COMP10002 Workshop Week 12

Problem-solving approaches :

- generate-and-test
- reduce-and-conquer and divide-and-conquer
- simulation
- adaptation

LAB:

- finish at least one exercise from the list: 9.3, 9.7, 9.10, 9.11

Some Strategies

1. ***reduce-&-conquer***: typically, reduce the task of size n to the same task of size n-1, solving that smaller instance, and then extending its solution to the original problem.
2. ***divide-&-conquer***: breaks down a problem into two or more smaller, independent subproblems of the same type, recursively solves each subproblem, and then combines their solutions to solve the original problem.
3. ***generate-&-test (brute-force)***: systematically generates candidate solutions and then tests each one to see if it satisfies the problem's requirements.
4. ***simulation***: creating a model of a real-world or hypothetical system and running experiments with that model to observe its behaviour, analyse its properties, and predict outcomes without directly interacting with the real system.

Reduce-and-conquer

recursive algorithms:
reduce the task of size n to the same task of size n-1

- Hanoi Tower
- Permutation: prints all permutations of numbers from `1` to `n`
- subset sum: Given a set of numbers (as an array), print, if any, a subset that adds up to `k`

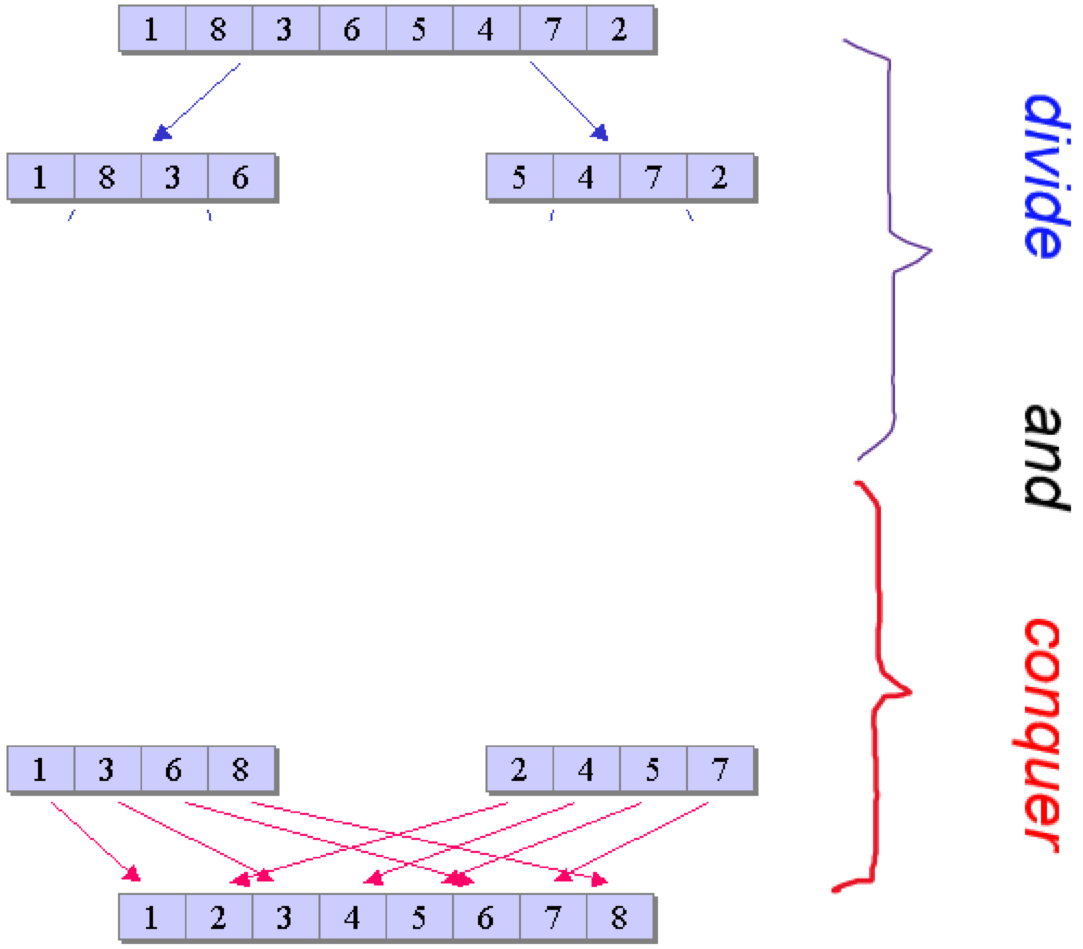
Recursion:

- what's are the inputs and outputs of the process?
- what are parameters that are changing?
- how describe the task of parameter n through the same task(s) of smaller parameter(s)?

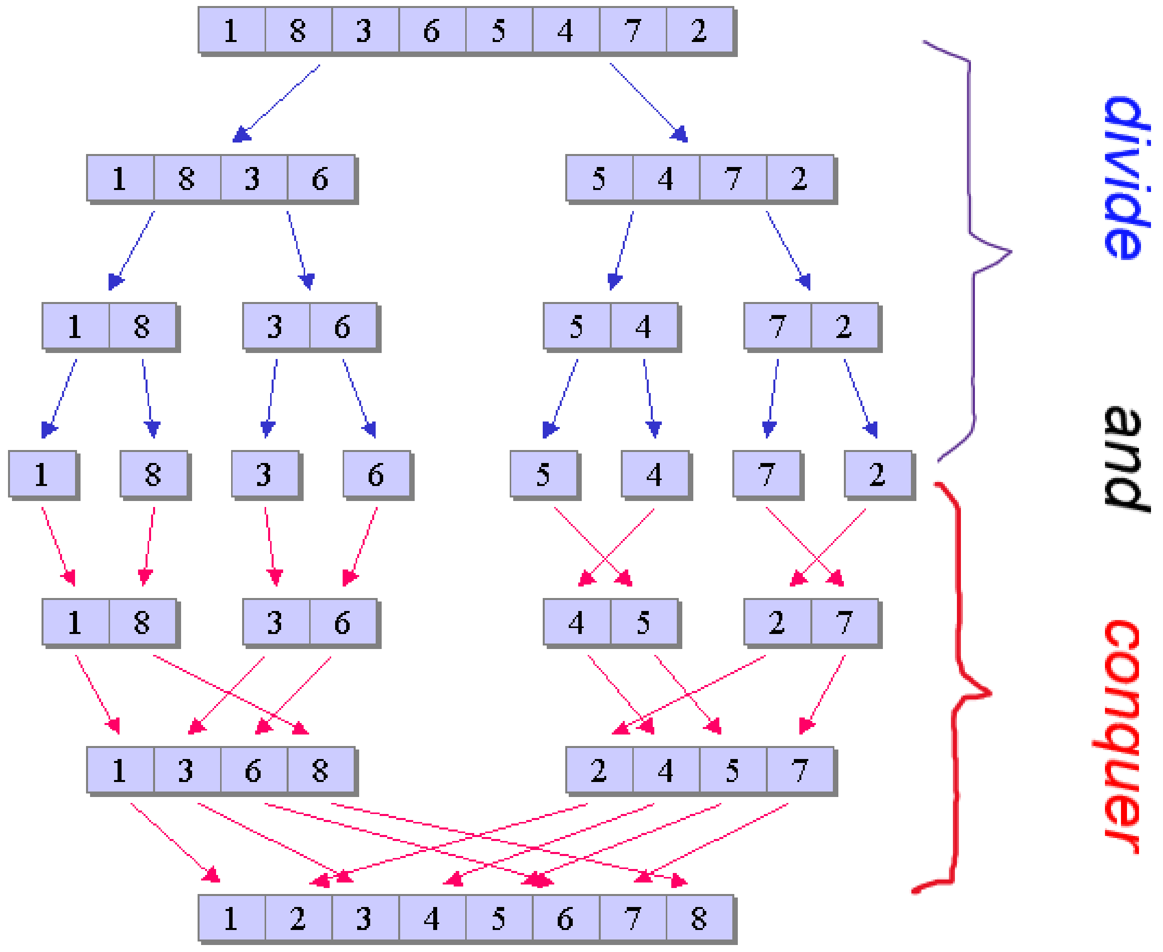
Base Cases:

- which values of parameter give a trivial or easy-found solution?
- what does the solution look like?

Divide-and-Conquer example: Top-Down MergeSort (on an array or a linked list)



Divide-and-Conquer example: Top-Down MergeSort (array-based)



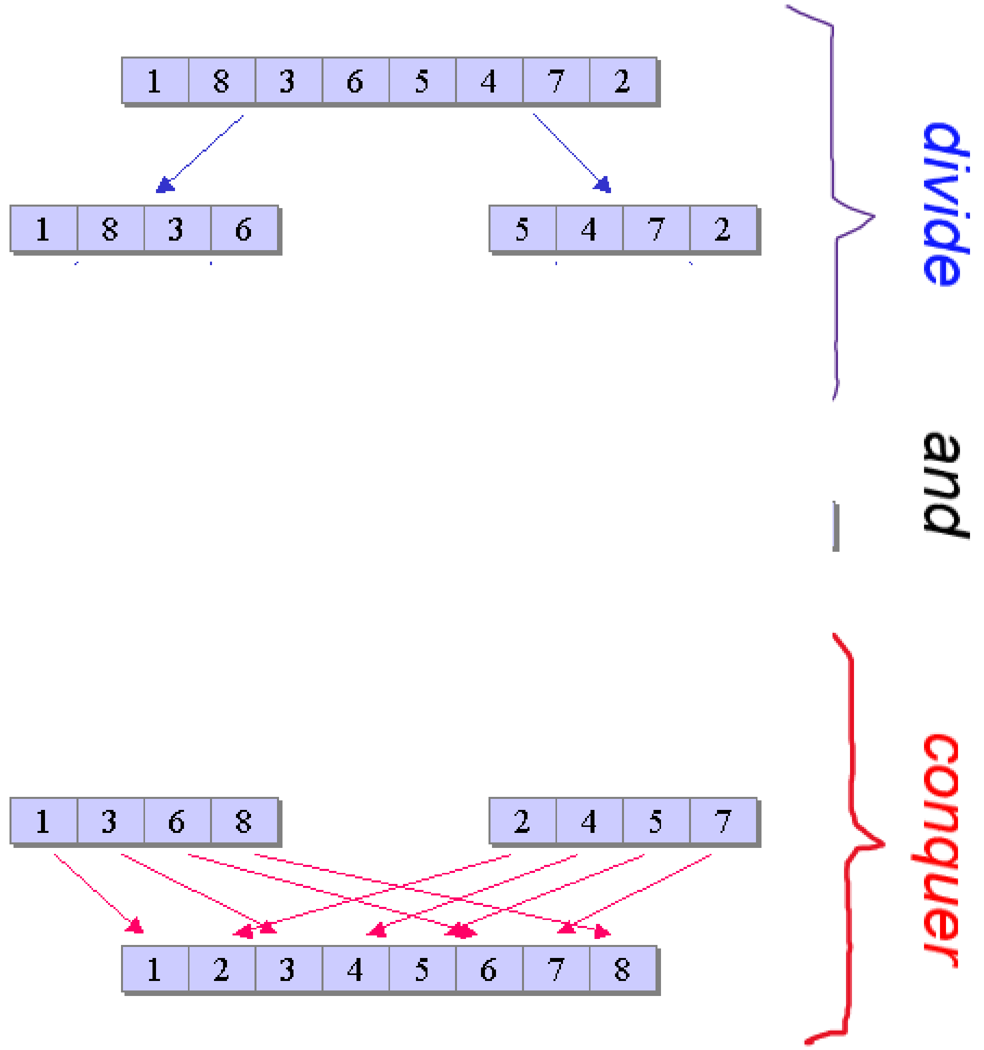
the sorting algorithm is :

- "*divide*":
    - break the array into 2 halves
    - ditto with sub-arrays until getting singleton arrays

and

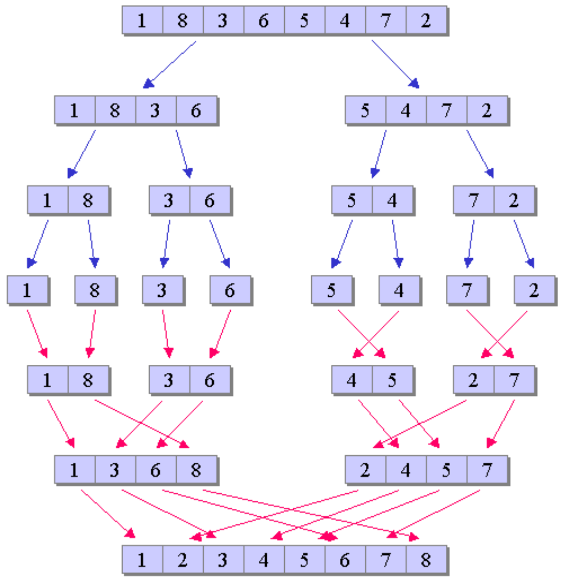
- "*conquer*":
    - merge 2 sorted arrays into one
    - ditto until getting a single (sorted) arrays

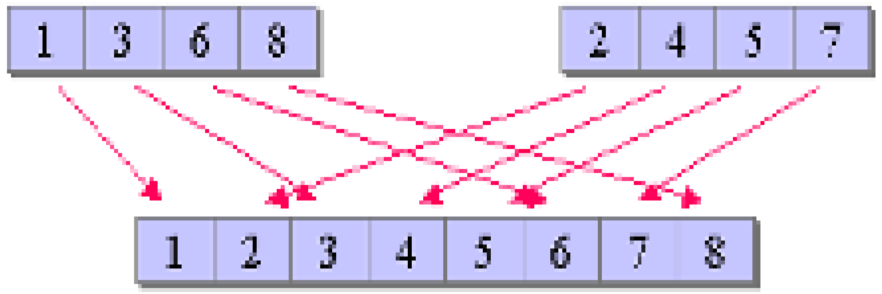
Divide-and-Conquer example: Top-Down MergeSort on Arrays



```
void MergeSort(int A[], int n) {
  if (n<=1) return;
  int mid= n/2;

  // array B[] = A[lo...mid];
  // array C[] = A[mid+1..hi];

  MergeSort(B, mid+1);
  MergeSort(C, n – mid);

  // merge sorted B[] and sorted C[] to A
  Merge(B, C, A);
}
```

How to merge 2 sorted arrays?





Merge(int B[], int nB, int C[], int nC, int A[]):

Input: two sorted sequences B and C(could be of different sizes)

Output: A= sorted union sequence B and C

Algorithm:

- Iterate B and C in parallel, and when doing so append the smaller value to A
- When B or C finishes, append the remainders of the other to A

	Merging 2 sorted arrays	Whole array-based mergesort algorithm
Time Complexity	*O(?)*	*O(?)*
Additional-Space Complexity	*O(?)*	*O(?)*

Generate & Test: ex9.7: The maximum subarray problem

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.

i :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A[i]:	2	1	3	-4	-5	2	6	1	-8	9	2	3	1	5	-7	1	-2

1. What is the problem? Give a function prototype.

??? max_subarray(???)

2. How to apply generate-and-test?

Generate & Test: ex9.7: The maximum subarray problem

```
  i  :   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
A[i]:    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 *lo, int *hi) {
  // generate all candidate subarray & check!






}
```

What's the complexity of the algorithm?

e9.7: The maximum subarray problem: approaches

Problem: Given int A[n], *find* lo *and* hi *so that the sum of elements from* lo *to* hi *is the maximal possible.*

Approach 1: Complexity= ?

Approach 2: any better, O(n) solution?

How do we do it manually?

```
   i :    0   1   2   3   4   5   6   7   8   9 10  11  12  13  14  15  16
A[i]:  2  1  3 -4 -5  2  6  1 -8  9  2  3  1  5 -7  1 -2
      -------
      ............  -------
```

e9.7: solution2 – simulate a smart manual method

```
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 *lo, int *hi) {




}
```

What’s the complexity of the algorithm?

Ex 9.3: Simulation (Monte-Carlo Method)?

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)
```

e9.3: let’s simulate a game

General Defs

```
#define FACES       13
#define SUITS        4
#define CARDS (FACES*SUITS)  /* number of cards */
#define PLAYERS      4
#define CARDSINHAND 5
const char *faces[] = {“Ac", “2", “3", “4", “5”, “6”,
                  “7", “8", “9", "10", ”Je", ”Qu", ”Ki"};
const char suits[] = {'S', 'C', 'D', 'H’};
```

Main Variables, Method 1

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

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

Main Variables, Method 2

```
// we have 52 different cards
// a card is just an int between 0 and 51 inclusively

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

a card value **i** is an int, which can be interpreted as:

- **i/4** defines the face of the card
- **i%4** defines the suit

How to simulate the games

Review: How to have random numbers

- rand() : generate a random non-negative integer
- rand()%n : generate a random integer in the range 0..n-1
- srand(k) : initializes the rand() function with a seed k
- srand(time(NULL)) : a good way to start the program before calling rand()

How to simulate the game:

- how to have a new card deck
- how to “shuffle” the deck?
- how to divide cards to players and counts pairs...

Thank You

&

Good Luck

Lab Time is Chat Time!

Any questions about all the topics covered in the course?

Implement:

9.3: poker statistics OR 9.7: maximal subarray

9.10: subset sum

OR: finish the permutation: print out all permutations of numbers from 1 to n

9.11: modeling a rocket flight [try this only if you remember well your high-school physics such as Newton's laws ☺]

After exam: *Possible follow-up subjects next year with algorithms and C implementation:*

sem1: `comp20007` – Design of Algorithms (DoA)

sem2: `comp20003` – Algorithms & Data Structures (ADS)

Additional Slides

Review: Recursive functions

Definition: A recursive function is a function that calls itself directly or indirectly.

Base case: A recursive function must have a base case that terminates the recursion.

Recursive case: The recursive case calls the function itself with a smaller input.

```
int factorial( int n ) {
  if (n <= 1) {
    return 1;
  }
  return n*factotial(n-1);
}
```

This function compute f(n)= n!, relying on the recurrence:

$$f(n) = \begin{cases} n * f(n-1) & if \quad n \geq 1 \\ 1 & if \quad otherwise \end{cases}$$

Good trick: start with an if statement to deal with the base case.

Reduce & Conquer: Recursion for reducing size n to n-1 (or smaller)

To apply recursion we need to identify

- Recursion: express the task of size n through *the same tasks*, but of smaller sizes
- Base Cases: identify the *base cases* where solutions are trivial or easy.

To write a recursive function, we normally (but not always):

- use if statements to deal with all the base cases first, then
- deal with the general cases.

Example 2: 5.13: int_pow or HanoiTower

the task

what's: relationship between the input size a smaller size?

what's: the base case?