Randomised Algorithms I

1 Deterministic algorithms

The algorithms we saw until now are deterministic:

- Process the input and produce an output
- If it runs many times with the same input it produced the same output
- For many "hard" computational problems
  - It is difficult to design efficient algorithms
  - or the running time is very high

Where is the difficulty:

- We often need to make choices between two (or more) alternatives
- to make a correct choice we often need to know many **structural parameters** of the input
- decisions in **previous** steps may influence the range of choices in future steps

What can we do for that?

- Design efficient heuristics:
  - They run quickly
  - But do not always compute the optimum solution
- Design approximation algorithms
  - an optimum solution is not guaranteed
  - but it is guaranteed that their output is "not far" from the optimum (small approximation ratio)
- Use randomisation

2 Randomised algorithms

- An algorithm is randomised if its behaviour (output, running time) is determined by:
  - The input and
  - The values produced by a random generator (correspond to the random choices that it makes)
- If it runs many times with the same input: it has different outputs/running times (depending on the random values each time)

3 Randomised numbers

- The procedure $RANDOM(a, b)$:
  - returns an integer between a and b
  - each of then with equal probability
  - every call of RANDOM is independent from the other ones
- In practice: real randomness is difficult
  - we use pseudo-random number generators
  - deterministic algorithms returning number that **look random**

An application of RANDOM(a,b)

- We are given a graph G with vertices 1.2,...,n

- We want to compute the random distance of a pair of vertices in G

Recall

- The algorithm BFS(G,a,b) computes the distance between the vertices a and b in G
- The randomised algorithm:

Listing 1: Random_Distance(G)

```
a=RANDOM(1,n)                  // randomly choose the first vertex a
b=a                                            // initialization
while b=a
    b=RANDOM(1,n)             // randomly choose the second vertex
return BFS(G,a,b)               //coompute the distance of a and b
```

Another application of RANDOM(a,b)

- We are given an array A with n numbers
- we want to permute its elements randomly

How can we do that

- Key idea
  - Assign each element A[i] of the array A with a ”priority” P[i]
  - Then sort the elements of A according to these priorities
- The randomised algorithm:

Listing 2: Permute-By-Sorting(A)

```
n ← length[A]
for i ← 1 to n
    do P[i]=RANDOM(1,n^3)
sort A using P as sort keys
return A
```

4 Randomised algorithms

- In a randomised algorithm:
  - The input is **not** randomly chosen: only the decisions of the algorithm
  - The running time may depend on its random choices. We calculate the ”expected running time”

Terminology:

Randomised algorithms	Average-case analysis
• Make random choices • Input is not random • expected running time	• No random choices (deterministic algorithms) • Random input • Average case running time

5 Basic probability notions

In a random experiment (e.g. a coin toss):

- First we define a sample space S
- Each subset of S is an event
- Each single element of S is also an elementary events (elementary events are also events)
- We assign to each elementary event $x \in S$ a number $Pr(x) \geqslant 0$
  Such that for every event $A = \{x_1, x_2, x_3, ..., x_k\}$
  - $\Pr(A) = \Pr(x_1) + \Pr(x_2) + \Pr(x_3) + \ldots + \Pr(x_k)$
  - Pr(S)=1 i.e.S is the certain event

The probability of an event shows:

- How probable is that the experiment has an outcome
- For two events A and B, if $A \cap B = \emptyset$, then
  - A and B are called **mutually exclusive** and
  - $Pr(A \cup B) = Pr(A) + Pr(B)$
- Two events A and B are independent if:
  - $Pr(A \cap B) = Pr(A) \cdot Pr(B)$
- In general we consider A and B as independent if they correspond to random experiments defined on different time and space domains

E.g for the random experiment "two independent tosses of a 'fair' coin":

- The sample space: $S = \{HH, HT, TH, TT\}$
- The event "one head and one tail is":
  $$A = \{HT, TH\}$$
  and has probability
  $$Pr(A) = Pr(HT) + Pr(TH)$$
  $$= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$$
- A random variable X is a function from S to the real numbers
- The expected value of X is
  $$E[X] = \sum_{x \in \mathbb{R}} x \cdot Pr(\{X = x\})$$

E.g for the experiment "toss two 'fair' coins":

- You earn £3 for each head but lose £2 for every tail
- The random variable is X="your earnings"
- The expected value of X is:

$$E[X] = 6 \cdot P(HH) + (3-2) \cdot P(\{HT, TH\}) - 4 \cdot P(TT) = 6 \cdot (1/4) + 1 \cdot (1/4 + 1/4) - 4 \cdot (1/4) = 1$$

Theorem: for any two random variables X and Y, the expected value of the random variable X+Y is

$$E[X + Y] = E[X] + E[Y]$$

- This property is called "linearity of expectation"
- It is particularly useful for applications

6 Randomised algorithms

Two main types of randomised algorithms

- Las Vegas algorithms
  - always find the correct solution
  - the running time varies for every execution
- Monte Carlo algorithms
  - not always the correct solution
  - but we can bound the probability of an incorrect solution
- Both very useful depending on the application

7 Monte Carlo algorithms

For decision problems (answer is YES/NO), there are two kinds of Monte Carlo algorithms

- Algorithms with two-sided error:
  - there is a non zero probability that the answer is wrong, when it answers YES and when it answers NO
- Algorithms with one-sided error:
  - there is a non zero probability that the answer is wrong, when it answers NO
  - the answer is always correct, when it answers YES
  - if we run the algorithm repeatedly:
    * the failure probability becomes very small
    * the running time increases

8 One sided error

- You have two £1 coins
- Bob claims that the first one is real and the second is fake
- You want to verify Bob's claim:
  - So you ask hum questions
  - what do you ask
- Main idea:
  - if the coins are not the same, you finally believe Bob
  - if the coins are the same, you will catch Bob lying
- A Monte-Carlo algorithm:
  - Put the coins behind your back
  - Pick one randomly (you know which one you have picked)
  - ask Bob whether the picked coin is real or fake
- Why does it work
  - If the coins are different Bob will always recognise the fake one
  - If the coins are the same:
    * In this case, Bob lies
    * He does not know which one you picked
    * he will guess $\Rightarrow$ he answers incorrectly with probability $\frac{1}{2}$

Our decision is ”does Bob lie?”

- if you answer YES
  - then you caught Bob lying
  - $\Rightarrow$ you are sure he lies (always correct YES answer)
- If you answer NO
  - then you did not catch Bob lying
  - $\Rightarrow$ either he is honest, or he was just lucky (erroneous NO answer with probability $\frac{1}{2}$)
- If you repeat the experiment k times:
  - You give an erroneous NO answer with probability $\frac{1}{2^k}$ i.e. if he is too lucky (all k times)
- The graph isomorphism problem: ”given two graphs G_1, G_2, are they the same graph?
- A hard problem - nobody knows whether an efficient algorithm exists
- If Bob claims that G_1 and G_2 are not the same
  - randomly pick one of the graphs
  - ask Bob whether this is G_1 or G_2
  - similar idea with the coins

$\Rightarrow$ we have a Monte Carlo algorithm with one-sided error for the problem ”graph non-isomorphism”