Introduction to algorithms

University of Technology and Engineering
Vietnam National University Hanoi

Algorithms?

What is an algorithm?

An algorithm is a step by step procedure to solve a problem. An algorithm can be described by nature languages or programming languages.

Example: How to cook rice?

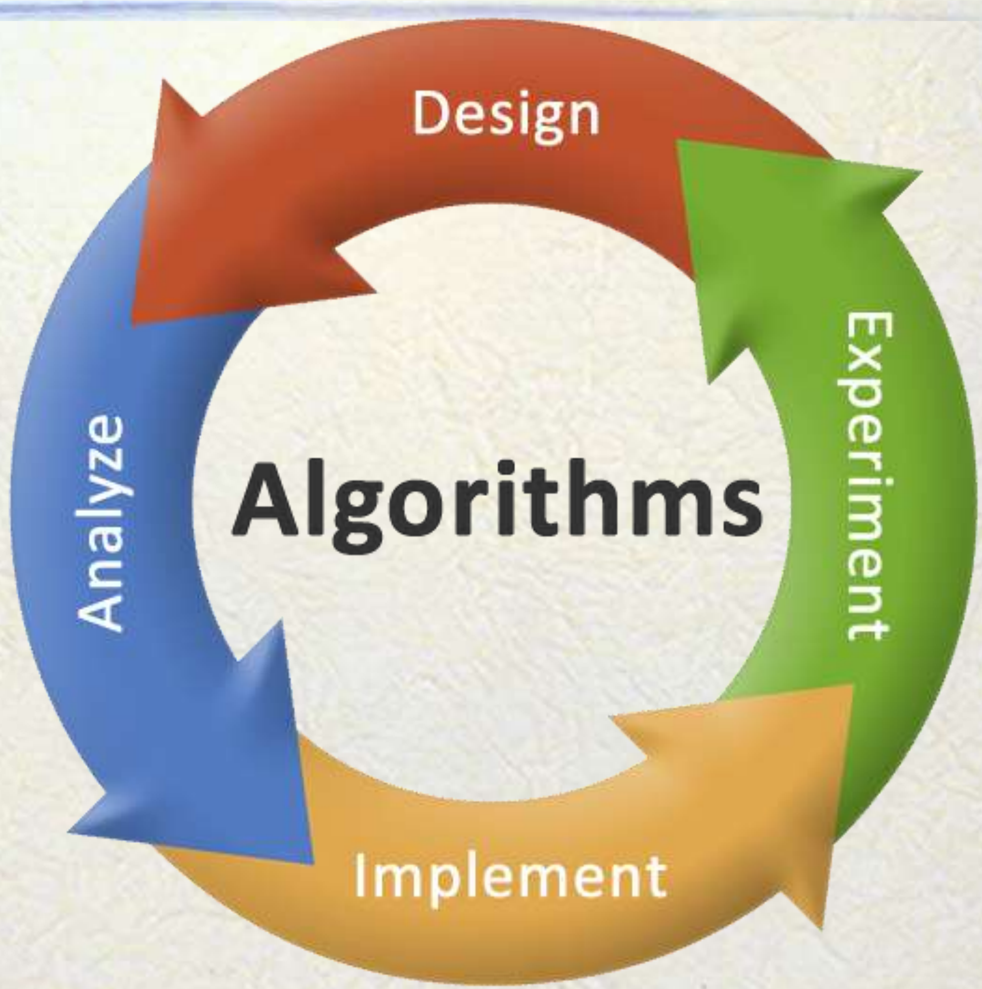
- ❖ Step 1: Get rice
- ❖ Step 2: Rinse the rice
- ❖ Step 3: Put the rice and water into a rice cooker
- ❖ Step 4: Turn on the rice cooker
- ❖ Step 5: Check the rice when the rice cooker turns off

Algorithms?

What is a good algorithm?

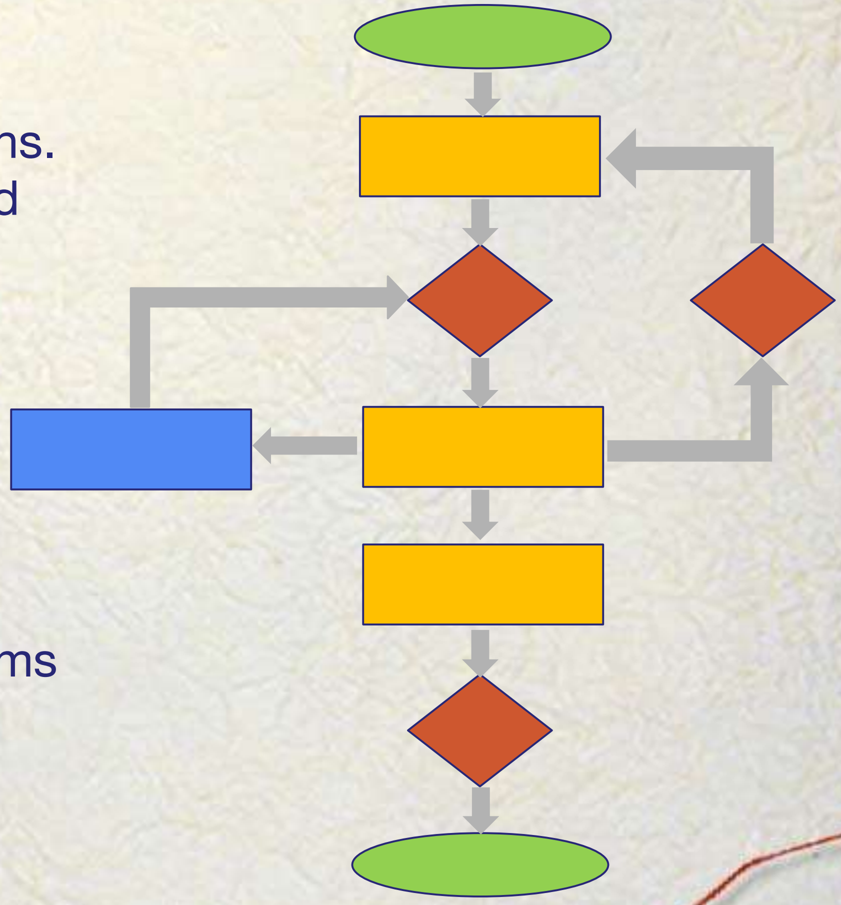
- Correctness
- Efficiency
- Simple/understandable
- Implementable

Algorithms



How to design algorithm?

- Step 1: Define the problem clearly
- Step 2: Analyze the problem from different perspectives and constrains.
- Step 3: Use popular techniques and strategies:
    - Basic algorithms
    - Brute force algorithms
    - Greedy algorithms
    - Divide and conquer
    - Dynamic programming
    - Graph theories
    - String/text processing algorithms
- Step 4: Use flow chart to represent algorithms



Searching problem

Problem: Given a list **A** consisting of *n* items. Check if an item X exists on list **A**?

Simple search algorithm: Iterate from the begin to the end of **A** to check if X equals to any item of A.

Complexity: O(n)

Sorting problem

Problem: Given a list **A** consisting of *n* items. Sort items of list **A** increasingly?

Example:

A = (1, 2, 5, 3, 8, 9, 2)

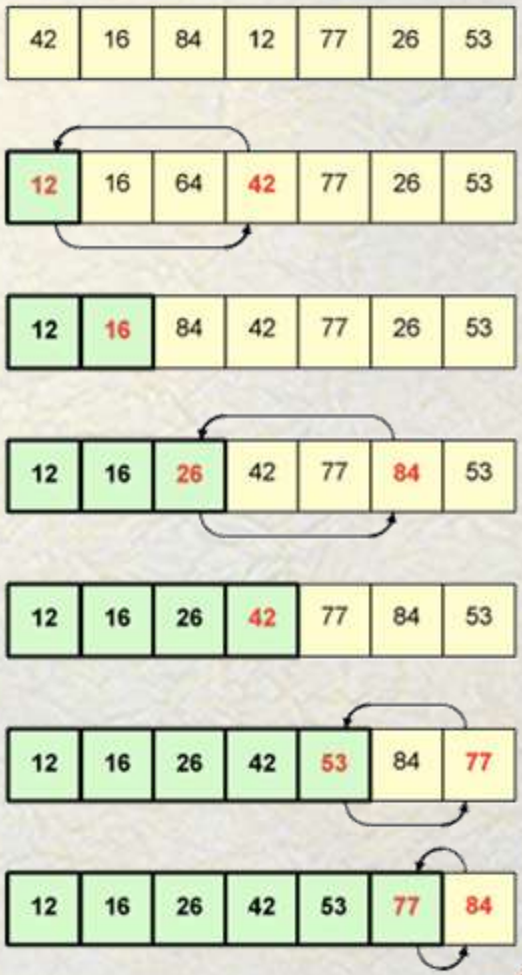
Sorting result:

A = (1, 2, 2, 3, 5, 8, 9)

Selection sort

Algorithm

- ➢ The list is divided into two parts: sorted part and unsorted part.
- ➢ Repeatedly finding the minimum element from unsorted part and moving it to the end of the sorted part.



Selection sort

```
Algorithm selection_sort (A, n):
    for iter from 0 to n - 2 do
        min_idx = iter;
        for idx from (iter + 1) to n - 1 do
            if A[idx] < A[min_idx] then
                min_idx = idx;
        swap (a[iter], a[min_idx]);
```

Complexity:

Searching on sorted list

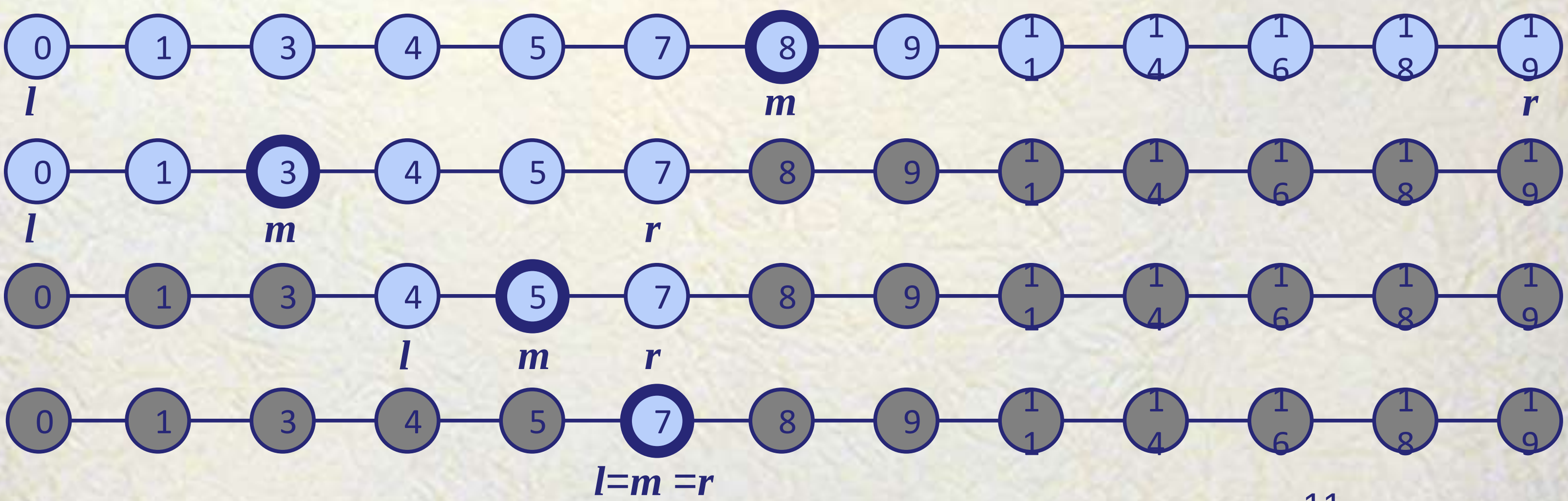
Problem: Given a list **A** consisting of *n* items sorted increasingly. Check if an item X exists on list **A**?

Binary search algorithm (A):

- ❖ If A is empty, return False
- ❖ Compare X with the item Y at the middle of **A**.
  - If X = Y, return True.
  - If X < Y, Perform binary search on the left half of **A**
  - If X > Y, Perform binary search on the right half of **A**

Complexity:

Example



0 1 3 4 5 7 8 9 11 14 16 18 19
l m r
0 1 3 4 5 7 8 9 11 14 16 18 19
l m r
0 1 3 4 5 7 8 9 11 14 16 18 19
l m r
0 1 3 4 5 7 8 9 11 14 16 18 19
l=m =r

Brute Force Search

Systematically evaluate all possible solutions for the problem to determine objective solutions.

Example:

- ❖ Find all prime numbers smaller than 100
- ❖ Find all binary numbers of length *n*
- ❖ Travel sale man problem

The largest row

Problem: Given a matrix A of *m* rows and *n* columns containing integer numbers. Your task is to find the row with the largest sum.

Method Find_largest_row (A, *m, n*):

Complexity:

The largest rectangle

Problem: Given a matrix of A of *m* rows and *n* columns containing integer numbers. Your task is to find the rectangle in the matrix with the largest sum.

Method Find_largest_rectangle (A, *m, n*):

Complexity:

Recursion

- **Recursion:** when a method calls itself
  Classic example (the factorial function):

  n! = n * (n-1)!

- Recursive definition:

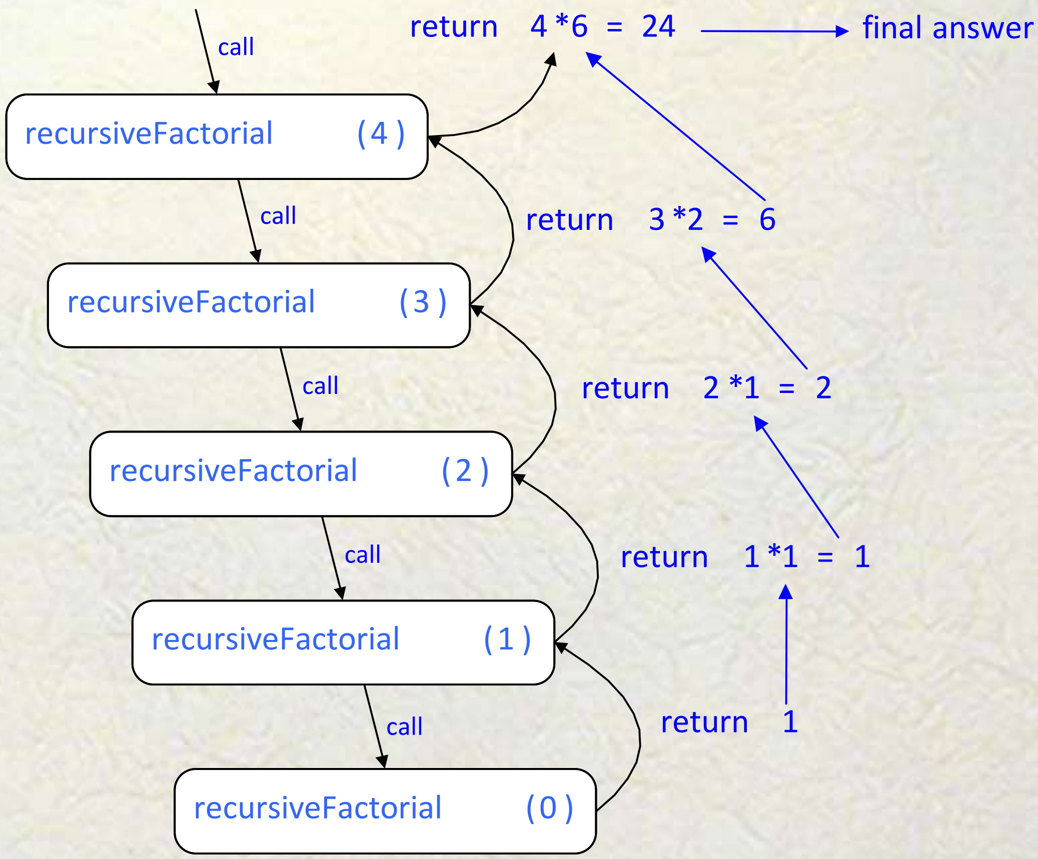
$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

```
Algorithm RecursiveFactorial (int n) {
        if (n == 0) return 1; // base case
        else return n * RecursiveFactorial (n - 1); // recursive case
}
```

Content of a Recursive Method

- **Base case(s).**
  - The cases for which we perform no recursive calls
  - Every possible chain of recursive calls **must** eventually reach a base case.
- ***Recursive calls.***
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion



Computing Powers

- The power function, p(*x, n*) = x^n, can be defined recursively:

$$p(x,n)=\begin{cases} 1 & \text{if } n=0 \\ x\cdot p(x,n-1) & \text{else} \end{cases}$$

- This leads to a power function that runs in O(n) time

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

- For example,

$$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$
$$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$
$$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$
$$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$$

A Recursive Squaring Method

```
Algorithm Power(x, n):
    Input: A number x and integer n = 0
    Output: The value x^n
  if n = 0        then
        return 1;
  if n is odd then
        y  = Power(x, (n - 1)/ 2);
        return x · y ·y;
  else
        y = Power(x, n/ 2);
        return y · y;
```

Analyzing the Recursive Squaring Method

Algorithm Power(x, n):
 Input: A number x and integer $n = 0$
 Output: The value x^n
 if $n = 0$ **then**
  return 1;
 if n is odd **then**
  $y = \text{Power}(x, (n - 1)/ 2)$;
  return $x \cdot y \cdot y$;
 else
  $y = \text{Power}(x, n/ 2)$;
  return $y \cdot y$;

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we used a variable twice here rather than calling the method twice.

Find all binary numbers

Problem: Find all binary numbers of length *n*.

Example: $n = 3$

000,001,010,011,100,101,110,111

Algorithm binary_number(b, *k, n*):

for v for 0 to 1 do

b[k] = v;

if (k==n) then

print b;

else

binary number (b, k + 1, n);

Complexity*:*

Find permutations

Problem: Find all permutations of length n.

Example: $n = 3$

123, 132, 213, 231, 312, 321

Algorithm: Find_permutations (n):

Complexity: