ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

Algorithmic paradigms

CONTENT

- Recursion
- Recursion with memoization
- Backtracking
- Branch and Bound
- Greedy
- Divide and conquer
- Dynamic Programming

Recursion

- A code in which the name of a function (procedure) appears in the function itself
- Base case
  - Input parameter of the function is small enough so that the result of the function can be computed directly without calling the function recursively

```
Example: S = 1 + 2 + … + n

int sum(int n) {
  if (n <= 1) return  1;
  int s = sum(n-1);
  return n + s;
}
```

Recursion

- Example

  $$C(k, n) = \frac{n!}{k!(n-k)!}$$

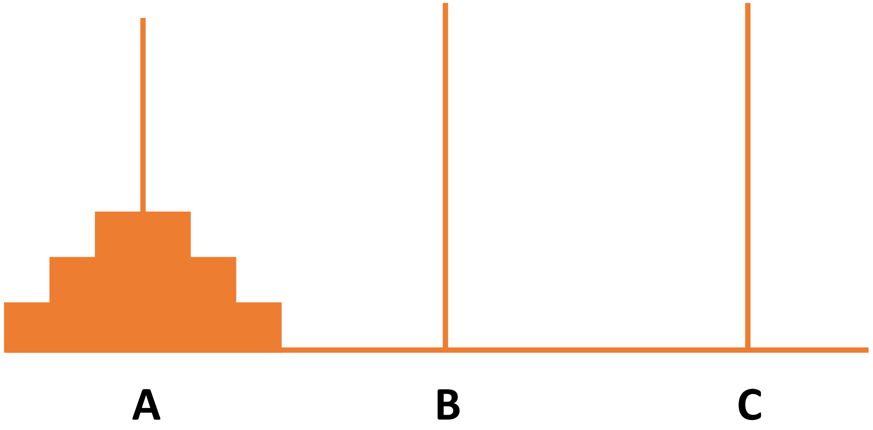
- Recurrent relation
  - $C(k, n) = C(k\text{-}1, n\text{-}1) + C(k, n\text{-}1)$
- Base case:

  $$C(0, n) = C(n, n) = 1$$

```
int C(int k, int n) {
  if (k == 0 || k == n)
    return  1;
  int C1 = C(k-1,n-1);
  int C2 = C(k,n-1);
  return C1 + C2;
}
```

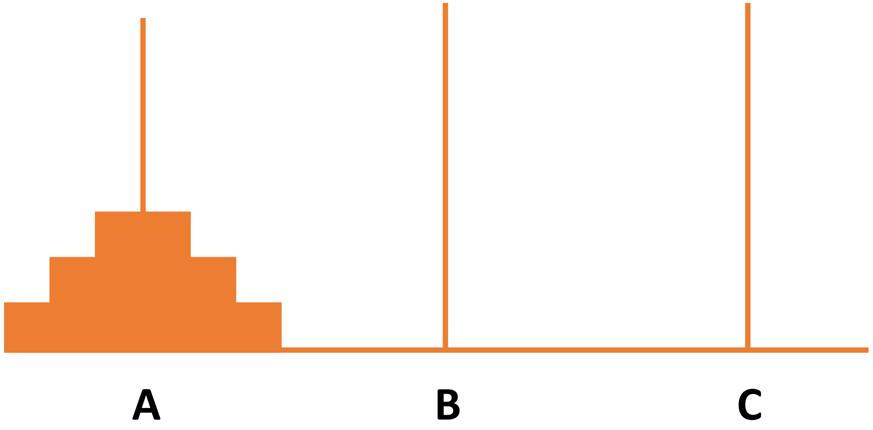
Hanoi tower

- There are *n* disks with different sizes and 3 rods A, B, C
- Initialization: *n* disks are in the rod A such that larger disks are below smaller ones



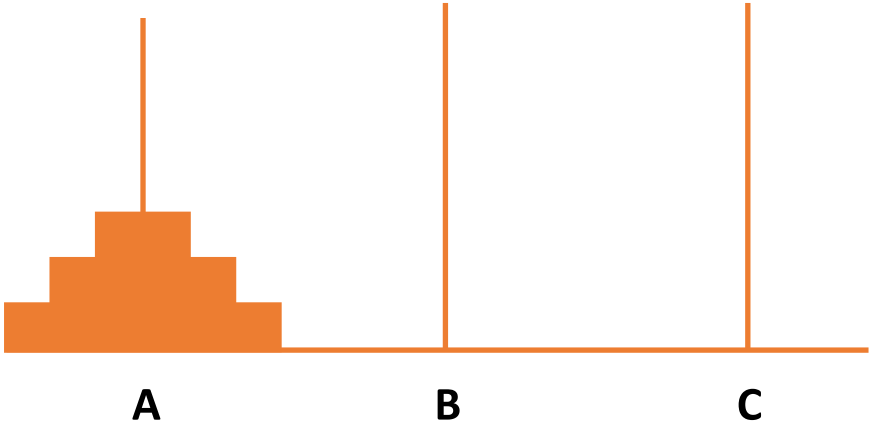
Hanoi tower

- There are *n* disks with different sizes and 3 rods A, B, C
- Initialization: *n* disks are in the rod A such that larger disks are below smaller ones
- Goal: Move *n* disks from A to B such that
  - Only one disk can be moved at a time
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
  - No larger disk may be placed on top of a smaller disk



Hanoi tower

- There are *n* disks with different sizes and 3 rods A, B, C
- Initialization: *n* disks are in the rod A such that larger disks are below smaller ones
- Goal: Move *n* disks from A to B such that
  - Only one disk can be moved at a time
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
  - No larger disk may be placed on top of a smaller disk



Lời giải

B1: A → B

B2: A → C

B3: B → C

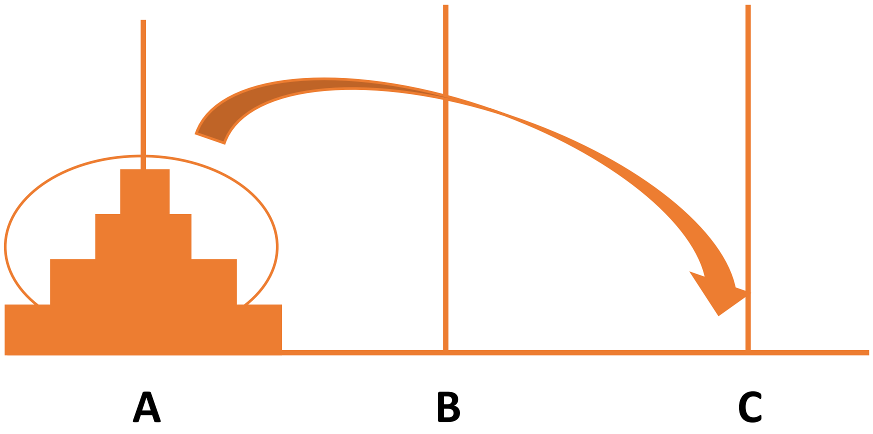
B4: A → B

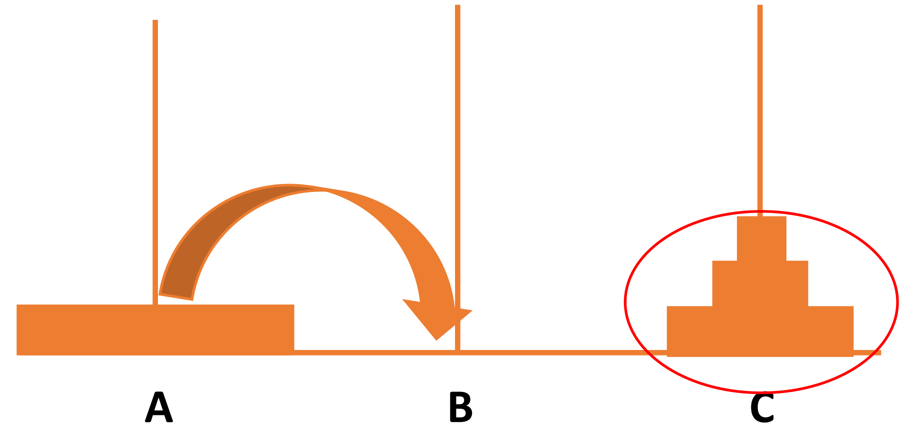
B5: C → A

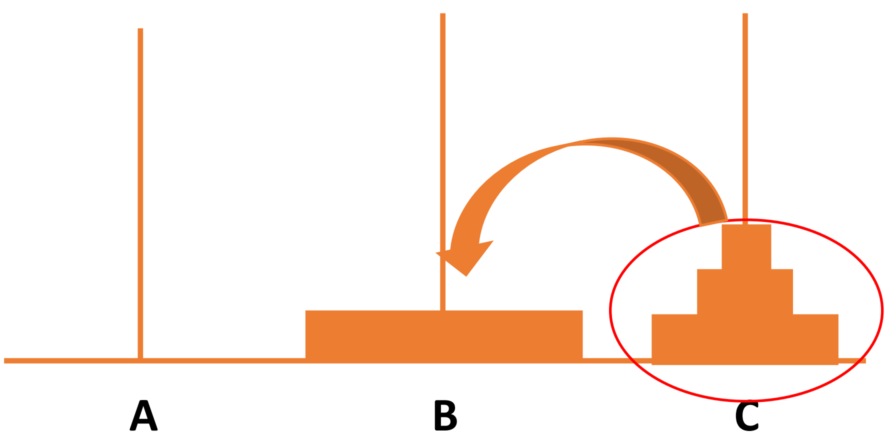
B6: C → B

B7: A → B

Hanoi tower

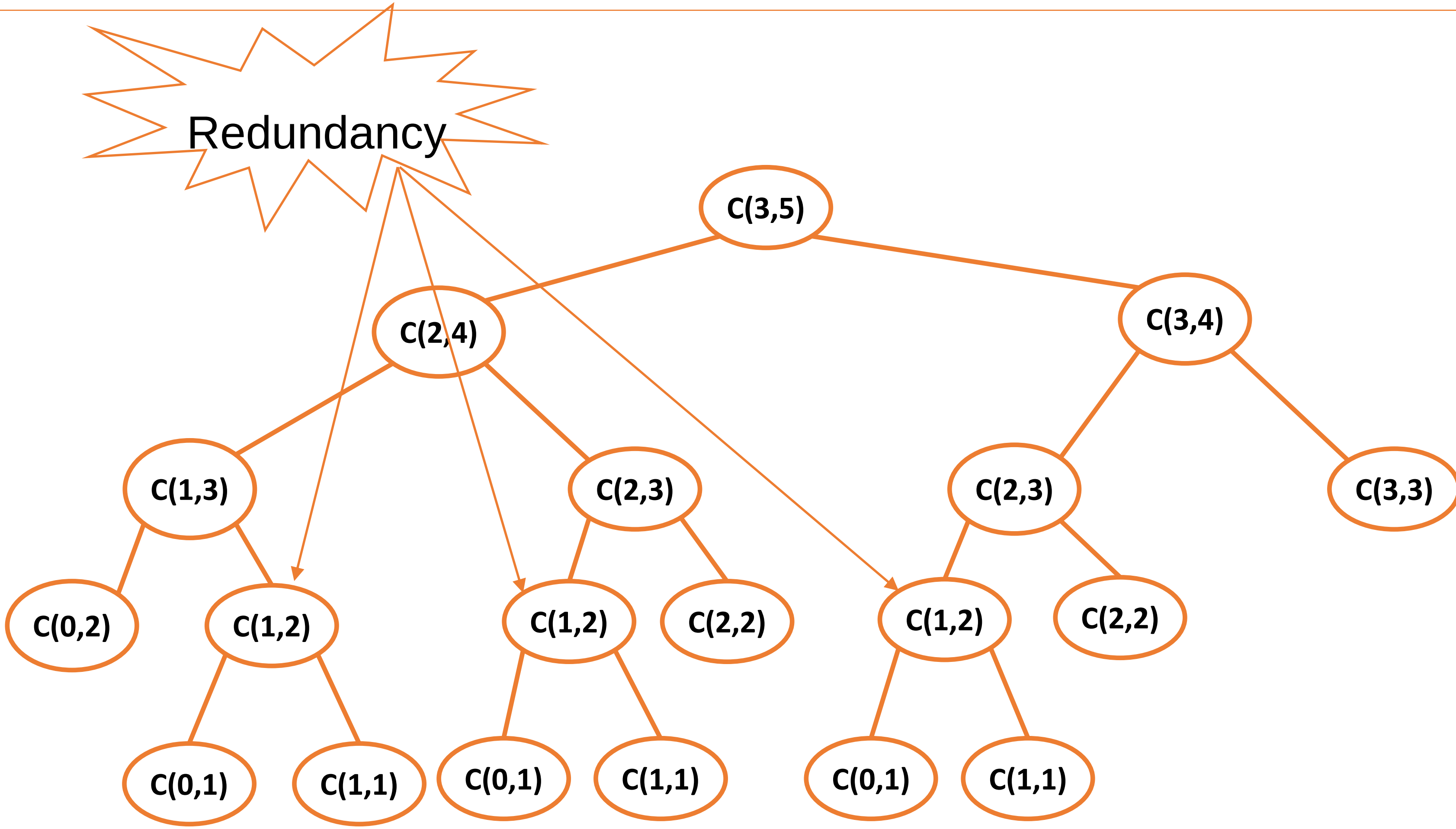







```
move(n, A, B, C) {
  if(n == 1) {
    print(“Move 1 disk from A to B”)
  }else{
    move(n-1, A, C, B);
    move(1, A, B, C);
    move(n-1, C, B, A);
  }
}
```

Recursion with memoization



Recursion with memoization

- Use memory (table) to store results of subproblems
- Map each subproblem with a specified parameter to an element of the table
- Initialize the table with special values indicating that no results of any subproblem are available

```
int m[MAX][MAX];

int C(int k, int n) {
  if (k == 0 || k == n)
    m[k][n] = 1;
  else {
    if(m[k][n] < 0){
      m[k][n] = C(k-1,n-1) +
                C(k,n-1);
    }
  }
  return m[k][n];
}

int main() {
  for(int i = 0; i < MAX; i++)
    for(int j = 0; j < MAX; j++)
       m[i][j] = -1;
  printf("%d\n",C(16,32));
}
```

Backtracking

- Application: solve combinatorial enumeration/optimization problems
- A = $\{(x_1, x_2, \ldots, x_n) \mid x_i \in A_i, \forall\, i = 1, \ldots, n\}$
- Generate all configurations $x \in A$ satisfying some property P
- Recursive procedure Try(k):
  - Try all values v that can be assigned to x_k without violating P
  - For each feasible value v:
    - Assign v to x_k
    - If $k < n$: call recursively Try(k+1) to try values for x_{k+1}
    - If $k = n$: get a solution

Backtracking

```
Try(k){
  foreach v of A_k
     if check(v,k) /* check feasibility of v */
       {
         x_k = v;
         [Update a data structure D]
         if(k = n) solution();
         else Try(k+1);
         [Recover D when backtracking]
       }
}
Main(){
  Try(1);
}
```

Backtracking

- Example: generate all binary sequences of length n
- Modelling:
  - Array `x[n]` where `x[i]` $\in\{0,1\}$ is the i^{th} bit of the sequence
    `(i= 0, . . ., n-1)`

```
void solution(){
  for(int k = 0; k < n; k++)
    printf("%d",x[k]);
  printf("\n");
}

int TRY(int k) {
  for(int v = 0; v <= 1; v++){
    x[k] = v;
    if(k == n-1) solution();
    else TRY(k+1);
  }
}

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

Backtracking

- Example: generate all binary sequences of length n containing no 2 consecutive bit 1
- Modelling: Array `x[n]` where `x[i]` $\in$ {0,1} is the i[th] bit of the sequence `(i= 0, . . ., n-1)`

```
void solution(){
  for(int k = 0; k < n; k++)
     printf("%d",x[k]);
  printf("\n");
}
int check(int v, int k){
  if(k == 0) return 1;
  return v + x[k-1] <= 1;
}
```

```
int TRY(int k) {
  for(int v = 0; v <= 1; v++){
    if(check(v,k)){
      x[k] = v;
      if(k == n-1) solution();
      else TRY(k+1);
    }
  }
}
int main() {
  TRY(0);
}
```

Backtracking

- Generate all *k*-subsets (sets having k elements) of 1, 2, …, *n*
- Modelling: Array $x[k]$ where $x[i] \in \{1, . . ., n\}$ is the i^{th} of the configuration ($i = 1, . . ., k$)
- Constraint *P*: $x[i] < x[i+1]$, for all $i = 1, 2, …, k-1$

```
int TRY(int i) {
  for(int v = x[i-1]+1; v <= n-k+i;
          v++){
     x[i] = v;
     if(i == k)
        printSolution();
      else TRY(i+1);
     }
}

int main() {
  x[0] = 0;
  TRY(1);
}
```

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Backtracking

- Generate all permutations of 1, 2, …, n
- Modelling: Array $x[1,. . . , n]$ where $x[i] \in \{1, . . ., n\}$ is the i^{th} element of the permutation ($i = 1, . . ., n$)
- Property P: $x[i] \neq x[j]$, với mọi $1 \leq i < j \leq n$
- Makring: $m[v]$ = true (false) if v has been appeared (not appeared) in the partial solution, for all $v = 1, …, n$

```
void TRY(int i) {
  for(int v = 1; v <= n; v++){
    if(!m[v]) {
      x[i] = v;
      m[v] = true; // mark v as appeared
      if(i == n)
        solution();
      else TRY(i+1);
      m[v] = false;// recover status of v
    }
  }
}

void main() {
  for(int v = 1; v <= n; v++)
     m[v] = false;
  TRY(1);
}
```

Backtracking: queen

- Place n queens on a chees board such that no two queens attach each other
- Modelling: $x[1, . . ., n]$ where $x[i]$ is the row index of the queen on column i, $i = 1, \ldots, n$
- Property P
  - $x[i] \neq x[j]$, $1 \leq i < j \leq n$
  - $x[i] + i \neq x[j] + j$, $1 \leq i < j \leq n$
  - $x[i] – i \neq x[j] - j$, $1 \leq i < j \leq n$

	1	2	3	4
1		X		
2				X
3	X			
4			X	

solution x = (3, 1, 4, 2)

Backtracking: queen

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

```
void TRY(int k) {
  for(int v = 1; v <= n; v++) {
    if(check(v,k)) {
      x[k] = v;
      if(k == n) solution();
      else TRY(k+1);
    }
  }
}
void main() {
    TRY(1);
}
```

Backtracking: sudoku

- 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

Backtracking: sudoku

- Modelling: Array $x[0..8, 0..8]$
- Property P
  - $x[i, j_2] \neq x[i, j_1]$, with $i = 0,\ldots,8$, và $0 \leq j_1 < j_2 \leq 8$
  - $x[i_1, j] \neq x[i_2, j]$, with $j = 0,\ldots,8$, và $0 \leq i_1 < i_2 \leq 8$
  - $x[3I+i_1, 3J+j_1] \neq x[3I+i_2, 3J+j_2]$, with $I, J = 0,\ldots, 2$, and $i_1, j_1, i_2, j_2 \in \{0,1, 2\}$ such that $i_1 \neq i_2$ or $j_1 \neq j_2$

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

Backtracking: sudoku

- Order of variables to be considered: from up to down, and from left to right
- First tried variable is x[0,0]

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				

Backtracking: sudoku

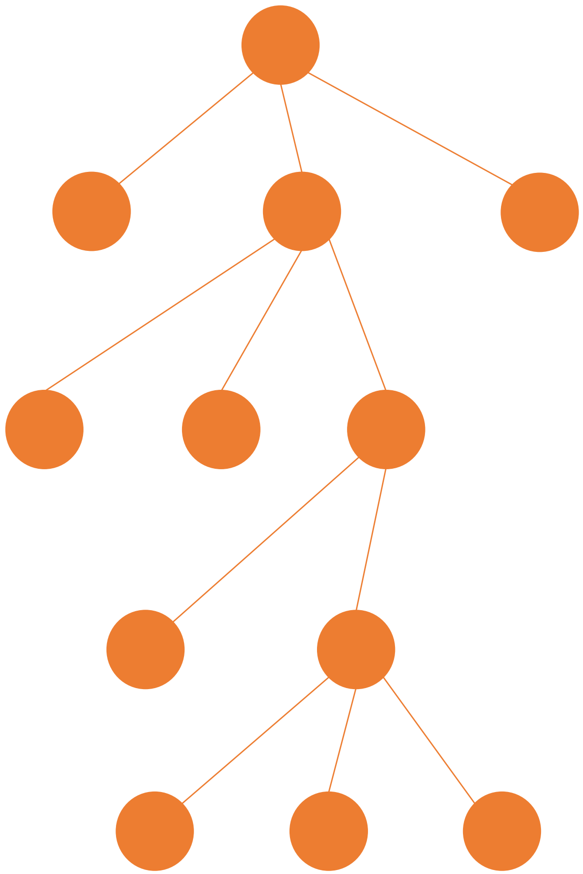
```
bool check(int v, int r, int c){
  for(int i = 0; i <= r-1; i++)
    if(x[i][c] == v) return false;
  for(int j = 0; j <= c-1; j++)
    if(x[r][j] == v) return false;
  int I = r/3; int J = c/3;
  int i = r - 3*I;  int j = c - 3*J;
  for(int i1 = 0; i1 <= i-1; i1++)
    for(int j1 = 0; j1 <= 2; j1++)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

```
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){
        solution();
      }else{
        if(c == 8) TRY(r+1,0);
        else TRY(r,c+1);
      }
    }
  }
}

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

Branch and Bound

- Solve combinatorial optimization problems: find a solution that minimizes or maximizes an objective function
- Base on exhaustive search
- Each node of the search tree corresponds to a ***partial solution***
- Evaluate the lower/upper bound of the objective function of solutions developed from a current partial solution
  - For minimization problem: if the lower bound is greater than or equal to the value of the objective function of the best solution found so far, then ***cut-off*** (do not expand from the current partial solution to compete solutions)
  - For maximization problem: if the upper bound is less than or equal to the value of the objective function of the best solution found so far, then cut-off



Branch and Bound

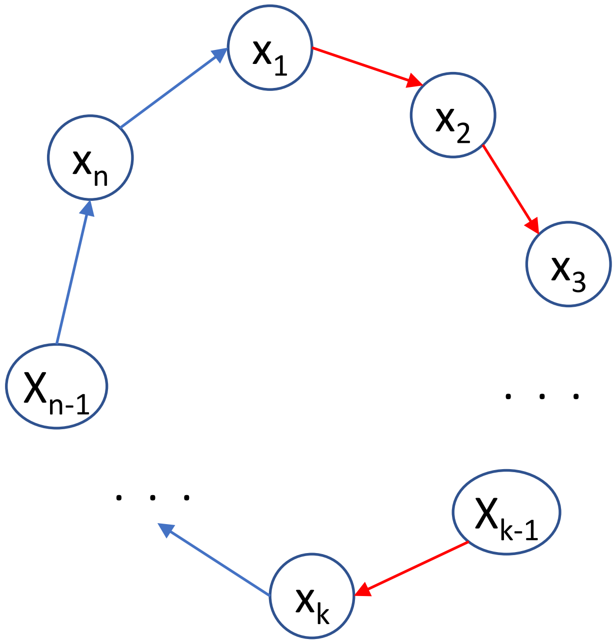
- Travelling Salesman Problem (TSP)
  - There are n cities 1, 2, …, n. The cost for travelling from city i to j is $c(i, j)$. Find a closed tour starting from city 1, visiting other cities exactly once and coming back to 1 having minimal total cost
- Modelling: Solution $x = (x_1, x_2, \ldots, x_n)$ where $x_i \in \{1, 2, \ldots, n\}$
  - Constraint: $x_i \neq x_j$, $\forall\ 1 \leq i < j \leq n$
  - Objective function

$$f(x) = c(x_1, x_2) + c(x_2, x_3) + \ldots + c(x_n, x_1) \rightarrow \min$$

Branch and Bound: TSP

- c_m : minimum cost among the cost between two cities
- Partial solution $(x_1,\ldots, x_k)$
  - Cost of the partial solution $f = c(x_1, x_2) + c(x_2, x_3) + \ldots + c(x_{k-1}, x_k)$
  - Lower bound

$$g(x_1,\ldots, x_k) = f + c_m \times (n-k+1)$$



Branch and Bound: TSP

```
void TRY(int k){
  for(int v = 1; v <= n; v++){
    if(marked[v] == false){
      a[k] = v;
      f = f + c[a[k-1]][a[k]];
      marked[v] = true;
      if(k == n){
        solution();
      }else{
        int g = f + cmin*(n-k+1);
        if(g < f_min)
          TRY(k+1);
      }
      marked[v] = false;
      f = f - c[a[k-1]][a[k]];
     }
   }
}
```

```
void solution() {
  if(f + c[x[n]][x[1]] < f_min){
    f_min = f + c[x[n]][x[1]];
  }
}
void main() {
  f_min = 9999999999;
  for(int v = 1; v <= n; v++)
    marked[v] = false;
  x[1] = 1; marked[1] = true;
  TRY(2);
}
```

Greedy algorithms

- Problem-based heuristics
- Each step, choose the best decision based on local information, do not take into account negative impacts of that decision in the future
- Not sure to find global optimal solutions

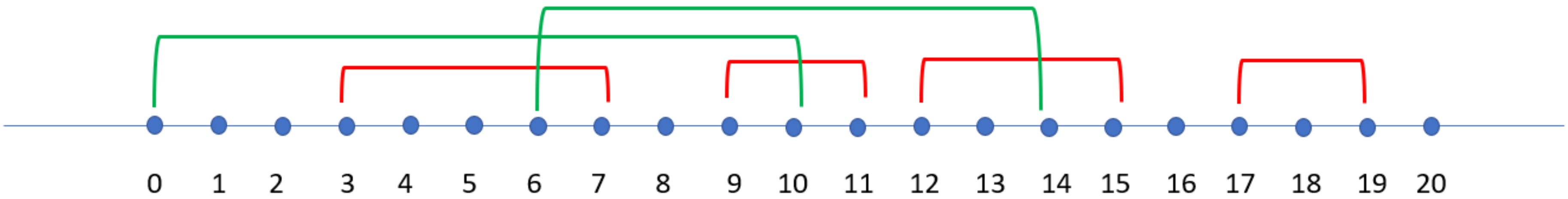
Greedy algorithms

- S: solution under construction (set of components)
- C: set of candidates
- select(C): select the most promising component to add to S
- solution(S): return true if S is a solution to the problem
- feasible(S): return true if S does not violate any constraints

```
Greedy() {
  S = {};
  while C ≠ ∅ and
        not solution(S){
    x = select(C);
    C = C \ {x};
    if feasible(S ∪ {x}) {
      S = S ∪ {x};
    }
  }
  return S;
}
```

Greedy algorithms: disjoint segments

- Given a set of segments $X = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ in which $a_i < b_i$ are coordinates of the segment i on a line, $i = 1, \ldots, n$.
- Goal: Find a subset of X having largest cardinality in which no two segments of the subset intersect



Optimal solution: S = {(3,7), (9,11), (12,15), (17,19)}

Greedy algorithms: disjoint segments

```
Greedy1() {
  S = {};
  L = sort segments of X in a non-decreasing order of a_i;
  while (X ≠ ∅) {
    (a_c,b_c) = select first segment of L;
    remove (a_c,b_c) from L;
    if feasible(S ∪ {(a_c,b_c)}) {
      S = S ∪ {(a_c,b_c)};
    }
  }
  return S;
}
```

Greedy algorithms: disjoint segments

- Greedy 1 cannot ensure to find optimal solutions, example

$$X = \{(1,11), (2,5), (6,10)\}$$

→ Greedy 1 returns solutions {(1,11)}, however, optimal solution is {(2,5), (6,10)}

Greedy algorithms: disjoint segments

```
Greedy2() {
  S = {};
  L = sort segments of X in a non-decreasing order of bi-ai;
  while (X ≠ ∅) {
    (ac,bc) = select first segment of L;
    remove (ac,bc) from L;
    if feasible(S ∪ {(ac,bc)}) {
      S = S ∪ {(ac,bc)};
    }
  }
  return S;
}
```

Greedy algorithms: disjoint segments

- Greedy 2 cannot ensure to find optimal solutions, example

$$X = \{(1,5), (4,7), (6,11)\}$$

→ Greedy 2 returns solution {(4,7)}, however, optimal solution is {(1,5), (6,11)}

Greedy algorithms: disjoint segments

```
Greedy3() {
  S = {};
  L = sort segments of X in a non-decreasing order of bᵢ;
  while (X ≠ ∅) {
    (a_c,b_c) = select first segment of L;
    remove (a_c,b_c) from L;
    if feasible(S ∪ {(a_c,b_c)}) {
      S = S ∪ {(a_c,b_c)};
    }
  }
  return S;
}
```

Greedy algorithms: exercise

- Given a set of jobs 1, 2, …, n where each job i has a deadline $d(i)$ and associated profit $p(i)$ if the job is finished before the deadline. It is assumed that every job takes the single unit of time and each job is executed at a time. Compute a subset of jobs to be scheduled such that each job is not finished after the deadline and total profits of jobs is maximal.

Greedy algorithms: exercise

- Example: there are 4 jobs with deadline d[i] and profit p[i] are given as follows:
  - Job 1: 4 20
  - Job 2: 1 10
  - Job 3: 1 40
  - Job 4: 1 30
- Optimal solution consists of jobs 1 and 3 with the following schedule:
  - Job 3 starts at time-point 0 and finishes at time-point 1
  - Job 1 starts at time-point 1 and finishes at time-point 2
- Total profit is 20 + 40 = 60

Divide and conquer

- Divide the original problem into independent subproblems
- Solve recursively subproblems
- Combine the results of subproblems to establish the solution to the original problem

Divide and conquer: binary search

- Given a sequence $x[1..n]$ which is sorted in an increasing order and a value y. Find the index i such that $x[i] = y$

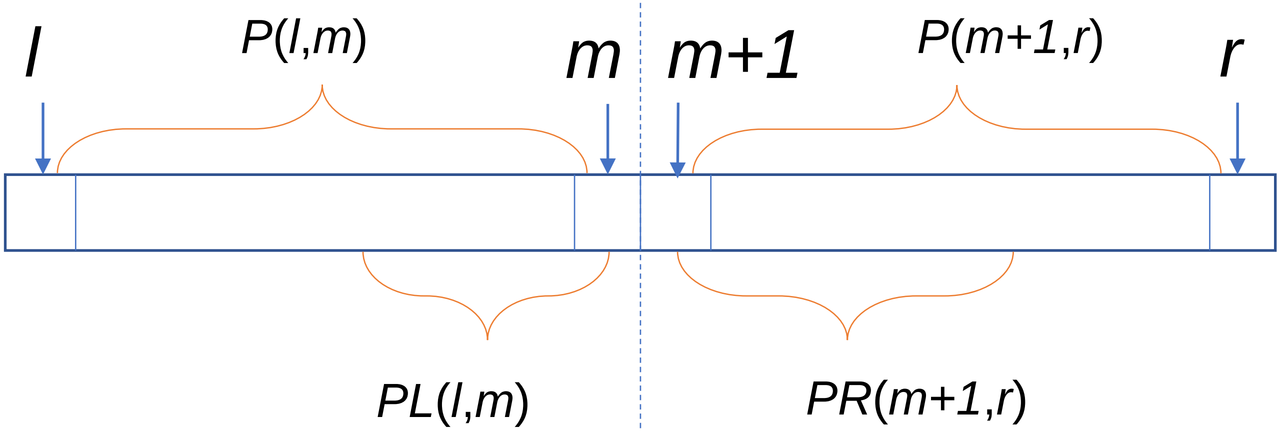
```
bSearch(x, start, finish, y) {
  if(start == finish) {
    if(x[start] == y)
      return start;
    else return -1;
  }else{
    m = (start + finish)/2;
    if(x[m] == y) return m;
    if(x[m] < y)
      return bSearch(x, m+1,finish,y);
    else
      return bSearch(x,start,m-1,y);
  }
}
```

Divide and conquer: largest subsequence

- Given a sequence of numbers $a = a_1, a_2, \ldots, a_n$. Find a subsequence (consecutive elements) such that the sum of its elements is maximal
- Division: $P(i, j)$ is the sum of elements of the largest subsequence of $a_i, a_{i+1}, \ldots, a_j$
- Combination
  - Denote $PL(i, j)$ the sum of elements of the largest subsequence of $a_i, a_{i+1}, \ldots, a_j$ such that the last element of that largest subsequence is a_j
  - Denote $PR(i, j)$ the sum of elements of the largest subsequence of $a_i, a_{i+1}, \ldots, a_j$ such that the first element of that largest subsequence is a_i

Divide and conquer: largest subsequence

- Consider the interval $[l, l+1, ..., r]$. Denote $m = (l+r)/2$
- $P(l,r) = \text{MAX}\{P(l, m), P(m+1,r), PL(l,m) + PR(m+1,r)\}$



Divide and conquer: largest subsequence

```
maxLeft(a, l, r){
  max = -∞;  s = 0;
  for i = r downto l do{
    s += a[i];
    if(s > max) max = s;
  }
  return max;
}
maxRight(a, l, r){
  max = -∞;  s = 0;
  for i = l to r do{
    s += a[i];
    if(s > max) max = s;
  }
  return max;
}
```

```
maxSeq(a[…], l, r){
  if l = r then return a[r];
  m = (l+r)/2;
  ml = maxSeq(a,l,m);
  mr = maxSeq(a,m+1,r);
  mlr = maxLeft(a,l,m) +
              maxRight(a,m+1,r);
  return max(ml,mr,mlr);
}

main() {
  input a[1..n]
  result = maxSeq(a,1,n);
}
```

Divide and conquer: running time analysis

- Divide the original problem into a subproblems of size *n*/*b*
- *T*(*n*): running time of the problem of size *n*
- Division time (line 4): *D*(*n*)
- Combination time (line 6): *C*(*n*)
- Recurrence relation:

$$T(n) = \begin{cases} \Theta(1), & n \le n0 \\ aT\left(\frac{n}{b}\right) + C(n) + D(n), & n > n0 \end{cases}$$

```
procedure D-and-C(n) {
1.  if(n ≤ n0)
2.     Process directly
3.  else{
4.     Divide original problem
              into a subproblems of size n/b
5.     Solve a subproblems recursively
6.     Combination of solutions
7.  }
}
```

Master theorem

- Recurrence relation:

$$T(n) = aT(n/b) + cn^k\text{, with constants } a \geq 1,\ b > 1,\ c > 0$$

- If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$
- If $a = b^k$ then $T(n) = \Theta(n^k \log n)$ with $\log n = \log_2 n$
- If a < b^k then $T(n) = \Theta(n^k)$

Dynamic programming

- Break the given problem down into a set of simpler subproblems
- Solve these subproblems once and store solutions in memory (e.g., tables) in a bottom-up fashion
- Smallest subproblems can be solved trivially
- Recurrence relation: solutions to a subproblem can be computed based on solutions (solved and stored in memory) to smaller subproblems

Dynamic programming

- Given an integers sequence $a = (a_1, a_2, \ldots, a_n)$. A subsequence of a is defined to be $a_i, a_{i+1},\ldots,a_j$. The weight of a subsequence is the sum of it s elements. Find the subsequence having the highest weight

Dynamic programming

- Subproblem definition
  - S_i: weight of the highest subsequence of $a_1,\ldots,a_i$ in which the last element of the subsequence is a_i, $i = 1,\ldots,n$
  - Smallest problem can be solved directly: $S_1 = a_1$
  - Recurrence relation

$$S_i = \begin{cases} S_{i-1} + a_i, \text{ if } S_{i-1} > 0 \\ a_i, \text{ otherwise} \end{cases}$$

Dynamic programming

- Longest increasing subsequence: given a sequence $a = a_1, a_2, \ldots, a_n$. A subsequence of a is created by removing some elements of a. Find a subsequence of a which is an increasing sequence and the length is longest

Dynamic programming

- Division
  - Denote P_i the problem of finding the longest increasing subsequence of $a_1, \dots, a_i$ such that the last element is a_i, for all $i = 1,\dots, n$
  - Denote S_i the number of element of the solution to P_i, $\forall\ i = 1,\dots, n$

Dynamic programming

- Division
  - Denote P_i the problem of finding the longest increasing subsequence of $a_1, \ldots, a_i$ such that the last element is a_i, for all $i = 1,\ldots, n$
  - Denote S_i the number of element of the solution to P_i, $\forall\, i = 1,\ldots, n$
  - $S_1 = 1$
  - $S_i = \max\{1, \max\{S_j + 1 \mid j < i \wedge a_j < a_i\}\}$
  - Solution to the original problem is $\max\{S_1, S_2, \ldots, S_n\}$

Dynamic programming

- Division
  - Denote P_i the problem of finding the longest increasing subsequence of $a_1, \ldots, a_i$ such that the last element is a_i, for all $i = 1,\ldots, n$
  - Denote S_i the number of element of the solution to P_i, $\forall\ i = 1,\ldots, n$
  - $S_1 = 1$
  - $S_i = \max\{1, \max\{S_j + 1 \mid j < i \wedge a_j < a_i\}\}$
  - Solution to the original problem is $\max\{S_1, S_2, \ldots, S_n\}$

```
void solve(a[1…n]){
  S[1] = 1;
  rs = S[1];
  for i = 2 to n do{
    S[i] = 1;
    for j = i-1 downto 1 do{
      if(a[i] > a[j]){
        if(S[j] + 1 > S[i])
          S[i] = S[j] + 1;
      }
    }
    rs = max(S[i], rs);
  }
  return rs
}
```

Dynamic programming

- Longest common subsequence
  - Denote $X = \langle X_1, X_2, \ldots, X_n \rangle$, a subsequence of X is created by removing some element from X
  - Input: Given two sequences $X = \langle X_1, X_2, \ldots, X_n \rangle$ and $Y = \langle Y_1, Y_2, \ldots, Y_m \rangle$
  - Output: Find a common subsequence of X and Y such that the length is longest

Dynamic programming

- Longest common subsequence
  - Division
    - Denote $S(i, j)$ the length of the longest common subsequence of $\langle X_1, \ldots, X_i \rangle$ and $\langle Y_1, \ldots, Y_j \rangle$, $\forall\, i = 1, \ldots, n$ and $j = 1, \ldots, m$
    - Base case
      - $\forall\, j = 1,\ldots, m$: $S(1, j) = \begin{cases} 1, \text{ if } X_1 \text{ appears in } Y_1, \ldots, Y_j \\ 0, \text{ otherwise} \end{cases}$
      - $\forall\, i = 1,\ldots, n$: $S(i, 1) = \begin{cases} 1, \text{ if } Y_1 \text{ appears in } X_1, \ldots, X_i \\ 0, \text{ otherwise} \end{cases}$
  - Aggregation

$$S(i, j) = \begin{cases} S(i\text{-}1, j\text{-}1) + 1, \text{ nếu } X_i = Y_j \\ \max\{S(i\text{-}1, j), S(i, j\text{-}1)\} \end{cases}$$

Dynamic programming

X

3	7	2	5	1	4	9

Y

4	3	2	3	6	1	5	4	9	7

S(i,j)

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	1	1	1	1	2
3	0	1	2	2	2	2	2	2	2	2
4	0	1	2	2	2	2	3	3	3	3
5	0	1	2	2	2	3	3	3	3	3
6	1	1	2	2	2	3	3	4	4	4
7	1	1	2	2	2	3	3	4	5	5

Dynamic programming

```
solve(X[1..n], Y[1..m]){
  rs = 0;
  for i = 0 to n do S[i][0] = 0;
  for j = 0 to m do S[0][j] = 0;

  for i = 1 to n do{
    for j = 1 to m do{
      if(X[i] == Y[j]) S[i][j] = S[i-1][j-1] + 1;
      else{
        S[i][j] = max(S[i-1][j], S[i][j-1]);
      }
      rs = max(S[i][j], rs);
    }
  }
  return rs;
}
```