Exercises

1.

	1	2	3	4	5	6
h	O	O	O	O	S	S
t	S	S	S	S	S	S

(a) Siggi makes:

$$E[] = \Sigma x P(x) = \frac{4}{6}\frac{1}{2}1 + \frac{4}{6}\frac{1}{2}(-1) + \frac{2}{6}1 = \frac{1}{3}$$

Ollie makes:

$$E[] = \Sigma x P(x) = \frac{4}{6}\frac{1}{2}1 + \frac{4}{6}\frac{1}{2}(-1) + \frac{2}{6}(-1) = -\frac{1}{3}$$

(b)

$$P(\text{Siggi makes \$}) = \frac{4}{6}\frac{1}{2} + \frac{2}{6} = \frac{2}{3}$$

$$P(\text{Ollie makes \$}) = \frac{4}{6}\frac{1}{2} = \frac{1}{3}$$

(c) 0

(d)

$$P(\text{roll 1 }|\text{Siggi makes \$}) = \frac{P(\text{roll 1} \cap \text{Siggi makes \$})}{P(\text{Siggi makes \$})} = \frac{\frac{1}{12}}{\frac{2}{3}} = \frac{1}{8}$$

2. Table:

Algorithm	Monte Carlo or Las Vegas?	Expected running time	Worst-case running time	Probability of returning a truthful toad
Algorithm 1	LV	O(n)	inf	1
Algorithm 2	MC	O(n)	O(n)	$\geq 1 - \frac{1}{2}^{101}$
Algorithm 3	LV	O(n)	O(n^2)	1

(a) Comment: "Choose" can be with or without replacement. Analysis for the former is simpler. The latter can be calculated from recursion. The conclusion is the same.
Algorithm 1:
Here "choose" is interpreted as "draw without replacement." Say n=10 with 6 trustworthy toads.

$$\begin{aligned}\text{Expected time} \propto n * [&\frac{6}{10} * 1\\ &\frac{4}{10}\frac{6}{9} * 2\\ &\frac{4}{10}\frac{3}{9}\frac{6}{8} * 3\\ &\frac{4}{10}\frac{3}{9}\frac{2}{8}\frac{6}{7} * 4\\ &\frac{4}{10}\frac{3}{9}\frac{2}{8}\frac{1}{7}\frac{6}{6} * 5]\end{aligned}$$

For all n, he series in the square brackets is finite because it is smaller than $\Sigma i/2^i$ which is finite. Therefore E[run time]=O(n).

(b) Algorithm 2:

Let L be the probability that a trustworthy toad is found in the loop.

$$L \geq \sum_{i=1}^{100} \frac{1}{2^i}$$

$$P(\text{A trustworthy toad is found})$$

$$= L + \frac{1}{2}(1-L) = \frac{1}{2} + \frac{1}{2}L \geq \frac{1}{2} + \frac{1}{2}(1 - \frac{1}{2^{100}}) = 1 - \frac{1}{2^{101}}$$

(c) Algorithm 3: Same reasoning as algorithm 1.

3. (a) Yes. No. No. Yes.

   (b) Stability. FIFO s.t. order in lower bits can be preserved when radixsort proceeds to higher order bits.

Problems

---

1. Find a center or a right. Use it as a pivot to partition the list. Find a right. Partition again. Each sentence above takes O(n).

```
def sortFlamingos(fs):
  # fs: a list of flamingos
  def partition(f, s, e, p, cmp):
    # f: array
    # s: start index
    # e: end index
    # c: pivot index
    # cmp: compare function
    f[p], f[e] = f[e], f[p]
    j = e-1
    while s <= j:
      if cmp(f[s], f[e]) < 0:
        s += 1
      else:
        f[s], f[j] = f[j], f[s]
        j -= 1
    f[s], f[e] = f[e], f[s]

  def cmp1(f1, f2):
    # f1 and f2 are flamingos
    if isLeft(f1):
      return -1
    return 0

  def cmp2(f1, f2):
    if isCenter(f1) or isLeft(f1):
```

```
      return -1
    return 0

  def findFirst(fs, fun):
    c = -1
    for i, f in enumerate(fs):
      if fun(f):
        c = i
        break
    return c

  p = findFirst(fs, isRightOrCenter)
  if p != -1: # has right or center
    partition(fs, 0, len(fs)-1, p, cmp1)
  else: # all left
    return

  p = findFirst(fs, isRight)
  if p != -1: # has right
    partition(fs, 0, len(fs)-1, p, cmp2)
  #else: # subarray is all center
```

2. (a) Binary search (O(log n)).

```
def binsearch(fs):
  # fs is a list of flamingos, sorted
  return binsearch_helper(fs, 0, len(fs)-1)

def binsearch_helper(fs, i, j):
  if j <= i:
    if compareToStick(fs[i]) == "the same":
      return fs[i]
    else:
      return "No such flamingo"
  mid = (i & j) + ((i ^ j)>>1) # knuth midpoint
  cmp = compareToStick(fs[mid])
  if cmp == "the same":
    return mid
  if cmp == "taller":
    return binsearch_helper(fs, i, mid-1)
  if cmp == "shorter":
    return binsearch_helper(fs, mid+1, j)
```

   (b) Each step eliminate half from consideration.

3. (a) For each k, check the number k's in A in the same fashion of binary search. So O(k log n).

```
def checkNum(k, m, M):
  if M <= m:
    return m
  mid = (m & M) + ((m ^ M)>>1) # knuth midpoint
  if isThereFewerks(k, mid):
    return checkNum(k, m, mid-1)
```

```
  if isThereMoreks(k, mid):
    return checkNum(k, mid+1, M)
  return mid

def probeA(n_given, k_given):
  out = []
  for k in range(1, k_given+1):
    n_left = n_given-len(out)
    ans = checkNum(k, 0, n_left)
    out += [k]*ans
  return out
```