COMP20005 Workshop Week 12

1	`Problem Solving Strategies`
	Discuss and then implement a solution to Exercise 9.7.
	`Skimming: 9.3, 9.11`

2	`Implement at least one of the exercises`
	`9.3, 9.6, 9.8, 9.9, 9.10, 9.11, 9.12`
	
	Write a fun program to end the semester!

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 some heuristics*

Ex 9.7

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

Write a function named `max_subarray` for that.

What is the problem? Give a function prototype.

e9.7: The maximum subarray problem

Problem: Given `int A[n]`*, find* `left` *and* `right` *so that the sum of elements from* `left` *to* `right` *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,* `left`=0 *and* `right`= -1*).*

Q: what technique can we apply here?

But first, write the function prototype...

```
?? max_subarray(???)
```

e9.7: understanding

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

```
answer: largest sum= 20?
        left= 9    right= 13 ?
```

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*

How to apply generate-and-test?

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
      ???
    }
  }
}
```

Strategy: Simulation

- Simulate what done manually
- Simulate physical processes

e9.7: solution2 - simulation

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

- What would we do when solving this problem manually?

e9.7: solution2 - simulation

```
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?

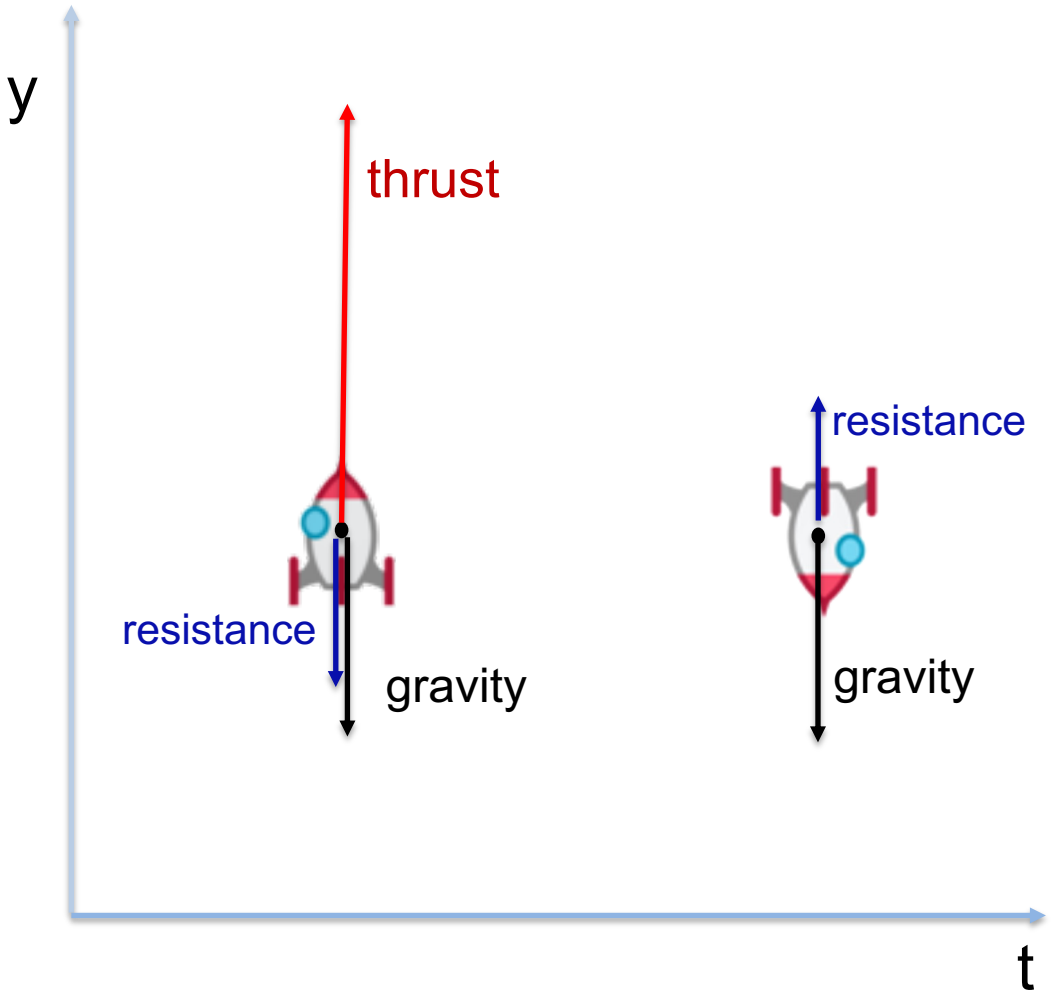
Nope! what should be new starting value for i?

Build a fast solution by developing the above points.

Ex 9.11: See grok for exercise and beautiful output

- Simulate the flight of a toy rocket – simulate a physical process

Ex 9.11: See grok W11 for exercise and beautiful output



High School Physics :-)

F= sum of all forces

m= total weight

a= F/m

t= t0 + Δt

v= v0 + a Δt

y= y0 + v Δt

--- for this task ---

m= W_{rocket} + W_{fuel}

W_{rocket} = 10

W_{fuel}= 8 – t x rate

rate= 0.8 (kg/s)

You can choose Δt as a parameter

e9.3: let's simulate a game

```
#define FACES      13
#define SUITS       4
#define CARDS (FACES*SUITS)  /* number of cards */

#define PLAYERS     4
#define CARDSINHAND 5

const char *faces[FACES] = {“Ac", “2", “3", “4", “5”, “6”,
                  “7", “8", “9", "10", ”Ja", ”Qu", ”Ki"};
const char suits[SUITS] = {'S', 'C', 'D', 'H'};

typedef struct {
    int face, suit;      // index to the above arrays
} card_t;

card_t players[PLAYERS][CARDSINHAND];
card_t deck[CARDS];



How to give each of the players random CARDINHAND cards?
```

e9.3: let's simulate a game

```
const char *faces[FACES] = {"Ac", "2", "3", "4", "5", "6",
                  "7", "8", "9", "10", "Ja", "Qu", "Ki"};
const char suits[SUITS] = {'S', 'C', 'D', 'H'};

typedef struct {
    int face, suit;      // index to the above arrays
} card_t;

card_t players[PLAYERS][CARDSINHAND];
card_t deck[CARDS];
```

How to give each of the players random CARDINHAND cards?

- we start with a "brand new" deck that has some pre-defined order (for example from Ac to Ki, from S to H (easy, right?)
- we "shuffle" the deck by re-arranging its element in a *random* order (**how**?)
- then, we deliver the cards one-by-one to the players in the round-robin manner (easy, right?)

Our last hour:

Write a fun program to end the semester!

Implement at least one of the exercises **9.6, 9.8, 9.9, 9.10, 9.11, 9.12**

Recommended:

9.06 more with numerical method: Newton-Raphson method
9.11 if you love high school physics
9.3 if you like playing poker
9.8: more with array – k-th largest value

Good Luck!