

-
1. (a) *Use the subroutine `probefluxcapacitor()` to implement the routine `randbit()` that outputs a uniformly random bit.*

Pseudocode

```
def randbit():  
    while true:  
        (bitA, p) = probefluxcapacitor()  
        (bitB, p) = probefluxcapacitor()  
        if bitA does not equal bitB:  
            return bitA
```

Correctness

If the bits were uniformly random then a 1 or a 0 would be equally probable. This would mean that the probability that either one would occur would need to be $\frac{1}{2}$ in order to be uniformly random.

For the problem we are given that $P(1) = p$ and $P(0) = (1 - p)$. The first thing we have to figure out is how do we make an algorithm such that the probability of returning a 1 is equal to the probability of returning a 0. The second thing we have to figure out is how do we make sure the final probability for any run of `randbit()` is equal $\frac{1}{2}$.

Notice the probability of two independent events, A and B , occurring together is $P(A \cap B) = P(A) \cdot P(B)$ [1]. If we let those two events be the probability of `randbit()` returning a 1, $P(1)$, and the probability of `randbit()` returning a 0, $P(0)$, then we can say that the probability of them both occurring, $P(1 \cap 0)$, is $p \cdot (1 - p)$. This helps because we know that one call to `probefluxcapacitor()` will either return p , for bit value 1, or $(1 - p)$, for bit value 0. But if we combine two

calls to `probefluxcapacitor()` sequentially and then compare the results we can create a probability like $P(1 \cap 0)$.

Two calls to `probefluxcapacitor()` could result in 4 four combinations:

- $P(1 \cap 1) = p \cdot p$
- $P(0 \cap 0) = (1 - p) \cdot (1 - p)$
- $P(1 \cap 0) = p \cdot (1 - p)$
- $P(0 \cap 1) = (1 - p) \cdot p$

The pseudocode above only returns a bit if the probability combination was $p \cdot (1 - p)$ or $(1 - p) \cdot p$ which are in fact equal to each other. This tells us that returning a 1 — bitA is 1 and bitB is 0 — and returning a 0 — bitA is 0 and bitB is 1 — have equal probability now. And since the if statement only looks at two possible combinations from the two calls to `probefluxcapacitor()` and out of only one of those possible combinations will a 1 be returned, the probability of returning a 1, $P(1)$, is $\frac{1}{2}$.

Runtime

In order to find the running time of a randomize algorithm, we need to look at the expectation of the running time. The two calls to `probefluxcapacitor()` each take $O(1)$ because they are just looking at at most two numbers. The checking of the two bits in the if statement is also $O(1)$. This means that one time through the loop takes $O(1)$. The probability of getting out of the while loop and returning is $p \cdot (1 - p)$ for a 1 or $(1 - p) \cdot p$ for a 0.

Let X be the indicator random variable associated with the returning of `rand-bit()`. X_1 means a 1 is returned and X_0 means a 0 is returned. This also means $E[X_1] = P(1)$, the probability of 1 being returned, and $E[X_0] = P(0)$, the prob-

ability of 0 being returned. This gives us

$$\begin{aligned} E[X] &= \sum_{i=0}^1 E[X_i] \\ &= E[X_0] + E[X_1] \\ &= p \cdot (1 - p) + (1 - p) \cdot p \\ &= 2p(1 - p) \end{aligned}$$

This means the running time of `randbit()` is $O(2p(1 - p) \cdot 1) = O(2p(1 - p))$.

- (b) *Implement the algorithm `randbit(p)` that outputs an independent random bit with $P(\text{randbit}(p)=1)=p$.*

Pseudocode

```
def randbit(p):
    (k, n) = p
    l = list of bits
    for i from 1 to n:
        bit = randbit()
        append bit to l
    if 1 in l:
        return 1
    else:
        return 0
```

Correctness

`randbit()` will return a 1 with a probability p of $\frac{1}{2}$. We consider putting the independent event A of running `randbit()` a single time into a collection. Then running `randbit()` n times would give us a collection of n events/bits. Since these

events are all independent then the probability of them happening together is

$$\begin{aligned}
 &= P(A_1) \cdot P(A_2) \cdot \dots \cdot P(A_n) \\
 &= \frac{1}{2} \cdot \frac{1}{2} \cdot \dots \cdot \frac{1}{2} \\
 &= \frac{1}{2^n}
 \end{aligned}$$

k is the number of outcomes we want out of the combined runs of `randbit()`. For example, if we want $P(\text{randbit}(\frac{3}{4}) = 1) = \frac{3}{4}$, then we would need look at the combined output of calling `randbit()` two times because $n = 2$. This would give us four possible possible outcomes: (0,0), (1,0), (0,1), (1,1). Out of those four possible outcomes there is a 1 contained in three of them. This means that if we return a 1 when a 1 is in a possible outcome we get $p = 3/4$.

In the pseudocode above we generalize this idea by running `randbit()` n times and keeping a list of the bits that are returned by `randbit()`. If a 1 exists in the list then we return a 1 otherwise we return a 0.

This technique doesn't appear to work for any other example so it is not correct. I was unable to figure out a solution to this problem.

(c) *Show the correctness and runtime for `randbit(p)` when p is not of the form $k/2^n$*

Correctness

Let's look at the alorithm when $p = 3/4$. This means $p = 0.11$ and i will be in the range 0 to 1. d is first set to b_1 which is a 1. Then `randbit()` is called and its output is compared to d . The probability of `randbit()` returning a 0 which would end the function is $1/2$. The next time through d is set to b_2 which is a 1. Again `randbit()` is called and its output is compared to d and the function returns if the output of `randbit()` is not the same as d . Each time, the probability of the output of `randbit()` not being equal to d is $1/2$.

I don't have any answer for proving the correctness of this problem.

Runtime

We are given that `getdigit(p,i)` runs in $O(i^c)$ time for some c . The if statement where the output of `randbit()` is compared to d has a probability of $1/2$ of being correct each time through the loop. The loop itself could run a maximum total of i times which is equal to the number of digits in the binary representation of p .

We can then say the upper bound on this function is $O(i(i^c + 1/2))$

2. *Give a linear time algorithm that determines if an element appears more than $n/2$ times in an array with probability $1 - p$ where p is small.*

Pseudocode

```
def AppearsOften(A):
    n = total elements in A
    counter = 0
    i = 1
    if n > 1:
        for j from 1 to n:
            if arethesame(i,j):
                counter = counter + 1
            elif !arethesame(i,j):
                counter = counter - 1
            if counter == 0:
                if j < n:
                    i = j + 1
                    counter = 0
        if counter > 1:
            return 'YES'
        else:
            return 'NO'
    else:
        return 'YES' # Single element in the array so it must occur > n/2 times
```

Correctness

The algorithm `AppearsOften()` goes through all of the elements in array A and attempts to keep a count of the element that appears most often in A . The base case,

when $i = 1$, occurs when A has only one element and the algorithm returns 'Yes' because $n/2 = 1/2$ and the element occurs one time which is greater than $1/2$.

For the inductive step we assume $n > 1$ and $i, j \leq n$. Each iteration through the loop, we compare the i th element with the j th element. If they are the same we increment the counter otherwise we decrement the counter. The counter helps to determine if after having gone through the elements in the array there is an element that occurs $> n/2$ times.

Runtime

The algorithm goes all n elements of array A at most one time. Keeping a counter and checking if elements are the same and checked the value of the counter take constant time, $O(1)$, each. This means the runtime for this algorithm is $O(n)$.

References

- [1] <https://www.mathsisfun.com/data/probability-events-independent.html>
- [2] <https://www.mathsisfun.com/data/binomial-distribution.html>