# COMP20005 Workshop Week 11

| | |
|---|---|
| **1** | Discuss strategies and Ex. 9.3, 9.7 |
| **2a** | Assignment2 Q&A |
| **2b** | Working on your assignment 2, OR<br><br>Do exercises from chapters 9 (**9.2, 9.3-W11x, 9.7-W11**, 9.8, 9.11-W11) if you are 100% sure that you have submitted a perfect work for assignment 2. |

# Which strategies we learnt?

# Some strategies we learnt:

- *Generate & Test:* Ex 9.7
- *Divide & Conquer:* write a function to print out all permutations of the numbers from 1 to n, one per line
- *Simulation:* Ex 9.3 – watch workshop video for hints,
- *Physical simulation:* Ex 9.11

# Ex 9.3: Simulation and rand()?

# Simulation: using rand()

At the start of program, use

```
srand(time(NULL));
```

(with `#include <stdlib.h>` and `<time.h>`)

Then: `rand()` gives a random `int`

`rand() % n` gives a random number in range `0..n-1`

`a + rand(b-a+1)` gives a random number in range `a..b`

# Ex 9.3: Simulation?

Write a program that deals four random five-card poker hands from a standard 52-card deck. You need to implement a suitable "shuffling" mechanism, and ensure that the same card does not get dealt twice.

Then modify your program to allow you to estimate the probability that a player in a four-person poker game obtains a simple pair (two cards with the same face value in different suits) in their initial hand. Compute your estimate using 40,000 hands dealt from 10,000 shuffled decks.

How about three of a kind (three cards of the same face value)?

And a full house (three of a kind plus a pair with the other two cards)?

For example:

```
./program
player 1: 3-S, Ac-C, Qu-D, 4-H, Qu-H
player 2: 10-C, 2-H, 5-H, 10-H, Ki-H
player 3: 2-C, 6-D, 10-D, Ki-D, 9-H
player 4: 8-S, 9-S, 10-S, Qu-S, 4-D
Over  40000 hands of cards:
 19680 (49.20%) have a pair (or better)
   900 ( 2.25%) have three of a kind (or better)
    48 ( 0.12%) have a full house (or better)
```

# Ex 9.3: Simulation: How to represent the card deck?

Write a program that deals four random five-card poker hands from a standard 52-card deck. You need to implement a suitable "shuffling" mechanism, and ensure that the same card does not get dealt twice.

For example:

```
./program
player 1: 3-S, Ac-C, Qu-D, 4-H, Qu-H
```

How to declare a variable for the 52-card deck? What could be used to represent a single card?

# Ex 9.3: Simulation: How to represent the card deck?

```
./program
player 1: 3-S, Ac-C, Qu-D, 4-H, Qu-H
```

How to declare a variable for the 52-card deck? What could be used to represent a single card?

`int cards[52];` // a card is a number in range 0..51, then we can map:

0..3 → Ac-H, Ac-D, Ac-S, Ac-C // 4 in one group

4..7 → 2-H, 2-D, 2-S, 2-C

50-53 → Ki-H .. Ki-C

So if `c` is a card value (from 0 to 51)

`c/4` gives index to `{"Ac","2","3",…,"10","Ja","Qu","Ki"}`

`c%4` gives index to `{"H", "D", "S", "C"}`

Note: The mapping to the name need to be done only when we want to print the cards!

# Ex 9.3: Simulation: How to represent the card deck?

Write a program that deals four random five-card poker hands from a standard 52-card deck. You need to implement a suitable "shuffling" mechanism, and ensure that the same card does not get dealt twice.

Watch the workshop video for more hints.

Declare a card deck

Make a “brand-new” card deck

Shuffle it

And give cards one-by-one to the players

# Strategy: generate-and-test

*Other names: brute-force search, exhaustive search.*

*Principle:*

- *systematically enumerate all possible candidates for the solution, and*
- *check whether each candidate satisfies the search requirements.*

*Notes:*

- *The search space can be very large.*
- *Sometimes it is possible to significantly reduce the search space using heuristics*

# Ex 9.7

Devise a mechanism for determining the set of adjacent elements in an array of n that has the largest sum.

Which of the problem solving techniques discussed in chapter 9 might yield algorithms for this problem?

*What is the problem? What are input and outputs?*

# e9.7: The maximum subarray problem

*Problem: Given* `int a[n]`, *find* l *and* r *so that the sum of elements from* l *to* r *is the maximal possible.*

*Note 1: Array* `a[]` *can also be of type* `float` *or* `double`.
*Note 2: since the sum of zero-length sequence is* 0, *if the array contains only negative elements, the answer is a zero-length sequence (for example,* l=0 *and* r= -1*).*

*Q: what technique can we apply here?*

*But first, write the function prototype...*

# e9.7: understanding

```
 2  1  3 -4 -5  2  6  1 -8  9  2  3  1  5 -7  1 -2
 -----       -----     ----------    --
```

answer: sum= 6?  9?  20? 1?

# e9.7: understanding

```
 2  1  3  -4  -5  2  6  1  -8  9  2  3  1  5  -7  1  -2
 -----        ------------------    --
```

answer= {l=5, r=13, sum= 21} ?

*How to find answer? Think about techniques we know...*

*Build 2 solutions:*

- *Solution 1: slow, using generate-and-test*
- *Solution 2: a faster one (simulating manual job)*

# e9.7: solution1 (brute force)

```
2  1  3  -4  -5  2  6  1  -8  9  2  3  1  5  -7  1  -2
-----       ----------------------    --
```

```
int
max_subarray(int A[], int n, int *left, int *right) {
  // ANY pair (i,j) (i<=j) can be a potential answer.

  for (i=  ;    ;    ) {
    for (j=  ;      ;    ) {
      sum= sum from i to j
      if(sum>maxsum) {
        ???
      }
    }
}
```

# e9.7: solution2 – mimic a smart manual job

```
2  1  3  -4  -5  2  6  1  -8  9  2  3  1  5  -7  1  -2
-----       ------------------    --
```

```
start with maxsum= 0 (solution= empty subarray)

Checking: which pair (i,j) gives better maxsum?
For the above array:
     i=0 j=0, sum= 2 (new maxsum)
         j=1, sum= 3 (new maxsum)
         j=2, sum= 6 (new maxsum)
         j=3, sum= 2 NO new maxsum
              but should continue because …
         j=4, sum= -3 should stop increasing j...
     should we start again from i=1?
     Nope! what should be new starting value for i?

Should we start from i=0 if A[0]= -1 ?
Build a fast solution by developing the above points.
```

# Assignment 2

# marking rubric: structure

A single violation will cost you *at least* 0.5 mark. Examples:

- **avoidance of structs `(-1.0)`**

- **duplicate of code segments**
- unnecessary duplication/copying of data

- avoidance struct pointers
- inappropriate or over-complicated structs
- other abuses of structs

- functions too long/complex especially `main()`
- insufficient use of functions
- overly complex function argument lists

- global variables
- over-complicated approach

# marking rubric: execution

*Compilation warning or error messages:*

- errors in compilation that prevent testing, -4.0;
- **unnecessary warning messages in compilation, -1.0;**

*Segfaults:*

- runtime segmentation fault on any test with no output generated, -2.0;
- runtime segmentation fault on any other test with no output generated, -2.0;

*Outputs:*

- incorrect Stage 1 output on any test, -1.0;
- differently incorrect Stage 1 output on any test, -1.0;
- numerically incorrect Stage 2 output on any test, -1.0;
- different numerically incorrect Stage 2 output on any test, -1.0;
- *visually incorrect* Stage 3 output on any test, -1.0;
- different visually incorrect Stage 3 output on any test, -1.0;
- incorrect Stage 4 output on any test, -1.0;
- other execution-time issue not already captured, -1.0;