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

Applied Algorithms

Nguyễn Khánh Phương

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

Contents

Contents

Contents

1. Exhaustive search (Brute force)

Exhaustive search (Brute force):

- When the problem requires finding objects that satisfy a certain properties from a given se of projects, we can apply the brute force:
  - Browse all the objects: for each object, check whether it satisfies the required properties or not, if so, that object is the solution to find, if not, keep searching.

Example (Traveling Salesman Problem): A tourist wants to visit n cities T_1, T_2, ..., T_n. *Itinerary is a way of going from a certain city through all the remaining cities, each city exactly once, and then back to the starting city. Let* d_{ij} the cost of going fron T_i to T_j (i, j = 1, 2,..., n). Find itinerary with minimum total cost.

Answer: browse all n! possible itineraries, each itinerary calculates the corresponding trip cost, and compares these n! values to get the one with minimum value.

- Brute force: simple, but computation time is inefficient.

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Example: Stock span problem

This problem is often asked in interviews of Google and Amazon:

> Given a list of prices of a single stock for N number of days, find stock span for each day. Stock span is defined as a number of consecutive days prior to the current day when the price of a stock was less than or equal to the price at current day.

Example:

Prices of stock in period of 6 days are {100, 60, 70, 65, 80, 85}, then stock span = {0,0,1,0,3,4}.

Solve by brute force:

Browse for each day i (from left to right `for i = 0,..,5`):

- `span[i] = 0;`
- scan each previous day j of i (`for j = i-1,..,0`):
  - `if price[j] <= price[i] then span[i]++;`
  - `else break;`

Complexity: $O(N^2)$ where N is number of days.

Exercise: propose algorithm with complexity of O(N) and memory O(N)

Example: Another form of the stock problem

Various signal towers are present in a city.Towers are aligned in a straight horizontal line(from left to right) and each tower transmits a signal in the right to left direction.Tower A shall block the signal of Tower B if Tower A is present to the left of Tower B and Tower A is taller than Tower B. So,the range of a signal of a given tower can be defined as :

{(the number of contiguous towers just to the left of the given tower whose height is less than or equal to the height of the given tower) + 1}.

You need to find the range of each tower.

INPUT

First line contains an integer T specifying the number of test cases.

Second line contains an integer n specifying the number of towers.

Third line contains n space separated integers(H[i]) denoting the height of each tower.

OUTPUT

Print the range of each tower (separated by a space).

Constraints

1 <= T <= 10

2 <= n <= 10^6

1 <= H[i] <= 10^8

A B

SAMPLE INPUT	SAMPLE OUTPUT
1 7 100 80 60 70 60 75 85	1 1 1 2 1 4 6

Exercise: Vito's family

The world-known gangster Vito Deadstone is moving to New York. He has a very big family there, all of them living in Lamafia Avenue. Since he will visit all his relatives very often, he is trying to find a house close to them.

Vito wants to minimize the total distance to all of them and has blackmailed you to write a program that solves his problem.

Input

The input consists of several test cases. The first line contains the number of test cases.

For each test case you will be given the integer number of relatives r ($0 < r < 500$) and the street numbers (also integers) $s_1, s_2, \ldots, s_i, \ldots, s_r$ where they live ($0 < s_i < 30000$). Note that several relatives could live in the same street number.

Output

For each test case your program must write the minimal sum of distances from the optimal Vito's house to each one of his relatives. The distance between two street numbers s_i and s_j is $d_{ij} = |s_i - s_j|$.

Sample Input

```
2
2 2 4
3 2 4 6
```

Sample Output

```
2
4
```

Among of r relatives' houses, find out which relative's house should Vito stay with, so that the total distance that Vito will visit all these r relatives is the smallest.

Exercise: Vito's family

- Method 1: Brute force: O(r^2)
  - Init: `min_Distance =0;`
  - For each element *ith* in the array (`for i =0 to r-1`) consider that as where Vito will be
    - Calculate the distance from i^{th} to every j^{th} element of the array:

```
for i =0 to r-1
{
   distance = 0;
   for j =0 to r-1
      distance += |s_i - s_j|;
```

    - Compare them to find `min_Distance`:

```
if (min_Distance < distance) min_Distance = distance;
```

➔ Time limit exceeded when r is large

Improve: $O(r\log r)$ or $O(r)$???

Exercise: Vito's family

Suppose there is a set S containing real numbers. Find the element $x \epsilon S$, such that $\sum_{y \epsilon S} |y - x|$ is minimum.

Answer: when x = median of set S

So the algorithm consists of 2 steps:

Step 1: Find the median of set S

- Method 1: O($r\log r$)
  - Sort r elements of S in ascending order: $O(r\log r)$
  - Median element = element at position $r/2$ of array `//median = A[r/2];`
- Method 2: O(r) "Find the median in linear time"

Step 2: The solution need to determine is the sum of the distances from the median element to all the remaining elements in the array : $O(r)$

```
min_Distance = 0;
for i =0 to r-1
    min_Distance += abs(A[i] - median);
```


Exercise: Vito's family: Implementation O(r log$r$)

Step 1: Find the median of set S

- Sort r elements of S in ascending order: $O(r\log r)$
- Median element = element at position $r/2$ of array `//median = A[r/2];`

Step 2: The solution need to determine is the sum of the distances from the median element to all the remaining elements in the array : $O(r)$

```
min_Distance = 0;
for i =0 to r-1
    min_Distance += abs(A[i] - median);
```

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

int main() {
    int T, r, i, A[501];
    scanf("%d", &T);
    while(T--) {
        scanf("%d", &r);
        for(i = 0; i < r; i++) scanf("%d", &A[i]);
        sort(A,A+r);
        int min_Distance = 0,  median =A[r/2];
        for(i = 0; i < r; i++)
            min_Distance += abs(A[i] - median);
        printf("%d\n", min_Distance);
    }
    return 0;
}
```


Contents

2. Backtracking

2.1. Algorithm diagram

2.2. Some examples

2. Backtracking

2.1. Algorithm diagram

2.2. Some examples

Backtracking diagram

- ***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 satisfy property P* $\}$.

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

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

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

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

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:

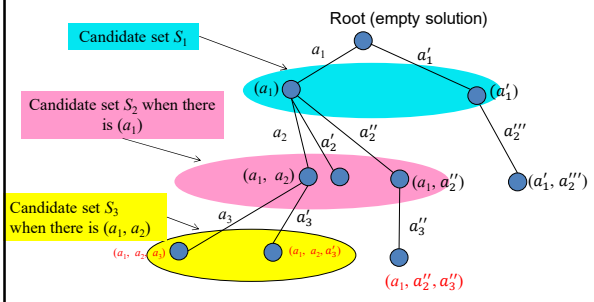
  $$(a_1, a_2, ..., a_{k-1}, a_k)$$

  - 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:
    - $S_k \neq \emptyset$
    - $S_k = \emptyset$

Backtracking diagram

- **$S_k \neq \emptyset$:** 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.
- **$S_k = \emptyset$:** 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 $(k-1)^{th}$ 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 $(k-1)^{th}$ 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 $(k-2)^{th}$ 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)

```
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;
    [maybe: update values of variables]
    if (k == n) then <Record (a_1, a_2, ..., a_k) 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.

Backtracking algorithm (not recursive)

```
void Backtracking ( )
{
  k=1;
  <Build S_k>;
  while (k > 0) {
    while (S_k ≠ ∅ ) {
      a_k ← S_k; // Take a_k from S_k
      if (k == n)> then <Record (a_1,a_2,...,a_k) as a
                          complete solution >;
      else {
            k = k+1;
            <Build S_k>;
      }
    }
    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 ∈ S_k`).
    - 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.

2. Backtracking

2.1. Algorithm diagram

2.2. Some examples

2.2. Some examples

1. **Enumerate all binary strings of length *n***
2. Enumerate all *m*-element subsets of the set of *n*-element
3. Enumerate all permutations of *n* elements
4. n-Queens problem
5. Enumerate all solutions to the positive integer linear equation

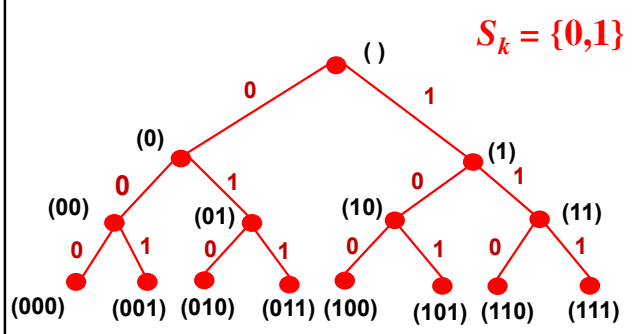
Example 1: Enumerate all binary strings of length *n*

- 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++
```


Decision tree to enumerate binary strings of length 3

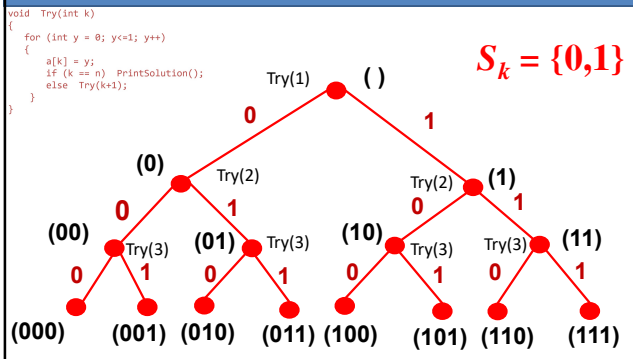


Program in C (Recursive)

```
#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");
}
```

```
void  Try(int k)
{
   for (int y = 0; y<=1; y++)
   {
       a[k] = y;
       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);
}
```


Decision tree to enumerate binary strings of length 3



Program in C (non recursive)

```
#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");
}
```

```
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)

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

2.2. Some examples

1. Enumerate all binary strings of length n
2. **Enumerate all m-element subsets of the set of n-element**
3. Enumerate all permutations of n elements
4. n-Queens problem
5. Enumerate all solutions to the positive integer linear equation

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

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Backtracking algorithm (recursive)

```
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;
      [maybe: update values of variables]
      if (k == n) then <Record (a_1, a_2, ..., a_k) 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 Try(int k)
{
   for (y=a[k-1]+1; y<=n-m+k; y++)
   {
      a[k] =y;
      if (k ==m) PrintSolution( );
      else Try(k+1);
   }
}
Try(1);
```

Program in C (Recursive)

```c
#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");
}
```

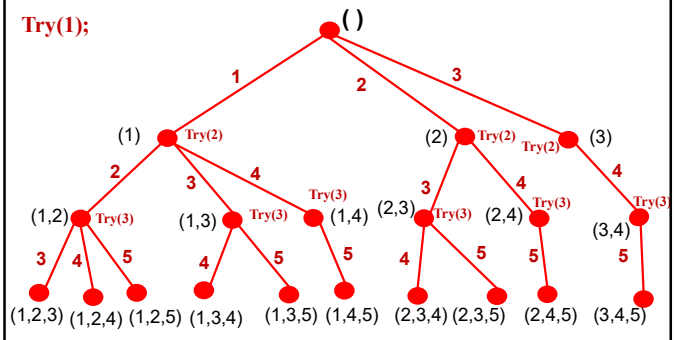
```c
void  Try(int k){
   int y;
   for (y = a[k-1] +1; y<= n-m+k; y++) {
         a[k] = y;
         if (k==m)  PrintSolution();
         else Try(k+1);
    }
}
int main() {
   printf("Enter n, m = ");
   scanf("%d %d",&n, &m);
   a[0]=0; cnt = 0; Try(1);
   printf("Number of %d-element subsets of set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

Program in C (Non Recursive)

```c
#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");
}
```

```c
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)



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

2.2. Some examples

1. Enumerate all binary strings of length n
2. Enumerate all m-element subsets of the set of n-element
3. **Enumerate all permutations of n elements**
4. n-Queens problem
5. Enumerate all solutions to the positive integer linear equation

40

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

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

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

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 k^th
     . . .
   }
```

Program in C (Recursive)

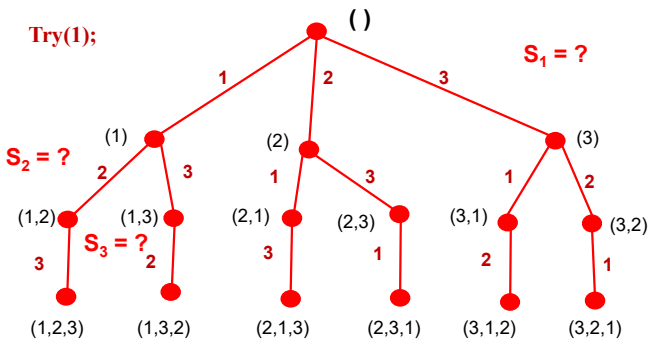
```
#include <stdio.h>

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);
}
```

Decision tree to enumerate permutations of 1, 2, 3



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

Example 3. Enumerate permutations

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

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

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

2.2. Some examples

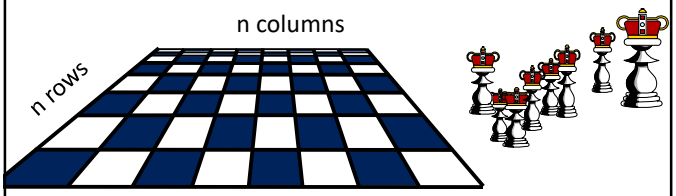
1. Enumerate all binary strings of length n
2. Enumerate all m-element subsets of the set of n-element
3. Enumerate all permutations of n elements
4. **n-Queens problem**
5. Enumerate all solutions to the positive integer linear equation

49

Example 4. The n-Queens problem

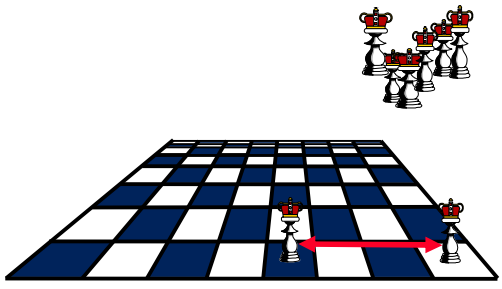
Enumerate all possibilities to place n queens on a chess board nxn such that no two queens attack to each other (no two queens on the same row, same column, same diagonal of the chessboard.

Constraint P



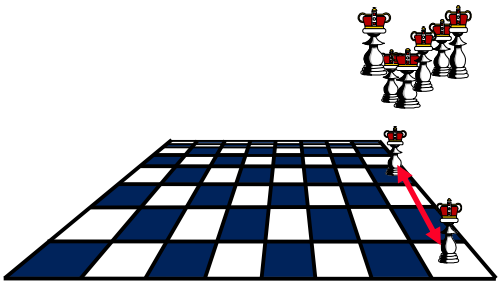
The n-Queens Problem

No two queens on the same row..



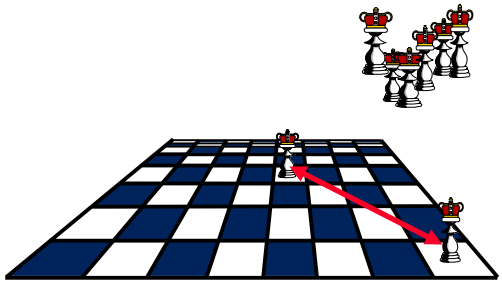
The n-Queens Problem

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 $(a_1, a_2, ..., a_n)$, where a_i is the index of the column of the queen on the i^{th} row.
- $(a_1, a_2, ..., a_n)$: 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).

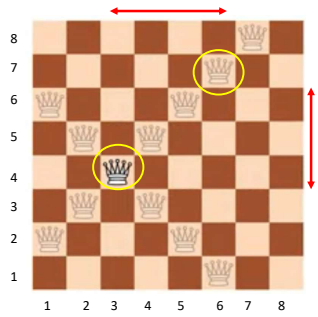
The n-Queen problem could be seen as enumerate all elements of:

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

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, …, a_{k-1}): that is, already placed (k-1) queens on the cell (1, a_1), (2, a_2), …(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,..,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 (UCVh(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
    }
}
```

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

int n, dem;
int a[20];

void Ghinhan() {
   int i;
   dem++;
   cout<<dem<<") ";
   for (i=1; i<=n;i++) cout<<a[i]<<"  ";
   cout<<endl;
}

int UCVh(int j, int k) {
   int i;
   for (i=1; i<k; i++)
      if ((j == a[i]) || (fabs(j-a[i])== k-i)) return 0;
   return 1;
}
void Try (int k)
{
    for (int j=1; j<=n; j++)
        if  (UCVh(j, k))
        {
            a[k] = j;
            if  (k == n) Ghinhan();
            else Try (k+1);
        }
}

int main() {
    cout<<"Nhap kich thuoc ban co: n =";cin>>n;
    cout<<"\n==== CAC CACH XEP QUAN HAU ====\n";
    dem = 0;
    Try (1);
    if (dem == 0) cout<<"Khong co cach xep nao!\n";
}
```

59

2.2. Some examples

1. Enumerate all binary strings of length n
2. Enumerate all m-element subsets of the set of n-element
3. Enumerate all permutations of n elements
4. n-Queens problem
5. **Enumerate all solutions to the positive integer linear equation**

60

Example 5

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

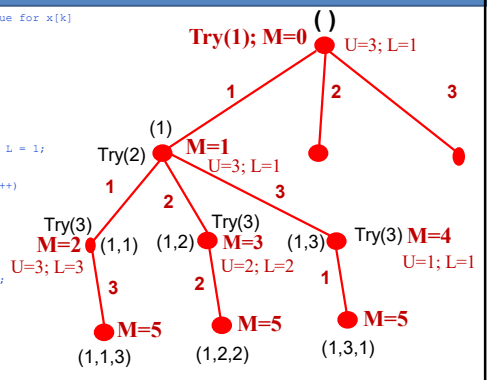
Backtracking: Solution to problem $(x_1, x_2, \ldots, x_n)$

```
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(k+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

Cây liệt kê lời giải n =5, k = 3

```
void Try(int k) //find value for x[k]
{
    if (k==n)
    {
        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();
        else  Try(k+1);
        M = M - j;
    }
}
```



Backtracking: another way

```
#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;
}
```

```
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 1: The Hamming Distance problem

The Hamming distance between two strings of bits (binary integers) is the number of corresponding bit positions that differ. This can be found by using XOR on corresponding bits or equivalently, by adding corresponding bits (base 2) without a carry. For example, in the two bit strings that follow:

```
A       0 1 0 0 1 0 1 0 0 0
B       1 1 0 1 0 1 0 1 0 0
A XOR B = 1 0 0 1 1 1 1 1 0 0
```

The Hamming distance (H) between these 10-bit strings is 6, the number of 1's in the XOR string.

Input

Input consists of several datasets. The first line of the input contains the number of datasets, and it's followed by a blank line. Each dataset contains N, the length of the bit strings and H, the Hamming distance, on the same line. There is a blank line between test cases.

Output

For each dataset print a list of all possible bit strings of length N that are Hamming distance H from the bit string containing all 0's (origin). That is, all bit strings of length N with exactly H 1's printed in ascending lexicographical order.

The number of such bit strings is equal to the combinatorial symbol $C(N, H)$. This is the number of possible combinations of $N - H$ zeros and H ones. It is equal to

$$\frac{N!}{(N-H)!H!}$$

This number can be very large. The program should work for $1 \leq H \leq N \leq 16$.
Print a blank line between datasets.

Sample Input

```
1

4 2
```

Sample Output

```
0011
0101
0110
1001
1010
1100
```

Exercise 2: Optimal Slots

- https://codeforces.com/gym/102219/problem/E

E. Optimal Slots

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

The main hall of your residency is open for use by the local community and public. Since it was built on public donations, there is no charge of using it. Every weekend particularly on public holidays there are up to 50 reservations to use it for multiple events with different durations.

You have been assigned by the residents to develop a program in choosing events to get most out of allocation time per weekend and have **as short unused time as possible**. Program should find the event(s) which fill(s) the allocation time best and print it in the same sequence as appears in the reservation list.

Input
Each test case consist of a single line.

The line starts with two integer T and N which is the time allocated for the hall to be used in the particular weekend and the number of events. The next N integer are the durations of the events (as appear in the reservation list). For example from the first line in sample data: $T = 5$ hours, N, number of events $= 5$, first event lasts for 1 $hour$, second is 2 $hours$, third is 3 $hours$, fourth is 4 $hours$, and the last one is 5 $hours$.

The input process will be terminated by a line containing 0.

Output
For each line of input value, in a single line, first, output a list of integers which are the selected events duration and another integer which is the sum of the selected events duration.

If there are multiple possible list of events, events that appear earlier in the list takes priority.

67

Exercise 2: Optimal Slots

- https://codeforces.com/gym/102219/problem/E

Example

input

```
5 5 1 2 3 4 5
10 9 11 9 3 5 8 4 9 3 2
16 8 12 6 11 11 13 1 10 7
13 5 10 12 2 13 10
28 14 18 19 26 15 18 24 7 21 14 25 2 12 9 6
0
```

output

```
1 4 5
3 5 2 10
6 10 16
13 13
19 7 2 28
```

68

Contents

3. Branch and bound

3. Branch and bound

3.1. Combinatorial optimization problem

- In many practical application problems of combinatorics, each configuration is assigned to a value equal to the rating of the worth using of the configuration for a particular use purpose.
- Then it appears the problem: Among possible combination configurations, determine the one that the worth using is the best. Such kind of problems is called the combinatorial optimization problem.

Example: The assignment problem: there are a number of *artists* and a number of *pictures*. Any artist can be assigned to perform any picture, incurring some *cost* that may vary depending on the artist-picture assignment. Find a way to assign pictures to artists such that the *total cost* of the assignment is minimized.

Example: The nurse scheduling problem

It involves the assignment of shifts and holidays to nurses. Each nurse has their own wishes and restrictions, as does the hospital. The problem is described as finding a schedule that both respects the constraints of the nurses and fulfills the objectives of the hospital. Conventionally, a nurse can work 3 shifts: day shift, night shift, late night shift.

Some constraints are:

- A nurse does not work the day shift, night shift and late night shift on the same day.
- A nurse may go on a holiday and will not work shifts during this time.
- A nurse does not do a late night shift followed by a day shift the next day.

State the combinatorial optimization problem

The combinatorial optimization problem in general could be stated as follows:

Find the min (or max) of function
$f(x) \rightarrow$ min (max),
with condition:
$x \in D$,
where D is a finite set.

Terminologies:

- $f(x)$ – objective function of problem,
- $x \in D$ - a solution
- D – set of solutions of problem.
- Set D is often described as a set of combinatorial configurations that satisfy given properties.
- Solution $x^* \in D$ having minimum (maximum) value of the objective function is called ***optimal solution***, and the value $f^*=f(x^*)$ is called ***optimal value*** of the problem.

3. Branch and bound

3.2. Some examples of combinatorial optimization problem

Traveling Salesman Problem – TSP
(Bài toán người du lịch)

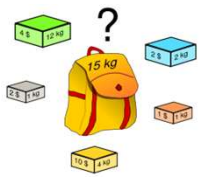
- A salesman wants to travel n cities: 1, 2, 3,…, n.
- *Itinerary is a way of starting from a city, and going through all the remaining cities, each city exactly once, and then back to the starting city.*
- Given c_{ij} is the cost of going from city i to city j ($i, j = 1, 2, ..., n$),
- Find the itinerary with minimum total cost.

3.2. Some examples of combinatorial optimization problem

1. Traveling Salesman Problem
2. **Knapsack Problem**
3. Bin Backing
4. Vehicle Routing Problem
5. Scheduling
6. Timetabling
7. Resource allocations

Example 2. Knapsack Problem (Bài toán cái túi)

- Problem Definition
  - Want to carry essential items in one bag
  - Given a set of items, each has
    - An weight (i.e., 12kg)
    - A value (i.e., 4$)



- Goal
  - To determine the number of each item to include in the bag so that
    - The total weight is less than some given weight that the bag can carry
    - And the total value is as large as possible

Example 2. Knapsack Problem (Bài toán cái túi)

- Three Types:
  - 0/1 Knapsack Problem
    - restricts the number of each kind of item to zero or one
  - Bounded Knapsack Problem
    - restricts the number of each item to a specific value
  - Unbounded Knapsack Problem
    - places no bounds on the number of each item
- Complexity Analysis
  - The general knapsack problem is known to be NP-hard
    - No polynomial-time algorithm is known for this problem

0/1 Knapsack Problem

- Problem: John wishes to take n items on a trip
  - The weight of item i is w_i and items are all different
  - The items are to be carried in a knapsack whose weight capacity is c
    - When sum of item weights ≤ c, all n items can be carried in the knapsack
    - When sum of item weights > c, some items must be left behind

Which items should be taken/left?

3.2. Some examples of combinatorial optimization problem

1. Traveling Salesman Problem
2. Knapsack Problem
3. **Bin Backing**
4. Vehicle Routing Problem
5. Scheduling
6. Timetabling
7. Resource allocations

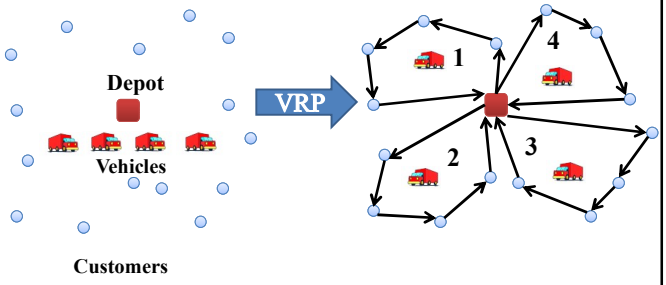
Example 3: Bin packing (Bài toán đóng thùng)

- Given n items: the weight of items are w_1, w_2, ..., w_n. Need to find a way to place these n items into bins of same size b such that the number of bins used is minimal.

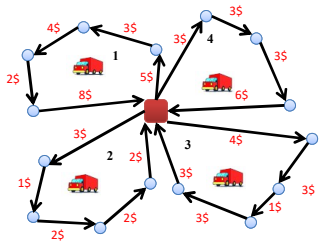
3.2. Some examples of combinatorial optimization problem

1. Traveling Salesman Problem
2. Knapsack Problem
3. Bin Backing
4. **Vehicle Routing Problem**
5. Scheduling
6. Timetabling
7. Resource allocations

Vehicle Routing Problem



Vehicle Routing Problem



Input:

- Each customer *i* request to deliver a demand of d_i (kg)
- Cost to travel from i to j is c_{ij}
- Time to travel from i to j is t_{ij}
- Capacity of vehicle: Q (example: 3000kg)
- The duration of working: D hours (example: 4 hours).
- Each customer is serviced once and by exactly 1 vehicle.

Output: Solution with minimum travel cost

Assume we have solution as in the figure, then the travel cost is:

- Vehicle 1: 5 + 3 + 4 + 2 + 8 = 22$
- Vehicle 2: 3 + 1 + 2 + 2 + 2 = 10$
- Vehicle 3: 4 + 3 + 1 + 3 + 3 = 14$
- Vehicle 4: 3 + 3 + 3 + 6 = 15$

Total cost = 22 + 10 + 14 + 15 = 71$

Brute force: Method description

- One of the most obvious methods to solve the combinatorial optimization problem is: On the basis of the combinatorial enumeration algorithms, we go through each solution of the problem, and for each solution, we calculate its value of objective function; then compare values of objective functions of all solutions to find the optimal solution whose objective function is minimal (maximal).
- The approach based on such principles is called the brute force (phương pháp duyệt toàn bộ).

Example: 0/1 knapsack problem

- Consider 0/1 knapsack problem

$$\max\{f(x)=\sum_{j=1}^{n} p_j x_j : x \in D\},$$

$$\text{where } D=\{x=(x_1,x_2,...,x_n)\in A^n : \sum_{j=1}^{n} w_j x_j \le c\}$$

- p_j, w_j, c are positive integers, j=1,2,…, n.
- Need algorithm to enumerate all elements of set D

Backtracking: enumerate all possible solutions

- **Construct set S_k:**
  - S_1={ 0, t_1 }, where t_1=1 if $c \geq w_1$; t_1 = 0, otherwise
  - Assume the current partial solution is (x_1, …, x_{k-1}). Then:
    - The remaining capacity of the bag is:

      $c_{k-1} = c - w_1x_1 - \ldots - w_{k-1}x_{k-1}$
    - The value of items already in the bag is:

      $f_{k-1} = p_1x_1 + \ldots + p_{k-1}x_{k-1}$

  Therefore: S_k = {0, t_k}, where t_k=1 if $c_{k-1} \geq w_k$; t_k = 0, otherwise
- **Implement S_k?**

```
for (y = 0; y++; y <= tk)
```

Program in pseudo code

```
int x[20], xopt[20], p[20], w[20];
int  n,b, ck, fk, fopt;

void InputData ( )
{
    <Enter value of n, p, w, b>;
}
void PrintSolution ( )
{
   <Optimal solution: xopt;
    Optimal value of objective
   function: fopt>;
}

int main( )
{
   InputData( );
   ck=c;
   fk= 0;
   fopt= 0;
   BackTrack(1);
    PrintSolution( );
}
```

```
void  Try(int k)
{
   int j, t;
  if (ck>=w[k])  t=1  else t=0;      //if: Remaining weight of bag >= weight of item i
  for (j= t; j--; j>=0)
  {
     x[k] = j;//j = 1→ x[k] = 1 tức là cho đồ vật k vào túi; j = 0→ x[k] = 0 tức là không cho đồ vật k vào túi
     ck= ck - w[k]*x[k];   //bk: trọng lượng còn lại của túi
     fk= fk + p[k]*x[k];   //fk: giá trị hiện tại của túi
     if  (k == n)   //nếu tất cả n đồ vật đều đã được xét
     {
        if (fk>fopt) {  //nếu giá trị hiện tại của túi > giá trị tốt nhất của túi hiện có
              xopt:=x; fopt:=fk;   //Cập nhật giá trị tốt nhất
        }
     }
     else Try(k+1);              //xét tiếp đồ vật thứ k+1
     ck = ck+w[k]*x[k];
     fk = fk - p[k]*x[k];
  }
}
```

Comment

- Brute force is difficult to do even on the most modern super computer. Example to enumerate all

15! = 1 307 674 368 000

permutations on the machine with the calculation speed of 1 billions operations per second, and if to enumerate one permutation requires 100 operations, then we need 130767 seconds > 36 hours!

20! ===> 7645 years

Comment

- However, it must be emphasized that in many cases (for example, in the traveling salesman problem, the knapsack, bin backing problem), we have not found yet any effective methods other than the brute force.

Comment

- Then, a problem arises that in the process of enumerating all solutions, we need to make use of the found information to eliminate solutions that are definitely not optimal.
- In the next section, we will look at such a search approach to solve the combinatorial optimization problems. In literature, it is called Branch and bound algorithm (Thuật toán nhánh cận).

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

3. Branch and bound

3. Branch and bound (Thuật toán nhánh cận)

3.1. General diagram

- Branch and bound algorithm consists of 2 procedures:
  - Branching Procedure (Phân nhánh)
  - Bounding Procedure (Tính cận)

- **Branching procedure:** The process of partitioning the set of solutions into subsets of size decreasing gradually until the subsets consists only one element. (*Quá trình phân hoạch tập các phương án ra thành các tập con với kích thước càng ngày càng nhỏ cho đến khi thu được phân hoạch tập các phương án ra thành các tập con một phần tử)*
- **Bounding procedure:** It is necessary to give an approach to calculate the bound for the value of the objective function on each subset A in the partition of the set of solutions. (*Cần đưa ra cách tính cận cho giá trị hàm mục tiêu của bài toán trên mỗi tập con A trong phân hoạch của tập các phương án.)*

3.1. General diagram

- We will describe the idea of algorithm on the model of the following general combinatorial optimization problem:

  $$\min \{ f(x) : x \in D \},$$

  where D is the finite set.
- Assume set D is described as following:

  $$D = \{x = (x_1, x_2, ..., x_n) \in A_1 \times A_2 \times ... \times A_n:$$

  $$x \text{ satisfies property } P\},$$

where $A_1, A_2, ..., A_n$ are finite set, and P is property on the Descartes product $A_1 \times A_2 \times ... \times A_n$.

Comment

- The requirement about describe the set D is to be able to use the backtracking algorithm to enumerate all solutions of the problem.
- Problem

  $$\max \{f(x): x \in D\}$$

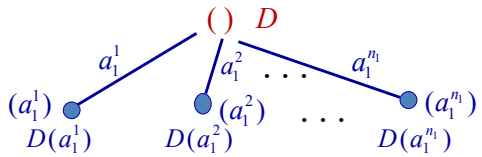
  is the equivalent of the problem

  $$\min \{g(x): x \in D\}, \text{ where } g(x) = -f(x)$$

Therefore, we can limit to considering the minimize problem

Branching procedure

Branching procedure can implement by using backtracking:



where $D(a_1^i) = \{x \in D: x_1 = a_1^i\}, i = 1,2,...,n_1$

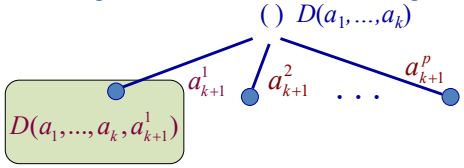
is the set of solutions which can be obtained from partial solution (a_1^i)

We have the partition:

$$D = D(a_1^1) \cup D(a_1^2) \cup ... \cup D(a_1^{n_1})$$

Branching

- Branching can be described as following:



➢ We have partition: $D(a_1,...,a_k)=\bigcup_{i=1}^{p} D(a_1,...,a_k,a^i_{k+1})$

$D(a_1,..., a_k)= \{ x\in D: x_i = a_i , i = 1,..., k \}$ is a subset of solutions where the first k elements of solutions are already known: $x_1 = a_1, x_2 = a_2, ..., x_k = a_k$

Bounding

- We need to determine function g defined on the set of all partial solutions that satisfies the following inequality:

$g(a_1,..., a_k) \le \min\{f(x): x\in D(a_1, ..., a_k)\}$ (*)

The value of objective function of all solutions having the first k elements as $(a_1, a_2,...,a_k)$

For each k-level partial solution $(a_1, a_2, ..., a_k)$, $k = 1, 2, ...$

- The inequality (*) means that the value of g of partial solution $(a_1, a_2, ..., a_k)$ is not greater than the minimum value of objective function of solution set $D(a_1, ..,a_k)$

$D(a_1,..., a_k)= \{ x\in D: x_i = a_i , i = 1,..., k \}$,

In other word, $g(a_1, a_2, . . . , a_k)$ is the **lower bound** of the value of objective function of solution set $D(a_1, a_2, ..., a_k)$.

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Cut branch by using lower bound

- Assume we already have function g defined as above. We will use this function to reduce the amount of searching during the process to consider all possible solutions in the backtracking algorithm.
- In the process to enumerate solutions, assume we already obtain some solutions. Thus, denote x^* the solution with objective function is minimum among all solutions obtained so far, and denote $f^* = f(x^*)$
- We call
  - x^* is the current best solution (optimal solution),
  - f^* is the current best value of objective function (optimal objective value).

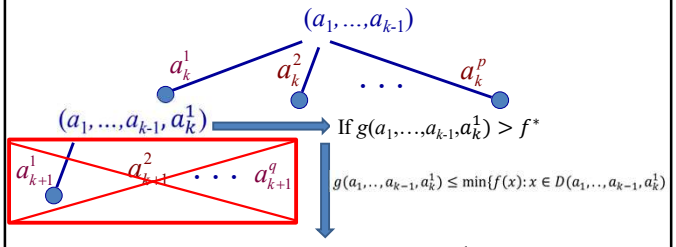
NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Cut branch by using lower bound

f^* is the current best objective value

If $g(a_1,...,a_{k-1},a^2_k) > f^*$

If $g(a_1,...,a_{k-1},a^p_k) > f^*$



→all solutions with first k elements as $(a_1,...,a_{k-1},a^1_k)$ certainly have the objective value > f^* ➔ we do not need to browse this branch

$g(a_1,..., a_k)$ is lower bound of partial solution $(a_1,..., a_k)$

Branch and bound

```
void  Try(int k)
{
    //Construct x_k from partial solution (x_1, x_2, ..., x_{k-1})
   for  a_k∈A_k
      if   (a_k∈S_k )
      {
        x_k =  a_k;
        if   (k == n)  <Update Record>;
        else if   (g(x_1,..., x_k)  ≤ f*) Try(k+1);
       }
}

void  BranchAndBound ( )
{
   f* = +∞;
   //if you know any solution x* then set f* = f(x*)
   Try(1);
   if  (f*  <  +∞)
          <f* is the optimal objective value, x* is optimal solution >
   else  <  problem does not have any solutions >;
}
```

Note that if in the procedure `Try`, we replace the statement

```
if (k==n) <Update record>;
else
if(g(x_1,...,x_k) ≤ f*)  Try(k+1);
```

by

```
if (k==n) <Update record>;
else Try(k+1);
```

then the algorithm is the **exhaustive search** (not Branch and bound anymore)

Note:

$$g(a_1,..., a_k) \le \min\{f(x): x \in D(a_1, ..., a_k)\} \quad (*)$$

The construction of g function depends on each specific combinatorial optimization problem. Usually we try to build it so that:

- Calculating the value of g must be simpler than solving the combinatorial optimization problem on the right side of (*).
- The value of $g(a_1, ..., a_k)$ must be close to the value of the right side of (*).

Unfortunately, these two requirements are often contradictory in practice.

3. Branch and bound

3.1. General diagram

3.2. Example

3.2.1. Traveling salesman problem

3.2.2. Knapsack problem

3.2.3. Assignment problem

Sir William Rowan Hamilton
1805 - 1865

Traveling Salesman Problem – TSP
(Bài toán người du lịch)

- A salesman wants to travel n cities: 1, 2, 3,…, n.
- *Itinerary is a way of starting from a city, and going through all the remaining cities, each city exactly once, and then back to the starting city.*
- Given c_{ij} is the cost of going from city i to city j (i, j = 1, 2, ..., n),
- Find the itinerary with minimum total cost.

3.2.1. Traveling salesman problem

Fix the starting city as city 1, the TSP leads to the problem:

- Determine the minimum value of

$f(1,x_2,\ldots, x_n) = c[1,x_2]+c[x_2,x_3]+\ldots+c[x_{n-1},x_n] + c[x_n,1]$

where

$(x_2, x_3, \ldots, x_n)$ is permutation of natural numbers $2, \ldots, n$.

Lower bound function (Hàm cận dưới)

- Denote

$c_{min} = \min \{ c[i, j] , i, j = 1, 2, \ldots, n, i \neq j \}$

the smallest cost between all pairs of cities.

- We need to evaluate the lower bound for the partial solution $(1, x_2, \ldots, x_k)$ corresponding to the partial journey that has passed through k cities

$1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_{k-1} \rightarrow x_k$

Lower bound function

- The cost need to pay for this partial solution is

$\sigma = c[1,x_2] + c[x_2, x_3] + \ldots + c[x_{k-1}, x_k]$.

- To develop it as the complete journey:

$1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_{k-1} \rightarrow x_k \rightarrow x_{k+1} \rightarrow x_{k+2} \rightarrow \ldots\ldots \rightarrow x_n \rightarrow 1$

(1) (2) (3)

Cost: σ Cost: $(n\text{-}k+1)c_{min}$

We still need to go through n-k+1 segments, each segment with the cost at least c_{min}, thus the lower bound of the partial solution $(1, x_2, \ldots, x_k)$ can be calculated by the formula:

$g(1, x_2, \ldots, x_k) = \sigma + (n\text{-}k+1)\, c_{min}$

Implementation

```
void  BranchAndBound()
{
    f* = +∞; cmin = min{c_ij : 1 ≤i,j ≤n}
    for (v = 1; v<=n;v++) visited[v] = FALSE;
    f=0; x_1 = 1; visited[x_1] = TRUE;
    Try(2);
    return f*;
}
```

```
void  BranchAndBound ( )
{
   f* = +∞;
   //if you know any solution x* then set f* = f(x*)
   Try(1);
   if  (f*  <  +∞)
          <f* is the optimal objective value, x* is optimal solution >
   else  <  problem does not have any solutions >;
}
```

Implementation

```
void  Try(int k)
{
    for (int v = 1; v<=n;v++) {
       if (visited[v] == FALSE) {
              x_k = v; visited[v] = TRUE;
              f = f + c(x_{k-1},x_k);
              if (k == n) //Update record
              {  if (f + c(x_n,x_1) < f*) f* = f + c(x_n,x_1); }
              else {
                     g = f + (n-k + 1)*cmin; //calculate bound
                     if (g < f* ) Try(k + 1);
                   }
               f = f - c(x_{k-1},x_k);
               visited[v] = FALSE;
        }//end if
    }//end for
}
```

```
void  Try(int k)
{
   //Construct x_k from partial solution (x_1, x_2, ..., x_{k-1})
   for  a_k∈A_k
      if   (a_k∈S_k )
      {
        x_k =  a_k;
        if   (k == n)  <Update Record>;
        else if   (g(x_1,..., x_k)  ≤ f*) Try(k+1);
      }
}
```

3. Branch and bound

3.1. General diagram

3.2. Example

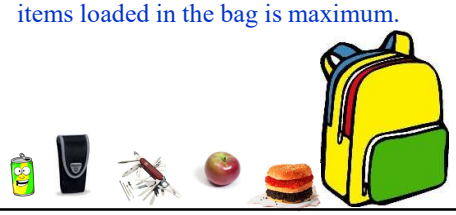
3.2.1. Traveling salesman problem

3.2.2. Knapsack problem

3.2.3. Assignment problem

3.2.2. Knap sack problem (Bài toán cái túi)

- There are n types of items.
- Item type j has
  - weight w_j and
  - profit p_j ($j = 1, 2,..., n$) .
- We need to select subsets of these items to put it into the bag of capacity c such that the total profit obtained from items loaded in the bag is maximum.



3.2.2. Knap sack problem (Bài toán cái túi)

- We have the variable

  x_j – number items type j loaded in the bag, j=1,2, ..., n
- Mathematical model of problem: Find

$$f^* = \max\{f(x) = \sum_{j=1}^{n} p_j x_j : \sum_{j=1}^{n} w_j x_j \le c, x_j \in Z_+, j = 1,2,...,n\}$$

where Z_+ is the set of nonnegative integers

Knapsack problem with integer variables

- Denote D the set of solutions to the problem:

$$D = \{x = (x_1,...,x_n) : \sum_{j=1}^{n} p_j x_j \le c, x_j \in Z_+, j = 1,2,...,n\}$$

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Construct upper bound

- Assume we index the item in the order such that the following inequality is satisfied:

$$p_1 / w_1 \geq p_2 / w_2 \geq \ldots \geq p_n / w_n .$$

(it means items are ordered in descending order of profit per one unit of weight)

- To construct the upper bound function, we consider the following Knapsack continuous variables (KPC). Find

$$g^* = \max \{ f(x) = \sum_{j=1}^{n} p_j x_j : \sum_{j=1}^{n} w_j x_j \leq c, \; x_j \geq 0, \; j = 1,2,\ldots,n \}$$

Construct upper bound function

Proposition. *The optimal solution to the KPC is vecto* $(x^* = x_1^*, x_2^*, .., x_n^*)$ *where elements are determined by the formula:*

$x_1^* = c/w_1, \; x_2^* = x_3^* = \cdots = x_n^* = 0$

and the optimal value is $g^* = v_1 b / w_1$.

Proof. Consider $x = (x_1, \ldots, x_n)$ as a solution to the KPC. Then

$p_j \leq (p_1 / w_1) \, w_j , \; j = 1, 2, \ldots, n$

as $x_j \geq 0$, we have

$p_j x_j \leq (p_1 / w_1) \, w_j x_j , \; j = 1, 2, \ldots, n.$

- Therefore

$$\sum_{j=1}^{n} p_j x_j \leq \sum_{j=1}^{n} (p_1 / w_1) w_j x_j$$
$$= (p_1 / w_1) \sum_{j=1}^{n} w_j x_j$$
$$\leq (p_1 / w_1) c = g^*$$

Proposition is proved.

Calculate the upper bound

- Now we have the k-level partial solution: $(u_1, u_2, \ldots, u_k)$, then the profit of items currently loaded in the bag is

$\sigma_k = p_1 u_1 + p_2 u_2 + \ldots + p_k u_k$

and the remaining capacity of the bag is

$c_k = c - (w_1 u_1 + w_2 u_2 + \ldots + w_k u_k)$

- We have:

$$\max \{ f(x) : x \in D, x_j = u_j, j = 1,2,\ldots,k \}$$
$$= \max \{ \sigma_k + \sum_{j=k+1}^{n} p_j x_j : \sum_{j=k+1}^{n} w_j x_j \leq c_k, \; x_j \in Z_+, j = k+1, k+2, \ldots, n \}$$
$$\leq \sigma_k + \max \{ \sum_{j=k+1}^{n} p_j x_j : \sum_{j=k+1}^{n} w_j x_j \leq c_k, \; x_j \geq 0, j = k+1, k+2, \ldots, n \}$$
$$= \sigma_k + p_{k+1} c_k / w_{k+1}$$

With remaining capacity c_k: take the item (k+1)th having the most profit to put into the bag

- Thus, we can calculate the upper bound for the partial solution $(u_1, u_2, \ldots, u_k)$ by formula

$g(u_1, u_2, \ldots, u_k) = \sigma_k + p_{k+1} \, c_k / w_{k+1}$

Calculate the upper bound

- **Note:** When continuing build the $(k+1)$th element of solution, candidates for x_{k+1} are $0, 1, \ldots, [c_k / w_{k+1}]$
- Using the result of the proposition, when selecting value for x_{k+1}, we browse candidates for x_{k+1} in the descending order: $[c_k / w_{k+1}], [c_k / w_{k+1}]\text{-}1, \ldots, 1, 0$

Implementation

```
void  BranchAndBound()
{
    f* = +∞;
    Sort items in the order p1/w1 ≥ p2/w2 ≥ …. ≥ pn/wn
    ck=c; //Remaining capacity of the bag
    fk=0; //Current value of the bag
    Try(1);
    return f*;
}
```

```
void  BranchAndBound ( )
{
   f* = +∞;
   //if you know any solution x* then set f* = f(x*)
   Try(1);
   if  (f*  <  +∞)
            <f* is the optimal objective value, x* is optimal solution >
   else  <  problem does not have any solutions >;
}
```

```
void  Try(int k)
{
   int j, t;
   if (ck >= w[k])  t=1  else t=0;    //if: Trọng lượng còn lại của túi >= trọng lượng đồ vật k
   for (j= t;  j>=0;j--)
   {
      x[k] = j;//j = 1 → x[k] = 1 tức là cho đồ vật k vào túi; j = 0→ x[k] = 0 tức là ko cho đồ vật k vào túi
      ck= ck-w[k]*x[k];    //bk: trọng lượng còn lại của túi
      fk= fk + p[k]*x[k];    //fk: giá trị hiện tại của túi
      if  (k == n)   //nếu tất cả n đồ vật đều đã được xét
      {
         if (fk>f*) {      //nếu giá trị hiện tại của túi > giá trị tốt nhất của túi hiện có
               x*=x; f*=fk;          //Cập nhật giá trị tốt nhất
         }
      }
      else Try(k+1); //xét tiếp đồ vật thứ k+1          else{
      ck= ck+w[k]*x[k];                                      g = fk + p[k+1] * (ck/w[k+1]);
      fk= fk - p[k]*x[k];                                    if (g >= f*) Try(k+1);
   }                                                       }
}
```

3. Branch and bound

3.1. General diagram

3.2. Example

3.2.1. Traveling salesman problem

3.2.2. Knapsack problem

3.2.3. Assignment problem

3.2.3. Assignment Problem(Bài toán giao việc)

- State the problem: there are n jobs and n workers. The cost that worker i does the job j is c_{ij}. It is necessary to assign these n jobs to n workers such that each worker does only 1 job and each job is done by only 1 worker. Assign each job to each worker so that the total cost is minimal.

Example: n = 4. The optimal solution is (A:job2, B: job1, C: job3, D:job4), and total cost = 2 + 6 + 1 + 4 = 13

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	**6**	4	3	7
C	5	8	**1**	8
D	7	6	9	**4**

3.2.3. Assignment Problem(Bài toán giao việc)

- Brute force: $n!$
- Branch and bound:
  - Calculate the lower bound:
    - Method 1: Each worker is assigned the least-cost job which nobody does it yet (each line finds the element with the smallest value).

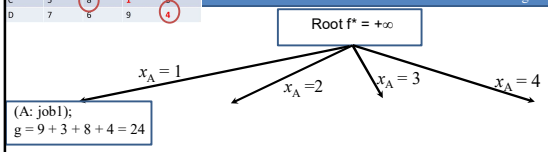
      Example: (A: job2, B: job 3, C: job1, D: job4): is a solution of the problem
    - Method 2: Each job is assigned to the worker with the minimum cost (Each column finds the element that has not been assigned to anyone with the smallest value)

      Example: (Job1: C, Job2: A, Job3: B, Job4: D): is a solution of the problem

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	**6**	4	3	7
C	5	8	**1**	8
D	7	6	9	**4**

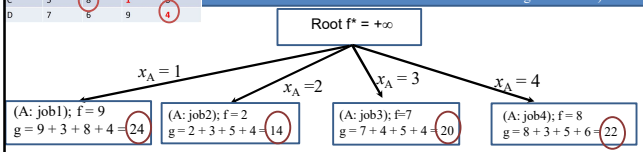
Consider the lower bound calculated by 1st method

Each worker is assigned the least-cost job which nobody does it yet (each line finds the element that has the smallest unassigned element)



Consider the lower bound calculated by 1st method

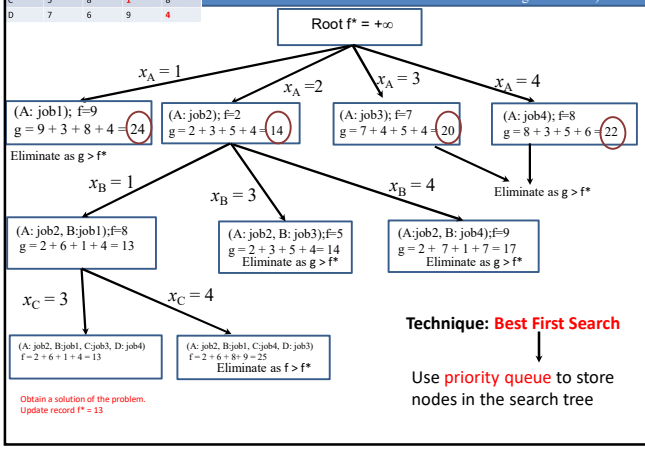
Each worker is assigned the least-cost job which nobody does it yet (each line finds the element that has the smallest unassigned element)



Technique: Best First Search

Consider the lower bound calculated by 1st method

Each worker is assigned the least-cost job which nobody does it yet (each line finds the element that has the smallest unassigned element)



Technique: Best First Search

Use priority queue to store nodes in the search tree

3.2.3. Assignment Problem(Bài toán giao việc)

```
#include <bits/stdc++.h>
using namespace std;
#define N 4

// state space tree node
struct Node
{
    // stores parent node of current node
    // helps in tracing path when answer is found
    Node* parent;
    //Thong tin tai nut hien tai:
    int f; //Chi phi tinh den nut hien tai
    int lowerbound; //Can duoi cho nut hien tai
    int workerID; //workerID gán tai nut hien tai
    int jobID; //jobID gan tai nut hien tai
    bool assigned[N]; /*tai nut hien tai neu job i chua duoc gan
      cho worker nao thi assigned[i] =true; nguoc lai  = false*/
};
```


3.2.3. Assignment Problem(Bài toán giao việc)

```
//Tao nut moi tren cay: gan cong nhan worker cho cong viec job
Node* newNode(int worker, int job, bool assigned[], Node* parent)
{
    Node* node = new Node;

    for (int j = 0; j < N; j++) node->assigned[j] = assigned[j];
    node->assigned[job] = true;

    node->parent = parent;
    node->workerID = worker;
    node->jobID = job;

    return node;
}
```


3.2.3. Assignment Problem(Bài toán giao việc)

```
/*Tinh Lowerbound cho nut hien tai (sau khi da gan cong nhan worker lam cv job):
 of node after worker x is assigned to job y. */
int calculateLowerbound(int costMatrix[N][N], int worker,
                        int job, bool assigned[])
{   //Mang assigned danh dau nhung cong viec nao chua duoc thuc hien o nut hien tai:
    int lowerbound = 0;
    bool available[N] = {true};// to store unavailable jobs
    // start from next worker
    for (int i = worker + 1; i < N; i++)
    {
        //Voi moi cong nhan i: gan cho no cong viec chua duoc ai thuc hien ma co chi phi nho nhat
        int min = INT_MAX, minIndex = -1;
        // do for each job
        for (int j = 0; j < N; j++)
        {
            // if job is unassigned
            if (!assigned[j] && available[j] && costMatrix[i][j] < min)
            {
                minIndex = j;   // store job number
                min = costMatrix[i][j];  // store cost
            }
        }
        lowerbound += min;// add cost of next worker
        available[minIndex] = false; // job becomes unavailable
    }
    return lowerbound;
}
```


3.2.3. Assignment Problem(Bài toán giao việc)

```
/* Ham so sanh su dung tren priority queue:
nut nao co lowerbound nho hon thi duoc uu tien hon */
struct comp
{
    bool operator()(const Node* lhs,
                    const Node* rhs) const
    {
        return lhs->lowerbound > rhs->lowerbound;
    }
};

// In loi giai
void printSolution(Node *min)
{
    if(min->parent==NULL) return;

    printSolution(min->parent);
    cout << "Gan cong nhan " << char(min->workerID + 'A')
         << " thuc hien cong viec " << min->jobID << endl;

}
```

3.2.3. Assignment Problem(Bài toán giao việc)

```
// Finds minimum cost using Branch and Bound.
int BranchAndBound(int costMatrix[N][N])
{
    /*Tao priority queue de luu tru cac nut tren cay
    (Create a priority queue to store live nodes of search tree)*/
    priority_queue<Node*, std::vector<Node*>, comp> pq;

    // initialize heap to dummy node with cost 0
    bool assigned[N] = {false};
    Node* root = newNode(-1, -1, assigned, NULL);
    root->f = root->lowerbound = 0;
    root->workerID = -1;

    // Add dummy node to list of live nodes;
    pq.push(root);

    // Finds a live node with least cost,
    // add its childrens to list of live nodes and
    // finally deletes it from the list.
    while (!pq.empty())
    {
      //Lay khoi queue nut co lowerbound min
      Node* min = pq.top();
      //Xoa nut do khoi queue:
      pq.pop();

      // i stores next worker
      int i = min->workerID + 1;
```


3.2.3. Assignment Problem(Bài toán giao việc)

```
    //Neu tat ca cac cong nhan deu da duoc gan viec thi in ket qua:
    if (i == N)
    {
        printSolution(min);
        return min->lowerbound;
    }

    // do for each job
    for (int j = 0; j < N; j++)
    {
      // If unassigned
      if (!min->assigned[j])
      {
        // create a new tree node
        Node* child = newNode(i, j, min->assigned, min);

        // cost for ancestors nodes including current node
        child->f = min->f + costMatrix[i][j];

        // calculate its lower bound
        child->lowerbound = child->f +
          calculateLowerbound(costMatrix, i, j, child->assigned);

        // Add child to list of live nodes;
        pq.push(child);
      }
    }
  }
}
```


3.2.3. Assignment Problem(Bài toán giao việc)

```
int main()
{
    // x-cordinate represents a Worker
    // y-cordinate represents a Job
    int costMatrix[N][N] =
    {
        {9, 2, 7, 8},
        {6, 4, 3, 7},
        {5, 8, 1, 8},
        {7, 6, 9, 4}
    };
    cout << "\n Chi phi toi uu = " << BranchAndBound(costMatrix);
    return 0;
}
```