CS3230 – Design and Analysis of Algorithms (S1 AY2024/25)

Lecture 9: Problem Reductions

Problem reductions

- In Week 6, we saw that the problem of finding a **longest palindromic subsequence** of a string reduces to finding an **LCS** of two strings.

Longest palindromic subsequence of x	$\longleftrightarrow$	LCS of x and reverse(x)

Problem reductions

- In Week 6, we saw that the problem of finding a **longest palindromic subsequence** of a string reduces to finding an **LCS** of two strings.

Longest palindromic subsequence of x ⟷ LCS of x and reverse(x)

LCS of two strings of length n can be computed in $O(n^2)$ time. ▷ Longest palindromic subsequence of a string of length n can be computed in $O(n^2)$ time.

No need to design a new algorithm from scratch.

Problem reductions

- Some other examples:
  - Homework problems. Reduction to Single-Source Shortest Paths.
  - Midterm exam. Reduction to the Knapsack problem.

Reductions between computational problems is a fundamental idea in algorithm design.

Problem reductions

- Viewed another way, reductions also allow us to show hardness of a problem from hardness of some other problem.

Such a lower bound is currently not known.

Suppose we can show that longest palindromic subsequence <u>cannot</u> be computed in $O(n^{1.99})$ time.

$\bigtriangledown$

This implies that LCS also <u>cannot</u> be computed in $O(n^{1.99})$ time.

Problem reductions

- We informally say that problem A reduces to problem B if A can be solved as follows.

Another word for "input"

Input: An instance α of A

1. Convert α to an instance β of B.
2. Solve β to obtain a solution $B(\beta)$ for β.
3. Convert $B(\beta)$ to a solution $A(\alpha)$ for α.

Output: A solution $A(\alpha)$ for α.

Problem reductions

Longest palindromic subsequence

LCS

- We informally say that problem A reduces to problem B if A can be solved as follows.

$A = x$
$B = \text{reverse}(x)$

Input string x

Input: An instance α of A

1. Convert α to an instance β of B.
2. Solve β to obtain a solution $B(\beta)$ for β.
3. Convert $B(\beta)$ to a solution $A(\alpha)$ for α.

Output: A solution $A(\alpha)$ for α.

Run an LCS algorithm on β.

$A(\alpha) = B(\beta)$

Problem reductions

Longest palindromic subsequence　　LCS

- We informally say that problem A reduces to problem B if A can be solved as follows.

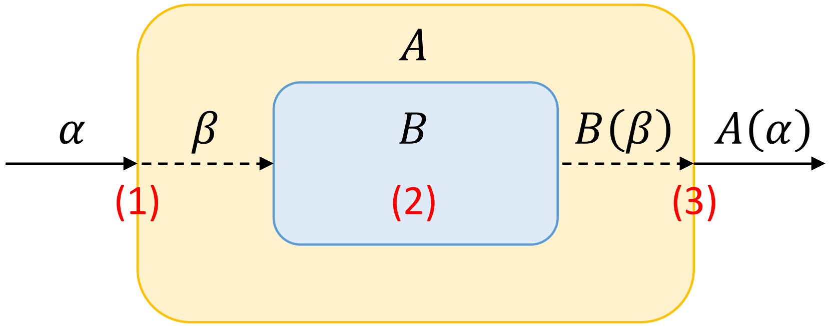
Input string x

$A = x$
$B = \text{reverse}(x)$

Input: An instance α of A

1. Convert α to an instance β of B.
2. Solve β to obtain a solution $B(\beta)$ for β.
3. Convert $B(\beta)$ to a solution $A(\alpha)$ for α.

Output: A solution $A(\alpha)$ for α.



Run an LCS algorithm on β.

$A(\alpha) = B(\beta)$

Matrix multiplication and squaring

MAT-MULTI

Input: two $(n \times n)$ matrices A and B

Output: $A \cdot B$

MAT-SQR

Input: one $(n \times n)$ matrix C

Output: C^2

Matrix multiplication and squaring

MAT-MULTI $\longleftarrow$ MAT-SQR

Input: two $(n \times n)$ matrices A and B

Output: $A \cdot B$

Input: one $(n \times n)$ matrix C

Output: C^2

Claim: MAT-SQR reduces to MAT-MULTI.

Proof: Given input matrix C for MAT-SQR, let $A = C$ and $B = C$ be the inputs for MAT-MULTI. Clearly, $A \cdot B = C^2$.

Matrix multiplication and squaring

MAT-MULTI $\longrightarrow$ MAT-SQR

Input: two $(n \times n)$ matrices A and B

Output: $A \cdot B$

Input: one $(n \times n)$ matrix C

Output: C^2

Claim: MAT-MULTI reduces to MAT-SQR.

Proof: Given input matrices A and B:

- Let $C = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}$ be the input for MAT-SQR.
- From $C^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$ we can learn AB.

Exercise

- Consider the following two problems:

SUM(0)

Input:
- ❑ An array B of length n

Output:
- ❑ $i, j \in [n]$ such that $B[i] + B[j] = 0$

if such indices (i, j) exist.

SUM(T)

Input:
- ❑ An array B of length n
- ❑ A number T

Output:
- ❑ $i, j \in [n]$ such that $B[i] + B[j] = T$

if such indices (i, j) exist.

Exercise

- By setting $T = 0$, **SUM(0)** reduces to **SUM(T)**.
- Can you reduce **SUM(T)** to **SUM(0)**?

- Consider the following two problems:

SUM(0)

Input:
- ❑ An array B of length n

Output:
- ❑ $i, j \in [n]$ such that $B[i] + B[j] = 0$

if such indices (i, j) exist.

SUM(T)

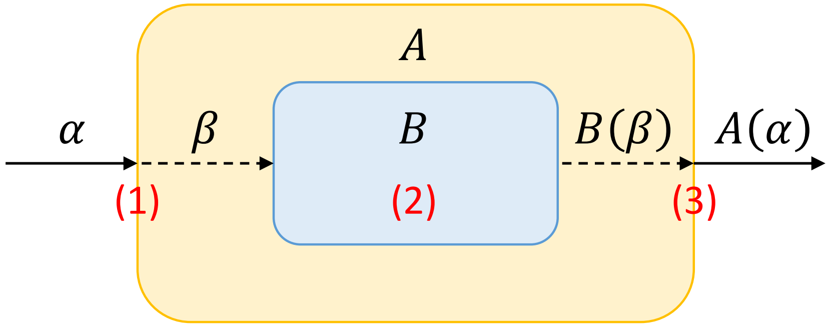
Input:
- ❑ An array B of length n
- ❑ A number T

Output:
- ❑ $i, j \in [n]$ such that $B[i] + B[j] = T$

if such indices (i, j) exist.

$p(n)$-time Reduction

- We say that there is a $p(n)$-time reduction from A to B if A can be solved as follows.



At most $p(n)$ size.

Input: An instance α of A

1. Convert α to an instance β of B. — At most $p(n)$ time.
2. Solve β to obtain a solution $B(\beta)$ for β.
3. Convert $B(\beta)$ to a solution $A(\alpha)$ for α. — At most $p(n)$ time.

Output: A solution $A(\alpha)$ for α.

$p(n)$-time Reduction

Examples:

- $O(n)$-time reduction from longest palindromic subsequence to LCS.
- $O(n)$-time reduction from matrix multiplication to matrix squaring.

At most $p(n)$ size.

Input: An instance α of A

1. Convert α to an instance β of B.
2. Solve β to obtain a solution $B(\beta)$ for β.
3. Convert $B(\beta)$ to a solution $A(\alpha)$ for α.

Output: A solution $A(\alpha)$ for α.

At most $p(n)$ time.

At most $p(n)$ time.

Running time composition

Claim:

Suppose the following two conditions are met:

- There is a $p(n)$-time reduction from problem A to problem B.
- There is a $T(n)$-time algorithm to solve problem B on instances of size n.

$\bigtriangledown$

There is a $\left(T\big(p(n)\big) + O\big(p(n)\big)\right)$-time algorithm to solve problem A on instances of size n.

Running time composition

At most $p(n)$ size.

Input: An instance α of A

1. Convert α to an instance β of B. — At most $p(n)$ time.
2. Solve β to obtain a solution $B(\beta)$ for β. — At most $T\big(p(n)\big)$ time.
3. Convert $B(\beta)$ to a solution $A(\alpha)$ for α. — At most $p(n)$ time.

Output: A solution $A(\alpha)$ for α.

There is a $\Big(T\big(p(n)\big) + O\big(p(n)\big)\Big)$-time algorithm to solve problem A on instances of size n.

Polynomial-time reduction

- If there is an $O(n^c)$-time reduction from A to B for some constant c, then:
  - We say that there is a **polynomial-time reduction** from A to B.
  - We write $A \leq_P B$.

Polynomial-time reduction

- If there is an $O(n^c)$-time reduction from A to B for some constant c, then:
  - We say that there is a **polynomial-time reduction** from A to B.
  - We write $A \leq_P B$.

$p(n) \in O(n^c)$

Suppose B has a polynomial-time algorithm. ← $T(n) \in n^{O(1)}$ time

▽ Running time composition

Then A also has a polynomial-time algorithm. ← $\left(T(p(n)) + O(p(n))\right) \subseteq n^{O(1)}$ time

Polynomial-time reduction

- If there is an $O(n^c)$-time reduction from A to B for some constant c, then:
  - We say that there is a **polynomial-time reduction** from A to B.
  - We write $A \leq_P B$. $p(n) \in O(n^c)$

Suppose A **cannot** be solved in polynomial time.

$\bigtriangledown$ $(p \rightarrow q)$ is the same as $(\neg q \rightarrow \neg p)$

Then B also **cannot** be solved in polynomial time.

Polynomial-time reduction

- If there is an $O(n^c)$-time reduction from A to B for some constant c, then:
  - We say that there is a **polynomial-time reduction** from A to B.
  - We write $A \leq_P B$.

Solvable in polynomial time

- **Intuition:**
  - If B is **easy**, then so is A.
  - If A is **hard**, then so is B.

Cannot be solved in polynomial time

Why polynomial time?

- A working definition of problems efficiently solvable in practice is that they have **polynomial-time algorithms** using "standard" computing hardware.

von Neumann (1953) · Nash (1955) · Gödel (1956) · Cobham (1964) · Edmonds (1965) · Rabin (1966)

Why polynomial time?

- Polynomial functions are **closed under compositions**.
  - If both $T(n)$ and $p(n)$ are polynomial, then $T\big(p(n)\big)$ is also polynomial.

or any other specific function

- Why is it not a good idea to define "efficiently solvable" as $O(n^2)$?
  - No composability.
  - Why $O(n^2)$ is considered efficient and $O(n^{2.001})$ is considered inefficient?

Why polynomial time?

- The notion of polynomial-time algorithms is **robust**:
  - Even if the underlying computing model/hardware is “reasonably” changed, the class of polynomial-time solvable problems remain unchanged.
    - Word-RAM vs. bit-RAM
    - RAM models vs. Turing machines

Why polynomial time?

- The notion of polynomial-time algorithms is **robust**:
  - Even if the underlying computing model/hardware is “reasonably” changed, the class of polynomial-time solvable problems remain unchanged.
    - Word-RAM vs. bit-RAM
    - RAM models vs. Turing machines

What is the time complexity of mergesort on an array of n integers of $O(\log n)$ bits?
- $O(n \log n)$?
- $O(n \log^2 n)$?

Why polynomial time?

- **Pros:**
  - Robustness.
  - Closure under composition.
  - ...

- **Cons:**
  - An $O(n^{100})$-time algorithm is clearly inefficient.

Fortunately, such algorithms are rare.

Input size

- **Recall:** The iterative algorithm to compute $\mathbf{Fib}(n)$ takes $O(n)$ time.

$\mathbf{IFib}(n)$
- If $n \leq 1$
  - return n
- Else,
  - prev2 $= 0$
  - prev1 $= 1$
  - for $i = 2$ to n
    - temp $=$ prev1
    - prev1 $=$ prev1+prev2
    - prev2 $=$ temp
  - return prev1

Is this polynomial-time?

Input size

$\ell = \lceil \log n \rceil$ is the length of the binary representation of n.

$O\left(2^{\ell}\right)$ time.

- **Recall:** The iterative algorithm to compute $\mathbf{Fib}(n)$ takes $O(n)$ time.

Is this polynomial-time?

$\mathbf{IFib}(n)$
- If $n \leq 1$
  - return n
- Else,
  - prev2 $= 0$
  - prev1 $= 1$
  - for $i = 2$ to n
    - temp $=$ prev1
    - prev1 $=$ prev1+prev2
    - prev2 $=$ temp
  - return prev1

A note on encoding

- When we say that an algorithm takes polynomial time, we usually mean that the runtime is polynomial in the **length of the encoding** of the problem instance, in terms of the number of bits.

We can encode n using $\ell = [\log n]$ bits by taking its binary representation.

$\mathbf{IFib}(n)$ **is not** a polynomial-time algorithm.

A note on encoding

- When we say that an algorithm takes polynomial time, we usually mean that the runtime is polynomial in the **length of the encoding** of the problem instance, in terms of the number of bits.

We can encode n using $\ell = [\log n]$ bits by taking its binary representation.

$\mathbf{IFib}(n)$ **is not** a polynomial-time algorithm.

We can also encode n using $\ell = n$ bits by $\overbrace{0 \cdots 0}^{n \text{ times}}$.

$\mathbf{IFib}(n)$ **is** a polynomial-time algorithm.

A note on encoding

- When we say that an algorithm takes polynomial time, we usually mean that the runtime is polynomial in the **length of the encoding** of the problem instance, in terms of the number of bits.

$\boxed{\text{We can encode } n \text{ using } \ell = \lceil \log n \rceil \text{ bits by taking its binary representation.}}$

$\mathbf{IFib}(n)$ **is not** a polynomial-time algorithm.

$\boxed{\text{We can also encode } n \text{ using } \ell = n \text{ bits by } \overbrace{0 \cdots 0}^{n \text{ times}}.}$

$\mathbf{IFib}(n)$ **is** a polynomial-time algorithm.

By default, we consider <u>the most natural</u> encoding, so $\mathbf{IFib}(n)$ is usually not considered a polynomial-time algorithm.

A note on encoding

- When we say that an algorithm takes polynomial time, we usually mean that the runtime is polynomial in the **length of the encoding** of the problem instance, in terms of the number of bits.

$\boxed{\text{We can encode } n \text{ using } \ell = \lceil \log n \rceil \text{ bits by taking its binary representation.}}$

$\mathbf{IFib}(n)$ **is not** a polynomial-time algorithm.

$\boxed{\text{We can also encode } n \text{ using } \ell = n \text{ bits by } \overbrace{0 \cdots 0}^{n \text{ times}}.}$

$\mathbf{IFib}(n)$ **is** a polynomial-time algorithm.

By default, we consider the most natural encoding, so $\mathbf{IFib}(n)$ is usually not considered a polynomial-time algorithm.

The divide-and-conquer algorithm to compute $\mathbf{Fib}(n)$, which takes $O(\log n) = O(\ell)$ time, is considered a polynomial-time algorithm.

A note on encoding

- When we say that an algorithm takes polynomial time, we usually mean that the runtime is polynomial in the **length of the encoding** of the problem instance, in terms of the number of bits.

- For some problems, there is **flexibility** in selecting the encoding.
  - How do you represent a graph $G = (V, E)$ as a binary string?
    - $O(|V|^2)$ bits?
    - $O(|E| \log|V|)$ bits?
    - …

A note on encoding

- When we say that an algorithm takes polynomial time, we usually mean that the runtime is polynomial in the **length of the encoding** of the problem instance, in terms of the number of bits.

- For some problems, there is **flexibility** in selecting the encoding.
  - How do you represent a graph $G = (V, E)$ as a binary string?
    - $O(|V|^2)$ bits?
    - $O(|E| \log|V|)$ bits?
    - …

Usually, the choice of the encoding **does not affect** the notion of polynomial-time algorithms.

Pseudo-polynomial time

The runtime could be exponential in the length of the input.

An algorithm that runs in time polynomial in the numeric value of the input is called a **pseudo-polynomial-time** algorithm.

Pseudo-polynomial time

The runtime could be exponential in the length of the input.

An algorithm that runs in time polynomial in the numeric value of the input is called a **pseudo-polynomial-time** algorithm.

$\mathbf{IFib}(n)$
- If $n \leq 1$
  - return n
- Else,
  - prev2 $= 0$
  - prev1 $= 1$
  - for $i = 2$ to n
    - temp $=$ prev1
    - prev1 $=$ prev1$+$prev2
    - prev2 $=$ temp
  - return prev1

This is pseudo-polynomial time.

The iterative algorithm to compute $\mathbf{Fib}(n)$ takes $O(n)$ time.

The knapsack problem

- **Recall:**

The dynamic programming algorithm for **knapsack** finishes in $O(nW)$ time.

The greedy algorithm for **fractional knapsack** finishes in $O(n \log n)$ time.

- $n =$ the number of items.
- $W =$ the capacity constraint.

Question 1 @ VisuAlgo online quiz

- **Recall:**

The dynamic programming algorithm for **knapsack** finishes in $O(nW)$ time.

The greedy algorithm for **fractional knapsack** finishes in $O(n \log n)$ time.

- $n =$ the number of items.
- $W =$ the capacity constraint.

Question: Are these algorithms polynomial-time?

- Yes for both
- Yes for **knapsack**, no for **fractional knapsack**
- No for **knapsack**, yes for **fractional knapsack**
- No for both

Recap

- Reduction is a basic idea in algorithm design:
  - Using an algorithm for one problem to solve another.

Recap

- Reduction is a basic idea in algorithm design:
  - Using an algorithm for one problem to solve another.

$A \leq_P B$ ▷ If A is hard, then so is B.

▽

If B is easily solvable, then so is A.

Recap

- Reduction is a basic idea in algorithm design:
  - Using an algorithm for one problem to solve another.

$A \leq_P B$ ▷ If A is hard, then so is B.

▽

If B is easily solvable, then so is A.

- **Intuition:** A can be seen as a special case of B.
  - Longest palindromic subsequence is a special case of LCS where one string is the reverse of the other.

Computational complexity theory

- A research field studying sets of computational problems and not individual computational problems.

Computational complexity theory

- A research field studying <u>sets of computational problems</u> and not <u>individual computational problems</u>.

- Some open questions:
  - $\mathbf{P} = \mathbf{PSPACE}$? — Is it true that any computational problem solvable in polynomial space also solvable in polynomial time?
  - $\mathbf{P} = \mathbf{BPP}$? — Is it true that any computational problem solvable in randomized polynomial time also solvable in deterministic polynomial time?
  - $\mathbf{ZPP} = \mathbf{BPP}$? — Is it true that any computational problem solvable in Monte Carlo randomized polynomial time also solvable in Las Vegas randomized polynomial time?

Computational complexity theory

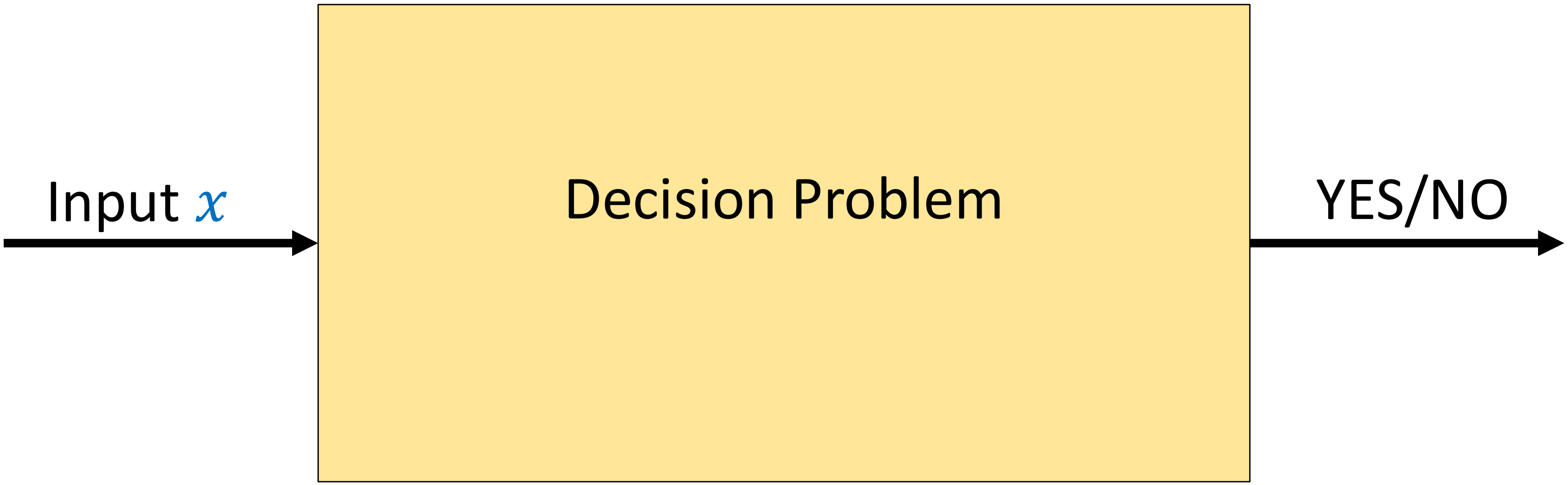
- A research field studying sets of computational problems and not individual computational problems.

Need a framework to talk about all computational problems using the same language.

Decision problems

The set of all binary strings that are valid encodings.

- A **decision problem** is a function that maps an instance space I to the solution space $\{\mathrm{YES}, \mathrm{NO}\}$.



Decision vs. optimization

- **Decision Problem**: Given a directed graph G and two vertices u and v, is there a path from u to v of length $\leq k$?

- **Optimization Problem**: Given a directed graph G and two vertices u and v, what is the length of a shortest path from u to v?

Decision vs. optimization

- Any given optimization problem can be converted into a decision problem:

Given an instance of the optimization problem and a number k, decide if there exists a solution whose value is $\underbrace{\leq k \text{ (or } \geq k\text{)}}$.

Depending on whether the optimization problem is maximization or minimization.

Decision reduces to optimization

Given the value of the optimal solution, simply check whether it is $\leq k$.

- The decision problem is no harder than the optimization problem.

in a different sense!

Optimization reduces to decision

We can use an algorithm for the decision problem to do a binary search for the value of the optimal solution.

in a different sense!

Optimization reduces to decision

We can use an algorithm for the decision problem to do a binary search for the value of the optimal solution.

- The decision problem can be solved in polynomial time **if and only if** the optimization problem can be solved in polynomial time.

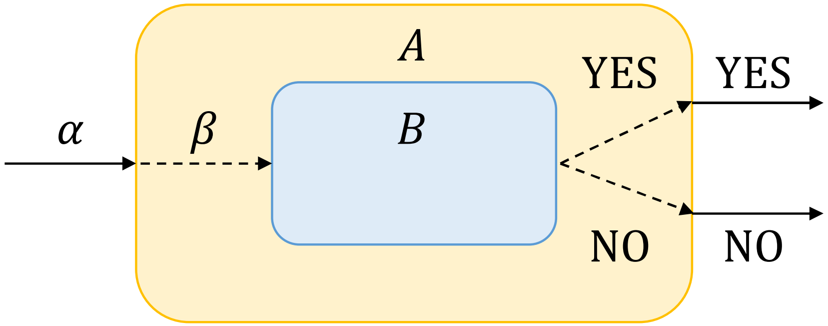
We can focus on decision problems.

Karp reduction

- Given two <u>decision</u> problems A and B, a **polynomial-time reduction** from A to B, denoted $A \leq_P B$, is defined as follows.

A transformation from instances α of A to instances β of B satisfying the two conditions:

- α is a YES-instance for A <u>if and only if</u> β is a YES-instance for B.
- The transformation takes polynomial time in the size of α.



Richard Karp

Question 2 @ VisuAlgo online quiz

Suppose that $A \leq_P B$.

Which of the following statements is **true**?

a) If A can be solved in polynomial time, then so can B.
b) If A can be solved in polynomial time, then B cannot be solved in polynomial time.
c) If A cannot be solved in polynomial time, then neither can B.
d) If A cannot be solved in polynomial time, then B can be solved in polynomial time.

Vertex cover

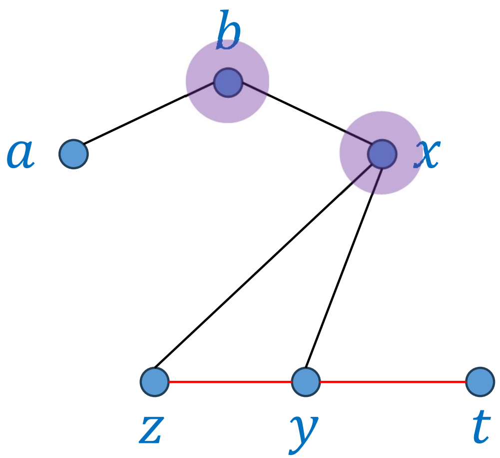
Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is a vertex cover if every edge $e = \{u, v\} \in E$ is covered by X.

$\{u, v\} \cap X \neq \emptyset$

Vertex cover

Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is a vertex cover if every edge $e = \{u, v\} \in E$ is covered by X.

$\{u, v\} \cap X \neq \emptyset$

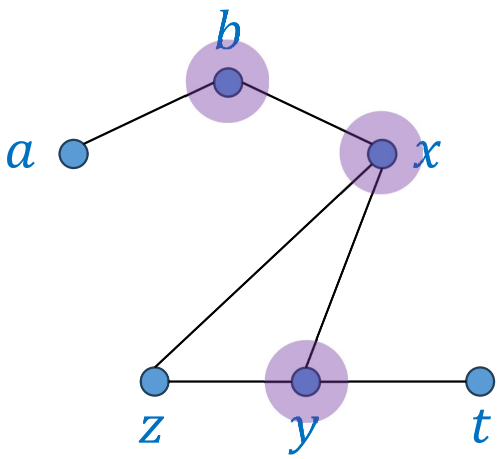


$X = \{b, x\}$ is **not** a vertex cover because $\{z, y\}$ and $\{y, t\}$ are not covered.

Vertex cover

Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is a vertex cover if every edge $e = \{u, v\} \in E$ is covered by X.

$\{u, v\} \cap X \neq \emptyset$

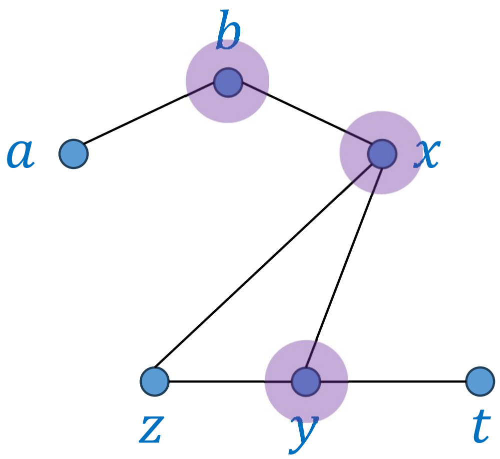


$X = \{b, x, y\}$ is a vertex cover because all edges are covered.

Vertex cover

Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is a vertex cover if every edge $e = \{u, v\} \in E$ is covered by X.

$\{u, v\} \cap X \neq \emptyset$



Optimization version: smallest cardinality

- Compute the size of a minimum vertex cover.

Decision version:

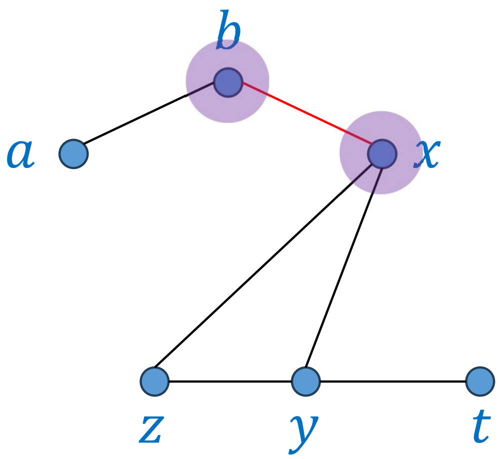
- Decide if there is a vertex cover of size $\leq k$.

Independent set

Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is an independent set if for every $u \in X$ and $v \in X$, we have $\{u, v\} \notin E$.

Independent set

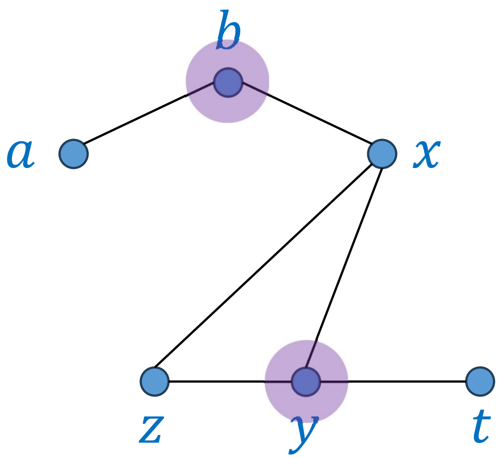
Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is an independent set if for every $u \in X$ and $v \in X$, we have $\{u, v\} \notin E$.



$X = \{b, x\}$ is **not** an independent set because $\{b, x\} \in E$.

Independent set

Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is an independent set if for every $u \in X$ and $v \in X$, we have $\{u, v\} \notin E$.

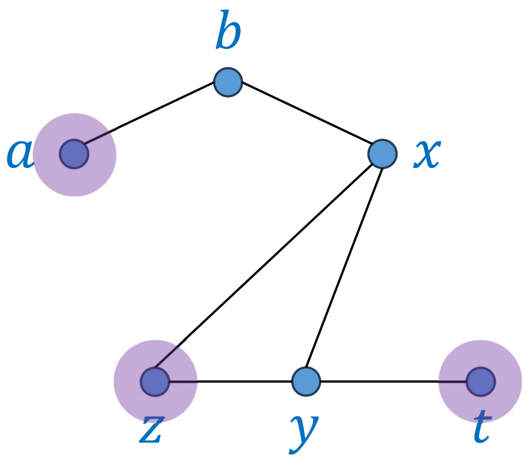


Is it a largest one?

$X = \{b, y\}$ is an independent set.

Independent set

Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is an independent set if for every $u \in X$ and $v \in X$, we have $\{u, v\} \notin E$.

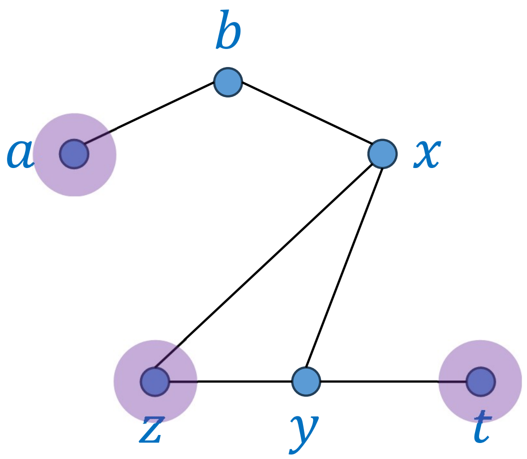


There can be more than one maximum independent set.

$X = \{a, z, t\}$ is a **maximum** independent set.

Independent set

Definition: For an undirected graph $G = (V, E)$, a subset $X \subseteq V$ is an independent set if for every $u \in X$ and $v \in X$, we have $\{u, v\} \notin E$.



Optimization version:

- Compute the size of a maximum independent set.

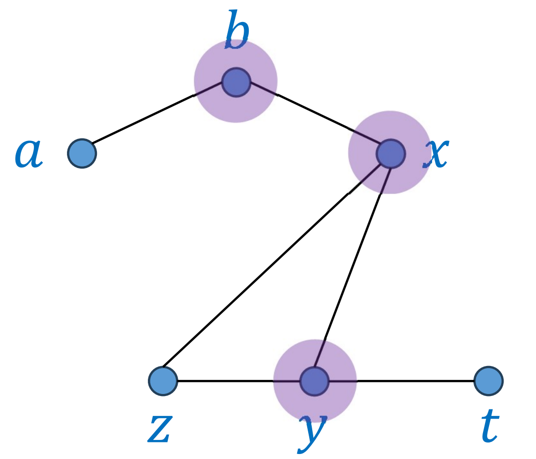
Decision version:

- Decide if there is an independent set of size $\geq k$.

VC vs. IS

VC: Vertex Cover

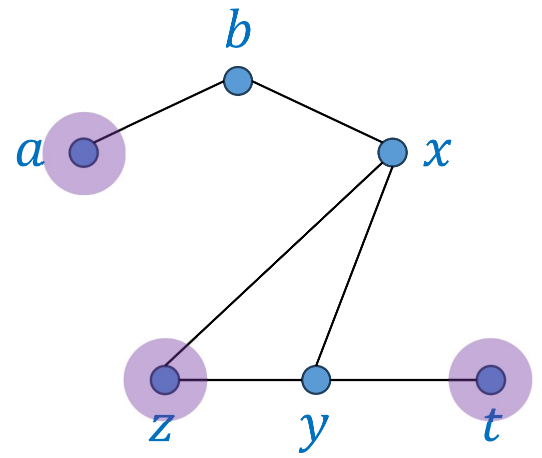
Input: A graph $G = (V, E)$ and a positive integer k.
Goal: Decide if there is a vertex cover of size $\leq k$.



Can you see a relation?

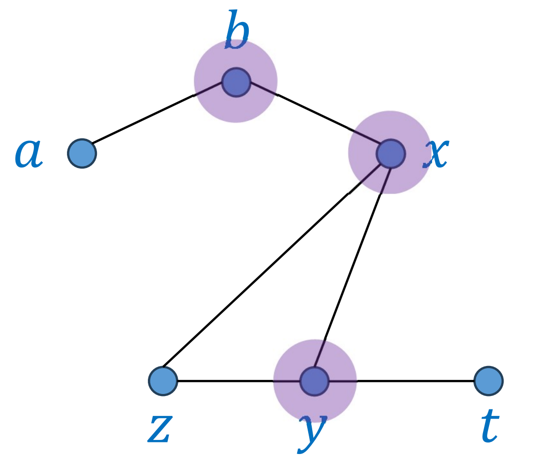
IS: Independent Set

Input: A graph $G = (V, E)$ and a positive integer k.
Goal: Decide if there is an independent set of size $\geq k$.



VC vs. IS

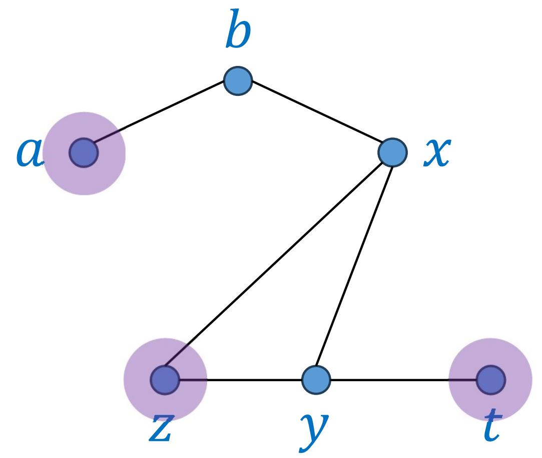
Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.



VC: Vertex Cover

Input: A graph $G = (V, E)$ and a positive integer k.
Goal: Decide if there is a vertex cover of size $\leq k$.

Can you see a relation?



IS: Independent Set

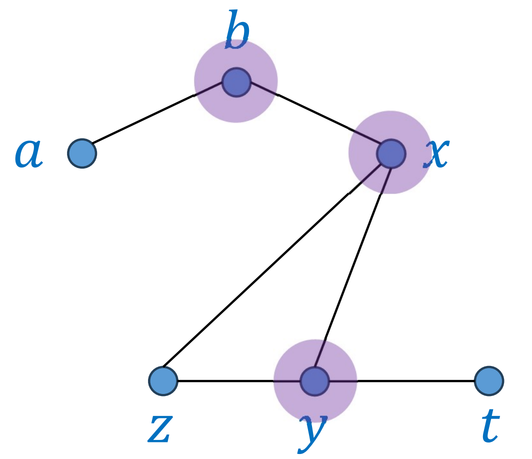
Input: A graph $G = (V, E)$ and a positive integer k.
Goal: Decide if there is an independent set of size $\geq k$.

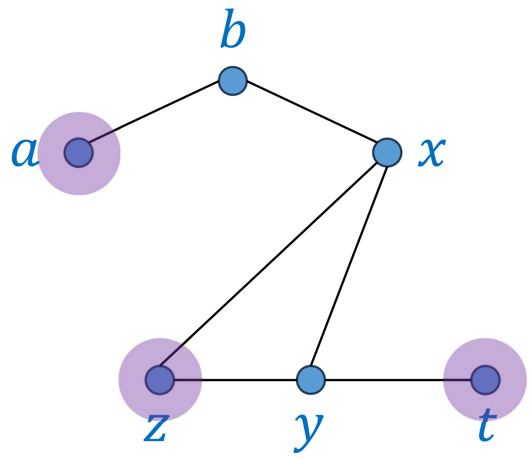
VC vs. IS

Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.

Proof (➔): If $X \subseteq V$ is a vertex cover, then $V \setminus X$ is an independent set.

- Consider any $u \in V \setminus X$ and $v \in V \setminus X$.
- We just need to show that $\{u, v\} \notin E$.





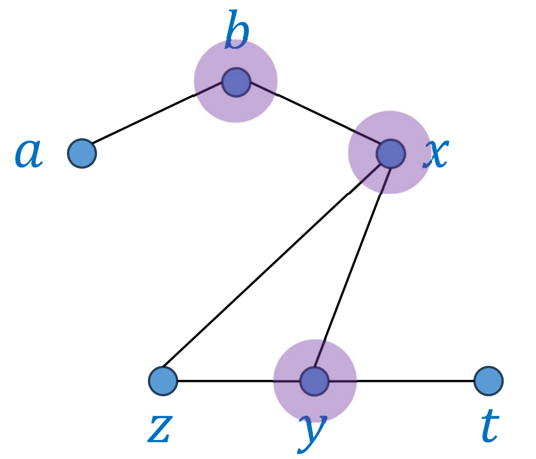
VC vs. IS

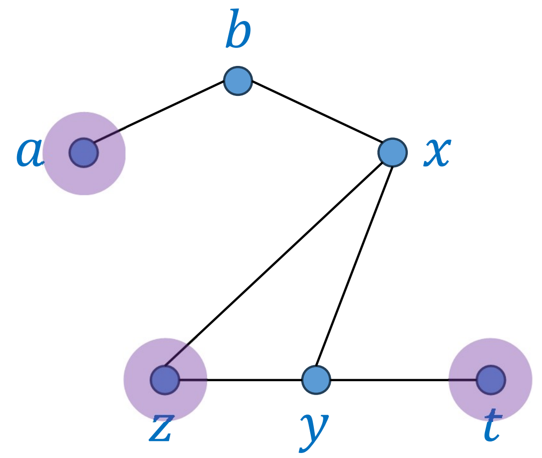
Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.

Proof (➔): If $X \subseteq V$ is a vertex cover, then $V \setminus X$ is an independent set.

- Consider any $u \in V \setminus X$ and $v \in V \setminus X$.
- We just need to show that $\{u, v\} \notin E$.

If $\{u, v\} \in E$, then $\{u, v\}$ is an edge not covered by X, which is impossible.



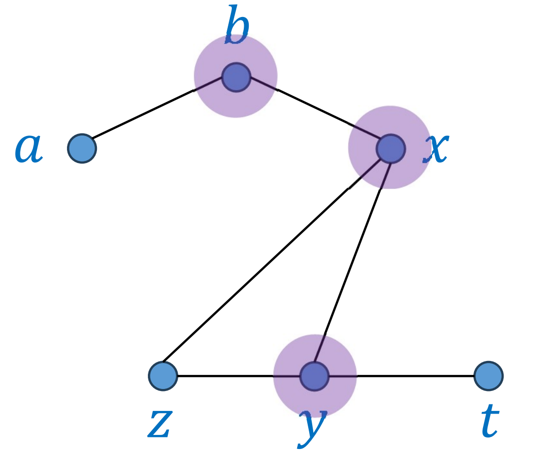


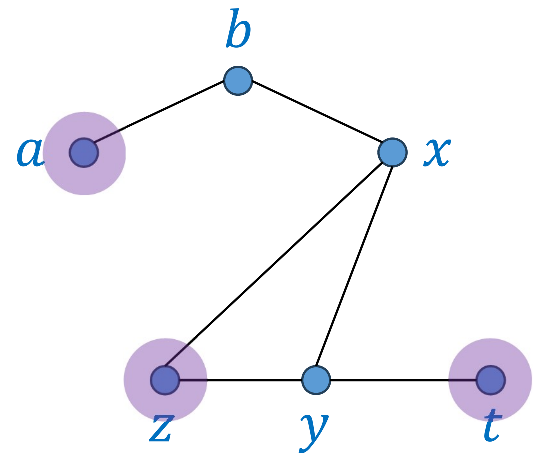
VC vs. IS

Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.

Proof (🡐): If $V \setminus X$ is an independent set, then $X \subseteq V$ is a vertex cover.

- Consider any $\{u, v\} \in E$.
- We just need to show that $\{u, v\}$ is covered by X, meaning that <u>at least one</u> of u and v is in X.





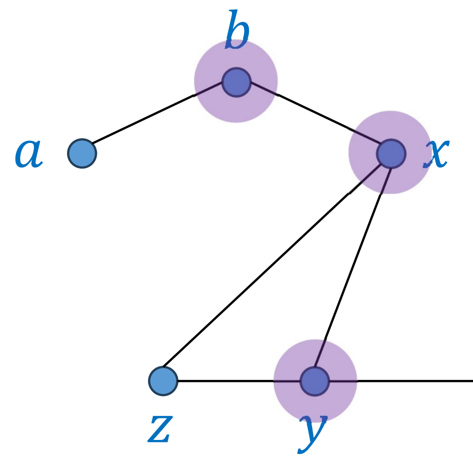
VC vs. IS

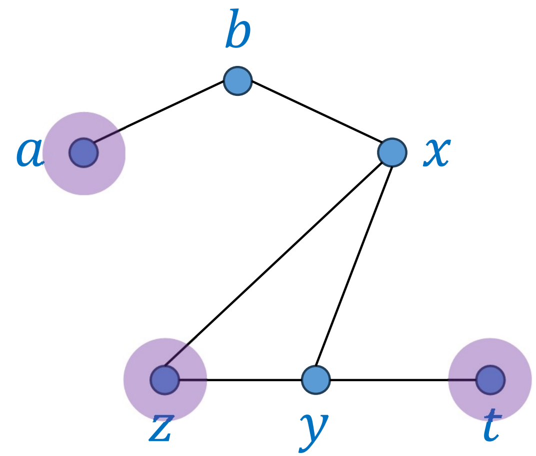
Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.

Proof (🡐): If $V \setminus X$ is an independent set, then $X \subseteq V$ is a vertex cover.

- Consider any $\{u, v\} \in E$.
- We just need to show that $\{u, v\}$ is covered by X, meaning that <u>at least one</u> of u and v is in X.

If both u and v are not in X, then $V \setminus X$ is not an independent set.





VC vs. IS

Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.

VC $\leq_P$ IS

An instance for **VC** — An instance for **IS**

Proof:

- Consider the transformation: $(G, k) \rightarrow (G, |V| - k)$.

VC vs. IS

Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.

VC $\leq_P$ IS

An instance for **VC**   An instance for **IS**

Proof:
- Consider the transformation: $(G, k) \to (G, |V| - k)$.
- The transformation can be done in polynomial time.
- Just need to show that (G, k) is a YES-instance <u>if and only if</u> $(G, |V| - k)$ is a YES-instance.

VC vs. IS

Claim: $X \subseteq V$ is a vertex cover <u>if and only if</u> $V \setminus X$ is an independent set.

VC $\leq_P$ IS

An instance for **VC** An instance for **IS**

Proof:

- Consider the transformation: $(G, k) \rightarrow (G, |V| - k)$.
- The transformation can be done in polynomial time.
- Just need to show that (G, k) is a YES-instance <u>if and only if</u> $(G, |V| - k)$ is a YES-instance.

(G, k) is a YES-instance for **VC**

G has a vertex cover $X \subseteq V$ of size $\leq k$.

if and only if ($\longleftrightarrow$)

$(G, |V| - k)$ is a YES-instance for **IS**

G has an independent set $V \setminus X$ of size $\geq |V| - k$.

VC vs. IS

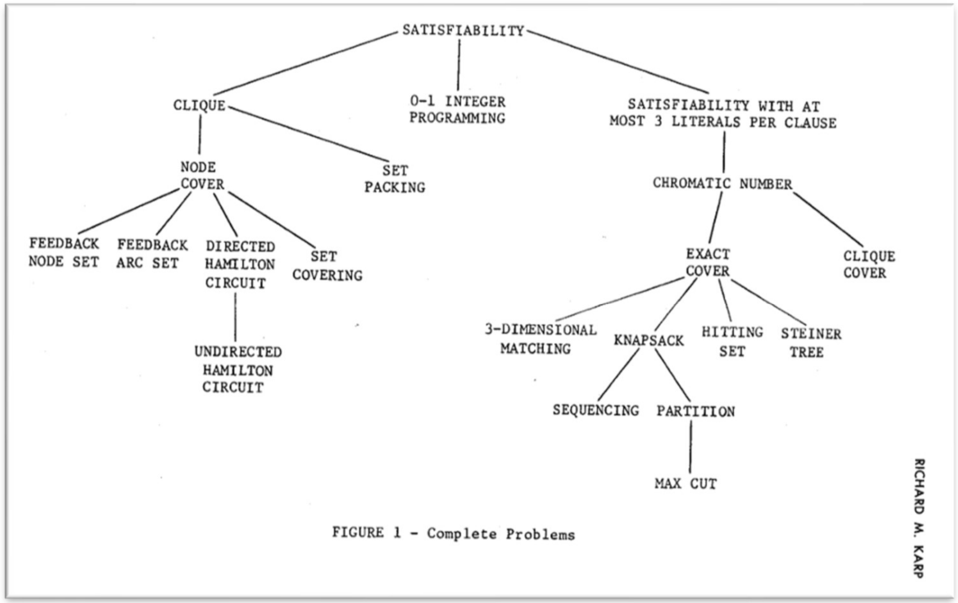
Claim: $X \subseteq V$ is a vertex cover if and only if $V \setminus X$ is an independent set.

VC $\leq_P$ IS ✓

IS $\leq_P$ VC The proof is similar.

Looking forward

- In the next week, we will see that there are many problems admitting polynomial-time reductions to and from **VC** and **IS**.



Richard Karp (1972) "Reducibility Among Combinatorial Problems"

If any one of these problems can be solved in polynomial time, then all these problems can be solved in polynomial time.

Acknowledgement

- The slides are modified from previous editions of this course and similar course elsewhere.
- **List of credits:**
    - Surender Baswana
    - Arnab Bhattacharya
    - Diptarka Chakraborty
    - Yi-Jun Chang
    - Erik Demaine
    - Steven Halim
    - Sanjay Jain
    - Charles Leiserson
    - Wing-Kin Sung
    - Kevin Wayne