TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms Basic Lab

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

Chapter 1. Basic data types, I/O with files
Chapter 2. Recursion
Chapter 3. Lists
Chapter 4. Stack and Queue
Chapter 5. Trees
Chapter 6. Sorting
Chapter 7. Searching

2

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 2. Recursion

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

1. Recursion
2. Recursion with memorization
3. Backtracking

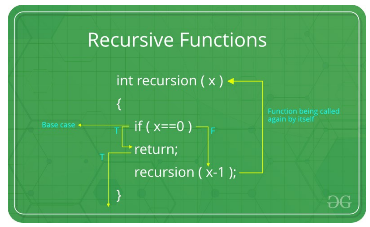
NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST 4

Contents

1. **Recursion**
2. Recursion with memorization
3. Backtracking

1. Recursion

- Recursive function
  - Call itself (with smaller input parameters)
  - Base case
    - Parameters are small so that the results are obtained easily
    - The function does not call itself



1. Recursion

- Recursive function
  - Call itself (with smaller input parameters)
  - Base case
    - Parameters are small so that the results are obtained easily
    - The function does not call itself

- $f(n) = 1 + 2 + \ldots + n$

Other form (Recursive form):

- $f(n) = \begin{cases} 1, \text{ if } n = 1 \\ f(n-1) + n, \text{ if } n > 1 \end{cases}$

```
#include <stdio.h>
int f(int n){
    if(n == 1) return 1;
    return n + f(n-1);
}
int main(){
    printf("%d\n",f(4));
}
```

1. Recursion

Recursive definition to calculate $n!$

$f(0) = 1$

$f(n) = n * f(n-1)$

To calculate the value of recursive function, we replace it gradually according to the recursive definition to obtain the expression with smaller and smaller arguments until we get the first condition.

For example:

recursive

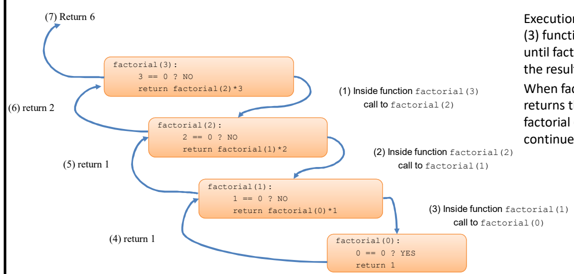
$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1!$
$= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$

First condition

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```

factorial(3);

```
int factorial(int n){
 if (n==0)
     return  1;
 else
     return n*factorial(n-1);
}
```



Execution of the factorial (3) function will stop until factorial (2) returns the result

When factorial (2) returns the result, the factorial (3) function continues to be executed

Example 1. Fibonacci sequence

- Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, …

$$f(n) = \begin{cases} 1, \text{ if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2), \text{ if } n > 1 \end{cases}$$

```
#include <stdio.h>
int f(int n){
    if(n <= 1) return 1;
    return f(n-1) + f(n-2);
}
int main(){
    for(int i = 0; i <= 10; i++)
        printf("%d ",f(i));
}
```

Example 2: Binary Search

Input: An array S consists of n elements: S[0],…,S[n-1] in ascending order; Value ***key*** with the same data type as array S.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

Binary search algorithm: The value *key* either

equals to the element at the middle of the array S,

or is at the left half (L) of the array S,

or is at the right half (R) of the array S.

(The situation L (R) happen only when *key* is smaller (larger) than the element at the middle of the array S)

```
int binsearch(int low, int high, int S[], int key)
```

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

key = 33

↑ low ↑ high

```
binsearch(0, 14, S, 33);
```


Example 2: Binary Search

Input: An array S consists of n elements: S[0],…,S[n-1] in ascending order; Value ***key*** with the same data type as array S.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo														hi

binsearch(0, 14, S, 33);

13

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	~~53~~	~~64~~	~~72~~	~~84~~	~~93~~	~~95~~	~~96~~	~~97~~
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo							mid							hi

binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);

The section to be investigated is halved after each iteration

14

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	~~53~~	~~64~~	~~72~~	~~84~~	~~93~~	~~95~~	~~96~~	~~97~~
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo						hi								

binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);

15

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

~~6~~	~~13~~	~~14~~	~~25~~	33	43	51	~~53~~	~~64~~	~~72~~	~~84~~	~~93~~	~~95~~	~~96~~	~~97~~
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo			mid			hi								

binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);

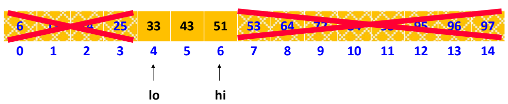
The section to be investigated is halved after each iteration

16

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < S[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
   }
   else return -1;
}
```

key=33



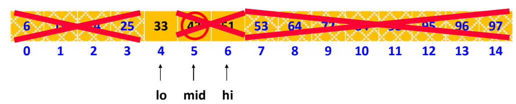
binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);

17

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < S[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
   }
   else return -1;
}
```

key=33



binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);
binsearch(4, 4, S, 33);

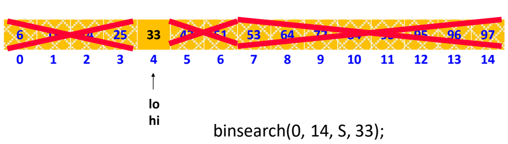
The section to be investigated is halved after each iteration

18

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < S[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
   }
   else return -1;
}
```

key=33



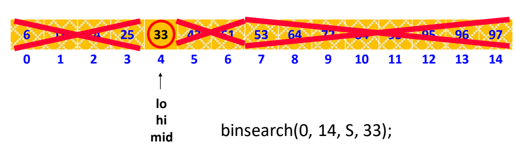
binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);
binsearch(4, 4, S, 33);

19

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < A[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
   }
   else return -1;
}
```

key=33



binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);
binsearch(4, 4, S, 33);

20

Example 3: Palindrome

- **Definition.** ***Palindrome*** is a string that reads it from left to right is the same as reading it from right to left.

Example: NOON, DEED, RADAR, MADAM

Able was I ere I saw Elba

- To determine if a given string is palindrome:
  - Compare the first character and the last character of the string.
    - If they are equal: new string = old string that removes the first and last characters. Repeat comparison step.
    - If they are different: given string is not palindrome

22

Example 3: Palindrome: str[start….end]

- Base case : string has <=1 character (`start >= end`)

```
return true
```

- Recursive step:

```
return true if (str[start]==str[end] &&
                palindrome(str, start+1, end-1))
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example 3. Recursion

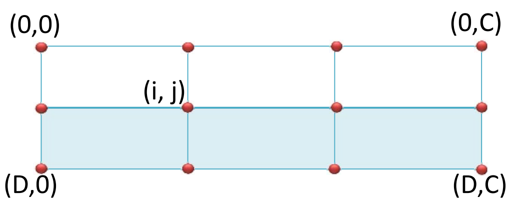
- How many ways to select *k* objects from *n* given objects

$$C(k,n) = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ C(k, n-1) + C(k-1, n-1) \text{ otherwise} \end{cases}$$

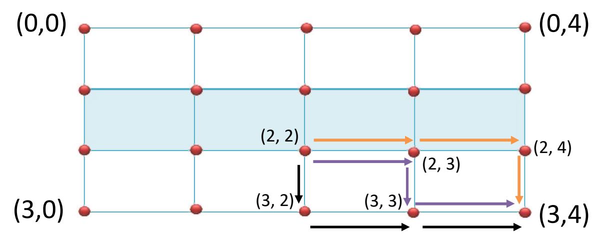
```
#include <stdio.h>
int C(int k, int n){
    if(k == 0 || k == n) return 1;
    return C(k,n-1) + C(k-1,n-1);
}
int main(){
    printf("%d ",C(3,5));
}
```

Example 4: Path on grid

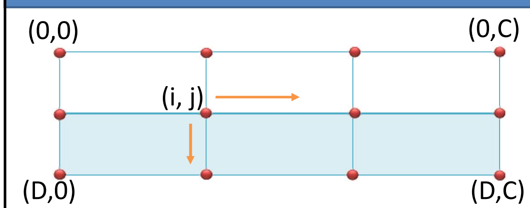
- Given the grid of size *D***C*.
- You are only allowed to move from one node to other node on the grid in either direction downwards or to the right.
- How many paths are there from node (*i*, *j*) to node (*D*, *C*).
  - Write function: `int CountPaths(int i, int j, int D, int C)`



Example 4: Path on grid



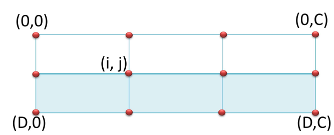
Example 5: Path on grid



- To solve the problem `CountPaths(i,j, D, C)`, we need to solve which subproblems ?
  - From node (*i*, *j*) there are only 2 ways:
    - Go down: to node (*i*+1, *j*) `CountPaths(i+1,j, D, C)`
    - Go right: to node (*i*, *j*+1) `CountPaths(i,j+1, D, C)`

➔`CountPaths(i,j, D, C) = CountPaths(i+1,j, D, C)`
`+ CountPaths(i,j+1, D, C)`

Basic case



- If you step over the edge of the grid (no longer on the grid), there is no path to the node (D, C):
  - `if (i > D) OR (j > C):CountPaths(i,j, D, C)` return 0

```
int CountPaths(int i, int j, int D, int C)
{
    if (i>D || j>C)  return 0;
    else
        return CountPaths(i + 1, j, D, C) +
               CountPaths(i, j + 1, D, C);
}
```

DONE ?

NO: because all subproblems will return 0

One more special case should be considered: when you are at line D or column C ➔ there is a path (this path does not need any steps)

28

Example 4: Path on grid

- Given the grid of size $D*C$.
- You are only allowed to move from one node to other node on the grid in either direction downwards or to the right.
- How many paths are there from node (i, j) to node (D, C).
  - Function: `int CountPaths(int i, int j, int D, int C)`

```
int CountPaths(int i, int j, int D, int C)
{
    if (i>D || j>C)  return 0;
    else if(i==D || j == C)
        return 1;
    else
        return CountPaths(i + 1, j, D, C) +
                CountPaths(i, j + 1, D, C);
}
```

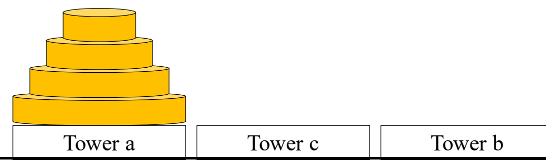
Exercise 1: Tower of Hanoi

The Tower of Hanoi, consists of three towers (a), (b), (c) together with n disks of different sizes. Initially these disks are stacked on the tower (a) in an ascending order, i.e. the smaller one sits over the larger one.

The objective of the game is to move all the disks from tower (a) to tower (c), following 3 rules:

- Only one disk can be moved at a time.
- Only the top disk can be moved
- No large disk can be sit over a smaller disk.

Let h_n denote the minimum number of moves needed to solve the Tower of Hanoi problem with n disks. What is the recurrence relation for h_n?

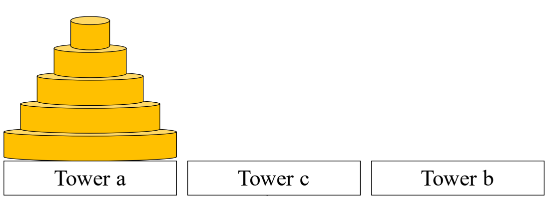


30

Tower of Hanoi: n=5

The objective of the game is to move all the disks from tower (a) to tower (c), following 3 rules:

1. Only one disk can be moved at a time.
2. Only the top disk can be moved
3. No large disk can be sit over a smaller disk.



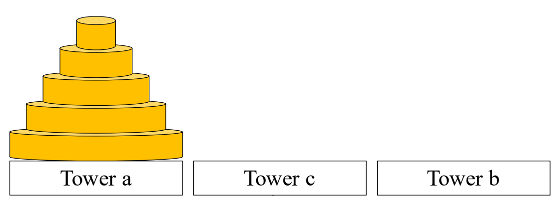
Exercise: Tower of Hanoi

- $h_1 = 1$
- For $n \geq 2$, we need to do 3 following steps to transfer all disks from tower (a) to tower (c):

  (1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

  (2) Move the largest disk to the tower (c)

  (3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c), placing them on top of the largest disk



Exercise 1: Tower of Hanoi

- $h_1 = 1$
- For $n \geq 2$, we need to do 3 following steps to transfer all disks from tower (a) to tower (c):

(1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

The problem of n-1 disks → #moves = h_{n-1}

(2) Move the largest disk to the tower (c)

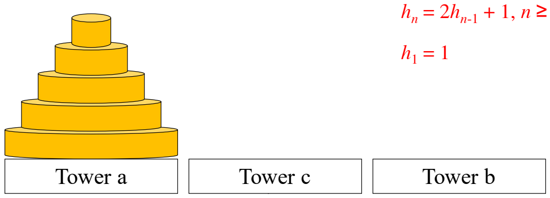
→ #moves = 1

(3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c), placing them on top of the largest disk

The problem of n-1 disks→ #moves = h_{n-1}

$h_n = 2h_{n-1} + 1, n \geq 2$

$h_1 = 1$



Tower of Hanoi: n=5

(1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

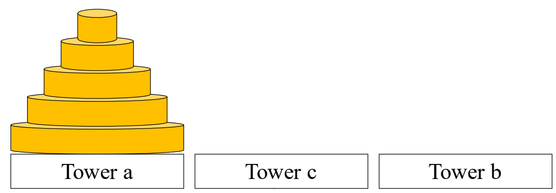
The problem of n-1 disks → #moves = h_{n-1}

(2) Move the largest disk to the tower (c)

→ #moves = 1

(3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c) , placing them on top of the largest disk

The problem of n-1 disks → #moves = h_{n-1}



Tower of Hanoi

The algorithm could be implemented as following:

//move n disks from tower a to tower c using tower b as an intermediary:

```
HanoiTower(n, a, c, b);
{
     if (n==1) then <move disk from tower a to tower c>
     else
     {
          HanoiTower(n-1,a,b,c);
          HanoiTower(1,a,c,b);
          HanoiTower(n-1,b,c,a);
     }
}
```

For $n \geq 2$, we need to do 3 following steps to transfer all disks from tower (a) to tower (c):

(1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

The problem of n-1 disks → #moves = h_{n-1}

(2) Move the largest disk to the tower (c)

→ #moves = 1

(3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c), placing them on top of the largest disk

The problem of n-1 disks→ #moves = h_{n-1}

Tower of Hanoi: Implementation

```
#include <bits/stdc++.h>
using namespace std;

void HanoiTower (int, char, char, char);
int i = 0;

int main()
{
    int n;
    cout<<" Input the number of disks = "; cin >>n;
    HanoiTower (n, 'a', 'c', 'b');
    cout <<"Total number of  disk movements = "<<i<<endl;
    return 0;
}

void HanoiTower (int n, char start, char finish, char spare)
{
    if (n == 1){
          cout<<" Move disk from tower "<<start<<" to tower "<<finish<<endl;
        i++;
        return;
    } else {
        HanoiTower (n-1, start, spare, finish);
        HanoiTower (1, start, finish, spare);
        HanoiTower (n-1, spare, finish, start);
    }
}
```

Exercise 1: Tower of Hanoi

- Let $T(n) = 2T(n-1) + 1$. What are the parameters?
  a =
  b =
  d =

  Therefore, which condition applies?

$$h_n = 2\, h_{n-1} + 1$$

Master Theorem

- Let $T(n)$ be a monotonically increasing function that satisfies
  $T(n) = a\,T(n/b) + f(n)$
  $T(1) = c$
  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Exercise 1: Tower of Hanoi

Backward substitution: this works exactly as its name suggests. Starting from the equation itself, work backwards, substituting values of the function for previous ones

$T(n) = 2\,T(n-1) + 1$

$= 2\,(2\,T(n-2) + 1) + 1 \quad = 2^2\,T(n-2) + 2 + 1$

$= 2^2(2\,T(n-3) + 1) + 2 + 1 \quad = 2^3\,T(n-3) + 2^2 + 2 + 1$

…

$= 2^{n-1}\,T(1) + 2^{n-2} + \ldots + 2 + 1$

$= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1$ (because $T(1) = 1$)

$= 2^n - 1$

➔ Time complexity: $O(2^n)$ which is exponential

Exercise 2

- Given set S consisting of n positive integer numbers, and a value `sum`. Determine whether or not there exists a subset S' of S that satisfies: the sum of all elements in S' is equal to = `sum`.

Example: S = {3, 34, 4, 12, 5, 2}, `sum` = 9

➔ Output: True //because there exist subset S' ={4, 5} in which sum of all elements of S' is equal to = 9.

Write recursive function:

```
int isSubsetSum(int S[], int n, int sum)
```


Exercise 2

- Given set S consisting of n positive integer numbers, and a value `sum`. Determine whether or not there exists a subset S' of S that satisfies: the sum of all elements in S' is equal to = `sum`.

Example: S = {3, 34, 4, 12, 5, 2}, `sum` = 9

➔ Output: True //because there exist subset S' ={4, 5} in which sum of all elements of S' is equal to = 9.

RECURSION: `int isSubsetSum(int S[], int n, int sum)`

There are 2 cases:

- Subset S' **consists of** the last element of set S

```
return isSubsetSum(S, n-1, sum - S[n-1])
```

- Subset S' **does not consists of** the last element of S

```
return isSubsetSum(S, n-1, sum)
```

Base case ???
```
1) n == 0 and sum > 0   ➔ return false
2) sum == 0             ➔ return true
```

Exercise 3: Calculate the binomial coefficient

- The binomial coefficient $C(n,k)$ is defined recursively as following:

$$C(n,0) = 1,\ C(n,n) = 1;\quad \text{where } n >= 0,$$

$$C(n,k) = C(n-1,k-1)+C(n-1,k),\quad \text{where } 0 < k < n$$

- Recursive implementation on C:

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```


Contents

2. Recursion with memorization

- In the previous section, we see that recursive algorithms for calculating Fibonacci numbers and calculating binomial coefficients were inefficient. To increase the efficiency of recursive algorithms without having to build iterative or recursive reduction procedures, we can use “recursion with memorization” technique.
- Using this technique, in many cases, we maintain the recursive structure of the algorithm and at the same time ensure its effectiveness. The biggest downside to this approach is the memory requirement.

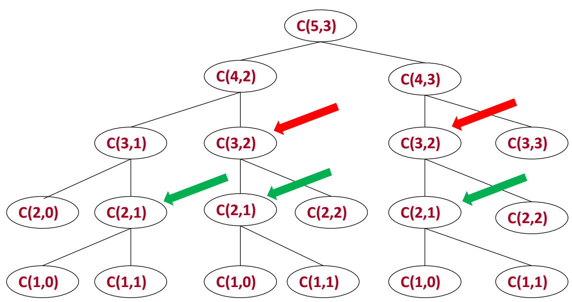
Duplication of subproblems

- Realizing that in recursive algorithms, whenever we need the solution of a subproblem, we must solve it recursively. Therefore, there are subproblems that are solved repeatedly. That leads to inefficiency of the algorithm. This phenomenon is called duplication of subproblem.

Example: Recursive algorithm to calculate C(5,3). The recursive tree that executes the call to function C(5,3) is shown in the following:

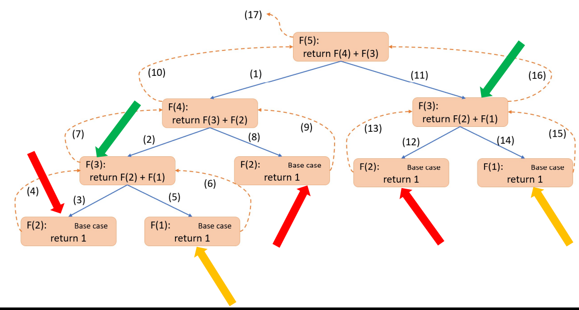
Example: Duplication of subproblems when calculating C(5,3)

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```



Example: Duplication of subproblems when calculating Fibonacci F(5)

```
int F (int n) {
  if  (n < 2) return n;
  else return F(n-1)+F(n-2);
}
```



Recursive with memorization

- To overcome this phenomenon, the idea of ***recursive with memorization*** is: We will use the variable to memorize information about the solution of subproblem right after the first time it is solved. This allows to shorten the computation time of the algorithm, because, whenever needed, it can be looked up without having to solve the subproblems that have been solved before.

Example: Recursive algorithm calculates binomial coefficients, we put a variable

- D[n][k] to record calculated value of C(n, k).
- Initially D[n][k]=0, when C(n, k) is calculated, this value will be stored in D[n][k]. Therefore, if D[n][k]>0 then it means there is no need to recursively call function C(n, k)

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```

Example: Recursive with memorization to calculate C(n,k)

```
#include <stdio.h>
#include <string.h>
#define MAX 100
int D[MAX][MAX];//D[n][k] stores the value of C(n,k)
int C(int n,int k){
    if(k == 0 || k == n) D[n][k] = 1;
    else if(D[n][k] == 0)
      M[n][k] = C(n-1,k-1) + C(n-1,k);
    return D[n][k];
}
int main(){
    memset(D,0,sizeof(D));
    printf("%d ",C(5,3));
}
```

Before calling function C(n, k), we need to initialize array D[ ][ ] :
D[i][j] = 0, where i, j = 0, 1,…,n

```
int C(int n,int k){
   if (D[n][k]>0)  return D[n][k];
   else{
      D[n][k] = C(n-1,k-1)+C(n-1,k);
      return D[n][k];
   }
}
```

Before calling function C(n, k), we need to initialize array D[ ][ ] :

- D[i][0] = 1, D[i][n]=1, where i = 0,1,…, n;
- D[i][j] = 0, for remaining values of i, j

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```

Test computation time

```
int C(int n, int k){
    if ((k==0)||(k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}
```

Recursive **without** using memorization

- $n = 30$; $k = 20$
- $n = 40$; $k = 30$
- $n = 50$; $k = 40$
- $n = 60$; $k = 50$

```
int C(int n,int k){
    if (D[n][k]>0)  return D[n][k];
    else{
        D[n][k] = C(n-1,k-1)+C(n-1,k);
        return D[n][k];
    }
}
```

Before calling function C(n, k), we need to initialize array D[ ][ ] :

- D[i][0] = 1, D[i][n]=1, where i = 0,1,..., n;
- D[i][j] = 0, for remaining values of i, j

Recursive using memorization

Contents

1. Recursion
2. Recursion with memorization
3. **Backtracking**

Backtracking (Thuật toán quay lui)

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length n
- Generate m-element subsets of the set of n elements
- Generate permutations of n elements

Backtracking

Used to solve enumeration problem:

- ***Enumeration problem (Q):*** *Given $A_1, A_2,..., A_n$ be finite sets. Denote*

  $A = A_1 \times A_2 \times ... \times A_n = \{ (a_1, a_2, ..., a_n): a_i \in A_i , i=1, 2, ..., n\}$.

  Assume P is a property on the set A. The problem is to enumerate all elements of the set A that satisfies the property P:

  $D = \{ a = (a_1, a_2, ..., a_n) \in A: a \text{ satisfy property } P \}$.

- Elements of the set D are called **feasible solution** (*lời giải chấp nhận được).*

Backtracking diagram

All basic combinatorial enumeration problem could be rephrased in the form of Enumeration problem (Q).

Example:

- The problem of enumerating all binary string of length n leads to the enumeration of elements of the set:

  $B^n = \{(a_1, ..., a_n): a_i \in \{0, 1\}, i=1, 2, ..., n\}$.

- The problem of enumerating all m-element subsets of set $N = \{1, 2, ..., n\}$ requires to enumerate elements of the set:

  $S(m,n) = \{(a_1,..., a_m) \in N^m: 1 \le a_1 < ... < a_m \le n\}$.

- The problem of enumerating all permutations of natural numbers 1, 2, ..., n requires to enumerate elements of the set

  $\Pi_n = \{(a_1,..., a_n) \in N^n: a_i \neq a_j ; i \neq j\}$.

Partial solution (Lời giải bộ phận)

The solution to the problem is an ordered tuple of n elements $(a_1, a_2, ..., a_n)$, where $a_i \in A_i$, $i = 1, 2, ..., n$.

Definition. The k-level partial solution $(0 \le k \le n)$ is an ordered tuple of k elements

$$(a_1, a_2, ..., a_k),$$

where $a_i \in A_i$, $i = 1, 2, ..., k$.

- When $k = 0$, 0-level partial solution is denoted as ( ), and called as the empty solution.
- When $k = n$, we have a complete solution to a problem.

Backtracking diagram

Backtracking algorithm is built based on the construction each component of solution one by one.

- Algorithm starts with empty solution ( ).
- Based on the property P, we determine which elements of set A_1 could be selected as the first component of solution. Such elements are called as **candidates** for the first component of solution. Denote candidates for the first component of solution as S_1. Take an element $a_1 \in S_1$, insert it into empty solution, we obtain 1-level partial solution: (a_1).

- ***Enumeration problem (Q):*** *Given $A_1, A_2,..., A_n$ be finite sets. Denote*

  $A = A_1 \times A_2 \times ... \times A_n = \{(a_1, a_2, ..., a_n): a_i \in A_i, i=1, 2, ..., n\}$.

  Assume P is a property on the set A. The problem is to enumerate all elements of the set A that satisfies the property P:

  $D = \{a = (a_1, a_2, ..., a_n) \in A: a \text{ satisfy property } P\}$.

Backtracking diagram

- General step: Assume we have k-1 level partial solution:

  $$(a_1, a_2, ..., a_{k-1}),$$

Now we need to build k-level partial solution:

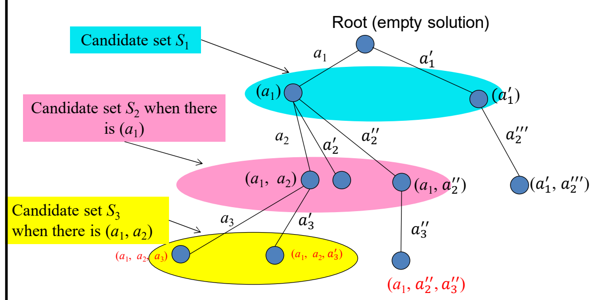
$$(a_1, a_2, ..., a_{k-1}, a_k)$$

- Based on the property P, we determine which elements of set A_k could be selected as the k^{th} component of solution.
- Such elements are called as candidates for the k^{th} position of solution when k-1 first components have been chosen as $(a_1, a_2, ..., a_{k-1})$. Denote these candidates by S_k.
- Consider 2 cases:
  - $S_k \neq \varnothing$
  - $S_k = \varnothing$

Backtracking diagram

- ***$S_k \neq \varnothing$:*** Take $a_k \in S_k$ to insert it into current (k-1)-level partial solution (a_1, a_2, ..., a_{k-1}), we obtain k-level partial solution (a_1, a_2, ..., a_{k-1}, a_k). Then
  - If $k = n$, then we obtain a complete solution to the problem,
  - If $k < n$, we continue to build the $(k+1)^{th}$ component of solution.
- ***$S_k = \varnothing$:*** It means the partial solution (a_1, a_2, ..., a_{k-1}) can not continue to develop into the complete solution. In this case, we **backtrack** to find new candidate for $(k\text{-}1)^{th}$ position of solution (note: this new candidate must be an element of S_{k-1})
  - If one could find such candidate, we insert it into $(k\text{-}1)^{th}$ position, then continue to build the k^{th} component.
  - If such candidate could not be found, we backtrack one more step to find new candidate for $(k\text{-}2)^{th}$ position,… If backtrack till the empty solution, we still can not find new candidate for 1[st] position, then the algorithm is finished.

Decision tree for backtracking



Backtracking algorithm (recursive)	Backtracking algorithm (not recursive)

```
void Try(int k)
{
   <Build Sk as the set to consist of candidates for the kth
      component of solution>;
  for y ∈ Sk  //Each candidate y of Sk
  {
      ak = y;
      [maybe: update values of variables]
      if (k == n) then <Record (a1, a2, ..., ak) as a
      complete solution >;
      else Try(k+1);
      [maybe: return the variables to their old values]
  }
}
```

The call to execute backtracking algorithm: Try(1);

- If only one solution need to be found, then it is necessary to find a way to terminate the nested recursive calls generated by the call to Try(1) once the first solution has just been recorded.
- If at the end of the algorithm, we obtain no solution, it means that the problem does not have any solution.

```
void Backtracking ( )
{
 k=1;
 <Build Sk>;
 while (k > 0) {
    while (Sk ≠ ∅ ) {
       ak ← Sk; // Take ak from Sk
       if (k == n)> then <Record (a1,a2,...,ak) as a
                             complete solution >;
       else {
            k = k+1;
            <Build Sk>;
       }
    }
     k = k - 1;   // Backtracking
  }
}
```

The call to execute backtracking algorithm: Backtracking ( );

Two key issues

- In order to implement a backtracking algorithm to solve a specific combinatorial problems, we need to solve the following two basic problems:
  - Find algorithm to build candidate set S_k
  - Find a way to describe these sets so that you can implement the operation to enumerate all their elements (implement the loop `for y ∈ Sk`).
    - The efficiency of the enumeration algorithm depends on whether we can accurately identify these candidate sets.

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Note

- If the length of complete solution is not known in advanced, and solutions are not needed to have the same length:

```
void Try(int k)
{
  <Build S_k as the set to consist of candidates for the k^th component of solution>;
  for y ∈ S_k  //Each candidate y of S_k
  {
    a_k = y;
    if  (k == n) then <Record (a_1, a_2, ..., a_k) as a complete solution >;
    else Try(k+1);
    Return the variables to their old states;
  }
}
```

- Then we need to modify statement

  if ($k == n$) then <Record (a_1, a_2, ..., a_k) as a complete solution >;
  else Try(k+1);

  to

  if <(a_1, a_2, ..., a_k) is a complete solution> then <Record (a_1, a_2, ..., a_k) as a complete solution >;
  else Try(k+1);

→ Need to build a function to check whether (a_1, a_2, ..., a_k) is the complete solution.

Backtracking (Thuật toán quay lui)

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

– **Generate binary strings of length *n***

– Generate *m*-element subsets of the set of *n* elements

– Generate permutations of *n* elements

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example 1: Enumerate all binary string of length *n*

- Problem to enumerate all binary string of length n leads to the enumeration of all elements of the set:

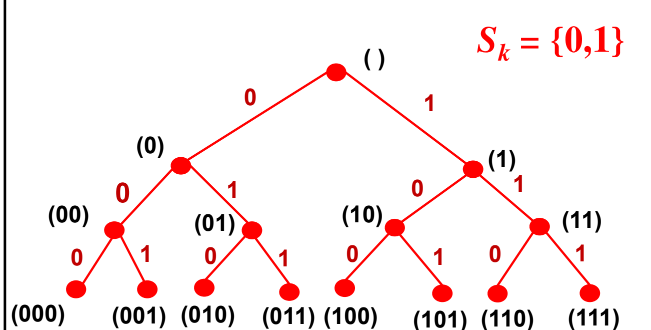
  $A^n = \{(a_1, ..., a_n): a_i \in \{0, 1\}, i=1, 2, ..., n\}$.

- We consider how to solve two issue keys to implement backtracking algorithm:
  - Build candidate set S_k: We have $S_1 = \{0, 1\}$. Assume we have binary string of length k-1 (a_1, ..., a_{k-1}), then $S_k = \{0,1\}$. Thus, the candidate sets for each position of the solution are determined.
  - Implement the loop to enumerate all elements of S_k: we can use the loop `for`

```
for (y=0; y<=1; y++) in C/C++
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Decision tree to enumerate binary strings of length 3



NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

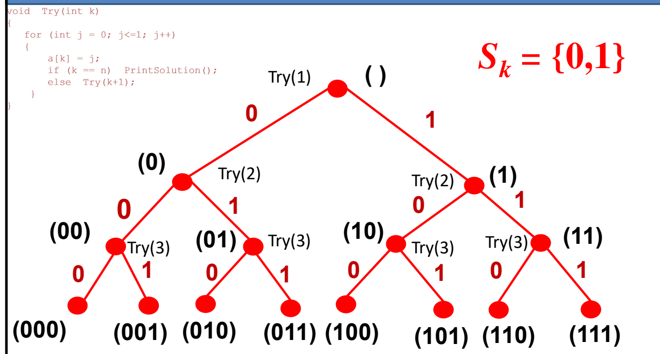
Program in C (Recursive)

```c
#include <stdio.h>
int  n, cnt;
int a[100];

void PrintSolution()
{
  int i;
  cnt++;
  printf("String # %d: ",cnt);
  for  (i=1 ; i<= n ;i++)
    printf("%d   ",a[i]);
  printf("\n");
}
```

```c
void  Try(int k)
{
   for (int j = 0; j<=1; j++)
   {
       a[k] = j;
       if (k == n)  PrintSolution();
       else  Try(k+1);
    }
}

int main()
{
   printf("Enter n = "); scanf("%d",&n);
   cnt = 0;
   Try(1);
   printf("Number of strings %d",cnt);
}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Decision tree to enumerate binary strings of length 3



NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Program in C (non recursive)

```c
#include <stdio.h>
int  n, cnt, k;
int a[100], s[100];

void PrintSolution()
{
  int i;
  cnt++;
  printf("String # %d: ",cnt);
  for  (i=1 ; i<= n ;i++)
    printf("%d   ",a[i]);
  printf("\n");
}
```

```c
void  GenerateString( )
{
     k=1;  s[k]=0;
     while (k > 0)
     {
           while (s[k] <= 1)
           {
                  a[k]=s[k];
                  s[k]=s[k]+1;
                  if(k==n)  PrintSolution();
                  else
                  {
                     k++; s[k]=0;
                  }
            }
            k--;   // BackTrack
      }
}
```

Program in C (non recursive)

```c
int main()
{
   printf("Enter n = "); scanf("%d",&n);
   cnt = 0;
   GenerateString();
   printf("Number of strings %d",cnt);
}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Backtracking (Thuật toán quay lui)

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- Generate *m*-element subsets of the set of *n* elements
- Generate permutations of *n* elements

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example 2. Generate *m*-element subsets of the set of *n* elements

Problem: Enumerate all *m*-element subsets of the set *n* elements $N = \{1, 2, ..., n\}$.

Example: Enumerate all 3-element subsets of the set 5 elements N = {1, 2, 3, 4, 5}

Solution: (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)

➔ Equivalent problem: Enumerate all elements of set:

$$S(m,n)=\{(a_1,..., a_m)\in N^m: 1 \le a_1<...<a_m\le n\}$$

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example 2. Generate *m*-element subsets of the set of *n* elements

We consider how to solve two issue keys to implement backtracking:

- Build candidate set S_k:
  - With the condition: : $1 \le a_1 < a_2 < ... < a_m \le n$

  we have $S_1 = \{1, 2, ..., n-(m-1)\}$.
  - Assume the current subset is $(a_1, ..., a_{k-1})$, with the condition $a_{k-1} < a_k < . . . < a_m \le n$, we have $S_k = \{a_{k-1}+1, a_{k-1}+2, ..., n-(m-k)\}$.
- Implement the loop to enumerate all elements of S_k: we can use the loop `for`

```
for (j=a[k-1]+1;j<=n-m+k;j++)
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Program in C (Recursive)

```
#include <stdio.h>

int  n, m, cnt;
int a[100];

void PrintSolution() {
    int i;
    count++;
    printf("The subset #%d: ,cnt);
    for  (i=1 ; i<= m ;i++)
       printf("%d ",a[i]);
    printf("\n");
}
```

```
void  Try(int k){
   int j;
   for (j = a[k-1] +1; j<= n-m+k; j++) {
         a[k] = j;
         if (k==m)  PrintSolution();
         else Try(k+1);
    }
}
int main() {
   printf("Enter n, m = ");
   scanf("%d %d",&n, &m);
   a[0]=0; cnt = 0; Try(1);
   printf("Number of %d-element subsets of set
   %d elements = %d \n",m, n, cnt);
}
```

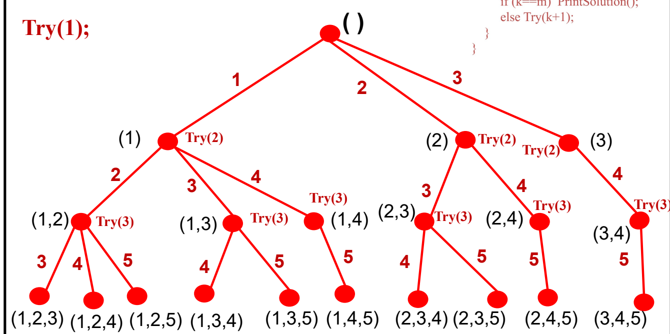
Program in C (Non Recursive)

```
#include <stdio.h>

int  n, m, cnt,k;
int a[100], s[100];

void PrintSolution() {
  int i;cnt++;
    printf("The subset #%d: ,cnt);
    for (i=1 ; i<= m ;i++)
       printf("%d ",a[i]);
    printf("\n");
}
```

```
void  MSet()
{
    k=1;  s[k]=1;
    while(k>0){
        while (s[k]<= n-m+k) {
             a[k]=s[k];  s[k]=s[k]+1;
             if (k==m) PrintSolution();
             else { k++; s[k]=a[k-1]+1;  }
        }
        k--;
    }
}

int main() {
  printf("Enter n, m = ");
  scanf("%d %d",&n, &m);
  a[0]=0; cnt = 0; MSet();
  printf("Number of %d-element subsets of set
   %d elements = %d \n",m, n, cnt);
}
```

Decision tree S(5,3)

```
void  Try(int k){
  int j;
  for (j = a[k-1] +1; j<= n-m+k; j++) {
      a[k] = j;
      if (k==m)  PrintSolution();
      else Try(k+1);
   }
}
```



$S_k = \{a_{k-1}+1, a_{k-1}+2, ..., n-(m-k)\}$

Backtracking (Thuật toán quay lui)

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length n
- Generate m-element subsets of the set of n elements
- **Generate permutations of n elements**

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Example 3. Enumerate permutations

Permutation set of natural numbers 1, 2, ..., n is the set:

$$\Pi_n = \{(a_1,..., a_n) \in N^n: \ a_i \neq a_j , i \neq j \}.$$

Problem: Enumerate all elements of Π_n

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example 3. Enumerate permutations

- Build candidate set S_k:
  - Actually $S_1 = N$. Assume we have current partial permutation (a_1, a_2, ..., a_{k-1}), with the condition $a_i \neq a_j$, for all $i \neq j$, we have

    $S_k = N \setminus \{ a_1, a_2, ..., a_{k-1} \}$.

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Describe S_k

Build function to detect candidates:

```
int check(int j, int k)
{
  //function returns true if and only if j ∈ Sk
   int i;
   for (i=1;i++;i<=k-1)
       if (j == a[i]) return 0;
   return 1;
}
```

Example 3. Enumerate permutations

- Implement the loop to enumerate all elements of S_k:

```
for (j=1; j <= n; j++;)
   if  (check(j, k))
   {
     // j is candidate for position kth
     . . .
   }
```

Program in C (Recursive)

```
#include <stdio.h>

int  n, m, cnt;
int a[100];

int PrintSolution() {
    int i, j;
    cnt++;
    printf("Permutation #%d: ",cnt);
    for  (i=1 ; i<= n ;i++)
        printf("%d  ",a[i]);
    printf("\n");
}
```

```
int check(int j, int k)
{
    int i;
    for (i=1; i<=k-1; i++)
       if (j == a[i])
              return 0;
    return 1;
}
```

Program in C (Recursive)

```
void  Try(int k)
{
   int j;
   for (j = 1; j<=n; j++)
      if (check(j,k))
      {  a[k] = j;
         if (k==n)  PrintSolution( );
         else  Try(k+1);
      }
}
int main() {
    printf("Enter n = ");scanf("%d",&n);
    cnt = 0; Try(1);
    printf("Number of permutations = %d",cnt);
}
```

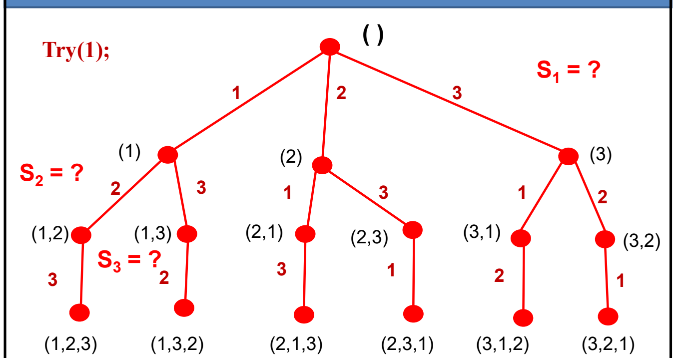
Example 3. Enumerate permutations (using marking array)

- Used an additional array to mark whether a value is used: `used[j] =0` if value j is not used to assign to any element a[k] for k = 1,..n; otherwise, `used[j]=1`

```
#include <stdio.h>
#define MAX 100
int n;
int a[MAX], used[MAX];
void PrintSolution(){
    int i;
    for(i = 1; i <= n; i++)
        printf("%d ",x[i]);
    printf("\n");
}
```

```
void Try(int k){
    int j;
    for(j = 1; j <= n; i++){
        if(used[j]==0){// i has not been used
            a[k] = j;
            used[j] = 1;// update used
            if(k == n) PrintSolution();
            else Try(k+1);
            used[j] = 0 ;// recover
        }
    }
}
int main(){
    n = 3; memset(used,0,sizeof(used));
    Try(1);
}
```

Decision tree to enumerate permutations of 1, 2, 3



$$S_k = N \setminus \{ a_1, a_2, \ldots, a_{k-1} \}$$

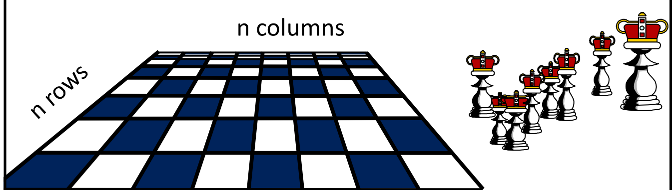
Exercise 1

Generate all binary sequences of length n not containing '11'

Exercise 2. The n-Queens problem

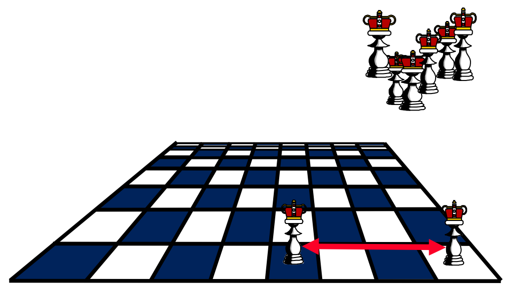
Enumerate all possibilities to place n queens on a chess board nxn such that no two queens attack to each other (no two queens on the same row, same column, same diagonal of the chessboard.

Constraint P



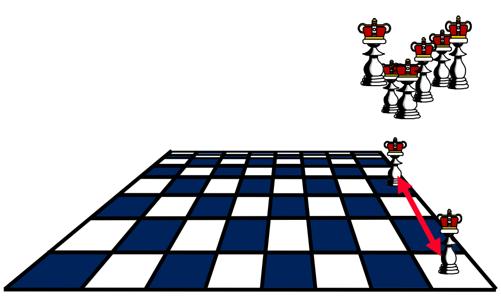
The n-Queens Problem

No two queens on the same row..



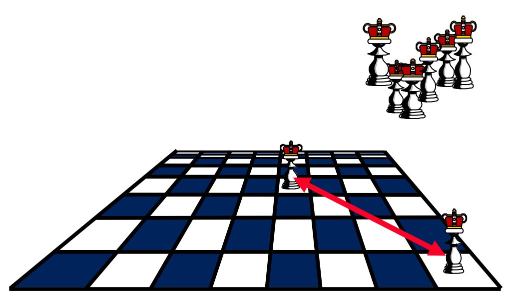
The n-Queens Problem

No two queens on the same column...



The n-Queens Problem

No two queens on the same diagonal...



The n-Queens Problem: Brute force

- Need to arrange n queens on a chess board of size nxn cells, how many ways are there?
  - The 1st queen could be placed on one of $n*n$ cells → there are n^2 ways
  - After placing the 1st queen on the chess board, we need to find a cell to place the 2nd queen: skip the cell where the 1st queen was places → there are only $n^2 - 1$ cells left to place the 2nd queen → there are $n^2 - 1$ ways to place the 2nd queen
  - …

  ==> In total there are **$(n^2)!$** Ways to place n queens on the chessboard such that there does not exist any cell having more than 1 queen. For each way, we need to check constraint P (no two queens attack to each other); if this constraint P is satisfied, then this way is a solution to the problem.
- Another algorithm: reduce the solution space from **$(n^2)!$** to **$n!$**

NGUYỄN KHÁNH PHƯƠNG
KHMT – SOICT - ĐHBK HN

Represent the solution

- Index the row and column of the chessboard from 1 to n.
- Each solution to the problem could be presented as an array of n elements $(a_1, a_2, \ldots, a_n)$, where a_i is the index of the column of the queen on the i^{th} row.
- $(a_1, a_2, \ldots, a_n)$: need to satisfy the constraint P
  - $a_i \neq a_j$, for all $i \neq j$ (two queens on i^{th} row and j^{th} row are not on the same column);
  - $| a_i - a_j | \neq | i - j |$, for all $i \neq j$ (two queens on cells (i, a_i) and (j, a_j) are not on the same diagonal).

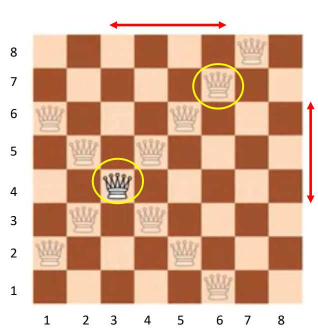
The n-Queen problem could be seen as enumerate all elements of:

$$D=\{(a_1, a_2, \ldots, a_n)\in N^n: a_i \neq a_j \text{ và } |a_i - a_j| \neq |i - j|, i \neq j \}$$

NGUYỄN KHÁNH PHƯƠNG
KHMT – SOICT - ĐHBK HN

The n-Queens Problem:

- The way to check two queens on cell (i_1, j_1) and cell (i_2, j_2) are not on the same diagonal



$|i_1 - i_2| \neq |j_1 - j_2|$

The n-Queens Problem: Backtracking

Algorithm:

1st iteration: place the 1st queen on cell (row 1, column a_1)

2nd iteration: place the 2nd queen on cell (row 2, column a_2)

…

Arrange each queen on each row of the chessboard in turn:

- Assume we already have the partial solution $(a_1, a_2, \ldots, a_{k-1})$: that is, already placed $(k-1)$ queens on the cell $(1, a_1), (2, a_2), \ldots (k-1, a_{k-1})$ satisfying the constraint P.
- We now need to find the value for a_k : so that we can place the k^{th} queen on cell (k, a_k) and satisfy the constraint P
  - Scan for each column $j = 1,2,\ldots,n$:
    - Check if we could place the k^{th} queen on cell (k, j) – row k column j: by using the function `Check(int j, int k)` return 1 if could place; otherwise return 0

The n-Queens Problem: Backtracking

```
int Check(int j, int k) //check if could place queen on cell (k, j)
{
  for (i=1; i<k; i++) //browse for each row 1..(k-1) already having queens
    if ((j == a[i]) || (fabs(j-a[i])== k-i)) return 0;

   return 1;
}
void Try(int k) //find value for a[k]: the column to place the kth queen
{
  for (int j=1; j<=n; j++) //browse each column 1..n: whether to place queen on cell (k, j)
    if (Check(j,k)) //if could place queen on cell (k, j)
    {
       a[k]=j;
       if (k==n) PrintSolution(); //already placed all n queens, so print solution on screen
       else Try(k+1); //continue find column to place the (k+1)th queen
    }
}
```

Exercise 3

Enumerate all solutions to the positive integer linear equation:

$$x_1 + x_2 + \ldots + x_n = N$$
$$x_1, x_2, \ldots, x_n > 0$$

Backtracking: Solution to problem $(x_1, x_2, \ldots, x_n)$

At each iteration k (k=1,..,n): we need to find value for x_k

- Assume we already have partial solution $(x_1, x_2, \ldots, x_{k-1})$: that means, we know already values of (k-1) variables, which are $x_1, x_2, \ldots, x_{k-1}$
- Now we need to find value for x_k:
  - Calculate: $M = x_1 + x_2 + \ldots + x_{k-1}$
  - Sum of remaining (N-k) variables $x_{k+1}, \ldots, x_n$ at least $= N - k$
  - The maximum value that x_k could be: $U = N - M - (N - k)$

  ➔ variable x_k could only be: $1 \le x_k \le U$

Backtracking: Solution to problem $(x_1, x_2, \ldots, x_n)$

```
void Try(int k) //find value for x[k]
{
    if (k==n) //there only the last variable x_n need to find value
    {
        U = N - M; L = U;
    }
    else {
        U = N - M - (N-k); L = 1;
    }
    for (j =  L; j <= U; j++)
    {
        x[k] = j;
        M = M + j;
        if (k == n) PrintSolution();//already have values of all n variables: x_1, x_2,...,x_n, so print solution on screen
        else  Try(i+1);//continue find value for x_k+1
        M = M - j;
    }
}
int main()
{   Enter value for n, N
    M = 0; //store sum of all variables that have found values already
    Try(1);
}
```

NGUYỄN KHÁNH PHƯƠNG
KHMT – SOICT - ĐHBK HN

Backtracking: another way

```c
#include <stdio.h>
#define MAX 100
int n,M,N;
int x[MAX];
void PrintSolution(){
    for(int i = 1; i <= n; i++)
        printf("%d ",x[i]);
    printf("\n");
}
int check(int j, int k){
    if(k == n) return M + j == N;
    return 1;
}
```

```c
void Try(int k){
    for(int j = 1; j <= N - M - (n-k); j++){
        if(check(j,k)){
            x[k] = j;
            M += j;
            if(k == n) PrintSolution();
            else Try(k+1);
            M -= j;
        }
    }
}
int main(){
    n = 3; N = 5; M = 0;
    Try(1);
}
```

Exercise 4: Sodoku

Generate all the ways to fill numbers 1, …, 9 to cells of a grid 9x9 such that:

- Numbers of each row are different
- Numbers of each column are different
- Numbers of each sub-grid (3x3) are different

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

Exercise 4: Sodoku

```c
int check(int v, int r, int c){
  for(int i = 0; i <= r-1; i++)
    if(x[i][c] == v) return 0;
  for(int j = 0; j <= c-1; j++)
    if(x[r][j] == v) return 0;
  int I = r/3;
  int J = c/3;
  int i = r - 3*I;
  int j = c - 3*J;
  for(int i1 = 0; i1 <= i-1; i1++)
   for(int j1 = 0; j1 <= 2; j1++)
     if(x[3*I+i1][3*J+j1] == v)return 0;
  for(int j1 = 0; j1 <= j-1; j1++)
   if(x[3*I+i][3*J+j1] == v) return 0;
  return 1;
}
```

```c
void TRY(int r, int c){
  for(int v = 1; v <= 9; v++){
    if(check(v,r,c)){
      x[r][c] = v;
      if(r == 8 && c == 8)
        PrintSolution();
      else{
        if(c == 8) TRY(r+1,0);
        else TRY(r,c+1);
      }
    }
  }
}

int main(){
  TRY(0,0);
}
```

Homework

1. Given positive integers M, N, and A_1, A_2, …, A_N. Find all solutions to the equation:

$$A_1X_1 + A_2X_2 + \ldots + A_NX_N = M$$

2. Implement the *n*-queens and sudoku problems using marking array