Chapter 6. Exhaustive search

Data Structures and Algorithms

FIB

Slides by Antoni Lozano

(with minor editions by other professors)

Q1 2022–2023

Brute-force algorithms

- Many problems consist in, given a set of constraints, find an object that satisfies them (a solution)
- For example, solving a sudoku.

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

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

- There are variants:
  - find/count all solutions
  - find the best solution (optimal solution)
  - etc.

Brute-force algorithms

Usually, the only way to solve a problem is to try all possible combinations. The resulting algorithm is called brute force or exhaustive search:

- It is usually exponential.
- It can be slow, but it is better than not having any algorithm at all....
- It can be practical for small input sizes.
- It can benefit from other techniques such as divide-and-conquer.

Brute-force algorithms

We want to write all strings of zeros and ones of size n.

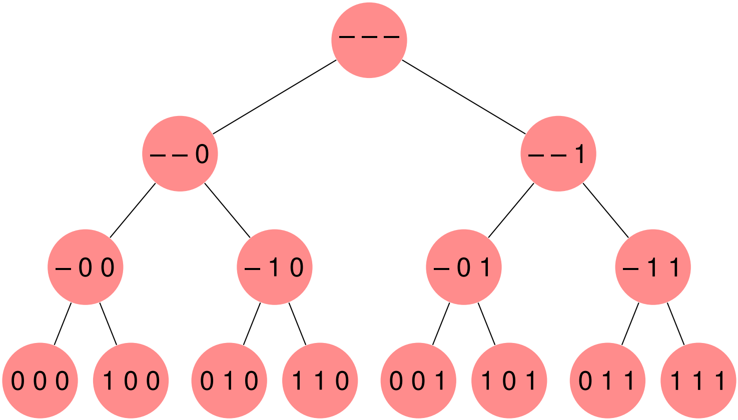
We have a procedure `write(vector<int>& A)` that writes vector `A`.
We need to call `binary(0, A)`, where $n =$ `A.size()` and `binary` is defined as follows:

```
// i is the next position in A we will assign
void binary(vector<int>& A, int i) {
  if (i == A.size()) write(A);    // base case
  else {                          // inductive case
    A[i] = 0; binary(A,i+1);
    A[i] = 1; binary(A,i+1);
} }

void binary(int n) {
  vector<int> A(n);
  binary(A,0);
}
```

Brute-force algorithms

For $n = 3$, we obtain the following recursion tree:



Leaves are solutions.

Edges express how we extend each partial solution.

Internal nodes are partial solutions.

Brute-force algorithms

What is the cost of exhaustive search?

- if there is an implicit tree or graph to be explored, they are usually exponential

- if the graph is given as input, they are polynomial

  For example, BFS and DFS are also exhaustive searchs.

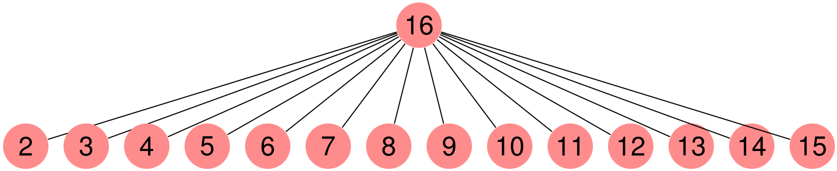
Example: prime numbers

```
bool is_prime (Integer x) {
    if (x <= 1) return false ;
    for (Integer i = 2; i < x; ++i)
        if (x % i == 0) return false ;
    return true; }
```

Maximum number of iterations: $(x-1)-2+1 = x-2$.

Cost as a function of x (the value of x): $\Theta(x)$

Cost as a function of $n = |x|$ (the length of the representation of x): $\Theta(2^n)$.



Implicit tree for $x = 16$

1 Brute-force algorithms

2 Backtracking
- String with ones
- Permutations
- Generic algorithm
- *n*-queens
- Latin square
- Knight's tour
- Knapsack problem
- Travelling Salesman Problem (TSP)
- Hamiltonian Graph

Backtracking

A backtracking algorithm can be seen as an intelligent implementation of exhaustive search with an improved cost (but still inefficient).

Example: furniture in an apartment

- Brute-force strategy: try all possible configurations of furniture in all rooms.
- The backtracking strategy uses that:
  - each piece usually goes to a concrete room
    (no sofa in the kitchen!)
  - there are pieces of furniture that go together
    (chairs and table, bed and bedside tables)
  - if a subdistribution is not satisfactory,
    we will not consider any distribution containing it

Backtracking – String with ones

Problem

We want to write all strings with zeros and ones of size n that contain exactly k ones.

- In a certain step of the algorithm, we will have a partial string and we will have to extend it in all possible ways.
- **First question:** how will we fill the string?

Backtracking – String with ones

Problem

We want to write all strings with zeros and ones of size n that contain exactly k ones.

- In a certain step of the algorithm, we will have a partial string and we will have to extend it in all possible ways.
- **First question:** how will we fill the string?
  - From left to right.

Backtracking – String with ones

Problem

We want to write all strings with zeros and ones of size n that contain exactly k ones.

- In a certain step of the algorithm, we will have a partial string and we will have to extend it in all possible ways.
- **First question:** how will we fill the string?
  - From left to right.
- **Second question:** how will we represent a partial string in C++?

Backtracking – String with ones

Problem

We want to write all strings with zeros and ones of size n that contain exactly k ones.

- In a certain step of the algorithm, we will have a partial string and we will have to extend it in all possible ways.
- **First question:** how will we fill the string?
  - From left to right.
- **Second question:** how will we represent a partial string in C++?
  - We will have a vector A of size n and an integer idx that indicates the first non-filled position.

A	0	1	1	0	?	?	?
					$\uparrow$		
					idx		

Problem

We want to write all strings with zeros and ones of size n that contain exactly k ones.

- In a certain step of the algorithm, we will have a partial string and we will have to extend it in all possible ways.
- **First question:** how will we fill the string?
  - From left to right.
- **Second question:** how will we represent a partial string in C++?
  - We will have a vector A of size n and an integer *idx* that indicates the first non-filled position.

A	0	1	1	0	?	?	?
					$\uparrow$		
					idx		

- **Third question:** given a partial string, which candidates do we have to fill position *idx*?

Backtracking – String with ones

Problem

We want to write all strings with zeros and ones of size n that contain exactly k ones.

- In a certain step of the algorithm, we will have a partial string and we will have to extend it in all possible ways.
- **First question:** how will we fill the string?
  - From left to right.
- **Second question:** how will we represent a partial string in C++?
  - We will have a vector A of size n and an integer *idx* that indicates the first non-filled position.

A	0	1	1	0	?	?	?
					↑		
					idx		

- **Third question:** given a partial string, which candidates do we have to fill position *idx*? 0 and 1.

Backtracking – String with ones

```
// A: partial string (size n)
// idx: first non-filled cell in A
// k: total number of 1s we want
void strings(vector<int>& A, int idx, int k) {
  if (idx == A.size()) {
    int c = 0;
    for (int x : A) c += x; // Counts 1s
    if (c == k) write(A);
  }
  else {
    A[idx] = 0;  strings(A, idx+1, k);
    A[idx] = 1;  strings(A, idx+1, k);
} }

int main(){
  int n, k;      cin >> n >> k;
  vector<int> A(n);
  strings(A,0,k);
}
```

- Do we really need to count from scratch the number of 1's in A every time?
  - We can keep, at every moment, the number of 1s of the partial string
  - That forces us to use one more parameter in the procedure

Backtracking – String with ones

```
// A: partial string (size n)
// idx: first non-filled cell in A
// u: number of 1s in A[0...idx-1] (already filled)
// k: total number of 1s we want
void strings2(vector<int>& A, int idx, int u, int k) {
  if (idx == A.size()) {
    if (u == k) write(A);
  }
  else {
    A[idx] = 0;      strings2(A, idx+1, u,   k);
    A[idx] = 1;      strings2(A, idx+1, u+1, k);
  }
}

int main(){
  int n, k;   cin >> n >> k;
  vector<int> A(n);
  strings2(A,0,0,k);    }
```

Backtracking – String with ones

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 3$, what happens in the following situation?

0	1	1	1	?	?	?

$$\uparrow$$
$$idx$$

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 3$, what happens in the following situation?

0	1	1	1	?	?	?
				$\uparrow$		
				idx		

At position *idx* we cannot place a 1

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 3$, what happens in the following situation?

0	1	1	1	?	?	?
				$\uparrow$		
				idx		

At position *idx* we cannot place a 1

- In general, if we have placed u 1s and we want k 1s, we can only place a 1 at position *idx* if $u < k$
- We have pruned the search space.

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 5$, what happens in the following situation?

0	1	0	1	?	?	?
				$\uparrow$		
				idx		

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 5$, what happens in the following situation?

0	1	0	1	?	?	?
				$\uparrow$ *idx*		

At position *idx* we cannot place a 0

Backtracking – String with ones

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 5$, what happens in the following situation?

0	1	0	1	?	?	?

$$\begin{array}{ccccccc} & & & & \uparrow & & \\ & & & & idx & & \end{array}$$

At position *idx* we cannot place a 0

- In general, if we want k ones, we will need $n - k$ zeros. If we have placed z zeros, we can only place a 0 at position *idx* if $z < n - k$.
- This is another way to prune the search space.

Backtracking – String with ones

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 5$, what happens in the following situation?

0	1	0	1	?	?	?
				$\uparrow$ *idx*		

  At position *idx* we cannot place a 0
- In general, if we want k ones, we will need $n - k$ zeros. If we have placed z zeros, we can only place a 0 at position *idx* if $z < n - k$.
- This is another way to prune the search space.
- **Fifth question:** how can this be implemented efficiently?

- **Fourth question:** can we detect situations in which a partial string cannot be extended to a total string with exactly k 1s?
- If $k = 5$, what happens in the following situation?

0	1	0	1	?	?	?
				$\uparrow$ *idx*		

  At position *idx* we cannot place a 0

- In general, if we want k ones, we will need $n - k$ zeros. If we have placed z zeros, we can only place a 0 at position *idx* if $z < n - k$.
- This is another way to prune the search space.
- **Fifth question:** how can this be implemented efficiently?
- An easy way is to also keep the number of zeros we have already placed.

Backtracking – String with ones

```
// A: partial string (size n)
// idx: first non-filled cell in A
// u: number of 1s in A[0...idx-1] (already placed)
// z: number of 0s in A[0...idx-1] (already placed)
// k: total number of 1s we want
void strings3(vector<int>& A, int idx, int z, int u, int k) {
  if (idx == A.size()) write(A);
  else {
    if (z < A.size() - k) { // not all 0s placed
      A[idx] = 0;  strings3(A, idx+1, z+1,   u, k); }

    if (u < k) { // not all 1s placed
      A[idx] = 1;  strings3(A, idx+1,   z, u+1, k); }
  }
}

int main(){
  int n, k;  cin >> n >> k;
  vector<int> A(n);
  strings3(A,0,0,0,k); }
```

What about the efficiency of the three solutions? (only counting solutions, no writing)

For example, if $n = 30$

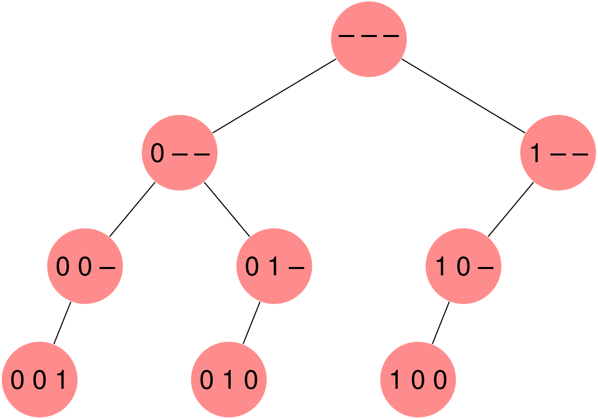
Algorithm	**Seconds (k=2)**	**Seconds (k=8)**	**Seconds (k=15)**
strings1	13.5	13.5	13.5
strings2	4	4	4
strings3	0.007	0.08	1.5

However, if we take $k = n/2$ we will observe that the third solution is also exponential in n, since it has to count $\binom{n}{n/2}$ strings.

n	10	16	22	28	34
$\binom{n}{n/2}$	252	12,870	705,432	40,116,600	2,333,606,220

Backtracking – String with ones

For $n = 3$ and $k = 1$, we have the following recursion tree:



It is better not to generate all possibilities and then check the number of ones.

But there are still an exponential number of nodes.

Backtracking – Permutations

Example: permutations of n elements

Which are the permutations of $\{1, \ldots, n\}$?

- There are n possibilities for the first element.
- Once the first is chosen, there are $n-1$ possibilities for the second one.
- Repeating the argument, we get

$$\prod_{k=1}^{n} k = n!.$$

For $n = 4$, we have the permutations:

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

Backtracking – Permutations

- In a certain step of the algorithm, we will have a partial permutation and we will have to extend it in all possible ways.
- **First question:** how will we fill the permutation?
  - From left to right.
- **Second question:** how will we represent a partial permutation in C++?
  - We will have a *A* of size *n* and an integer *idx* that indicates the first non-filled position.

A							
	3	4	6	1	?	?	?
					$\uparrow$ *idx*		

- **Third question:** given a partial permutation *A*, which candidates do we have to fill position *idx*? Elements in $\{1, 2, \ldots, n\}$ not already present in *A*.

Backtracking – Permutations

```
// n: we want permutations of {1,2,...,n}
// A: partial permutation (size n)
// idx: first non-filled cell in A
void write_permutations1(int n, vector<int>& A, int idx) {
  if (idx == A.size()) write(A);
  else {
    for (int k = 1; k <= n; ++k) {
      bool used = false; // Determine whether k has been used
      for (int i = 0; i < idx and not used; ++i)
        used = (A[i] == k);
      if (not used) {
        A[idx] = k;
        write_permutations1(n,A,idx+1);
      }  }  }  }

int main() {
  int n; cin >> n;
  vector<int> A(n);
  write_permutations1(n,A,0);
}
```

- Can we avoid computing *used* again and again?
- We can easily maintain this information with a vector *used* such that *used*[k] is true iff number k already appears in the partial permutation.
- **Careful:** this information has to be maintained also upon backtracking.
- If we count permutations of 12 elements (no writing), this improvement allows us to go from 112 to 30 segons.

Backtracking – Permutations

```
void write_permutations2(int n, vector<int>& A, int idx, vector
    <bool>& used) {
  if (idx == A.size()) write(A);
  else {
    for (int k = 1; k <= n; ++k) {
      if (not used[k]) {
        A[idx] = k;
        used[k] = true;
        write_permutations2(n,A,idx+1,used);
        used[k] = false; // restore upon backtracking
      }
    }
  }
}

int main() {
  int n; cin >> n;
  vector<int> A(n);
  vector<bool> used(n+1,false);
  write_permutations2(n,A,0,used); }
```

- **Fourth question:** can we detect situations in which a partial permutation cannot be extended to a total one? No.
- Hence, no additional pruning is possible. However, we have already pruned the search space when selecting the candidates for position *idx*.

Generic algorithm

We can define a generic backtracking algorithm:

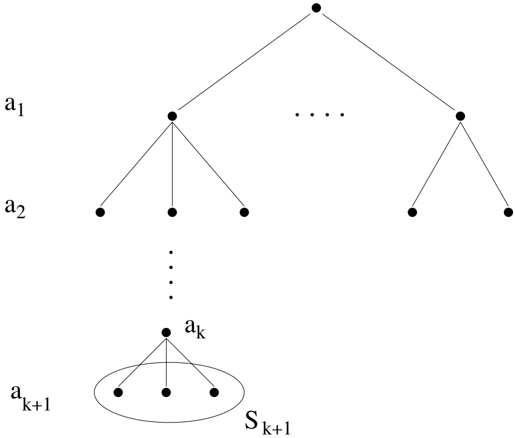
- The solution space is organized as a configuration tree.
- Each node or tree configuration is represented with a vector

$$A = (a_1, a_2, \ldots, a_k)$$

  that contains the choices already made (the partial solution).
- Vector A grows in the *forward* phase by choosing a_{k+1} from a set S_{k+1} of candidates (exploration in depth).
- A is reduced in the *backtracking* phase (backtrack).

Generic algorithm

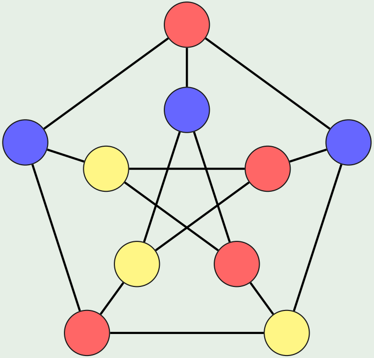
A backtracking algorithm is usually a DFS in a configuration tree:



Generic algorithm

Example: 3-Colorability

The 3-colorability problem consists of determining whether we can assign one of 3 available colors to each node such that adjacent vertices have different colors.

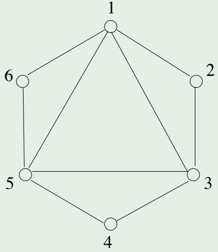


3-coloring of Petersen's graph

Generic algorithm

3-colorability

Given the graph



- Configurations are partial color assignments, i.e.,

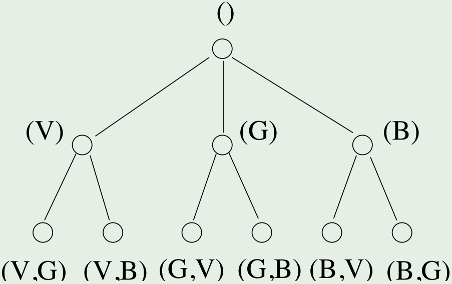
$$A = (a_1, a_2, \ldots, a_k)$$

  will represent that vertex i has color $a_i \in \{B, G, V\}$.

- The candidate set S_{k+1} for a_{k+1} will contain all colors compatible with the already colored neighbors.

Generic algorithm

The first 3 levels of the configuration tree would be:



But if we only want to find a solution,

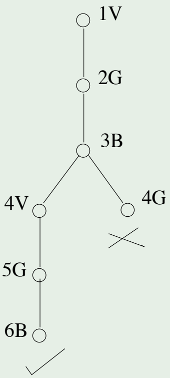
- we can fix a color for vertex 1
- we can fix a different color for vertex 2
- any other solution will be symmetric

Generic algorithm

Choosing

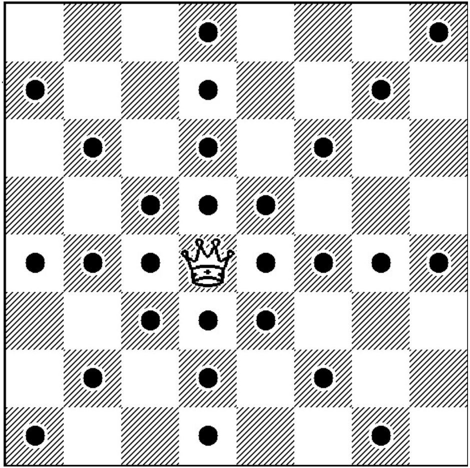
- $S_1 = \{V\}$
- $S_2 = \{G\}$

and defining $S_{k+1} = \{c \in \{V, G, B\} \mid \forall i \leq k \ \ (\{i, k+1\} \in A(G) \Rightarrow c \neq a_i)\}$, we obtain the configuration tree



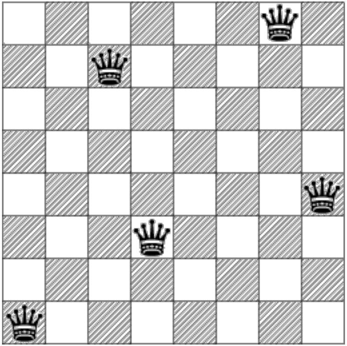
n-queens

Queen movements in chess:



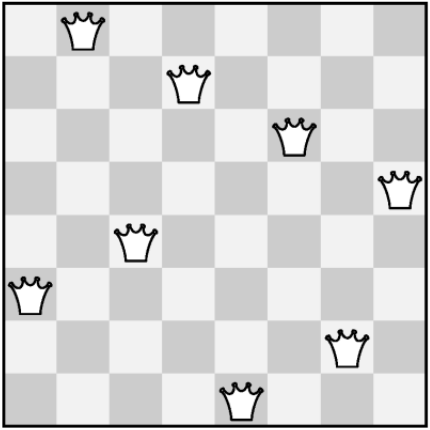
n-queens

How many queens can we place in a chessboard
so that no two queens threaten each other?
5? 6? 7? 8?



8-queens problem

Placing 8 queens in a chessboard so that no two queens threaten each other.



Brute-force solving strategies:

1. Choose 8 different positions in the board.

$$\binom{64}{8} = 4.426.165.368 \text{ configurations}$$

2. Choose 8 positions in different rows.

$$8^8 = 16.777.216 \text{ configurations}$$

3. Choose 8 positions in different rows and columns.

$$8! = 40.320 \text{ configurations}$$

With backtracking one can still do better.

n-queens

We will consider the generalized problem of the n-queens.

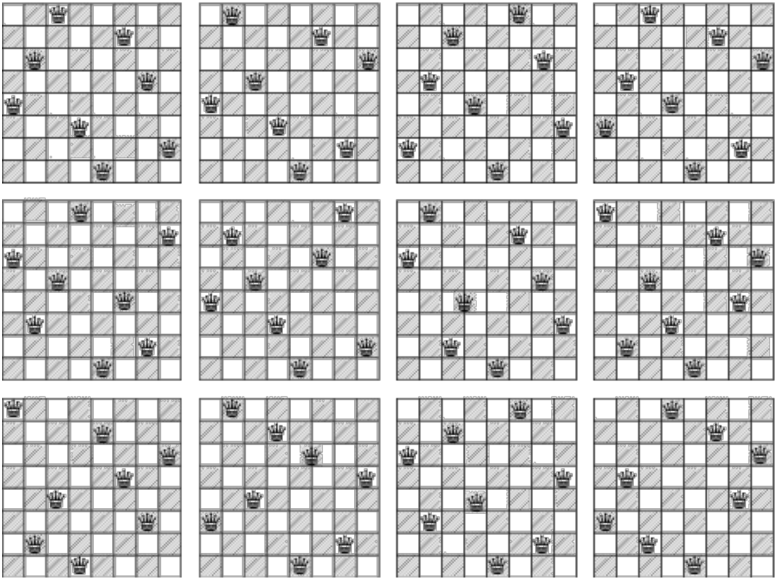
n-queens problem

Place n queens in an $n \times n$ chessboard so that no two queens threaten each other.

Number of non-isomorphic solutions (by rotation or reflection) of the *n*-queens problem:

n	solutions
1	1
2	0
3	0
4	1
5	2
6	1
7	6
8	12
9	46
10	92

The 12 non-isomorphic solutions for $n = 8$



n-queens

First implementation:

- finds all solutions
- with backtracking
- extends the partial solution whenever it is “legal”
  (can be extended to a complete solution)
- worst-cast cost in time: $\Theta(n^n)$

We will implement the position of the queens with a vector

```
vector<int> t;
```

that indicates that the queen in row i is at column $t[i]$.

n-queens

```
void write() {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j)
      cout << (t[i] == j ? "Q" : ".");
    cout << endl;
  }
  cout << endl;
}
```

In order to know whether queens in rows i and k ($i \neq k$) attack each other, note that their cells are $(i, t[i])$ and $(k, t[j])$.

- column, check whether $t[i] = t[k]$
- diagonal type 1 ($\searrow$), check whether $t[i] - i = t[k] - k$
- diagonal type 2 ($\nearrow$), check whether $t[i] + i = t[k] + k$

	0	1	2	3	4
0	E	F	G	H	I
1	D	E	F	G	H
2	C	D	E	F	G
3	B	C	D	E	F
4	A	B	C	D	E

Diag A: $(4, 0)$
Diag B: $(3, 0), (4, 1)$
Diag C: $(2, 0), (3, 1), (4, 2)$
Diag D: $(1, 0), (2, 1), (3, 2), (4, 3)$
Diag E: $(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)$
Diag F: $(0, 1), (1, 2), (2, 3), (3, 4)$
Diag G: $(0, 2), (1, 3), (2, 4)$
Diag H: $(0, 3), (1, 4)$
Diag I: $(0, 4)$

In order to know whether queen in rows i and k attack each other, note that their cells are $(i, t[i])$ and $(k, t[k])$.

- column, check whether $t[i] = t[k]$
- diagonal type 1 ($\searrow$), check whether $t[i] - i = t[k] - k$
- diagonal type 2 ($\nearrow$), check whether $t[i] + i = t[k] + k$

	0	1	2	3	4
0	A	B	C	D	E
1	B	C	D	E	F
2	C	D	E	F	G
3	D	E	F	G	H
4	E	F	G	H	I

Diag A: $(0, 0)$
Diag B: $(1, 0), (1, 1)$
Diag C: $(2, 0), (1, 1), (0, 2)$
Diag D: $(3, 0), (2, 1), (1, 2), (0, 3)$
Diag E: $(4, 0), (3, 1), (2, 2), (1, 3), (0, 4)$
Diag F: $(4, 1), (3, 2), (2, 3), (1, 4)$
Diag G: $(4, 2), (3, 3), (2, 4)$
Diag H: $(4, 3), (3, 4)$
Diag I: $(4, 4)$

```
bool legal(int i) {
  for (int k = 0; k < i; ++k)
    if (t[k]      == t[i]     or
        t[k] - k == t[i] - i or
        t[k] + k == t[i] + i)
      return false;
  return true;
}

void queens(int i) {
  if (i == n) write();
  else
    for (int j = 0; j < n; ++j){ // j is col for queen of row i
      t[i] = j;
      if (legal(i))
        queens(i+1);
    }
}
```

Second implementation:

- finds all solutions
- with backtracking
- extends partial solution whenever it is “legal”
  (can be extended to a complete solution)
- with marking (book-keeping)
- worst-case cost in time: $\Theta(n^n)$

MARKING STRATEGY:

- No need to mark used rows (one queen per row by construction)
- Easy to mark used columns (vector of Booleans of size n)
- Diagonals?

	0	1	2	3	4
0	A	B	C	D	E
1	B	C	D	E	F
2	C	D	E	F	G
3	D	E	F	G	H
4	E	F	G	H	I

Diag A: $(0, 0)$
Diag B: $(1, 0), (1, 1)$
Diag C: $(2, 0), (1, 1), (0, 2)$
Diag D: $(3, 0), (2, 1), (1, 2), (0, 3)$
Diag E: $(4, 0), (3, 1), (2, 2), (1, 3), (0, 4)$
Diag F: $(4, 1), (3, 2), (2, 3), (1, 4)$
Diag G: $(4, 2), (3, 3), (2, 4)$
Diag H: $(4, 3), (3, 4)$
Diag I: $(4, 4)$

We have 2n - 1 diagonals

Diagonal identified by $i + j$. This gives numbers in $[0, 2n - 2]$.

MARKING STRATEGY:

- No need to mark used rows (one queen per row by construction)
- Easy to mark used columns (vector of Booleans of size n)
- Diagonals?

	0	1	2	3	4
0	E	F	G	H	I
1	D	E	F	G	H
2	C	D	E	F	G
3	B	C	D	E	F
4	A	B	C	D	E

Diag A: $(4, 0)$
Diag B: $(3, 0), (4, 1)$
Diag C: $(2, 0), (3, 1), (4, 2)$
Diag D: $(1, 0), (2, 1), (3, 2), (4, 3)$
Diag E: $(0, 0), (1, 1), (2, 2), (3, 3)$
Diag F: $(0, 1), (1, 2), (2, 3), (3, 4)$
Diag G: $(0, 2), (1, 3), (2, 4)$
Diag H: $(0, 3), (1, 4)$.
Diag I: $(0, 4)$

We have 2n - 1 diagonals

Diagonal identified by $i - j$. This gives numbers in $[-(n-1), n-1]$.

Instead identify by $i - j + (n - 1)$. This gives number in $[0, 2n - 2]$.

n-queens

```
#include <iostream>
#include <vector>

using namespace std;

int n;
vector<int> t;
// mc[j] == queen at column j
// md1[k] == queen at diagonal i+j = k, etc.
vector<int> mc, md1, md2;

void queens(int i);

int main() {
  cin >> n;
  t = vector<int>(n);
  mc = vector<int>(n, false);
  md1 = md2 = vector<int>(2*n-1, false);
  queens(0);
}
```

n-queens

```
int diag1(int i, int j) { return i+j; }
int diag2(int i, int j) { return i-j + n-1; }

void queens(int i) {
  if (i == n) write();
  else
    for (int j = 0; j < n; ++j) // j is col for queen of row i
      if (not mc[j] and
          not md1[diag1(i, j)] and
          not md2[diag2(i, j)]) {
        t[i] = j;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        queens(i+1);
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
}
```

n-queens

If we only want one solution, we can stop as soon as we find the first one:

```
 // Returns whether there is solution completing the partial
bool queens(int i) {
  if (i == n) {
    write();
    return true;
  }
  else {
    for (int j = 0; j < n; ++j)
      if (not mc[j] and
          not md1[diag1(i, j)] and
          not md2[diag2(i, j)]) {
        t[i] = j;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        if (queens(i+1)) return true;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
    return false;
  }
}
```

n-queens

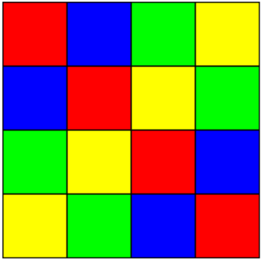
If we want to count solutions:

```
// Returns the number of sol extending the partial
int queens(int i) {
  if (i == n) {
    return 1;
  }
  else {
    int res = 0;
    for (int j = 0; j < n; ++j)
      if (not mc[j] and
          not md1[diag1(i, j)] and
          not md2[diag2(i, j)]) {
        t[i] = j;
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = true;
        res += queens(i+1);
        mc[j] = md1[diag1(i, j)] = md2[diag2(i, j)] = false;
      }
    return res;
  }
}
```

Latin square

A Latin square is any $n \times n$ grid filled with n different symbols so that each one appears once in every column and row.

1	2	3
2	3	1
3	1	2



A	B
B	A

Latin square

Number of Latin squares llatins $n \times n$ for $n \in \{1, \dots, 11\}$:

n	solutions
1	1
2	2
3	12
4	576
5	161280
6	812851200
7	61479419904000
8	108776032459082956800
9	5524751496156892842531225600
10	9982437658213039871725064756920320000
11	776966836171770144107444346734230682311065600000

Latin square

Latin square problem

Given n, find all Latin squares of size n.

Latin square

Backtracking solution with markings.
Cost: $\mathcal{O}(n^{n^2})$.

```
#include <iostream>
#include <vector>

using namespace std;

int n;

// q[i][j] == value at row i, column j
vector<vector<int>> q;

// r[i][v] == whether row i already uses value v
vector<vector<bool>> r;

// c[j][v] = whether column j already uses value v
vector<vector<bool>> c;
```

Latin square

```
void write() {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      cout << q[i][j] << '\t';
    }
    cout << endl;
  }
  cout << endl;
}
```

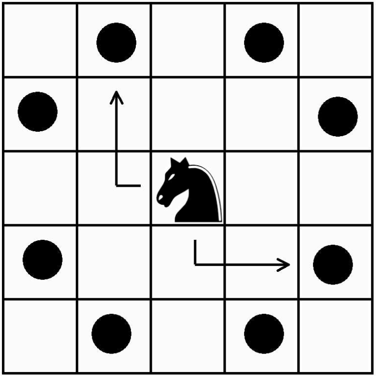
Latin square

```
// Find all Latin squares completing from (i, j)
void latin_square(int i, int j) {
  if (i == n) return write();
  if (j == n) return latin_square(i+1, 0);
  for (int v = 0; v < n; ++v) {
    if (not r[i][v] and not c[j][v]) {
      r[i][v] = c[j][v] = true;
      q[i][j] = v;
      latin_square(i, j+1);
      r[i][v] = c[j][v] = false;
    }
  }
}

int main () {
  cin >> n;
  q = vector<vector<int>>(n, vector<int>(n));
  r = c = vector<vector<bool>>(n, vector<bool>(n, false));
  latin_square(0, 0);
}
```

Knight's tour

Given an $n \times n$ chessboard and a starting cell, find, if possible, a knight's tour that starting from that cell visits all other cells with no repetitions.



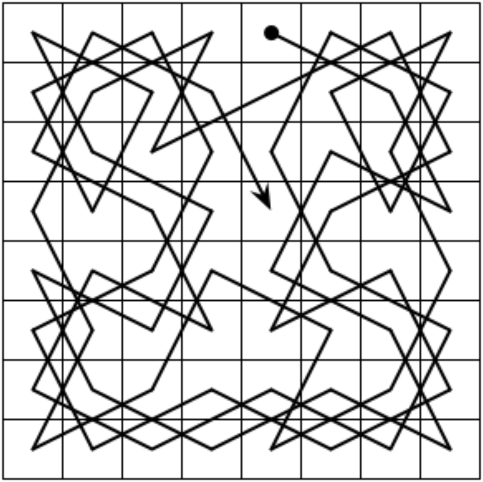
Knight's movements

Knight's tour

This problem has a long mathematical tradition. It comes from India (IX d.C.) and Euler worked on it (XVIII d.C).

- There are two versions:
  - closed: start and finish can be joined by a knight move
  - open: start and finish in arbitrary positions
- There are
  - 9.862 closed tours in a 6×6 board
  - 13.267.364.410.532 closed in a 8×8 board

Knight's tour



An open solution on a chessboard

Knight's tour

Knight's tour problem

Given an $n \times n$ board and a cell (i, j),
we want to find an open tour starting from (i, j).

Backtracking solution

Knight's tour

```
#include <iostream>
#include <vector>

using namespace std;

int n;

// t[i][j] == k if in the k-th jump we arrive at (i,j)
// -1 if we have not arrived at (i,j) yet
vector<vector<int>> t;

// Can we fill board starting at (i, j) having done s jumps?
bool possible(int i, int j, int s);

int main() {
  int i, j;
  cin >> n >> i >> j;
  t = vector<vector<int>>(n, vector<int>(n, -1));
  cout << possible(i, j, 0) << endl;
}
```

Knight's tour

```
bool possible(int i, int j, int s) {
  if (i >= 0 and i < n and j >= 0 and j < n and t[i][j] == -1){
    t[i][j] = s;
    if (s == n*n-1 or
        possible(i+2, j-1, s+1) or possible(i+2, j+1, s+1) or
        possible(i+1, j+2, s+1) or possible(i-1, j+2, s+1) or
        possible(i-2, j+1, s+1) or possible(i-2, j-1, s+1) or
        possible(i-1, j-2, s+1) or possible(i+1, j-2, s+1))
      return true;
    t[i][j] = -1;
  }
  return false;
}
```

Knight's tour

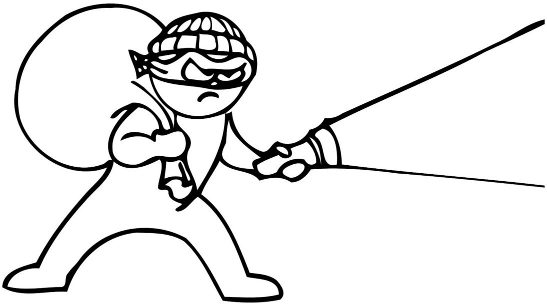
```
// Alternative implementation

vector<int> di = {1,  1, -1,  -1,  2,  2, -2,  -2};
vector<int> dj = {2, -2,  2,  -2,  1, -1,  1,  -1};

bool possible(int i, int j, int s) {
  if (i >= 0 and i < n and j >= 0 and j < n and t[i][j] == -1){
    t[i][j] = s;
    if (s == n*n-1) return true;
    for (int k = 0; k < 8; ++k)
      if (possible(i + di[k], j + dj[k], s+1))
        return true;
    t[i][j] = -1;
  }
  return false;
}
```

Knapsack problem

Let us assume that a thief enters a shop and wants to steal a set of objects with the largest possible value.



How can he find the best set using algorithms?

Knapsack problem

First of all, elaborate a list with weights and values of objects and knowing how much weigh he can handle.

Now he only needs an algorithm to act as quickly as possible.

Knapsack problem

The second step is to clearly define the problem.

Knapsack problem (integer version)

Given a knapsack that can store a weight C, and n objects with

- weights $p_1, p_2, \ldots, p_n$
- and values $v_1, v_2, \ldots, v_n$

find a selection $S \subseteq \{1, \ldots, n\}$ of the objects

- with maximum value $\sum_{i \in S} v_i$
- and that does not exceed the knapsack capacity

$$\sum_{i \in S} p_i \leq C.$$

Knapsack problem

First solution: prune when capacity is exceeded

```
#include <iostream>
#include <vector>

using namespace std;

int c;              // Capacity
int n;              // Number of objects
vector<int> w;      // Weights
vector<int> v;      // Values
vector<int> s;      // (Partial) Solution we are building
                    // s[i] == 1 iff i-th object is chosen

int bv = -1;        // Best value    so far
vector<int> bs;     // Best solution so far
```

Knapsack problem

```
void opt(int k, int swp, int svp) { // swp: sum weights partial
  if (swp > c) return; // Exceed capacity: do not continue
  if (k == n) {
    if (svp > bv) { // Improve best solution so far
      bs = s;
      bv = svp;
    }
    return;
  }
  s[k] = 0; opt(k+1, swp,         svp         ); // Discard obj. k
  s[k] = 1; opt(k+1, swp + w[k], svp + v[k]); // Choose obj. k
}

int main() {
  cin >> c >> n;
  w = v = s = vector<int>(n);
  for (int& x : p) cin >> x;
  for (int& x : v) cin >> x;
  opt(0, 0, 0);
  cout << bv << endl; }
```

Knapsack problem

Second solution: prune when we exceed capacity,
and when we cannot improve the best cost found so far (branch & bound)

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int c;             // Capacity
int n;             // Number of objects
vector<int> w;     // Weights
vector<int> v;     // Values
vector<int> s;     // (Partial) Solution we are building

int bv = -1;       // Best value    so far
vector<int> bs;    // Best solution so far
```

Knapsack problem

```
// svr: sum values remaining
void opt(int k, int swp, int svp, int svr) {
  if (swp > c  or  svp + svr <= bv) return;
  if (k == n) {
    bs = s;
    bv = svp;
    return;
  }
  s[k] = 0; opt(k+1, swp,         svp,         svr - v[k]);
  s[k] = 1; opt(k+1, swp + w[k], svp + v[k], svr - v[k]);
}

int main() {
  cin >> c >> n;
  p = v = s = vector<int>(n);
  for (int& x : p) cin >> x;
  for (int& x : v) cin >> x;
  opt(0, 0, 0, accumulate(v.begin(), v.end(), 0));
  cout << bv << endl;
}
```

Travelling Salesman Problem (TSP)

The Travelling Salesman Problems (TSP) consists in, given a network of cities, finding the order in which we should visit the cities such that:

- we start and finish in the same city
- we visit each city exactly once, and
- the total travelled distance is as small as possible

Optimal route visiting the 15 largest cities in Germany.

Source: `upload.wikimedia.org/wikipedia/commons/c/c4/TSP_Deutschland_3.png`

Travelling Salesman Problem (TSP)

- It is one of the most famous combinatorial optimization problems
- It is important from the theoretical point of view
- It has practical applications in
  - planning
  - logistics
  - microchips manufacturing
  - DNA sequence
  - astronomy
  - ...

Travelling Salesman Problem (TSP)

```
class TSP {

  int n;                      // number of cities
  matrix<int> M;              // matrix of distances
  vector<int> s;              // s[i] is next city after city i
                              // (−1 if not yet used)
                              // s is the partial solution we build
  vector<int> best_sol;       // best solution found so far
  double best_cost;           // cost of best solution found so far
```

Travelling Salesman Problem (TSP)

```
void recursive (int v, int t, double c) {
    // v = last vertex in partial solution s
    // t = size of the path given by s
    // c = cost of s
    if (t == n) {
        c += M[v][0];
        if (c < best_cost) {
            best_cost = c;
            best_sol = s;
            best_sol[v] = 0;
        }
    } else {
        for(int u = 0; u < n; ++u)
          if (u != v and s[u] == -1) {
            if (c + M[v][u] < best_cost) {
                s[v] = u;
                recursive(u, t+1, c + M[v][u]);
                s[v] = -1;
}   }   }   }
```

Travelling Salesman Problem (TSP)

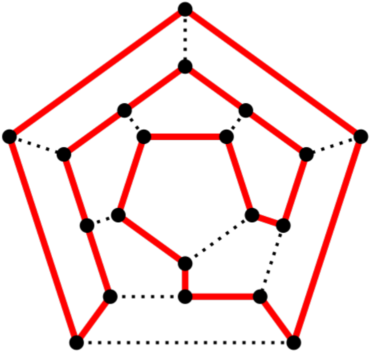
```
public:

    TSP (matrix& M) {
        this->M = M;
        n = M.rows();
        s = vector<int>(n, -1);
        best_sol = vector<int>(n);
        best_cost = infinite;
        recursive(0, 1, 0);
    }

    vector<int> solution (      ) { return best_sol;     }
    int         next     (int x) { return best_sol[x];  }
    double      cost     (      ) { return best_cost;    }
};
```

Hamiltonian Graph

- A Hamiltonian cycle is a cycle that visits each vertex exactly once



Source: `https://en.wikipedia.org/wiki/Hamiltonian_path`

- If a graph has a Hamiltonian cycle, we say the graph is Hamiltonian.
- Given a graph, we want to know whether it is Hamiltonian.

Hamiltonian Graph

- Let us assume that the graph is connected
- Let us assume it is represented with **sorted** adjacency lists

```
typedef vector< vector<int> > Graph;
typedef list<int>::iterator iter;


class HamiltonianGraph {

    Graph G;              // the graph
    int n;                // number of vertices


    vector<int> s;        // next of each vertex
                          // (−1 if not yet used)
                          // s is the partial solution we build

    bool found;           // whether we have already found a cycle
    vector<int> sol;      // solution (if found)
```

Hamiltonian Graph

```
void recursive (int v, int t) {
    // v = last vertex in the path, t = size of path
    if (t == n) {
        // should make sure path s can be closed to a cycle
        if (G[v][0] == 0) {
            s[v] = 0;
            found = true;
            sol = s;
            s[v] = −1;
        }
    } else {
        for (int u : G[v]) {
            if (s[u] == −1) {
                s[v] = u;
                recursive(u, t+1);
                s[v] = −1;
                if (found) return;
}   }   }   }
```

Hamiltonian Graph

```
public:

    HamiltonianGraph (Graph G) {
        this->G = G;
        n = G.size();
        s = vector<int>(n, -1);
        found = false;
        recursive(0,1);
    }

    bool te_solucio () {
        return found;
    }

    vector<int> solucio () {
        return sol;
    }
};
```