COMP9024 18s2 Data Structures and Algorithms

Week 12 Problem Set
Approximation and Randomised Algorithms

1. (Approximation)

   Write a C program to implement the root finding approximation algorithm from the lecture as the function:

   ```
   double bisection(double (*f)(double), double x1, double x2)
   ```

   Use your program to find roots for

   a. $f(x) = x^3 - 7x - 6$, in the interval $[0.0, 10.0]$

   b. $f(x) = \sin x$, in the interval $[2.0, 4.0]$

   c. $f(x) = \sin 5x + \cos 10x + \frac{x^2}{10}$, in the intervals $[0.0, 1.0]$ and $[1.0, 2.0]$

   with precision $\varepsilon=10^{-10}$.

   Answer:

   ```
   #include <stdlib.h>
   #include <stdio.h>
   #include <math.h>

   #define EPSILON 1.0e-10

   double bisection(double (*f)(double), double x1, double x2) {
      double mid;
      do {
         mid = (x1+x2) / 2;
         if (f(x1) * f(mid) < 0)  // root to the left of mid
            x2 = mid;
         else                     // root to the right of mid
            x1 = mid;
      } while (f(mid) != 0 && x2-x1 >= EPSILON);
      return mid;
   }

   double f1(double x) {
      return pow(x,3) - 7*x - 6;
   }

   double f3(double x) {
      return sin(5*x) + cos(10*x) + x*x/10;
   }

   int main(void) {
      printf("%.10f\n", root(f1, 0, 10));
      printf("%.10f\n", root(sin, 2, 4));
      printf("%.10f\n", root(f3, 0, 1));
      printf("%.10f\n", root(f3, 1, 2));
      return 0;
   }
   ```

   a. 3.0
   b. π (= 3.1415926536)
   c. 0.7371977788 and 1.1420071707

2. (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*.)

d. Write a C program that generates a random word (consisting of just lower-case letters a–z). When executed, the first command-line argument is the length of the word, and the second argument is the seed used by the random number generator, i.e. by `rand()`.

An example of the program executing could be

```
prompt$ ./randword 6 2
ebgnha
```

which seeds the random-number generator with 2 and generates a random word of length 6.

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to the problem set and week. It expects to find a file named* `randword.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun prob12
```

Note: It is important in this exercise that you call the random number generator as specified in the lecture, otherwise the outputs will not match. If you use a different method to generate random numbers, the testcases will 'fail', but your program could still be perfectly correct.

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);
```

d.

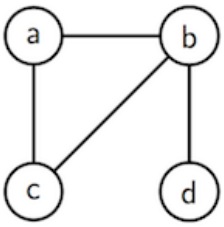
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  char *alphabet = "abcdefghijklmnopqrstuvwxyz";
  int length;
  unsigned int seed;
  if (argc == 3 &&
      sscanf(argv[1], "%d", &length) &&
```

```
      sscanf(argv[2], "%d", &seed)) {
    srandom(seed);     // seed comes from the command line
    int i;
    for (i=0; i<length; i++) {
       int ran = random()%ALPHABET_SIZE;   // 0 ≤ ran ≤ 25
       putchar(alphabet[ran]);
    }
    putchar('\n');
  }
  return 0;
}
```

3. (Karger's algorithm)

   Consider the graph *G*:



   a. Find a minimum cut of *G*.

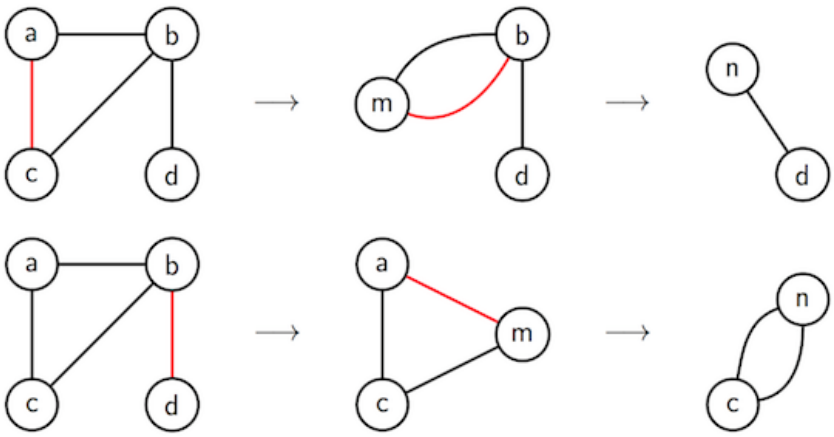
   b. Show two different ways in which randomised edge contraction may execute on *G*, one that results in a minimum cut and one that doesn't.

   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}. It has a weight of 1.

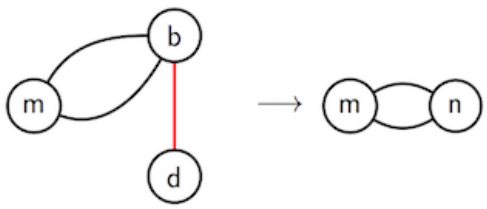
b. Two possible execution of the contraction algorithm are:



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

c. There are four edges to choose from in the first step.

- Selecting edge a-b or edge b-c results in an intermediate graph similar to the one in the first execution from above when edge a-c is selected. An intermediate state of this form is therefore reached with probability ¾. If you then choose the other edge between m and b, the result would be the same minimum cut. The likelihood for this to happen is therefore 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 you can continue to 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%.

4. (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 = [1,2,3]`, 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[3]   probability 1/3*1/3*1/3 = 1/27
i=1: swap A[1],A[1]   i=2: swap A[2],A[3]   i=3: swap A[3],A[2]   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[3]   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[1]   probability 1/3*1/3*1/3 = 1/27
```

Therefore the likelihood with which `randomiseWithAll` returns `[1,2,3]` is 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[1]   i=2: swap A[2],A[2]   i=3: swap A[3],A[3]   probability 1/3*1/2*1 = 1/6
```

5. (Feedback)

We (Michael and Michael, Michael and Shanush) want to hear from you how you liked COMP9024 and if you have any suggestions on how the course should be run in the future.

Log onto MyExperience to provide feedback. Please do so even if you just want to say, "I liked the way it was taught".

6. **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`⟶`y`, `h`⟶`r`, `s`⟶`h`, then the plaintext

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

would be encrypted by this ciphertext:

```
hry hyyh, ry hyyh.
```

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.

```
fhb fny jnvj: nmfbx fhxbb rnvj pcfhyef ixyqxnoockq, zcmb wblyobj obnkckqzbjj! (qbymmxbv
                                                                        dnobj)
```

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

Answer:

Jian Gao was the first to break the code. Andrew Cvetko came second. Well done!

Start by counting the frequencies in the ciphertext:

```
a: 1
b: 8
c: 4
d: 1
e: 2
f: 7
g: 1
h: 4
i: 2
j: 4
k: 2
l: 2
m: 3
n: 6
o: 4
p: 2
q: 3
r: 2
s: 1
t: 1
u: 1
v: 3
w: 2
x: 5
y: 5
z: 2
```

If we replace the three most frequent letters (`b,f,n`) by the most frequent letters in English (`e,t,a`) we 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)
```