Week 10 Problem Set
Randomised Algorithms

Congratulations on reaching the end of this course!

1. (Random numbers)

   a. A program that simulates the tosses of a coin is as follows:

```
#define NTOSSES 20
srand(time(NULL));
int i, count = 0;
for (i=0; i<NTOSSES; i++) {
   int toss = rand() % 2;     // toss = 0 or 1
   if (toss == 0) {
      putchar('H');
      count++;
   } else {
      putchar('T');
   }
}
printf("\n%d heads, %d tails\n", count, NTOSSES-count);
```

   Sample output is:

```
HHTTTTHHTTTHHTHHHHHH
12 heads, 8 tails
```

   1. What would an analogous program to simulate the repeated rolling of a die look like?
   2. What could be a sample output of this program?

   b. If you were given a string, say "hippopotamus", and you had to select a random letter, how would you do this?

   c. If you have to pick a random number between 2 numbers, say *i* and *j* (inclusive), how would you do this? (Assume *i<j*.)

Answer:

a. There are 6 outcomes of dice rolling: the numbers 1 to 6.
- So we'll need a fixed array *count[0..5]* of length 6 to count how many of each.
- This needs to be initialised to all zeros.

We roll the die *NROLLS* times, just like we tossed the coin.
- The outcome each time will be *roll = 1 + rand()%6* .
- This is a number between 1 and 6. (*Easy to see?*)

We increment the count for this *roll*:
- *count[roll-1]++*

We conclude by printing the contents of the *count* array.

Output such as the following would be possible (for 20 rolls of the die):

```
25426251423232651155
Counts = {3,6,2,2,5,2}
```

- The list of digits shows the sequence of numbers that are rolled.
- The count shows how often each number 1, 2, ..., 6 appeared.

b. We pick a random number between 0 and 11 (as there are 12 letters in the given string, and they will be stored at indices 0 .. 11), and then print that element in the character array.

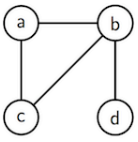
```
srand(time(NULL));             // an arbitrary seed
char *string = "hippopotamus";
int size = strlen(string);
int ran = rand() % size;       // 0 ≤ ran ≤ size-1
printf("%c\n", string[ran]);
```

c. We pick a random number between 0 and *j-i*, and add that number to *i*. Note that we must compute *rand()* modulo (*j-i+1*) because the number *j-i* should be included.

```
srand(time(NULL));             // an arbitrary seed
int ran = rand()%(j-i+1);      // 0 ≤ ran ≤ j-i
int num = i + ran;             // i ≤ num ≤ j
printf("%d\n", num);
```

2. (Karger's algorithm)

Consider the graph *G*:



   a. Find a minimum cut of *G*.

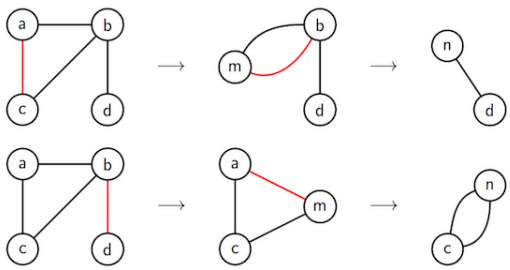
   b. Show two different ways in which randomised edge contraction may execute on *G*, one that results in a minimum cut and one that doesn't.

   c. Determine the total number of possible executions of the edge contraction algorithm on *G*. How many of these result in a minimum cut? What, therefore, is the probability for Karger's algorithm to find a minimum cut for *G*?

Answer:

a. The unique minimum cut is {a,b,c},{d}. The weight is 1.

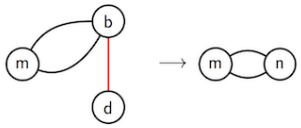
b. Two possible execution of the contraction algorithm:



The first execution results in a minimum cut of weight 1. The second execution results in a cut of weight 2.

c. In the first step there are four edges to choose from.

- Selecting edge a-b, edge a-c or edge b-c all result in an intermediate graph like the one in the first execution from above. This therefore happens with probability ¾. If in the next step you choose any of the two edges between m and b, the result would be the same minimum cut. The likelihood for this to happen is ¾·⅔ = ½. If instead edge b-d is selected, the result would be:



  This happens with probability ¾·⅓ = ¼.

- If edge b-d is selected first as shown in the second execution for Exercise b, then in the next step you can contract any of the edges a-c, a-m or c-m to always obtain a graph with two nodes and two edges between them, hence a cut of weight 2. The likelihood for this to happen is therefore ¼·1 = ¼.

To summarise, there are 4·3 possible executions, half of which result in the minimum cut. Hence, if randomised edge contraction is executed $\left\lceil \binom{4}{2} \cdot \ln 4 \right\rceil = 9$ times as required by Karger's algorithm, then the likelihood to find a minimum cut is 1 - (½)^9 = 99.8%.

3. (Analysis of randomised algorithms)

Random permutations can be a good way to test the average runtime behaviour of implementations, for example to test sorting algorithms, BST insertion strategies etc.

The following algorithm can be used to compute a uniform random permutation of a given array:

```
randomiseInPlace(A):
|  Input  array A[1..n]
|  Output random permutation of A
|
|  for all i=1…n do
|     swap A[i] with A[random number between i and n]
|  end for
|  return A
```

Does the following variation also produce each possible permutation with equal probability?

```
randomiseWithAll(A):
|  Input  array A[1..n]
|  Output random permutation of A
|
|  for all i=1…n do
|     swap A[i] with A[random number between 1 and n]
|  end for
|  return A
```

Answer:

Somewhat surprisingly, the answer is no. Take, for example, the array `[1,2,3]`. There are 6 permutations, so each one of them should be produced with probability 1/6. But the permutation `P = [3,2,1]`, say, is returned with lower probability, as we can see when we consider all the ways in which `P` can be obtained:

```
i=1: swap A[1],A[1]   i=2: swap A[2],A[2]   i=3: swap A[3],A[1]   probability 1/3*1/3*1/3 = 1/27
i=1: swap A[1],A[3]   i=2: swap A[2],A[2]   i=3: swap A[3],A[3]   probability 1/3*1/3*1/3 = 1/27
i=1: swap A[1],A[2]   i=2: swap A[2],A[1]   i=3: swap A[3],A[1]   probability 1/3*1/3*1/3 = 1/27
i=1: swap A[1],A[3]   i=2: swap A[2],A[3]   i=3: swap A[3],A[2]   probability 1/3*1/3*1/3 = 1/27
```

This shows that `randomiseWithAll` returns `[3,2,1]` with probability 4/27 ≈ 0.1481 < 1/6 ≈ 0.1667.

Contrast this with `randomiseInPlace`, where every possible permutation is the result of a unique sequences of swaps. For example:

```
i=1: swap A[1],A[3]   i=2: swap A[2],A[2]   i=3: swap A[3],A[3]   probability 1/3*1/2*1 = 1/6
```

4. **Challenge Exercise**

In cryptography, a *substitution cipher* encrypts a plaintext by replacing each letter with another letter according to a fixed system. For example, if e⟶o, h⟶r, s⟶h, then the plaintext

```
she sees, he sees.
```

would be encrypted by this ciphertext:

```
hro hooh, ro hooh.
```

Even though there are 26! ≈ 4·10^{26} different encryption keys, substitution ciphers are not very strong and cryptoanalysts can break them by guessing the meaning of the most frequent letters in a ciphertext. With the help of a letter-frequency table for English, try to decrypt the ciphertext below that's been encrypted using a randomly generated substitution cipher.

```
bjy bmz gmdg: mkbyo bjoyy tmdg xabjzfb uozvomeeaqv, haky iypzeyg eymqaqvhygg. (vyzkkoyd nmeyg)
```

(The first student to email me the correct plaintext will receive accolades on this webpage.)

Answer:

Congratulations to Mung Wah Hum, who was the first to crack the code.
Simon Sillitoe came in second.

Start by counting the frequencies in the ciphertext:

```
a: 4
b: 6
c: 0
d: 3
e: 5
f: 1
g: 7
h: 2
i: 1
j: 3
k: 4
l: 0
m: 7
n: 1
o: 5
p: 1
q: 3
r: 0
s: 0
t: 1
u: 1
v: 4
w: 0
x: 1
y: 12
z: 5
```

If you replace the most frequent letter (`y`) by the most frequent letter in English (`e`) and for the next two most frequent letters in English (`t,a`) try some of the other letters with high frequency (`b,g,m`), then you may obtain the following partial text:

```
t_e ta_ _a__: a_te_ t__ee _a__ __t___t _____a_____, ___e _e___e_ _ea_____e__! (_e___e_ _a_e_)
```

You may then guess that the first word (`t_e`) could be "the" and that (`t__ee`) is probably "three":

```
the ta_ _a__: a_ter three _a__ __th__t _r__ra_____, ___e _e___e_ _ea_____e__! (_e___re_ _a_e_)
```

Then "`a_ter`" is likely to mean "after" and so on. The actual plaintext is:

```
the tao says: after three days without programming, life becomes meaningless! (geoffrey james)
```