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).

E(X) = ∑ X * Pr(X=x)

= 0.5 * 1/n + 0.5 * (n-1)/n = 1/2n + (n-1)/2n = 1/2n * (1+n-1) = n/2n = 1/2

Because there is a guarantee that A & B will both appear then any n >= 2. Hence expected value is less than 2

2.

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.

E(X) = ∑ X * Pr(X=x)

= n * (n-1)/n + n2 * 1/n = n-1 + n = 2n − 1

Hence average asymptotic runtime will O(n)

3. A dice is tossed repeatedly.

A.

Probability (p) of getting a 6 in throwing a die is 1/6 and the probability (q) of not getting a 6 is 1−1/6=5/6. Let xth throw of the die results in a 6. This means that preceding (x-1) throws do not result in a 6. So, the probability function is

p(x)=qx−1 * p for x= 0,1,2,…….

Hence, E(x)=∑ x * qx−1 * p

=p * ∑x * qx−1

=p * (1+2q+3q2+4q3+....)

=p * (1−q)−2

=1/6 * (1−5/6)−2

=(1/6) * (1/6)−2

=1/6 ∗ 36=6

B.

There are 4 cases.

case 1. The first toss is a non-six (probability 5/6). Then we start again after one toss.

case 2. The first toss is a six, but the second is a non-six (probability 1/6∗5/6=5/36). Then we start over after two tosses.

case 3. The first two tosses are sixes, but the third is a non-six (probability 1/6∗1/6∗5/6=5/216). Then we start over after three tosses.

case 4. The first three tosses are sixes (probability 1/216). Event occurred

Thus we have

E=5/6(E+1)+5/36(E+2)+5/216(E+3)+1/216(3)⟹ E = 258

4. 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

Fucntion FindSumInSet

Input S: Array, index: int, K:int

Outpu: Boolean (true if exist, false, otherwise)

IF K =0 return true

If s.length == 0 && K != 0 false

If(S[index-1] >K)

Return FindSumInSet(S, index-1, K)

Return FindSumInSet(S,index-1,K) OR FindSumInSet(S,index-1,K-S[index-1])

5. 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. a. Step 1: Check if temp is sorted. If so, return. b. Step 2: Randomly permute the elements of arr and place the result in temp. c. 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?

Algorithm won’t work and there is no guarantee that it will never finish execution because there is no guarantee that randomizing elements permutation will be different each time and generate sorted set.

Best case: array is already sorted. Then running time is O(n)

Worst case: O(∞) as algorithm may never end

Average case: There are N! distinct permutation of the array. Each permutation would cost O(n) to generate. Then average running time will be O(n*n!)

6. 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, cache)

Input: a natural number n, cache is array of found solutions = []

Output: F(n)

if (n = 0 || n = 1) then return n

if(cache[n] != 0) return cache[n]

else cache[n] = fib(n-1) + fib(n-2)

return cache[n]

Because no node is called more than once, this memorization strategy has a time complexity of O(N), not O(2N).