

PROBLEM SET 2
Due: Friday, September 27

Reading: CLR Sections 6.1, 6.2, 6.3, & 6.6; Start reading Chapters 7 & 8.

Suggested Problems: 6.1-1, 6.1-2, 6.1-3, 6.1-4; 6.2-2, 6.2-3, 6.2-4, 6.2-11; 6.3-1, 6.3-2, 6.3-3 ; 6.6-1; 6-2;

Required Problems. Write up and turn in the solutions to the problems listed below. Notes:

- The last problem (Problem 5) does not depend on combinatorics or probability, so you may want to do it first. It is last only because its problem statement is longer than that of the other problems.
- Problems 4a and 5d require you to design algorithms rather than just analyze them. I recommend that you look at these problems early on so that you have sufficient time to think about them. Just having them in the back of your mind can sometimes lead to unexpected progress. The kind of creativity involved in algorithm design is difficult to summon at the last minute!

Problem 1: Let Me Count the Ways ... [20]

In the following parts, please show how you derive your answers:

a [5] How many ways can 12 students be assigned to the 15 computers in E101? (Assume each student is assigned to one computer.)

b [5] How many long-distance phone numbers are there? Assume that a long distance phone number is an area code followed by a 7-digit local number. An area code is a 3-digit number whose middle digit must be 0 or 1, whose first digit cannot be 0 or 1, and whose last two digits cannot be 11. (Note: there now exist area codes that violate these rules, but ignore this fact.) The first 3 digits of the 7-digit local number cannot be an area code.

c [5] How many terms are there in the expansion of $(w + x + y + z)^7$? Each term has the form $cw^{e1}x^{e2}y^{e3}z^{e4}$, where $e1 + e2 + e3 + e4 = 7$.

d [5] How many ways can 100 students be scheduled into 5 different writing classes so that each class has exactly 20 students and each student takes one class?

Problem 2: Randomizing Arrays [15]

Many array manipulation algorithms assume that the elements of an input array are *randomly permuted*. An array of n distinct items is randomly permuted if each of the $n!$ permutations is equally likely. In many cases, the array input to such an algorithm is the output of another process that is decidedly non-random. So it is desirable to find an efficient way to randomly permute a given array.

Suppose that:

- A is a length n array containing all the integers from 1 to n .
- $\text{swap}(A, i, j)$ swaps the values in $A[i]$ and $A[j]$.
- $\text{random}(x, y)$ is a routine that returns a random integer between x and y , inclusive.

For each of the following algorithms, indicate whether the algorithm randomly permutes the input array A . Justify your answer in each case. (*Hint*: try the algorithms for $n = 3$, and generalize.)

a [5] `for i <- 1 to n`
 `do A[i] <- random(1, n)`

b [5] `for i <- 1 to n`
 `do swap(A, i, random(1, n))`

c [5] `for i <- 1 to n`
 `do swap(A, i, random(i, n))`

Problem 3: The Infamous Monty Hall Problem [10]

Do CLR Exercise 6.2-10 (p. 110). Justify your answer with a probability tree. It is important to note that the emcee *knows* where the prize is hidden, and so will always be able to reveal an empty stage behind one of the curtains you have not picked. (This is a classic problem that got national attention a few years back when it appeared in a popular column by Marilyn vos Savant. Many readers of her column, including those with math PhDs, made fools of themselves by adamantly arguing the wrong answer.)

Problem 4: Coin Flipping [25]

We all know that a flip of a fair coin can be used to choose one of two possibilities with equal probability. It is not hard to see that a n flips of a fair coin can choose one of 2^n possibilities with equal probability. But did you know that a fair coin can be used to choose among m possibilities, where m is not a power of 2? In this problem, you will explore the case of $m = 3$, but the strategy generalizes to any m .

Suppose you are given a black-box procedure `Flip()` that returns one of the symbols `H` or `T`, each with a probability of $1/2$.

a [15] Using `Flip()`, write pseudocode for a procedure `One-Third()` that returns one of the symbols `A`, `B`, or `C`, each with a probability of $1/3$.

b [5] Demonstrate that your `One-Third()` procedure from part a is correct by showing that the probability of returning `A` is $1/3$. (Presumably, the argument for `B` and `C` will be similar, so you need not show it.)

c [5] Determine the expected number of times that your `One-Third()` procedure will call `Flip()` before returning a result.

Problem 5: 2D-Searching [30]

Call a 2-dimensional array of integers *2d-sorted* if each row is sorted from low to high value and each column is also sorted from low to high value. Here is an example of a 2d-sorted 8 x 8 array named *B*

2	4	7	11	12	13	19	23
3	8	14	16	17	24	25	27
6	10	19	27	31	35	37	42
9	13	26	30	32	40	43	48
11	15	29	34	41	42	45	50
14	17	33	37	44	49	51	61
18	22	35	38	46	53	57	63
21	25	39	42	47	54	60	64

The notation $B[i, j]$ refers to the element in the i th row and j th column of B , where row and column indices start at 1. For example, $B[3, 2]$ is 10.

In this problem, we will consider various algorithms for determining whether a 2d-sorted $n \times n$ array contains a given number k . The algorithms will be expressed as a function `2D-Search` that take a 2d-sorted $n \times n$ array and a number k as arguments and returns a boolean that is *true* if k is in the array and is *false* otherwise.

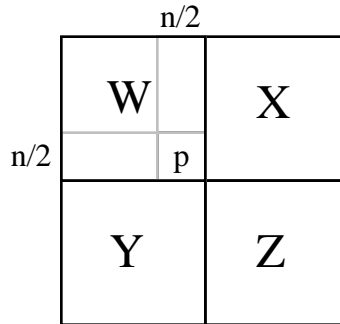
a [5]. Below is a straightforward iterative algorithm for searching a 2d-sorted $n \times n$ array for k . Give the worst-case running time of this algorithm as a function of n .

```
2D-Search-a (A, k)
  for i <- 1 to n    {number of rows}
    for j <- 1 to n  {number of columns}
      if A[i, j] = k then return true  {return exits the function immediately}
  return false
```

b [5]. Suppose there is a function `Binary-Search(A, i, k)` that performs binary search on the i th row of A to determine if k is in the row; it returns *true* if k is in the row and *false* otherwise. Below is an algorithm that uses `Binary-Search` to search for k in a 2d-sorted $n \times n$ array. Give the worst-case running time of this algorithm as a function of n .

```
2D-Search-b (A, k)
  for i <- 1 to n    {number of rows}
    if Binary-Search(A, i, k) then return true
  return false
```

c. [5] Another way to search for k in a 2d-sorted $n \times n$ array is to use divide and conquer to generalize Binary-Search to work on a 2d-sorted array A . Here's the idea: suppose that p is the value in the array at $A[n/2, n/2]$. (See the following picture. For simplicity, assume that n is a power of 2 throughout this part.)



Then if $k = p$, the search terminates with *true*; if $k < p$, the search continues in the three square quadrants W, X, and Y; and if $k > p$, the search continues in the three square quadrants X, Y, and Z. Give the worst-case running time of this algorithm as a function of n . Use the iteration method on a recurrence equation to justify your answer.

d. [10] Develop an $O(n)$ worst-case running time algorithm that implements 2D-Search on a 2d-sorted $n \times n$ array. Give pseudocode for your algorithm, and justify why it is $O(n)$. *Hint:* Any number k can be used to divide a 2d-sorted array into two parts: one consisting of the elements $\leq k$, and the other consisting of the elements $> k$. For example, the following figure shows a thick line separating the two parts for the sample array B and $k = 20$.

2	4	7	11	12	13	19	23
3	8	14	16	17	24	25	27
6	10	19	27	31	35	37	42
9	13	26	30	32	40	43	48
11	15	29	34	41	42	45	50
14	17	33	37	44	49	51	61
18	22	35	38	46	53	57	63
21	25	39	42	47	54	60	64

e. [5] Rank the algorithms from parts a through d by running time. An algorithm with running time f should appear above an algorithm with running time g if $f = O(g)$, and should appear on the same line if $f = \Theta(g)$.

Extra Credit Problem: Envelopes [20] (The best probability problem I know of! The following wording comes from Uri Wilensky's MIT Phd thesis)

There are two envelopes in front of you. One contains x dollars and the other contains $2x$ dollars. You choose one, open it, and get \$100. You are then offered a switch if you want it. You can take the other envelope instead.

Here are two arguments:

1. There were 2 envelopes, one contained x dollars and the other $2x$. You got either x or $2x$. Each was equally likely, so switching doesn't help.
2. The other envelope contains either \$200 or \$50. These are equally likely. So the expected value of the other envelope is $(\$200 + \$50)/2 = \$125$. You should switch.

Which if any of these arguments do you believe? What is the bug in the other argument?

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS231 Problem Set 2

Due Friday, September 27, 1996

Name:

Date & Time Submitted (*only if late*):

Collaborators (*anyone you collaborated with in the process of doing the problem set*):

*In the **Time** column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [20]		
Problem 2 [15]		
Problem 3 [10]		
Problem 4 [25]		
Problem 5 [30]		
Extra Credit [20]		
Total		