Chapter 2. Exhaustive search

Algorithmics and Programming III

FIB

Q1 2019–2020

Version September 30, 2019

Brute-force algorithms

- Many problems can be seen as, given a finite set of objects, finding one that satisfies some constraints (a solution to the problem)
- The set of objects to be explored is called the search space

Brute-force algorithms

- Many problems can be seen as, given a finite set of objects, finding one that satisfies some constraints (a solution to the problem)
- The set of objects to be explored is called the search space
- Variations: finding
  - all solutions,
  - or an optimal solution if there is some cost/merit function

Brute-force algorithms

- Many problems can be seen as, given a finite set of objects, finding one that satisfies some constraints (a solution to the problem)
- The set of objects to be explored is called the search space
- Variations: finding
    - all solutions,
    - or an optimal solution if there is some cost/merit function
- Sometimes the only way to solve these problems is to try all possibilities: generate all possible candidates and test if they are solutions
- The resulting algorithm is called brute force or exhaustive search:
    - It is usually exponential
    - It can be slow, but is better than not having any algorithm at all...
    - It can be practical for inputs of small size

Example

- We want to write all binary sequences of size n
- E.g., for $n = 3$: 000, 001, 010, 011, 100, 101, 110, 111

Example

- We want to write all binary sequences of size n
- E.g., for $n = 3$: 000, 001, 010, 011, 100, 101, 110, 111
- We will generate and print sequences one at a time
- We need some data structure to store the current sequence; e.g., a `vector<int>` of size n

Example

- We want to write all binary sequences of size n
- E.g., for $n = 3$: 000, 001, 010, 011, 100, 101, 110, 111
- We will generate and print sequences one at a time
- We need some data structure to store the current sequence; e.g., a `vector<int>` of size n
- Let us implement a function

```
void gen(int k, vector<int>& seq)
```

that, given

- a natural `k`, and
- a vector `seq` whose first `k` positions are already filled,

writes all possible ways to extend `seq` to a full sequence

Example

```
#include <iostream>
#include <vector>
using namespace std;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq) {
  ...




}

int main() {
  int n;
  cin >> n;
  vector<int> seq(n);
  gen(?, ???); }
```

Example

```
#include <iostream>
#include <vector>
using namespace std;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq) {
  ???




}

int main() {
  int n;
  cin >> n;
  vector<int> seq(n);
  gen(0, seq); }
```

Example

```
#include <iostream>
#include <vector>
using namespace std;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq) {
  if (k == seq.size()) ???
  ...



}

int main() {
  int n;
  cin >> n;
  vector<int> seq(n);
  gen(0, seq); }
```

Example

```
#include <iostream>
#include <vector>
using namespace std;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq) {
  if (k == seq.size()) write(seq);
  else {
    seq[?] = ?; gen(???, seq);
    ...
} }

int main() {
  int n;
  cin >> n;
  vector<int> seq(n);
  gen(0, seq); }
```

Example

```
#include <iostream>
#include <vector>
using namespace std;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq) {
  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq);
    seq[k] = ?; gen(???, seq);
} }

int main() {
  int n;
  cin >> n;
  vector<int> seq(n);
  gen(0, seq); }
```

Example

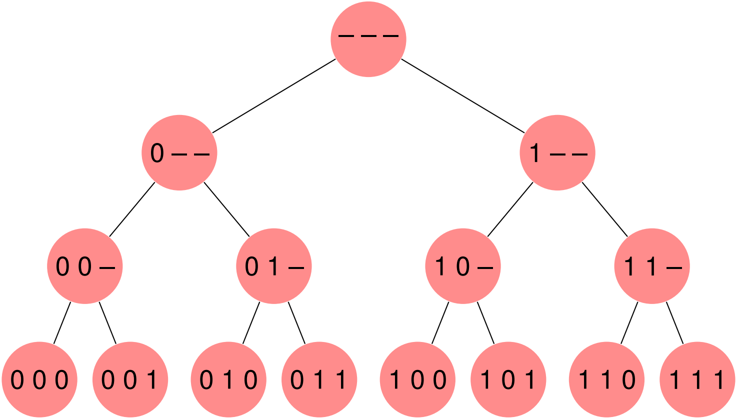
```cpp
#include <iostream>
#include <vector>
using namespace std;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq) {
  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq);
    seq[k] = 1; gen(k+1, seq);
} }

int main() {
  int n;
  cin >> n;
  vector<int> seq(n);
  gen(0, seq); }
```

Example

```
#include <iostream>
#include <vector>
using namespace std;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq) {
  if (k == seq.size()) write(seq);
  else {
    for (int v = 0; v <= 1; ++v) { // Alternative code
       seq[k] = v; gen(k+1, seq);
} } }

int main() {
  int n;
  cin >> n;
  vector<int> seq(n);
  gen(0, seq); }
```

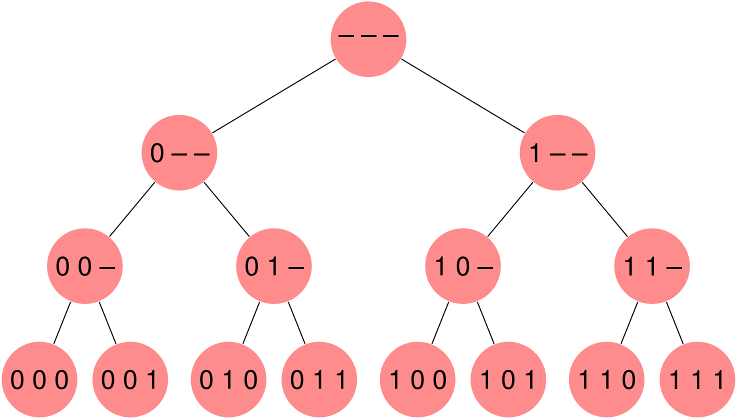
Example

- For $n = 3$ we get this recursion tree (a.k.a. search tree, state space tree)



- Leaves correspond to (candidate) solutions
- Internal nodes correspond to partial solutions
- Root corresponds to the empty partial solution

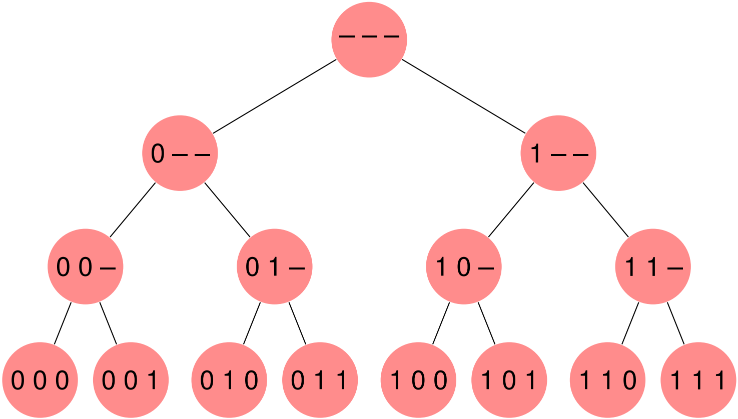
- For $n = 3$ we get this recursion tree (a.k.a. search tree, state space tree)



- Each level corresponds to a position of the sequence
- Each internal node has as many descendants as possible choices

Example

- For $n = 3$ we get this recursion tree (a.k.a. search tree, state space tree)



- Exhaustive search is a DFS on the search tree: the root is visited first, and a visit to a node is followed immediately by visits to its descendants

Generation of subsets

- We want to write a program that, given n different words $s_1, \ldots, s_n$, prints all the subsets that can be made up with the words.
- For example, if $n = 3$ and $s_1 = \texttt{hola}, s_2 = \texttt{adeu}, s_3 = \texttt{hi}$:

```
{}
{hi}
{adeu}
{adeu,hi}
{hola}
{hola,hi}
{hola,adeu}
{hola,adeu,hi}
```

Generation of subsets

```
vector<string> s;

void write(const vector<int>& seq) {
  cout << '{';
  string aux = "";
  for (int i = 0; i < seq.size(); ++i)
    if (seq[i]) {
      cout << aux << s[i];
      aux = ",";
    }
  cout << '}' << endl; }

 // The same as in the generation of binary sequences
void gen(int k, vector<int>& seq) { ... }

int main() {
  int n;
  cin >> n;
  s = vector<string>(n);
  for (auto& si : s) cin >> si;
  vector<int> seq(n);
  gen(0, seq); }
```

Generic algorithm

- A clean way to implement brute force is recursively

Generic algorithm

- A clean way to implement brute force is recursively
- The algorithm considers one candidate solution at a time
- Candidate solutions are viewed as sequences of items
- So the current partial solution is stored in a vector (or a matrix, ...)

Generic algorithm

- A clean way to implement brute force is recursively
- The algorithm considers one candidate solution at a time
- Candidate solutions are viewed as sequences of items
- So the current partial solution is stored in a vector (or a matrix, ...)
- For example, let us imagine we want to generate all solutions
- Let us consider a function

```
void gen(int k, vector<T>& s) //T may be int, char, ...
```

that, given

- a natural `k`, and
- a vector `s` whose first `k` positions are filled,

generates all possible ways to extend the partial solution `s` to a solution

Generic algorithm

```
void gen(int k, vector<T>& s) {
   ...







}

int main() {
  ...
  vector<T> s(...); // Allocate space
  gen(0, s);
}
```

Generic algorithm

```
void gen(int k, vector<T>& s) {
   if (k == s.size()) { // Base case: partial solution complete
      if (satisfies_constraints(s)) // s is a solution
         process(s); // may be: write, increment a counter, ...
   }
   else {
      ...



   }
}

int main() {
  ...
  vector<T> s(...); // Allocate space
  gen(0, s);
}
```

Generic algorithm

```
void gen(int k, vector<T>& s) {
   if (k == s.size()) { // Base case: partial solution complete
      if (satisfies_constraints(s)) // s is a solution
         process(s); // may be: write, increment a counter, ...
   }
   else {
      for (T v : possible values for s[k]) {
         s[k] = v;
         gen(k+1, s);
      }
   }
}

int main() {
  ...
  vector<T> s(...); // Allocate space
  gen(0, s);
}
```

Generic algorithm

- Now let us imagine we want to generate just one solution
- We consider a function

```
bool gen(int k, vector<T>& s)
```

that, given

- a natural `k`, and
- a vector `s` whose first `k` positions are filled,

returns `true` if there is a way to extend the partial solution `s` to a solution

Generic algorithm

```
bool gen(int k, vector<T>& s) {
   if (k == s.size()) {
      if (satisfies_constraints(s)) {
         process(s);
         return true;
      }
      else return false;
   }
   else {
      for (T v : possible values for s[k]) {
         s[k] = v;
         if (gen(k+1, s)) return true;
      }
      return false;
}  }

int main() {
  ...
  vector<T> s(...);
  if (not gen(0, s)) cout << "No solution" << endl;
}
```

Cost of brute force

- What is the cost of brute force?
- In general, proportional to the size of the search tree
- So the cost depends on how the search tree is represented in the input

Cost of brute force

- What is the cost of brute force?
- In general, proportional to the size of the search tree
- So the cost depends on how the search tree is represented in the input
  - If the search tree is explicitly part of the input it is polynomial

    For example, DFS and BFS are exhaustive search algorithms running in polynomial time if the graph is given

Cost of brute force

- What is the cost of brute force?
- In general, proportional to the size of the search tree
- So the cost depends on how the search tree is represented in the input
  - If the search tree is explicitly part of the input it is polynomial

    For example, DFS and BFS are exhaustive search algorithms running in polynomial time if the graph is given
  - If the search tree is implicit the cost is usually exponential (or worse)

Backtracking

- Backtracking algorithms are like brute force but the search is stopped when we detect that the current partial solution will not lead to a solution

  (and then we backtrack in the search, i.e., we undo the last choice)

- Typically this happens when the partial solution does not satisfy one of the constraints, and this makes it impossible to be part of a solution

Example

- We want to write all binary sequences of size n with at most m ones
- E.g., for $n = 3$, $m = 1$: 000, 001, 010, 100,
- We store our partial solution in a `vector<int>` of size n
- Let us consider a function

```
void gen(int k, vector<int>& seq, int u)
```

  that, given

  - a natural `k`,
  - a natural `u`, and
  - a vector `seq` whose first `k` positions are filled, of which `u` are 1's,

  writes all possible ways to extend `seq` to a solution

Example

```
int n, m;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq, int u) {
  ...






} }

int main() {
  cin >> n >> m;
  vector<int> seq(n);
  gen(0, seq, ?); }
```

Example

```
int n, m;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq, int u) {
  ...
  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq, ???);
    seq[k] = 1; gen(k+1, seq, ???);
} }

int main() {
  cin >> n >> m;
  vector<int> seq(n);
  gen(0, seq, 0); }
```

Example

```
int n, m;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq, int u) {
  ???
  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq, u);
    seq[k] = 1; gen(k+1, seq, u+1);
} }

int main() {
  cin >> n >> m;
  vector<int> seq(n);
  gen(0, seq, 0); }
```

Example

```
int n, m;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq, int u) {
  if (???) return;
  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq, u);
    seq[k] = 1; gen(k+1, seq, u+1);
} }

int main() {
  cin >> n >> m;
  vector<int> seq(n);
  gen(0, seq, 0); }
```

Example

```
int n, m;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq, int u) {
  if (u > m) return; // Too many 1's
  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq, u);
    seq[k] = 1; gen(k+1, seq, u+1);
} }

int main() {
  cin >> n >> m;
  vector<int> seq(n);
  gen(0, seq, 0); }
```

Example

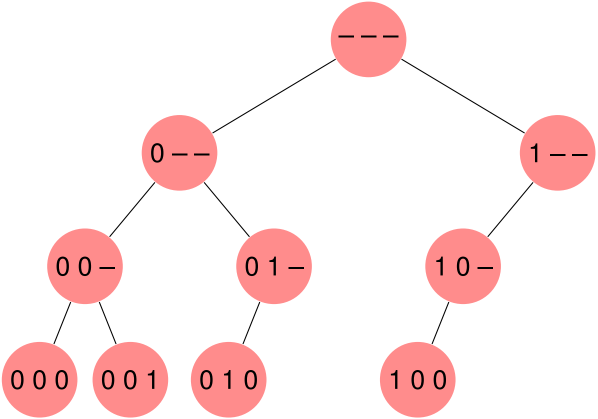
```
int n, m;

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl;
}

void gen(int k, vector<int>& seq, int u) {

  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq, u);
    if (u < m) {seq[k] = 1; gen(k+1, seq, u+1);}
} }

int main() {
  cin >> n >> m;
  vector<int> seq(n);
  gen(0, seq, 0); }
```

Example

```
int n, m, u;  // u is a global variable instead of a parameter

void write(const vector<int>& v) {
  for (int x : v) cout << x;
  cout << endl; }

void gen(int k, vector<int>& seq) {
  if (k == seq.size()) write(seq);
  else {
    seq[k] = 0; gen(k+1, seq);
    if (u < m) {
       ++u;
       seq[k] = 1; gen(k+1, seq);
       --u;   // as u is global, changes must be restored!
} } }

int main() {
  u = 0;
  cin >> n >> m;
  vector<int> seq(n);
  gen(0, seq); }
```

Example

- For $n = 3$ and $m = 1$, we obtain the following search tree:



- Better than generate and test all possibilities: fewer nodes are explored
- When we discard a partial solution we say we prune the search tree

Generic algorithm

- For example, let us imagine we want to generate all solutions
- Let us consider a function

```
void gen(int k, vector<T>& s)
```

that, given

- a natural `k`, and
- a vector `s` whose first `k` positions are filled,

generates all possible ways to extend the partial solution `s` to a solution

Generic algorithm

```
void gen(int k, vector<T>& s) {
   if (partial solution s can be pruned) return;
   if (k == s.size()) {
      if (satisfies_constraints(s))
         process(s); // ^ may not be necessary due to pruning
   }
   else {
      for (T v : possible values for s[k]){ // prune here too?
         s[k] = v;
         gen(k+1, s);
      }
   }
}

int main() {
  ...
  vector<T> s(...);
  gen(0, s);
}
```

Cost of backtracking

- Cost in time: $\mathcal{O}$(size of search tree), usually exponential in the input
- Cost in space: $\mathcal{O}$(depth of search tree), usually polynomial in the input

Cost of backtracking

- Cost in time: $\mathcal{O}$(size of search tree), usually exponential in the input
- Cost in space: $\mathcal{O}$(depth of search tree), usually polynomial in the input

- Hence in the worst case backtracking is not better than exhautive search
- But in practice the difference in performance may be dramatic
- Backtracking algorithms are useful because
  they can be efficient for many large instances,
  not because they are efficient for all large instances
- Pruning can be of critical importance

Cost of backtracking

- Cost in time: $\mathcal{O}$(size of search tree), usually exponential in the input
- Cost in space: $\mathcal{O}$(depth of search tree), usually polynomial in the input

- Hence in the worst case backtracking is not better than exhautive search
- But in practice the difference in performance may be dramatic
- Backtracking algorithms are useful because
  they can be efficient for many large instances,
  not because they are efficient for all large instances
- Pruning can be of critical importance
- There is a tradeoff between
  - the time spent on detecting that a node can be pruned, and
  - the size of the search tree that is pruned

  In the end, what we want is to reduce the execution time!

  (not the number of visited nodes of the search tree)

Generation of permutations

- A permutation of n different objects is any sequence of the n objects
  The order in which the objects appear in the sequence matters
- E.g., $(1, 2, 3)$ and $(1, 3, 2)$ are different permutations of $\{1, 2, 3\}$
- The number of permutations of n objects is

Generation of permutations

- A permutation of n different objects is any sequence of the n objects
  The order in which the objects appear in the sequence matters
- E.g., $(1, 2, 3)$ and $(1, 3, 2)$ are different permutations of $\{1, 2, 3\}$
- The number of permutations of n objects is $n! = n \cdot (n-1) \cdots 2 \cdot 1$

Generation of permutations

- Given n, we want to write all permutations of numbers 1, 2, . . ., n
- For example, for $n = 3$:

$$\begin{array}{cc} (1,2,3) & (1,3,2) \\ (2,1,3) & (2,3,1) \\ (3,1,2) & (3,2,1) \end{array}$$

Generation of permutations

- Given n, we want to write all permutations of numbers 1, 2, . . ., n
- For example, for $n = 3$:

$$\begin{array}{ll} (1,2,3) & (1,3,2) \\ (2,1,3) & (2,3,1) \\ (3,1,2) & (3,2,1) \end{array}$$

- A natural way to represent a permutation is as a `vector<int>`
- E.g., if vector `p` represents $(1, 2, 3)$ then `p[0]= 1`, `p[1]= 2`, `p[2]= 3`

Generation of permutations

- Given n, we want to write all permutations of numbers 1, 2, . . ., n
- For example, for $n = 3$:

$$\begin{array}{cc} (1,2,3) & (1,3,2) \\ (2,1,3) & (2,3,1) \\ (3,1,2) & (3,2,1) \end{array}$$

- A natural way to represent a permutation is as a `vector<int>`
- E.g., if vector `p` represents $(1, 2, 3)$ then `p[0]= 1`, `p[1]= 2`, `p[2]= 3`
- First implementation: let us consider a function

```
void gen(int k, vector<int>& s)
```

  that, given
  - a natural `k`, and
  - a vector `s` whose first `k` positions are filled,

  generates all ways to extend the partial solution `s` to a permutation

Generation of permutations

```
bool legal(vector<int>& s, int k) { // is s[k] new?
  ???




}

void gen(int k, vector<int>& s) {
  if (k == s.size()) write(s);
  else {
    for (int v = 1; v <= s.size(); ++v) {
      s[k] = v;
      if (legal(s, k)) gen(k+1, s); // prune here
} } }

int main() {
  int n;
  cin >> n;
  vector<int> s(n);
  gen(0, s);
}
```

Generation of permutations

```
bool legal(vector<int>& s, int k) { // is s[k] new?
  for (int i = 0; i < k; ++i)
    if (s[i] == s[k])
      return false; // Found a repetition, so it is not legal
  return true;
}

void gen(int k, vector<int>& s) {
  if (k == s.size()) write(s);
  else {
    for (int v = 1; v <= s.size(); ++v) {
      s[k] = v;
      if (legal(s, k)) gen(k+1, s); // prune here
} } }

int main() {
  int n;
  cin >> n;
  vector<int> s(n);
  gen(0, s);
}
```

Generation of permutations

- A more efficient way to discard repeated values: mark the ones we use
- We will maintain a `vector<bool> mkd` such that
  `mkd[v]` is true iff `v` has already been used in the partial solution
- Second implementation: let us consider a function

  ```
  void gen(int k, vector<int>& s, vector<bool>& mkd)
  ```

  that, given

  - a natural `k`,
  - a vector `s` whose first `k` positions are filled, and
  - a vector `mkd` indicating the values that have been used,

  generates all ways to extend the partial solution `s` to a permutation

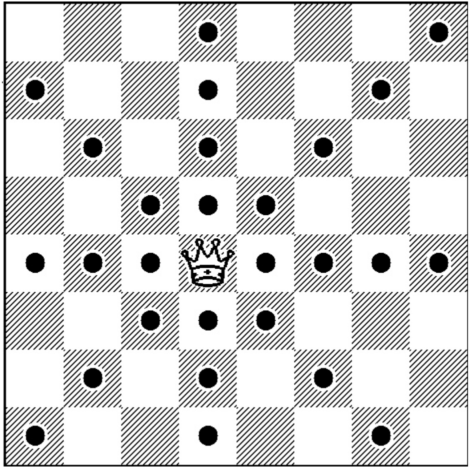
Generation of permutations

```
void gen(int k, vector<int>& s, vector<bool>& mkd) {
  if (k == s.size()) write(s);
  else {
    for (int v = 1; v <= s.size(); ++v) {
      if (???) {
        ???;
        s[k] = v;
        gen(k+1, s, mkd);
        ???;
} } } }

int main() {
  int n;
  cin >> n;
  vector<int> s(n);
  vector<bool> mkd(n+1, false); // n+1 as values are 1..n
  gen(0, s, mkd);
}
```

Generation of permutations

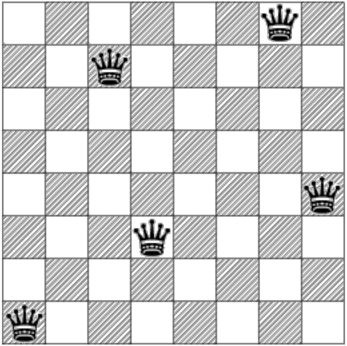
```
void gen(int k, vector<int>& s, vector<bool>& mkd) {
  if (k == s.size()) write(s);
  else {
    for (int v = 1; v <= s.size(); ++v) {
      if (not mkd[v]) {
        mkd[v] = true;
        s[k] = v;
        gen(k+1, s, mkd);
        mkd[v] = false;
} } } }

int main() {
  int n;
  cin >> n;
  vector<int> s(n);
  vector<bool> mkd(n+1, false); // n+1 as values are 1..n
  gen(0, s, mkd);
}
```

Queen movements in chess:



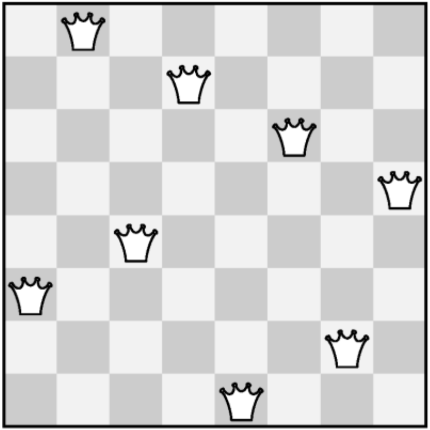
How many queens can we place on a board
so that no two queens threaten each other?

5? 6? 7? 8?



8-queens problem

Placing 8 queens on a board so that no two queens threaten each other.



Brute-force solving strategies:

Brute-force solving strategies:

1. Choose 8 different positions on the board.

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

Brute-force solving strategies:

1. Choose 8 different positions on the board.

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

2. Choose 8 positions on different rows.

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

Brute-force solving strategies:

1. Choose 8 different positions on the board.

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

2. Choose 8 positions on different rows.

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

3. Choose 8 positions on different rows and columns.

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

Brute-force solving strategies:

1. Choose 8 different positions on the board.

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

2. Choose 8 positions on different rows.

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

3. Choose 8 positions on different rows and columns.

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

With backtracking one can still do better.

n-queens

We consider the generalized problem of the n-queens:

n-queens problem

Place n queens on an $n \times n$ board so that no two queens threaten each other.

n-queens

We consider the generalized problem of the *n*-queens:

n-queens problem

Place n queens on an $n \times n$ board so that no two queens threaten each other.

So we have to place n queens so that there cannot be two queens

- in the same row
- in the same column
- in the same ascending diagonal
- in the same descending diagonal

n-queens

We consider the generalized problem of the *n*-queens:

_n_-queens problem

Place n queens on an $n \times n$ board so that no two queens threaten each other.

So we have to place n queens so that there cannot be two queens

- in the same row
- in the same column
- in the same ascending diagonal
- in the same descending diagonal

Let us consider the problem of writing all solutions

First implementation:

- with backtracking
- extends the partial solution as long as it is “legal”
  (can be extended to a complete solution)

- If n queens are placed on an $n \times n$ board
  so that there cannot be two queens on the same row,
  then each row must have exactly one queen

- We implement the position of the queens with a `vector<int> T`
  that indicates that queen of row i is in column $T[i]$ $\quad (0 \leq i < n)$.

- This way there are not two queens on the same row by construction

- If n queens are placed on an $n \times n$ board so that there cannot be two queens on the same row, then each row must have exactly one queen
- We implement the position of the queens with a `vector<int> T` that indicates that queen of row i is in column $T[i]$ $(0 \leq i < n)$.
- This way there are not two queens on the same row by construction
- In order to know if the queens in rows i and k share
  - column, we will check if $T[i] = T[k]$
  - descending diagonal ($\searrow$), we will check if $T[i] - i = T[k] - k$
  - ascending diagonal ($\nearrow$), we will check if $T[i] + i = T[k] + k$

n-queens

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

bool legal(int i) { does queen of row i attack previous queens?
  for (int k = 0; k < i; ++k)
    if (t[k] == t[i] or t[i]-i == t[k]-k or t[i]+i == t[k]+k)
      return false;
  return true;
}

void reines(int i) {
  if (i == n) return write();
    for (int j = 0; j < n; ++j) {
      t[i] = j;
      if (legal(i)) reines(i+1);
}   }

int main() {
  cin >> n;
  t = vector<int>(n);
  reines(0);
}
```

Second implementation:

- with backtracking
- extends the partial solution as long as it is “legal”
  (can be extended to a complete solution)
- with marking: we mark “used” columns and diagonals

n-queens

```
int n;
vector<int> t, mc, md1, md2;

int d1(int i, int j) { return i+j; }
int d2(int i, int j) { return i-j + n-1; }

void reines(int i) {
  if (i == n) return write();
  for (int j = 0; j < n; ++j)
    if (not mc[j] and not md1[d1(i,j)] and not md2[d2(i,j)]) {
      t[i] = j;
      mc[j] = md1[d1(i, j)] = md2[d2(i, j)] = true;
      reines(i+1);
      mc[j] = md1[d1(i, j)] = md2[d2(i, j)] = false;
}   }

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

Travelling salesman

The problem of the travelling salesman consists in, given a network of cities, to find the order to visit them so that:

- the salesman starts and ends at his city
- the salesman visits the rest of the cities exactly once, and
- the total distance of the tour is the shortest possible

Travelling salesman

The problem of the travelling salesman consists in, given a network of cities, to find the order to visit them so that:

- the salesman starts and ends at his city
- the salesman visits the rest of the cities exactly once, and
- the total distance of the tour is the shortest possible

Optimal tour of a salesman visiting the 15 largest cities in Germany

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

Travelling salesman

- Unlike the previous examples, this is an optimization problem:
  the goal is to find the optimal solution
- In fact it is one of the best studied problems in combinatorial optimization

Travelling salesman

- Unlike the previous examples, this is an optimization problem: the goal is to find the optimal solution
- In fact it is one of the best studied problems in combinatorial optimization
- It has practical applications to
  - planning
  - logistics
  - microchips
  - DNA sequencing
  - astronomy
  - ...
- Its decision version is NP-complete

The input data consists in:

- n: the number of cities (which are identified with numbers $0, \ldots, n-1$)
- D: an $n \times n$ matrix such that the distance between i and j is $D[i][j]$ ($D[i][j] \geq 0$ for all $0 \leq i, j < n$)

Travelling salesman

The input data consists in:

- n: the number of cities (which are identified with numbers $0, \ldots, n-1$)
- D: an $n \times n$ matrix such that the distance between i and j is $D[i][j]$ ($D[i][j] \geq 0$ for all $0 \leq i, j < n$)

Given the input:

- which is the shortest tour?
- what is its cost?

For example, if $n = 11$ and D is the following matrix:

	0	1	2	3	4	5	6	7	8	9	10
0	0	29	20	21	16	31	100	12	4	31	18
1	29	0	15	29	28	40	72	21	29	41	12
2	20	15	0	15	14	25	81	9	23	27	13
3	21	29	15	0	4	12	92	12	25	13	25
4	16	28	14	4	0	16	94	9	20	16	22
5	31	40	25	12	16	0	95	24	36	3	37
6	100	72	81	92	94	95	0	90	101	99	84
7	12	21	9	12	9	24	90	0	15	25	13
8	4	29	23	25	20	36	101	15	0	35	18
9	31	41	27	13	16	3	99	25	35	0	38
10	18	12	13	25	22	37	84	13	18	38	0

then the optimal sequence of cities is $(0, 7, 4, 3, 9, 5, 2, 6, 1, 10, 8, 0)$ and the cost of the tour is

$$12 + 9 + 4 + 13 + 3 + 25 + 81 + 72 + 12 + 18 + 4 = 253$$

Travelling salesman

```
int n;                       // Number of cities
vector<vector<double>> D;    // Distance matrix
vector<int> s, best_s;       // Current  &  best solution so far
double best_c;               // Cost of the best solution so far

// v = last city, t = #cities partial route, c = partial cost
void tsp(int v, int t, double c) { ... }

int main() {
  cin >> n;
  D = vector<vector<double>>(n, vector<double>(n));
  for (auto& R : D) for (auto& d : R) cin >> d; // Read data

  s = vector<int>(n,-1); // s[u] is the city that follows u
  best_c = DBL_MAX;      // DBL_MAX acts as +∞
  tsp(0, 1, 0);          // Wlog, the city of the salesman is 0

  cout << 0 << endl;       // Write sequence of cities
  for (int v = 0, i = 0; i < n; ++i) {
    v = best_s[v];
    cout << v << endl;
} }
```

Travelling salesman

```
int n;                        // Number of cities
vector<vector<double>> D;     // Distance matrix
vector<int> s, best_s;        // Current  &  best solution so far
double best_c;                // Cost of the best solution so far

// v = last city, t = #cities partial route, c = partial cost
void tsp(int v, int t, double c) {
  if (t == n) {
    c += D[v][0];
    if (c < best_c) {
      s[v] = 0;
      best_s = s;
      best_c = c;
  } }
  else for (int u = 0; u < n; ++u)
         if (u != v and s[u] == -1) {
           s[v] = u;
           tsp(u, t+1, c+D[v][u]);
           s[v] = -1;
}        }

int main() { ... }
```

```
int n;                          // Number of cities
vector<vector<double>> D;       // Distance matrix
vector<int> s, best_s;          // Current  &  best solution so far
double best_c;                  // Cost of the best solution so far

// v = last city, t = #cities partial route, c = partial cost
void tsp(int v, int t, double c) {
  if (t == n) {
    c += D[v][0];
    if (c < best_c) {
      s[v] = 0;
      best_s = s;
      best_c = c;
  } }
  else for (int u = 0; u < n; ++u)
         if (u != v and s[u] == -1 and c + D[v][u] < best_c) {
           s[v] = u;
           tsp(u, t+1, c+D[v][u]);
           s[v] = -1;
}        }

int main() { ... }
```

Branch & bound

- Branch & bound (B&B) is an improvement on backtracking
- Only applicable to optimization problems
- For the sake of presentation, let us consider a minimization problem

Branch & bound

- Let C be the cost function to be minimized
- Let p be the current partial solution
- Assume we have a lower bound L for the cost of any solution extending p
  That is, if s is a solution that extends p, then $L \leq C(s)$

Branch & bound

- Let C be the cost function to be minimized
- Let p be the current partial solution
- Assume we have a lower bound L for the cost of any solution extending p
  That is, if s is a solution that extends p, then $L \leq C(s)$
- Let C^* be the cost of the best solution found so far
- If $C^* \leq L$ then partial solution p can be pruned:
  if s is a solution that extends p, then

$$C^* \leq L \leq C(s)$$

  No solution extending p can improve on the best solution found so far

Branch & bound

- Let C be the cost function to be minimized
- Let p be the current partial solution
- Assume we have a lower bound L for the cost of any solution extending p
  That is, if s is a solution that extends p, then $L \leq C(s)$
- Let C^* be the cost of the best solution found so far
- If $C^* \leq L$ then partial solution p can be pruned:
  if s is a solution that extends p, then

$$C^* \leq L \leq C(s)$$

  No solution extending p can improve on the best solution found so far
- The computation of the lower bound depends on the problem to solve

Example

- Let us consider again the Travelling Salesman Problem

```
void tsp(int v, int t, double c) {
  if (t == n) {
    ...
  } }
  else for (int u = 0; u < n; ++u)
         if (u != v and s[u] == -1 and c + D[v][u] < best_c) {
           s[v] = u;
           tsp(u, t+1, c+D[v][u]);
           s[v] = -1;
}        }
```

- Here `c + D[v][u]` is a lower bound on the cost of any extension of `s`
- So this is an example of B&B algorithm

Example

- Let us consider again the Travelling Salesman Problem

```
void tsp(int v, int t, double c) {
  if (t == n) {
    ...
  } }
  else for (int u = 0; u < n; ++u)
           if (u != v and s[u] == -1 and c + D[v][u] < best_c) {
             s[v] = u;
             tsp(u, t+1, c+D[v][u]);
             s[v] = -1;
}          }
```

- Here `c + D[v][u]` is a lower bound on the cost of any extension of `s`
- So this is an example of B&B algorithm
- However, this lower bounding procedure is rather coarse:
  it only considers the part of the tour that we have already built
- Next we'll see a more precise lower bounding procedure

Example

- In any tour, the distance of the edge taken when leaving a vertex must be at least as great as the distance of the shortest edge from that vertex:
  - a lower bound on the cost of leaving vertex 0 is given by the minimum of all the non-diagonal entries in row 0 of D
  - a lower bound on the cost of leaving vertex 1 is given by the minimum of all the non-diagonal entries in 1 of D
  - . . .

Example

- In any tour, the distance of the edge taken when leaving a vertex must be at least as great as the distance of the shortest edge from that vertex:
  - a lower bound on the cost of leaving vertex 0 is given by the minimum of all the non-diagonal entries in row 0 of D
  - a lower bound on the cost of leaving vertex 1 is given by the minimum of all the non-diagonal entries in 1 of D
  - ...

	0	1	2	3	4
0	0	14	4	10	20
1	14	0	7	8	7
2	4	5	0	7	16
3	11	7	9	0	2
4	18	7	17	4	0

$$\begin{array}{lcl} 0 & \rightarrow & \min(14, 4, 10, 20) = 4 \\ 1 & \rightarrow & \min(14, 7, 8, 7) = 7 \\ 2 & \rightarrow & \min(4, 5, 7, 16) = 4 \\ 3 & \rightarrow & \min(11, 7, 9, 2) = 2 \\ 4 & \rightarrow & \min(18, 7, 17, 4) = 4 \end{array}$$

Lower bound on the cost of any tour: $4 + 7 + 4 + 2 + 4 = 21$

Example

- Assume that we have a partial solution in which we go from 0 to 1
- So the cost of leaving from 0 is the weight of the edge from 0 to 1: 14

	0	1	2	3	4
0	0	14	4	10	20
1	14	0	7	8	7
2	4	5	0	7	16
3	11	7	9	0	2
4	18	7	17	4	0

$$\begin{array}{lll} 0 & \rightarrow & 14 \\ 1 & \rightarrow & \min(7,8,7) = 7 \\ 2 & \rightarrow & \min(4,7,16) = 4 \\ 3 & \rightarrow & \min(11,9,2) = 2 \\ 4 & \rightarrow & \min(18,17,4) = 4 \end{array}$$

- For the min of 1 we don't include the edge to 0, as 1 can't return to 0
- For the min's of the other vertices
  we don't include the edge to 1, as we have already been at 1
- Lower bound on the cost of any tour that goes 0 to 1:

$$14 + 7 + 4 + 2 + 4 = 31$$

Example

- Assume that we have a partial solution in which $0 \to 1 \to 2$
- The cost of leaving from 0 is the weight of the edge from 0 to 1: 14
- The cost of leaving from 1 is the weight of the edge from 1 to 2: 7

	0	1	2	3	4
0	0	14	4	10	20
1	14	0	7	8	7
2	4	5	0	7	16
3	11	7	9	0	2
4	18	7	17	4	0

$$\begin{array}{lcl} 0 & \to & 14 \\ 1 & \to & 7 \\ 2 & \to & \min(7, 16) = 7 \\ 3 & \to & \min(11, 2) = 2 \\ 4 & \to & \min(18, 4) = 4 \end{array}$$

- For the min of 2 we don't include edges to 0, 1, as 2 can't return to 0, 1
- For the min's of the other vertices
  we don't include edges to 1, 2, as we have already been at 1, 2
- Lower bound on the cost of any tour in which $0 \to 1 \to 2$:

$$14 + 7 + 7 + 2 + 4 = 34$$

Example

```
void tsp (int v, int t, double c) {
  if (t == n) {
    c += D[v][0];
    if (c < best_c) {
      s[v] = 0;
      best_s = s;
      best_c = c;
    }
  }
  else if (lower_bound(v, c) < best_c) {
      for (int u = 0; u < n; ++u)
        if (u != v and s[u] == -1 and c + D[v][u] < best_c) {
          s[v] = u;
          tsp(u, t+1, c+D[v][u]);
          s[v] = -1;
        }
  }
}
```

Example

```
double min_weight_last(int w) {
  double min = DBL_MAX;
  for (int z = 0; z < n; ++z)
    if (z != w and s[z] == -1 and min > D[w][z])
      min = D[w][z];
  return min;
}

double min_weight_unas(int w, int v) {
  double min = DBL_MAX;
  for (int z = 0; z < n; ++z)
    if (z != w and
        (z == 0 or (s[z] == -1 and z != v)) and min > D[w][z])
      min = D[w][z];
  return min;
}

double lower_bound(int v, double c) {
  for (int w = 0; w < n; ++w)
    if (w == v)             c += min_weight_last(w);
    else if (s[w] == -1) c += min_weight_unas(w, v);
  return c; }
```

Best-first search

- We can compare the lower bounds of the partial solutions and continue the search with the one with the best bound (best-first search)
- Each node of the search tree must be explicitly represented, containing:
  - all data of the partial solution
  - the level in the search tree
  - the lower bound
- We can use a priority queue to manage pending nodes of search tree (instead of a stack as in DFS)
- In this way, we often arrive at an optimal solution more quickly than if we simply proceeded blindly in a predetermined order
- Downside: memory consumption is higher than in backtracking, many nodes may be pending at the same time

Best-first search

```
#include <iostream>
#include <vector>
#include <cfloat> // For DBL_MAX
#include <queue>  // For priority queues

using namespace std;

// Input data
int n;
vector<vector<double>> D;

// Output data
vector<int> best_s;
double      best_c = DBL_MAX;

struct Node { ... }; // A Node is a node of the search tree

int main() { ... }   // Finds the tour with minimum cost
```

Best-first search

```
int main() {
  cin >> n;
  D = vector<vector<double>>(n, vector<double>(n));
  for (auto& R : D) for (auto& d : R) cin >> d;

  priority_queue<Node> pq; // Manages nodes pending to explore
  pq.push(Node());// Push root of search tree on priority queue
  while (not pq.empty()) {      // While there are pending nodes
    Node x = pq.top(); // Take the most promising pending node
    pq.pop();
    if (x.lb < best_c) { // Else prune node
      if (x.t == n) { // If complete, lower bound is the cost
        best_c = x.c; // So this is better than the best so far
        best_s = x.s;
      }
      else for (int u = 0; u < n; ++u)
          if (u != x.v and x.s[u] == -1 and
              x.c + D[x.v][u] < best_c) {
            Node y = x;
            y.extend(u); // Add u to the partial route
            if (y.lb < best_c) pq.push(y); // Else prune node
} } }       }
```

Best-first search

```
struct Node {
  vector<int> s;   // s[u] is the city that follows u
  int v, t;        // v = last city, t = #cities partial route
  double c, lb;    // c = partial cost, lb = lower bound

  Node() : s(n, -1), v(0), t(1), c(0), lb(lower_bound()) { }

  void extend(int u) { // Adds u to the partial route
    s[v] = u;          // u follows v
    c += D[v][u];      // Add the weight of v -> u to the cost
    ++t;               // One more city
    v = u;             // Now u becomes the last city
    if (t == n) {      // u completes the tour, go back to 0
      c += D[u][0];
      s[u] = 0;
      lb = c;
    }
    else lb = lower_bound();
  }

  friend bool operator <(const Node& x, const Node& y) {
     return x.lb > y.lb; } // To have a min-priority queue
```

Best-first search

```
//These functions are essentially the same as before
  double min_weight_last() {
    double min = DBL_MAX;
    for (int z = 0; z < n; ++z)
      if (z != v and s[z] == -1 and min > D[v][z])
        min = D[v][z];
    return min;
  }
  double min_weight_unas(int w) {
    double min = DBL_MAX;
    for (int z = 0; z < n; ++z)
      if (z != w and
         (z == 0 or (s[z] == -1 and z != v)) and min > D[w][z])
        min = D[w][z];
    return min;
  }
  double lower_bound() {
    double sum = c;
    for (int w = 0; w < n; ++w)
      if (w == v)           sum += min_weight_last();
      else if (s[w] == -1) sum += min_weight_unas(w);
    return sum; }
```