EXACT ALGORITHMS

- We need to solve problem exactly
EXACT ALGORITHMS

• We need to solve problem exactly

• Problem takes exponential time solve exactly
Exact Algorithms

- We need to solve problem exactly

- Problem takes exponential time solve exactly

- Intelligent exhaustive search: finding optimal solution without going through all candidate solutions
## Running Time

<table>
<thead>
<tr>
<th>running time:</th>
<th>$n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than $10^9$:</td>
<td>$10^9$</td>
<td>$10^{4.5}$</td>
<td>$10^3$</td>
<td>$12$</td>
</tr>
</tbody>
</table>
## Running Time

<table>
<thead>
<tr>
<th>running time:</th>
<th>( n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than ( 10^9 ):</td>
<td>( 10^9 )</td>
<td>( 10^{4.5} )</td>
<td>( 10^3 )</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>running time:</th>
<th>( n! )</th>
<th>( 4^n )</th>
<th>( 2^n )</th>
<th>( 1.308^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than ( 10^9 ):</td>
<td>12</td>
<td>14</td>
<td>29</td>
<td>77</td>
</tr>
</tbody>
</table>
Traveling Salesman Problem (TSP)
Traveling Salesman Problem

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once.
TRAVELING SALESMAN PROBLEM

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once.

```plaintext
length: 9
```
• Classical optimization problem with countless number of real life applications (see Lecture 1)
ALGORITHMS

• Classical optimization problem with countless number of real life applications (see Lecture 1)

• No polynomial time algorithms known
ALGORITHMS

• Classical optimization problem with countless number of real life applications (see Lecture 1)
• No polynomial time algorithms known
• We’ll see exact exponential-time algorithms
A naive algorithm just checks all possible $\sim n!$ cycles.
A naive algorithm just checks all possible $\sim n!$ cycles.

We’ll see

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$
**Brute Force Solution**

A naive algorithm just checks all possible $\sim n!$ cycles.

We’ll see

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$
- The running time is exponential, but is much better than $n!$
Dynamic Programming

• Dynamic programming is one of the most powerful algorithmic techniques
Dynamic Programming

• Dynamic programming is one of the most powerful algorithmic techniques
• Rough idea: express a solution for a problem through solutions for smaller subproblems
Dynamic Programming

- Dynamic programming is one of the most powerful algorithmic techniques.
- Rough idea: express a solution for a problem through solutions for smaller subproblems.
- Solve subproblems one by one. Store solutions to subproblems in a table to avoid recomputing the same thing again.
• For a subset of vertices $S \subseteq \{1, \ldots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at $i$ and visits all vertices from $S$ exactly once.
Subproblems

- For a subset of vertices $S \subseteq \{1, \ldots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at $i$ and visits all vertices from $S$ exactly once.
- $C(\{1\}, 1) = 0$ and $C(S, 1) = +\infty$ when $|S| > 1$. 
Consider the second-to-last vertex \( j \) on the required shortest path from 1 to \( i \) visiting all vertices from \( S \).
• Consider the second-to-last vertex \( j \) on the required shortest path from 1 to \( i \) visiting all vertices from \( S \)

• The subpath from 1 to \( j \) is the shortest one visiting all vertices from \( S - \{i\} \) exactly once
Recurrence Relation

- Consider the second-to-last vertex $j$ on the required shortest path from 1 to $i$ visiting all vertices from $S$
- The subpath from 1 to $j$ is the shortest one visiting all vertices from $S - \{i\}$ exactly once
- Hence
  \[
  C(S, i) = \min_j \{C(S - \{i\}, j) + d_{ji}\}, \text{ where the minimum is over all } j \in S \text{ such that } j \neq i
  \]
• Need to process all subsets $S \subseteq \{1, \ldots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed.
ORDER OF SUBPROBLEMS

• Need to process all subsets $S \subseteq \{1, \ldots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed.

• For example, we can process subsets in order of increasing size.
$C(\ast, \ast) \leftarrow +\infty$

$C(\{1\}, 1) \leftarrow 0$
ALGORITHM

\[
C(*, *) \leftarrow +\infty
\]

\[
C(\{1\}, 1) \leftarrow 0
\]

for \( s \) from 2 to \( n \):

for all \( 1 \in S \subseteq \{1, \ldots, n\} \) of size \( s \):

\[
C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\}
\]

return \( \min_i\{C(\{1, \ldots, n\}, i) + d_{i1}\} \)
ALGORITHM

\[ C(\ast, \ast) \leftarrow +\infty \]
\[ C(\{1\}, 1) \leftarrow 0 \]

for \( s \) from 2 to \( n \):

\[ \text{for all } 1 \in S \subseteq \{1, \ldots, n\} \text{ of size } s: \]

\[ \text{for all } i \in S, i \neq 1: \]

\[ \text{for all } j \in S, j \neq i \]

\[ C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\} \]
ALGORITHM

\[ C(\ast, \ast) \leftarrow +\infty \]

\[ C(\{1\}, 1) \leftarrow 0 \]

for s from 2 to n:

for all \(1 \in S \subseteq \{1, \ldots, n\}\) of size s:

for all \(i \in S, i \neq 1:\)

for all \(j \in S, j \neq i\)

\[ C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\} \]

return \(\min_i\{C(\{1, \ldots, n\}, i) + d_{i,1}\}\)
Satisfiability Problem (SAT)
\((x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3)\)
SAT

$$(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3)$$
\( k\text{-SAT} \)

\[ \phi(x_1, \ldots, x_n) = (x_1 \lor \neg x_2 \lor \ldots \lor x_k) \land \]

\[ \ldots \land \]

\[ (x_2 \lor \neg x_3 \lor \ldots \lor x_8) \]

\( \phi \) is satisfiable if \( \exists x \in \{0, 1\}^n : \phi(x) = 1 \).

Otherwise, \( \phi \) is unsatisfiable.
\( k\text{-SAT} \)

\[
\phi(x_1, \ldots, x_n) = (x_1 \lor \neg x_2 \lor \ldots \lor x_k) \land \\
\ldots \\
\land \\
(x_2 \lor \neg x_3 \lor \ldots \lor x_8)
\]

\( \phi \) is \textit{satisfiable} if

\[
\exists x \in \{0, 1\}^n : \phi(x) = 1.
\]

Otherwise, \( \phi \) is \textit{unsatisfiable}
\( k\text{-SAT} \)

\[
\phi(x_1, \ldots, x_n) = (x_1 \lor \neg x_2 \lor \ldots \lor x_k) \land \\
\cdots \\
(x_2 \lor \neg x_3 \lor \ldots \lor x_8)
\]

\( \phi \) is satisfiable if

\[
\exists x \in \{0, 1\}^n: \phi(x) = 1.
\]

Otherwise, \( \phi \) is unsatisfiable

\( n \) Boolean vars, \( m \) clauses
\( k\text{-SAT} \)

\[ \phi(x_1, \ldots, x_n) = (x_1 \lor \neg x_2 \lor \ldots \lor x_k) \land \\
\ldots \land \\
(x_2 \lor \neg x_3 \lor \ldots \lor x_8) \]

\( \phi \) is satisfiable if

\[ \exists x \in \{0, 1\}^n : \phi(x) = 1. \]

Otherwise, \( \phi \) is unsatisfiable

\( n \) Boolean vars, \( m \) clauses

\( k\text{-SAT} \) is SAT where clause length \( \leq k \)
$k$-SAT. EXAMPLES

$$(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3)$$
\( k\text{-SAT. EXAMPLES} \)

\((x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3)\)

\((x_1) \land (\neg x_2) \land (x_3) \land (\neg x_1)\)
Complexity of SAT

- 1-SAT
- 2-SAT
- 3-SAT
- ...
Complexity of SAT

- $P$
- 1-SAT
- 2-SAT
- 3-SAT
- $k$-SAT
- SAT
- NP
But how hard is SAT?
SAT in $2^n$

- $O^*(\cdot)$ suppresses polynomial factors in the input length:

$$2^n n^{10} m^2 = O^*(2^n)$$
SAT in $2^n$

- $O^*(\cdot)$ suppresses polynomial factors in the input length:

\[ 2^n n^{10} m^2 = O^*(2^n) \]

- SAT can be solved in time $O^*(2^n)$
SAT in $2^n$

- $O^*(\cdot)$ suppresses polynomial factors in the input length:

  \[ 2^n n^{10} m^2 = O^*(2^n) \]

- SAT can be solved in time $O^*(2^n)$

- We don’t know how to solve SAT exponentially faster: in time $O^*(1.999^n)$
3-SAT

- \((x_1 \lor x_2 \lor x_9) \land \ldots \land (x_2 \lor \neg x_3 \lor x_8)\)

Consider three sub-problems:
- \(x_1 = 1, \quad x_2 = 1, \quad x_9 = 1\)
- \(x_1 = 0, \quad x_2 = 1, \quad x_9 = 1\)
- \(x_1 = 0, \quad x_2 = 0, \quad x_9 = 1\)

The original formula is SAT if and only if at least one of these formulas is SAT.
3-SAT

- \((x_1 \lor x_2 \lor x_9) \land \ldots \land (x_2 \lor \neg x_3 \lor x_8)\)

Consider three sub-problems:
- \(x_1 = 1\)
- \(x_1 = 0, x_2 = 1, x_9 = 1\)
- \(x_1 = 0, x_2 = 0, x_9 = 1\)

The original formula is SAT iff at least one of these formulas is SAT.
3-SAT

- \((x_1 \lor x_2 \lor x_9) \land \ldots \land (x_2 \lor \neg x_3 \lor x_8)\)

- Consider three sub-problems:
  - \(x_1 = 1\)
  - \(x_1 = 0, x_2 = 1\)
  - \(x_1 = 0, x_2 = 0, x_9 = 1\)

The original formula is SAT iff at least one of these formulas is SAT.
3-SAT

- \((x_1 \lor x_2 \lor x_9) \land \ldots \land (x_2 \lor \neg x_3 \lor x_8)\)

- Consider three sub-problems:
  - \(x_1 = 1\)
  - \(x_1 = 0, x_2 = 1\)
  - \(x_1 = 0, x_2 = 0, x_9 = 1\)

- The original formula is SAT iff at least one of these formulas is SAT
3-SAT. ANALYSIS

• \( T(n) \leq T(n - 1) + T(n - 2) + T(n - 3) \)
3-SAT. Analysis

• \( T(n) \leq T(n-1) + T(n-2) + T(n-3) \)

• \( T(n) \leq 1.85^n \)
3-SAT. Analysis

- $T(n) \leq T(n - 1) + T(n - 2) + T(n - 3)$
- $T(n) \leq 1.85^n$:

\[
T(n) \leq T(n - 1) + T(n - 2) + T(n - 3) \\
\leq 1.85^{n-1} + 1.85^{n-2} + 1.85^{n-3} \\
= 1.85^n \left( \frac{1}{1.85} + \frac{1}{1.85^2} + \frac{1}{1.85^3} \right) \\
< 1.85^n (0.991) \\
< 1.85^n
\]
3-SAT. Analysis

- $T(n) \leq T(n - 1) + T(n - 2) + T(n - 3)$
- $T(n) \leq 1.85^n$

\[
T(n) \leq T(n - 1) + T(n - 2) + T(n - 3) \\
\leq 1.85^{n-1} + 1.85^{n-2} + 1.85^{n-3} \\
= 1.85^n \left( \frac{1}{1.85} + \frac{1}{1.85^2} + \frac{1}{1.85^3} \right) \\
< 1.85^n (0.991) \\
< 1.85^n
\]

- There are even faster algorithms: $1.308^n$ [HKZZ19]
How hard can SAT be?
Algorithmic Complexity of SAT

2-SAT $O(m)$

1-SAT $O(m)$
Algorithmic Complexity of SAT

1-SAT

2-SAT O(m)

3-SAT 1.308^n
Algorithmic Complexity of SAT

\[ \text{k-SAT} \quad 2^n(1-O(1/k)) \]

\[ \vdots \]

\[ 3\text{-SAT} \quad 1.308^n \]

\[ 2\text{-SAT} \quad O(m) \]

\[ 1\text{-SAT} \quad O(m) \]
Algorithmic Complexity of SAT

SAT \(2^n\)

\(k\)-SAT \(2^n(1-O(1/k))\)

3-SAT \(1.308^n\)

2-SAT \(O(m)\)

1-SAT \(O(m)\)