family of hash functions.

   (b) Prove that $\mathcal{M}$ is ***not*** uniform.

   (c) Now consider a modification of the previous scheme, where we specify a hash function by a random matrix $M \in \{0,1\}^{\ell\times w}$ and an independent random offset vector $b \in \{0,1\}^\ell$:

$$h_{M,b}(x) \;=\; (Mx + b) \bmod 2 \;=\; \left(\bigoplus_{i=1}^{w} M_i x_i\right) \oplus b$$

   Prove that the family $\mathcal{M}^+$ of all such functions is *strongly* universal (2-uniform).

   (d) Prove that $\mathcal{M}^+$ is ***not*** 4-uniform.

   (e) ***[Extra credit]*** Prove that $\mathcal{M}^+$ is actually 3-uniform.

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

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

At the end of the algorithm, the variable ℓ stores the length of the input stream S; this number is *not* known to the algorithm in advance. If S is empty, the output of the algorithm is (correctly!) undefined.

In the following questions, consider an arbitrary non-empty input stream S, and let n denote the (unknown) length of S.

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

(b) What is the *exact* expected number of times that GetOneSample(S) executes line ($\star$)?

(c) What is the *exact* expected value of ℓ when GetOneSample(S) executes line ($\star$) for the *last* time?

(d) What is the *exact* expected value of ℓ when either GetOneSample(S) executes line ($\star$) for the *second* time (or the algorithm ends, whichever happens first)?

3. (This is a continuation of the previous problem.) Describe and analyze an algorithm that returns a subset of k distinct items chosen uniformly at random from a data stream of length at least k. Prove that your algorithm is correct. Your algorithm should have the following form:

```
GetSample(stream S, k):
  ⟨⟨Do some preprocessing⟩⟩
  while S is not done
      x ← next item in S
      ⟨⟨Do something with x⟩⟩
  return ⟨⟨something⟩⟩
```

Both the time for each ⟨⟨*step*⟩⟩ in your algorithm and the space for any necessary data structures must be bounded by functions of k, *not* the length of the stream.

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

CS 473 ✧ Spring 2020

Homework 6

Due Wednesday, March 25, 2020 at 9pm

(after spring break)

For the rest of the semester, you are welcome to use randomized algorithms in your homework and exam solutions, but please report your running times carefully. Without further qualification, "$O(n^2)$ time" means $O(n^2)$ time *in the worst case*. If you mean $O(n^2)$ *expected* time, or $O(n^2)$ time *with high probability*, you must write that explicitly.

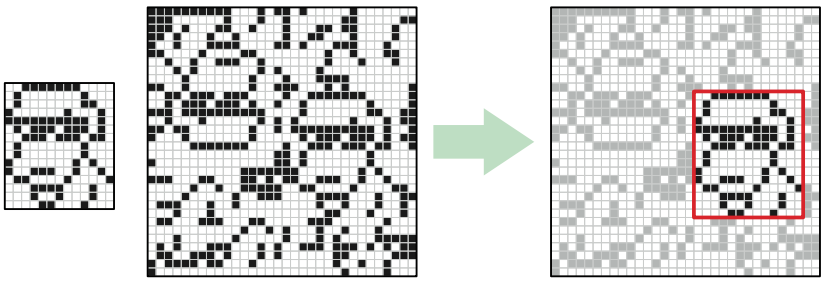
---

1. Describe and analyze an *even faster* algorithm to find the length of the longest substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples:

   - Given the input string `ALGORITHM`, your algorithm should return 0.
   - Given the input string `RECURSION`, your algorithm should return 1, for the substring `R`.
   - Given the input string `REDIVIDE`, your algorithm should return 3, for the substring `EDI`. (Remember: The forward and backward substrings must not overlap!)

   Yes, this *exact* problem appeared in Midterm 1, so you should already know how to solve it in $O(n^2)$ time. You can do better now.

2. Describe an efficient algorithm to determine if a given $p \times q$ rectangular pattern of bits appears anywhere in an $m \times n$ bitmap. (The pattern may be shifted horizontally and/or vertically, but it may not be rotated or reflected.)



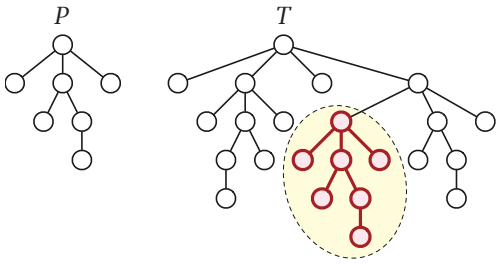
3. Describe an efficient algorithm to decide, given two rooted ordered trees P and T, whether P (the "pattern") occurs anywhere as a subtree of T (the "text").

   A *rooted ordered tree* is a rooted tree where every node has a (possibly empty) *sequence* of children. The order of these children matters: Two rooted ordered trees are identical if and only if their roots have the same number of children and, for each index i, the subtrees rooted at the ith children of both roots are identical.

   For purposes of this problem, a *subtree* of T contains some node and ***all*** its descendants in T, along with the edges of T between those vertices.

There is no data stored in the nodes, only pointers to children (if any). We want an algorithm that compares the *shapes* of the trees.

For example, in the figure below, P appears exactly once as a subtree of T.



CS 473 ✧ Spring 2020

Homework 7

Due Wednesday, April 1, 2020 at 9pm

---

1. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c : E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$.

   (a) Describe and analyze an efficient algorithm to determine whether f is a maximum (s, t)-flow in G.

   (b) Describe and analyze an efficient algorithm to determine whether f is the *unique* maximum (s, t)-flow in G.

   Do not assume ***anything*** about the function f.

2. A new assistant professor, teaching maximum flows for the first time, suggested the following greedy modification to the generic Ford-Fulkerson augmenting path algorithm. Instead of maintaining a residual graph, the greedy algorithm just reduces the capacity of edges along the augmenting path. In particular, whenever the algorithm saturates an edge, that edge is simply removed from the graph.

```
GreedyFlow(G, c, s, t):
  for every edge e in G
      f(e) ← 0

  while there is a path from s to t in G
      π ← arbitrary path from s to t in G
      F ← minimum capacity of any edge in π
      for every edge e in π
          f(e) ← f(e) + F
          if c(e) = F
              remove e from G
          else
              c(e) ← c(e) − F

  return f
```

   (a) Prove that GreedyFlow does not always compute a maximum flow.

   (b) Prove that GreedyFlow is not even guaranteed to compute a good approximation to the maximum flow. That is, for any constant $\alpha > 1$, describe a flow network G such that the value of the maximum flow is more than α times the value of the flow computed by GreedyFlow. *[Hint: Assume that GreedyFlow chooses the worst possible path π at each iteration.]*

3. Suppose you are given a flow network G with ***integer*** edge capacities and an ***integer*** maximum flow f^* in G. Describe algorithms for the following operations:

(a) INCREMENT(e): Increase the capacity of edge e by 1 and update the maximum flow.

(b) DECREMENT(e): Decrease the capacity of edge e by 1 and update the maximum flow.

Both algorithms should modify f^* so that it is still a maximum flow, but more quickly than recomputing a maximum flow from scratch.

CS 473 ✧ Spring 2020

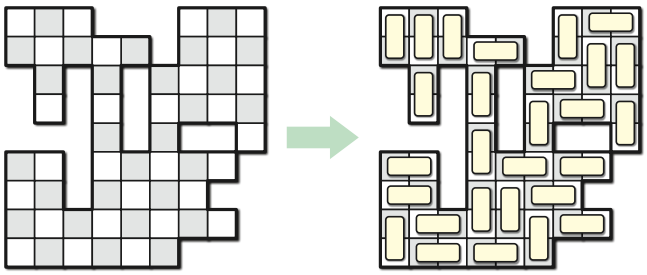
Homework 8

Due Wednesday, April 15, 2020 at 9pm

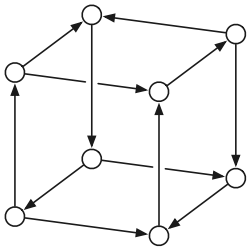
---

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

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



2. A ***k-orientation*** of an undirected graph G is an assignment of directions to the edges of G so that every vertex of G has at most k incoming edges. For example, the figure below shows a 2-orientation of the graph of the cube.



   Describe and analyze an algorithm that determines the smallest value of k such that G has a k-orientation, given the undirected graph G input. Equivalently, your algorithm should find an orientation of the edges of G such that the maximum in-degree is as small as possible. For example, given the cube graph as input, your algorithm should return 2.

3. Suppose you have a sequence of jobs, indexed from 1 to i, that you want to run on two processors. For each index i, running job i on processor 1 requires $A[i]$ time, and running job i on processor 2 takes $B[i]$ time. If two jobs i and j are assigned to different processors, there is an additional communication overhead of $C[i,j] = C[j,i]$. Thus, if we assign the jobs in some subset $S \subseteq \{1,2,\ldots,n\}$ to processor 1, and we assign the remaining $n - |S|$ jobs to processor 2, then the total execution time is

$$\sum_{i \in S} A[i] \;+\; \sum_{i \notin S} B[i] \;+\; \sum_{i \in S} \sum_{j \notin S} C[i,j].$$

Describe an algorithm to assign jobs to processors so that this total execution time is as small as possible. The input to your algorithm consists of the arrays $A[1..n]$, $B[1..n]$, and $C[1..n, 1..n]$.

CS 473 ✧ Spring 2020

Homework 9

Due Wednesday, April 22, 2020 at 9pm

---

1. Every year, Professor Dumbledore assigns the instructors at Hogwarts to various faculty committees. There are n faculty members and c committees. Each committee member has submitted a list of their *prices* for serving on each committee; each price could be positive, negative, zero, or even infinite. For example, Professor Snape might declare that he would serve on the Student Recruiting Committee for 1000 Galleons, that he would *pay* 10000 Galleons to serve on the Defense Against the Dark Arts Course Revision Committee, and that he would not serve on the Muggle Relations committee for any price.

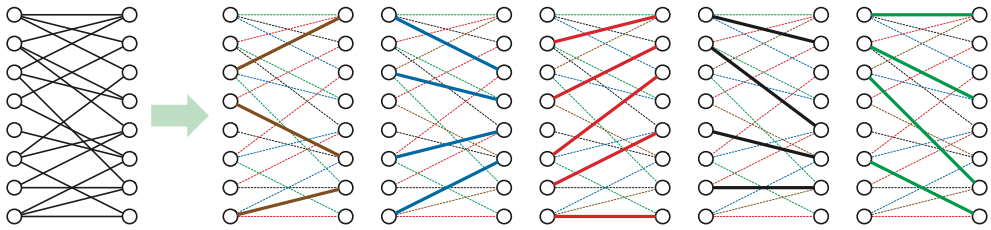
   Conversely, Dumbledore knows how many instructors are needed for each committee, and he has compiled a list of instructors who would be suitable members for each committee. (For example: "Dark Arts Revision: 5 members, anyone but Snape.") If Dumbledore assigns an instructor to a committee, he must pay that instructor's price from the Hogwarts treasury.

   Dumbledore needs to assign instructors to committees so that (1) each committee is full, (3) no instructor is assigned to more than three committees, (2) only suitable and willing instructors are assigned to each committee, and (4) the total cost of the assignment is as small as possible. Describe and analyze an efficient algorithm that either solves Dumbledore's problem, or correctly reports that there is no valid assignment whose total cost is finite.

2. Let $G = (L \sqcup R, E)$ be a bipartite graph, whose left vertices L are indexed $\ell_1, \ell_2, \ldots, \ell_n$ and whose right vertices are indexed $r_1, r_2, \ldots, r_n$. A matching M in G is ***non-crossing*** if, for every pair of edges $\ell_i r_j$ and $\ell_{i'} r_{j'}$ in M, we have $i < i'$ if and only if $j < j'$. If we place the vertices of G in index order along two vertical lines and draw the edges of G as straight line segments, a matching is non-crossing if its edges do not cross.

   Describe and analyze an algorithm to find the smallest number of disjoint non-crossing matchings $M_1, M_2, \ldots, M_k$ that cover G, meaning each edge in G lies in exactly one matching M_i.

   [Hint: How would you compute the largest non-crossing matching in G?]



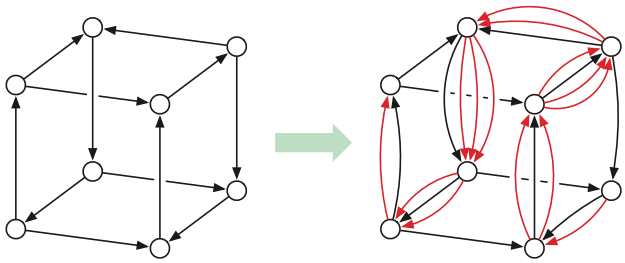
Decomposing a bipartite graph into non-crossing matchings.

3. An *Euler tour* in a directed graph G is a closed walk (starting and ending at the same vertex) that traverses every edge in G exactly once; a directed graph is *Eulerian* if it has an Euler tour. Euler tours are named after Leonhard Euler, who was the first person to systematically study them, starting with the Bridges of Königsberg puzzle.

   (a) Prove that a directed graph G with no isolated vertices is Eulerian if and only if (1) G is strongly connected[1] and (2) the in-degree of each vertex of G is equal to its out-degree.[2] *[Hint: Flow decomposition!]*

   (b) Suppose that we are given a strongly connected directed graph G with no isolated vertices that is *not* Eulerian, and we want to make G Eulerian by duplicating existing edges. Each edge e has a duplication cost $€(e) \geq 0$. We are allowed to add as many copies of an existing edge e as we like, but we must pay $€(e)$ for each new copy. On the other hand, if G does not already have an edge from vertex u to vertex v, we cannot add a new edge from u to v.

   Describe an algorithm that computes the minimum-cost set of edge-duplications that makes G Eulerian.



Making a directed cube graph Eulerian.

[1] A directed graph G is *strongly connected* if, for any two vertices u and v, there is a directed walk in G from u to v and a directed walk in G from v to u.

[2] The *in-degree* of a vertex is its number of incoming edges; the *out-degree* is its number of outgoing edges.

CS 473 ✧ Spring 2020

Homework 10

Due Wednesday, April 22, 2020 at 9pm

This is the last homework assignment.

---

1. (a) Give a linear-programming formulation of the ***bipartite maximum matching*** problem. The input is a bipartite graph $G = (U \cup V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G. Your linear program should have one variable for each edge. (Don't worry about the optimal solution being integral; it will be.)

   (b) Now derive the dual of your linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

2. Given points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in the plane, the ***linear regression problem*** asks for real numbers a and b such that the line $y = ax + b$ fits the points as closely as possible, according to some criterion. The most common fit criterion is minimizing the ***L_2 error***, defined as follows:[1]

$$\varepsilon_2(a, b) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

   But there are many other ways of measuring a line's fit to a set of points, some of which can be optimized via linear programming.

   (a) The ***L_1 error*** (or *total absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_1(a, b) = \sum_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

   Describe a linear program whose solution (a, b) describes the line with minimum L_1 error.

   (b) The ***L_∞ error*** (or *maximum absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_\infty(a, b) = \max_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

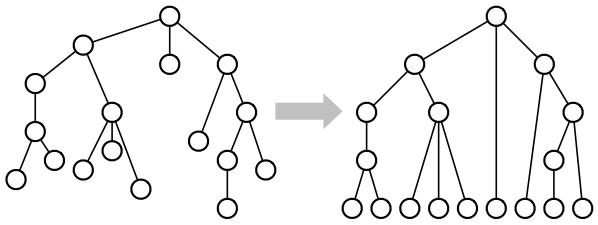
   Describe a linear program whose solution (a, b) describes the line with minimum L_∞ error.

---

[1] This measure is also known as *sum of squared residuals*, and the algorithm to compute the best fit is normally called *(ordinary/linear) least squares*.

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

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



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

   (b) Give a concise linear programming formulation of this problem.

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

   (d) Describe and analyze an algorithm that solves this problem in $O(n)$ time. Your algorithm should either compute the minimum total weighted stretch, or report correctly that the total weighted stretch can be made arbitrarily negative.