Algorithm Design
Algorithm Analysis

Prof. Dr. Brahim Hnich

March 11, 2013

1

\footnote{Slides based on those by Kevin Wayne. Copyright ©2005 Pearson-Addison Wesley. All rights reserved.}
Computational tractability

Asymptotic order of growth

A survey of common running times
"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.” - Francis Sullivan
"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time?" - Charles Babbage
Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution. Typically takes $2^N$ time or worse for inputs of size $N$. Unacceptable in practice.
Polynomial-time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution. Typically takes $2^N$ time or worse for inputs of size $N$. Unacceptable in practice.

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor $C$. 

\[
\text{There exists constants } c > 0 \text{ and } d > 0 \text{ such that on every input of size } N, \text{ its running time is bounded by } cN^d \text{ steps.}
\]

Definition. An algorithm is poly-time if the above scaling property holds.
Polynomial-time

**Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution. Typically takes $2^N$ time or worse for inputs of size $N$. Unacceptable in practice.

**Desirable scaling property.** When the input size doubles, the algorithm should only slow down by some constant factor $C$.

- There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $cN^d$ steps.
Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution. Typically takes $2^N$ time or worse for inputs of size $N$. Unacceptable in practice.

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor $C$.

- There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $cN^d$ steps.

Definition. An algorithm is poly-time if the above scaling property holds.
Polynomial-time

**Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution. Typically takes $2^N$ time or worse for inputs of size $N$. Unacceptable in practice.

**Desirable scaling property.** When the input size doubles, the algorithm should only slow down by some constant factor $C$.

- There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $cN^d$ steps.

**Definition.** An algorithm is poly-time if the above scaling property holds.

- Choose $C = 2^d$
Worst-Case Analysis

**Worst case running time.** Obtain bound on largest possible running time of algorithm on input of a given size $N$. 

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.
- Average case running time. Obtain bound on running time of algorithm on random input as a function of input size $N$.
- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
Worst-Case Analysis

**Worst case running time.** Obtain bound on largest possible running time of algorithm on input of a given size $N$.

- Generally captures efficiency in practice.
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size $N$.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size $N$.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size $N$. 
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size $N$.
  ▶ Generally captures efficiency in practice.
  ▶ Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size $N$.
  ▶ Hard (or impossible) to accurately model real instances by random distributions.
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size $N$.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size $N$.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
Worst-Case Polynomial-Time

**Definition.** An algorithm is efficient if its running time is polynomial.
Worst-Case Polynomial-Time

**Definition.** An algorithm is efficient if its running time is polynomial.

- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
Worst-Case Polynomial-Time

**Definition.** An algorithm is efficient if its running time is polynomial.

- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
Worst-Case Polynomial-Time

**Definition.** An algorithm is efficient if its running time is polynomial.

- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
Worst-Case Polynomial-Time

**Definition.** An algorithm is efficient if its running time is polynomial.

- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare. E.g., Simplex Method and Unix `grep`.
Why it matters?

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
Asymptotic order of growth

Upper bounds. \( T(n) \) is in \( O(f(n)) \) if there exist constants \( c > 0 \) and \( n0 \geq 0 \) such that for all \( n \geq n0 \) we have \( T(n) \leq cf(n) \).
Asymptotic order of growth

Upper bounds. \( T(n) \) is in \( O(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for all \( n \geq n_0 \) we have \( T(n) \leq cf(n) \).

Lower bounds. \( T(n) \) is in \( \Omega(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for all \( n \geq n_0 \) we have \( T(n) \geq cf(n) \).
Asymptotic order of growth

**Upper bounds.** $T(n)$ is in $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq cf(n)$.

**Lower bounds.** $T(n)$ is in $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq cf(n)$.

**Tight bounds.** $T(n)$ is in $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$. 
Asymptotic order of growth

**Upper bounds.**  \( T(n) \) is in \( O(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for all \( n \geq n_0 \) we have \( T(n) \leq cf(n) \).

**Lower bounds.**  \( T(n) \) is in \( \Omega(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for all \( n \geq n_0 \) we have \( T(n) \geq cf(n) \).

**Tight bounds.**  \( T(n) \) is in \( \Theta(f(n)) \) if \( T(n) \) is both \( O(f(n)) \) and \( \Omega(f(n)) \).

\[
T(n) = 32n^2 + 17n + 32
\]
Asymptotic order of growth

Upper bounds. $T(n)$ is in $O(f(n))$ if there exist constants $c > 0$ and $n0 \geq 0$ such that for all $n \geq n0$ we have $T(n) \leq cf(n)$.

Lower bounds. $T(n)$ is in $\Omega(f(n))$ if there exist constants $c > 0$ and $n0 \geq 0$ such that for all $n \geq n0$ we have $T(n) \geq cf(n)$.

Tight bounds. $T(n)$ is in $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

- $T(n) = 32n^2 + 17n + 32$
- $T(n)$ is in $O(n^2), O(n^3), \Omega(n^2), \Omega(n)$, and $\Theta(n^2)$
Asymptotic order of growth

Upper bounds. $T(n)$ is in $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq cf(n)$.

Lower bounds. $T(n)$ is in $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq cf(n)$.

Tight bounds. $T(n)$ is in $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

$T(n) = 32n^2 + 17n + 32$

$T(n)$ is in $O(n^2), O(n^3), \Omega(n^2), \Omega(n)$, and $\Theta(n^2)$

$T(n)$ is not in $O(n), \Omega(n^3), \Theta(n)$, or $\Theta(n^3)$
Properties

- Transitivity

If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.

If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$.

If $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$.
Properties

- **Transitivity**
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
Properties

- **Transitivity**
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
  - If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$. 
Properties

- Transitivity
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
  - If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$.
  - If $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$.
Properties

- **Transitivity**
  - If \( f \in O(g) \) and \( g \in O(h) \) then \( f \in O(h) \).
  - If \( f \in \Omega(g) \) and \( g \in \Omega(h) \) then \( f \in \Omega(h) \).
  - If \( f \in \Theta(g) \) and \( g \in \Theta(h) \) then \( f \in \Theta(h) \).

- **Additivity**
Properties

- Transitivity
  - If \( f \in O(g) \) and \( g \in O(h) \) then \( f \in O(h) \).
  - If \( f \in \Omega(g) \) and \( g \in \Omega(h) \) then \( f \in \Omega(h) \).
  - If \( f \in \Theta(g) \) and \( g \in \Theta(h) \) then \( f \in \Theta(h) \).

- Additivity
  - If \( f \in O(h) \) and \( g \in O(h) \) then \( f + g \in O(h) \).
Properties

- **Transitivity**
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
  - If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$.
  - If $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$.

- **Additivity**
  - If $f \in O(h)$ and $g \in O(h)$ then $f + g \in O(h)$.
  - If $f \in \Omega(h)$ and $g \in \Omega(h)$ then $f + g \in \Omega(h)$.
Properties

- **Transitivity**
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
  - If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$.
  - If $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$.

- **Additivity**
  - If $f \in O(h)$ and $g \in O(h)$ then $f + g \in O(h)$.
  - If $f \in \Omega(h)$ and $g \in \Omega(h)$ then $f + g \in \Omega(h)$.
  - If $f \in \Theta(h)$ and $g \in \Theta(h)$ then $f + g \in \Theta(h)$. 
Asymptotic Bounds for Some Common Functions

- Polynomials. $a_0 + a_1 n + \cdots + a_d n^d$ is in $\Theta(nd)$ if $a^d > 0$. 
Asymptotic Bounds for Some Common Functions

- Polynomials. \( a_0 + a_1 n + \ldots + a_d n^d \) is in \( \Theta(nd) \) if \( a^d > 0 \).
- Polynomial time. Running time is in \( O(nd) \) for some constant \( d \) independent of the input size \( n \).
Polynomials. $a_0 + a_1 n + + a_d n^d$ is in $\Theta(nd)$ if $a^d > 0$.

Polynomial time. Running time is in $O(nd)$ for some constant $d$ independent of the input size $n$.

Logarithms. $O(\log a n) \in O(\log b n)$ for any constants $a, b > 0$. 
Asymptotic Bounds for Some Common Functions

- Polynomials. \( a_0 + a_1 n + \ldots + a_d n^d \) is in \( \Theta(nd) \) if \( a^d > 0 \).
- Polynomial time. Running time is in \( O(nd) \) for some constant \( d \) independent of the input size \( n \).
- Logarithms. \( O(\log_a n) \in O(\log_b n) \) for any constants \( a, b > 0 \).
- Logarithms. For every \( x > 0 \), \( \log n \in O(nx) \). I.e., logs grow slower than any polynomial.
Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is in $\Theta(nd)$ if $a^d > 0$.

Polynomial time. Running time is in $O(nd)$ for some constant $d$ independent of the input size $n$.

Logarithms. $O(\log a n) \in O(\log b n)$ for any constants $a, b > 0$.

Logarithms. For every $x > 0$, $\log n \in O(nx)$. I.e., logs grow slower than any polynomial.

Exponentials. For every $r > 1$ and every $d > 0$, $n^d \in O(r^n)$. I.e., every exponential grows faster than every polynomial.
Linear time: Find maximum in a list

- Linear time. Running time is at most a constant factor times the size of the input.
- Example problem: Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

```python
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```
Linear time: Merging two sorted lists

- Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.
Linear time: Merging two sorted lists

- Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

\[ \begin{align*}
\text{Append the smaller of} & \quad a_i \text{ and } b_j \text{ to the output.} \\
\text{Merged result} & \quad \begin{array}{c}
]/\\ \quad a_i \\
]/\\ \quad b_j \\
\end{array} \\
A & \quad B
\end{align*} \]

**Figure 2.2** To merge sorted lists $A$ and $B$, we repeatedly extract the smaller item from the front of the two lists and append it to the output.
Linear time: Merging two sorted lists

\begin{verbatim}
i = 1, j = 1
while (both lists are nonempty) {
    if (a_i \leq b_j) append a_i to output list and increment i
    else append b_j to output list and increment j
}
append remainder of nonempty list to output list
\end{verbatim}
Linear time: Merging two sorted lists

Theorem

Merging two lists of size $n$ takes $O(n)$ time.

```
i = 1, j = 1
while (both lists are nonempty) {
    if ($a_i \leq b_j$) append $a_i$ to output list and increment $i$
    else append $b_j$ to output list and increment $j$
}
append remainder of nonempty list to output list
```
Linear time: Merging two sorted lists

Theorem

*Merging two lists of size \( n \) takes \( O(n) \) time.*

Proof.

After each comparison, the length of output list increases by 1. \( \square \)
\(O(n \log n)\): Sorting

- \(O(n \log n)\) time. Arises in divide-and-conquer algorithms.
O(nlogn): Sorting

- $O(n\log n)$ time. Arises in divide-and-conquer algorithms.
- Sorting. Mergesort and heapsort are sorting algorithms that perform $O(n\log n)$ comparisons.
\(O(n \log n)\): Sorting

- \(O(n \log n)\) time. Arises in divide-and-conquer algorithms.
- Sorting. Mergesort and heapsort are sorting algorithms that perform \(O(n \log n)\) comparisons.
- Largest empty interval. Given \(n\) time-stamps \(x_1, \ldots, x_n\) on which copies of a file arrive at a server, what is the largest interval of time when no copies of the file arrive?
$O(n\log n)$: Sorting

- $O(n\log n)$ time. Arises in divide-and-conquer algorithms.
- Sorting. Mergesort and heapsort are sorting algorithms that perform $O(n\log n)$ comparisons.
- Largest empty interval. Given $n$ time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is the largest interval of time when no copies of the file arrive?
- $O(n\log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic Time

- Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.
Quadratic Time

- Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.
- $O(n^2)$ solution. Try all pairs of points.
Quadratic Time

- Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

- $O(n^2)$ solution. Try all pairs of points.

\[
\text{min} \leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2 \\
\text{for } i = 1 \text{ to } n \{ \\
    \text{for } j = i+1 \text{ to } n \{ \\
        d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2 \\
        \text{if } (d < \text{min}) \\
            \text{min} \leftarrow d \\
    \}
\}
\]
Cubic Time

- Set disjointness. Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $\{1, 2, \ldots, n\}$, is there some pair of these which are disjoint?
Cubic Time

- Set disjointness. Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $\{1, 2, \ldots, n\}$, is there some pair of these which are disjoint?

- $O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.
Cubic Time

- Set disjointness. Given \( n \) sets \( S_1, \ldots, S_n \) each of which is a subset of \( \{1, 2, \ldots, n\} \), is there some pair of these which are disjoint?

- \( O(n^3) \) solution. For each pairs of sets, determine if they are disjoint.

```python
foreach set S_i {
    foreach other set S_j {
        foreach element p of S_i {
            determine whether p also belongs to S_j
        }
        if (no element of S_i belongs to S_j)
            report that S_i and S_j are disjoint
    }
}
```
$O(n^k)$ Time

- Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge? N.B., $k$ is a constant!
$O(n^k)$ Time

- Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge? N.B., $k$ is a constant!
- $O(n^k)$ solution. Enumerate all subsets of $k$ nodes.
Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge? N.B., $k$ is a constant!

$O(n^k)$ solution. Enumerate all subsets of $k$ nodes.

```python
foreach subset S of k nodes {
    check whether S in an independent set
    if (S is an independent set)
        report S is an independent set
}
```
$O(n^k)$ Time

- Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge? N.B., $k$ is a constant!
- $O(n^k)$ solution. Enumerate all subsets of $k$ nodes.

```java
foreach subset $S$ of $k$ nodes {
    check whether $S$ in an independent set
    if (S is an independent set)
        report $S$ is an independent set
}
```

- Checking whether $S$ is an independent set can be done in $O(n^2)$
\( O(n^k) \) Time

- Independent set of size \( k \). Given a graph, are there \( k \) nodes such that no two are joined by an edge? N.B., \( k \) is a constant!
- \( O(n^k) \) solution. Enumerate all subsets of \( k \) nodes.

```plaintext
foreach subset \( S \) of \( k \) nodes {
    check whether \( S \) is an independent set
    if (\( S \) is an independent set)
        report \( S \) is an independent set
}
```

- Checking whether \( S \) is an independent set can be done in \( O(n^2) \)
- Number of \( k \) element subsets: \( \binom{n}{k} \leq \frac{n^k}{k!} \)
Independent set of size $k$. Given a graph, are there $k$ nodes such that no two are joined by an edge? N.B., $k$ is a constant!

$O(n^k)$ solution. Enumerate all subsets of $k$ nodes.

```
foreach subset S of k nodes {
    check whether S in an independent set
    if  (S is an independent set)
        report S is an independent set
}
```

Checking whether $S$ is an independent set can be done in $O(n^2)$

Number of $k$ element subsets: $inom{n}{k} \leq \frac{n^k}{k!}$

$O(k^2 \frac{n^k}{k!})$ is in $O(n^k)$
Exponential Time

- Independent set. Given a graph, what is maximum size of an independent set?
Exponential Time

- Independent set. Given a graph, what is maximum size of an independent set?
- $O(n^22^n)$ solution. Enumerate all subsets.
Exponential Time

- Independent set. Given a graph, what is maximum size of an independent set?
- \( O(n^2 2^n) \) solution. Enumerate all subsets.

```plaintext
S* ← \emptyset
foreach subset S of nodes {
    check whether S is an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```