1. Suppose an array of length n is populated randomly with the letters A and B, with the

guarantee that both letters occur at least once and that A and B are equally likely to

occur. Let Search(S, x) be the usual algorithm for searching for an element x in

an array S, which returns the position of the first occurrence of x in S (or –1 if not

found). Prove that the average-case running time for Search(S,x) is < 2 whenever

x is either A or B, and therefore Search (for A or for B) has average case asymptotic

running time O(1).

**Let X be a random variable denoting the running time of the algorithm to search for element x in array S.**

**For any given index i in the array S there is a 50% chance of either getting an A or a B.**

**So if f(n) = c·n for finding A, for some constant c, and likewise g(n) = d·n, for some constant d.**

**E[X] = 0.5·f(n) + 0.5·g(n) = (c/2)·n + (d/2)·n**

**Which is O(n).**

1. A company uses a well-known sorting algorithm to sort its data. A best case for this

sorting algorithm occurs when its input is an already-sorted array. In such cases, it

runs in O(n) time. A worst case occurs when its input is reverse-sorted. In that case, it

runs in O(n2) time. The company knows from experience that all input arrays are

either sorted or reverse sorted, but nearly all input arrays are already sorted. In fact, it

is estimated that, for any collection of n arrays from the pool of all length-n arrays in

the company’s data store, only one of these arrays is ever reverse-sorted. What is the

average-case asymptotic running time of the algorithm, given this distribution of

inputs? Prove your answer. *Hint*: Review the Lesson 3 slides.

**Let X be a random variable denoting the running time of the algorithm on any given input array that could occur. Since already sorted arrays run in θ(n) time, there is a constant c and a function f such that running time for such arrays is asymptotically equal to f(n) = c·n for some c. Likewise, for reverse sorted arrays, there is a constant d and a function g such that running time to sort these arrays is asymptotically equal to g(n) =d·n2. We may therefore write:**

**X(arr) = f(arr.length) if arr already sorted**

**g(arr.length) if arr reverse sorted**

**Since the probability of each running time is 1/2, we have**

**E[X] = 0.5·f(n) + 0.5·g(n) = (c/2).n + (d/2)·n2**

**Which is θ(n2).**

1. A die is tossed repeatedly.
   1. What is the expected number of tosses required to get a 6?
   2. What is the expected number of tosses required to get a total of three 6’s?

In each case, prove your answer.

1. **Let Y be a random variable denoting the number of trials required to get heads.**

**Pr(Y=1) = 1/6**

**Pr(Y=2) = 1/36**

**Pr(Y=n) = 1/6n**

**Therefore,**

**E[Y] = 1.Pr(Y=1) + 2.Pr(Y=2).…**

**= For 1 = i < ∞: (i/6i)**

**= 6**

1. **Since each toss is an independent event.**

**A = {6}**

**Pr(A) = 1/6**

**Pr(A)Pr(A)Pr(A) = (1/6).(1/6).(1/6) = 1/63**

1. Design an algorithm that does the following: Input is a set S of n integers together

with an integer k. Your algorithm outputs “true” if there is some subset of S, the sum

of whose elements is exactly k; it outputs “false” if no such subset can be found.

What is the asymptotic running time of your algorithm? Explain.

**Algroithm CheckSum(S, k)**

**Input: S a set of n integers, k a non-negative integer**

**Output: True if subset summing up to k of set S exists, otherwise false.**

**for i** ← **0 to n – 1 do:**

**rollingSum** ← **S[i]**

**for j** ← **0 to n – 1 do:**

**if i ≠ j then:**

**rollingSum += S[j]**

**if rollingSum = k**

**return true**

**if rollingSum > k**

**rollingSum -= S[j]**

**return false**

**Running time: O(n2) because for all i in set S, j is checked.**

1. Goofy has thought of a new way to sort an array arr of n distinct integers: Create a

temp array and copy arr into temp.

* 1. Step 1: Check if temp is sorted. If so, return.
  2. Step 2: Randomly permute the elements of arr and place the result in temp.
  3. Step 3: Repeat Steps 1 and 2 until there is a return.

Will Goofy’s sorting procedure work at all? What is a best case for GoofySort? What

is the running time in the best case? What is the worst-case running time? What is the

average case running time?

**Will this work?**

**Yes. However, it’s not optimal.**

**Best case:**

**O(n)**

**Copy a sorted array arr into temp and terminate**

**Worst case running time:**

**Unbounded.**

**Average case running time:**

**Unbounded.**

1. Recall the recursive algorithm for computing the nth Fibonacci number and the fact

that it runs in exponential time. Improve this algorithm by thinking of the algorithm

as a procedure to solve many subproblems, and to put the solutions to these

subproblems together to obtain a final solution. For computing the nth Fibonacci

number, the subproblems are computations of the mth Fibonacci number for each

m < n. Putting these together allows the algorithm to compute the nth Fibonacci

number. The problem with the recursive algorithm is that it *re-computes* solutions to

these subproblems over and over again.

In this problem, re-work the recursive Fibonacci algorithm fib(n) so that, for each m < n, when fib(m) is computed for the first time, the return value is stored (in an array,

for example), and when this value is needed at a later stage, instead of re-computing,

the algorithm simply reads the stored value. (Storing values of solutions to

subproblems is called *memoization.*) What is the running time of your new algorithm?

**Algorithm fib(n)**

**Input: Non negative integer n**

**Output: fib(n) that’s fib(n-1) + fib(n-2)**

**F** ← **{n}**

**F[0] = 1**

**F[1] = 2**

**if F[n] > 0 then**

**return F[n]**

**else**

**F[n] = fib(n-1) + fib(n-2)**

**Running time: O(n).**