THE UNIVERSITY OF MELBOURNE

COMP90038
Algorithms and Complexity

Lecture 5: Brute Force Methods
(with thanks to Harald Søndergaard)

Toby Murray

toby.murray@unimelb.edu.au

DMD 8.17 (Level 8, Doug McDonell Bldg)

http://people.eng.unimelb.edu.au/tobym

@tobycmurray

Brute Force Algorithms

- Straightforward problem solving approach, usually based directly on the problem's statement.

- Exhaustive search for solutions is a prime example.
    - Selection sort
    - String matching
    - Closest pair
    - Exhaustive search for combinatorial solutions
    - Graph traversal

Example: Selection Sort

$$
\begin{array}{l}
\textbf{function } \textsc{SelSort}(A[\cdot], n) \\
\quad \textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad\quad min \leftarrow i \\
\quad\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad\quad \textbf{if } A[j] < A[min] \textbf{ then} \\
\quad\quad\quad\quad min \leftarrow j \\
\quad\quad t \leftarrow A[i] \qquad\qquad \text{// swap A[i] and A[min]} \\
\quad\quad A[i] \leftarrow A[min] \\
\quad\quad A[min] \leftarrow t
\end{array}
$$

Example: Selection Sort

THE UNIVERSITY OF MELBOURNE

$$
\begin{array}{l}
\textbf{function } \textsc{SelSort}(A[\cdot], n) \\
\quad \textbf{for } i \leftarrow 0 \text{ to } n-2 \textbf{ do} \\
\quad\quad min \leftarrow i \\
\quad\quad \textbf{for } j \leftarrow i+1 \text{ to } n-1 \textbf{ do} \\
\quad\quad\quad \textbf{if } A[j] < A[min] \textbf{ then} \\
\quad\quad\quad\quad min \leftarrow j \\
\quad\quad t \leftarrow A[i] \\
\quad\quad A[i] \leftarrow A[min] \\
\quad\quad A[min] \leftarrow t
\end{array}
$$

// swap A[i] and A[min]

Time Complexity:

Example: Selection Sort

function $\textsc{SelSort}(A[\cdot], n)$
  for $i \leftarrow 0$ to $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ to $n-1$ **do**
      if $A[j] < A[min]$ **then**
        $min \leftarrow j$
    $t \leftarrow A[i]$ // swap A[i] and A[min]
    $A[i] \leftarrow A[min]$
    $A[min] \leftarrow t$

Time Complexity: $\Theta(n^2)$

Example: Selection Sort

$$
\begin{array}{l}
\textbf{function } \text{SELSORT}(A[\cdot], n) \\
\quad \textbf{for } i \leftarrow 0 \textbf{ to } n-2 \textbf{ do} \\
\quad\quad min \leftarrow i \\
\quad\quad \textbf{for } j \leftarrow i+1 \textbf{ to } n-1 \textbf{ do} \\
\quad\quad\quad \textbf{if } A[j] < A[min] \textbf{ then} \\
\quad\quad\quad\quad min \leftarrow j \\
\quad\quad t \leftarrow A[i] \qquad \text{// swap A[i] and A[min]} \\
\quad\quad A[i] \leftarrow A[min] \\
\quad\quad A[min] \leftarrow t
\end{array}
$$

Time Complexity: $\Theta(n^2)$

We will soon meet better sorting algorithms

Properties of Sorting Algorithms

- A Sorting algorithm is:
  - **in-place** if it does not require additional memory except, perhaps, for a few units of memory
  - **stable** if it preserves the relative order of elements with identical keys
  - **input-insensitive** if its running time is fairly independent of input properties other than size

Properties of Selection Sort

- While running time is quadratic, selection sort makes only about ***n*** **exchanges**.
- So: **selection sort is a good algorithm for sorting small collections of large records.**
- In-place?
- Stable?
- Input-insensitive?

Properties of Selection Sort

- While running time is quadratic, selection sort makes only about ***n*** **exchanges**.
- So: **selection sort is a good algorithm for sorting small collections of large records.**
- In-place? yes
- Stable?
- Input-insensitive?

Properties of Selection Sort

- While running time is quadratic, selection sort makes only about ***n*** **exchanges**.
- So: **selection sort is a good algorithm for sorting small collections of large records.**
- In-place? yes
- Stable? ?
- Input-insensitive?

Properties of Selection Sort

- While running time is quadratic, selection sort makes only about ***n*** **exchanges**.
- So: **selection sort is a good algorithm for sorting small collections of large records.**
- In-place? yes
- Stable? ?
- Input-insensitive? yes

Selection Sort Stability

key: 4 val: ab	key: 3 val: bc	key: 4 val: de	key: 3 val: fg
0	1	2	3

↑
A[i]

Selection Sort Stability

key: 4 val: ab	key: 3 val: bc	key: 4 val: de	key: 3 val: fg
0	1	2	3

A[i]

Selection Sort Stability

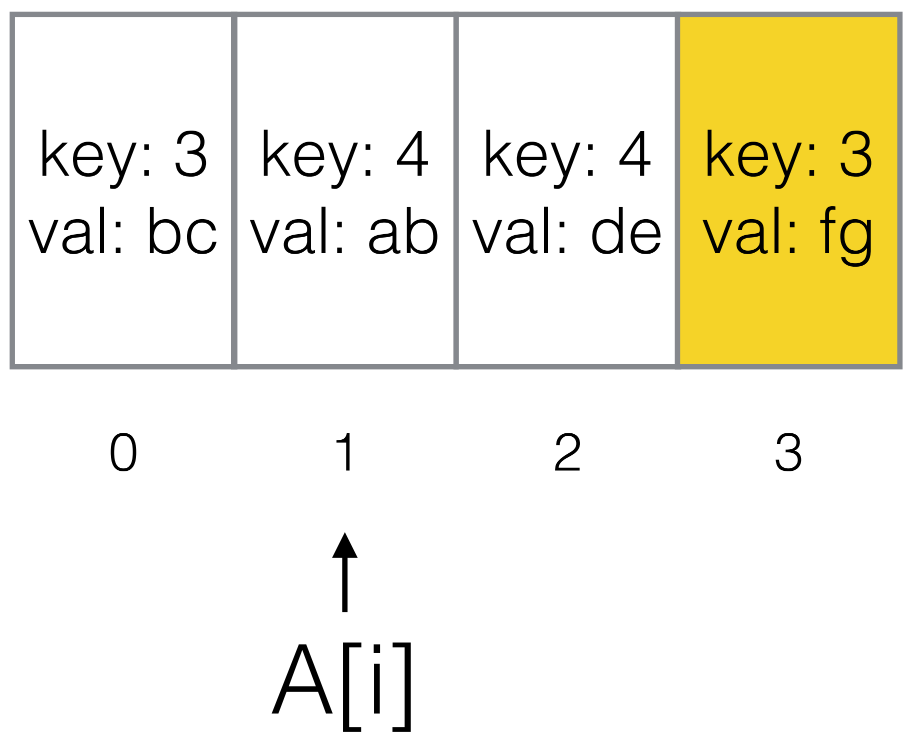
0	1	2	3
key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg

↑
A[i]

Selection Sort Stability

key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg
0	1	2	3
	↑ A[i]		

Selection Sort Stability



Selection Sort Stability

THE UNIVERSITY OF MELBOURNE

key: 3 val: bc	key: 3 val: fg	key: 4 val: de	key: 4 val: ab
0	1	2	3

↑
A[i]

Selection Sort Stability

key: 3 val: bc	key: 3 val: fg	key: 4 val: de	key: 4 val: ab
0	1	2	3

A[i]

the relative order
of the two "4" records
has changed!

Properties of Selection Sort

- While running time is quadratic, selection sort makes only about ***n*** **exchanges**.
- So: **selection sort is a good algorithm for sorting small collections of large records.**
- In-place? yes
- Stable? no
- Input-insensitive? yes

Brute Force String Matching

- **Pattern** *p*: a string of *m* characters to search **for**
- **Text** *t*: a long string of *n* characters to search **in**
- We use *i* to run through the text and *j* to run through the pattern

```
for i ← 0 to n − m do
    j ← 0
    while j < m and p[j] = t[i + j] do
        j ← j + 1
    if j = m then
        return i
return −1
```


Brute Force String Matching

THE UNIVERSITY OF MELBOURNE

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \textbf{ to } n - m \textbf{ do} \\
\quad j \leftarrow 0 \\
\quad \textbf{while } j < m \textbf{ and } p[j] = t[i+j] \textbf{ do} \\
\quad\quad j \leftarrow j + 1 \\
\quad \textbf{if } j = m \textbf{ then} \\
\quad\quad \textbf{return } i \\
\textbf{return } -1
\end{array}
$$

t:	N	O	B	O	D	Y		K	N	O	W	S	
	0	1	2	3	4	5	6	7	8	9	10	11	n

p:	N	O	T	
	0	1	2	m

Brute Force String Matching

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \textbf{ to } n - m \textbf{ do} \\
\quad j \leftarrow 0 \\
\quad \textbf{while } j < m \textbf{ and } p[j] = t[i+j] \textbf{ do} \\
\quad\quad j \leftarrow j + 1 \\
\quad \textbf{if } j = m \textbf{ then} \\
\quad\quad \textbf{return } i \\
\textbf{return } -1
\end{array}
$$

t[i] ↓

t:	N	O	B	O	D	Y		K	N	O	W	S	
	0	1	2	3	4	5	6	7	8	9	10	11	n

p:	N	O	T	
	0	1	2	m

Brute Force String Matching

for $i \leftarrow 0$ to $n - m$ **do**
    $j \leftarrow 0$
    while $j < m$ and $p[j] = t[i + j]$ **do**
        $j \leftarrow j + 1$
    if $j = m$ **then**
        return i
return -1

t[i+j]

t[i]

t:	N	O	B	O	D	Y		K	N	O	W	S	
	0	1	2	3	4	5	6	7	8	9	10	11	n

p:	N	O	T	
	0	1	2	m

p[j]

Brute Force String Matching

for $i \leftarrow 0$ **to** $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ **and** $p[j] = t[i + j]$ **do**
    $j \leftarrow j + 1$
  if $j = m$ **then**
    return i
return -1

t[i+j]

t[i]

t:	N	O	B	O	D	Y		K	N	O	W	S	
	0	1	2	3	4	5	6	7	8	9	10	11	n

p:	N	O	T	
	0	1	2	m

p[j]

Brute Force String Matching

THE UNIVERSITY OF MELBOURNE

for $i \leftarrow 0$ **to** $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ **and** $p[j] = t[i + j]$ **do**
    $j \leftarrow j + 1$
  if $j = m$ **then**
    return i
return -1

t[i+j]

t[i]

t:

0	1	2	3	4	5	6	7	8	9	10	11	n
N	O	B	O	D	Y		K	N	O	W	S	

p:

0	1	2	m
N	O	T	

p[j]

Brute Force String Matching

```
for i ← 0 to n − m do
    j ← 0
    while j < m and p[j] = t[i + j] do
        j ← j + 1
    if j = m then
        return i
return −1
```

t[i]

t:	N	O	B	O	D	Y		K	N	O	W	S	
	0	1	2	3	4	5	6	7	8	9	10	11	n

p:	N	O	T	
	0	1	2	m

Brute Force String Matching

for $i \leftarrow 0$ to $n - m$ **do**
    $j \leftarrow 0$
    while $j < m$ and $p[j] = t[i + j]$ **do**
        $j \leftarrow j + 1$
    if $j = m$ **then**
        return i
return -1

t[i]

t:	N	O	B	O	D	Y		K	N	O	W	S	
	0	1	2	3	4	5	6	7	8	9	10	11	n

p:	N	O	T	
	0	1	2	m

Brute Force String Matching

for $i \leftarrow 0$ to $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ and $p[j] = t[i + j]$ **do**
    $j \leftarrow j + 1$
  if $j = m$ **then**
    return i
return -1

t[i]

t:	N	O	B	O	D	Y		K	N	O	W	S	
	0	1	2	3	4	5	6	7	8	9	10	11	n

p:	N	O	T	
	0	1	2	m

Brute Force String Matching

for $i \leftarrow 0$ to $n - m$ **do**

  $j \leftarrow 0$

  while $j < m$ and $p[j] = t[i + j]$ **do**

    $j \leftarrow j + 1$

  if $j = m$ **then**

    return i

return -1

t[i+j]

t[i]

t:

N	O	B	O	D	Y		K	N	O	W	S	
0	1	2	3	4	5	6	7	8	9	10	11	n

p:

N	O	T	
0	1	2	m

p[j]

Brute Force String Matching

for $i \leftarrow 0$ to $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ and $p[j] = t[i + j]$ **do**
    $j \leftarrow j + 1$
  if $j = m$ **then**
    return i
return -1

t[i] ↓

t:

0	1	2	3	4	5	6	7	8	9	10	11	n
N	O	B	O	D	Y		K	N	O	W	S	

p:

0	1	2	m
N	O	T	

Brute Force String Matching

for $i \leftarrow 0$ to $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ and $p[j] = t[i + j]$ **do**
    $j \leftarrow j + 1$
  if $j = m$ **then**
    return i
return -1

t[i]

t:

0	1	2	3	4	5	6	7	8	9	10	11	n
N	O	B	O	D	Y		K	N	O	W	S	

p:

0	1	2	m
N	O	T	

Analysing Brute Force String Matching

Analysing Brute Force String Matching

Basic operation:

Analysing Brute Force String Matching

Basic operation: comparison p[j] = t[i+j]

Analysing Brute Force String Matching

Basic operation: comparison p[j] = t[i+j]

For each n - m + 1 positions in *t* we make up to m comparisons

Analysing Brute Force String Matching

Basic operation: comparison p[j] = t[i+j]

For each n - m + 1 positions in *t*
we make up to m comparisons

Assuming n much larger than m: $O(mn)$ comparisons.

Analysing Brute Force String Matching

Basic operation: comparison p[j] = t[i+j]

For each n - m + 1 positions in *t*
we make up to m comparisons

Assuming n much larger than m: $O(mn)$ comparisons.

But for **random text** over reasonably large alphabet (e.g. English), **average** running time is linear in *n*

Analysing Brute Force String Matching

Basic operation: comparison p[j] = t[i+j]

For each n - m + 1 positions in *t* we make up to m comparisons

Assuming n much larger than m: $O(mn)$ comparisons.

But for **random text** over reasonably large alphabet (e.g. English), **average** running time is linear in *n*

There are better algorithms, for smaller alphabets such as binary strings or strings of DNA nucleobases. But for many purposes, the brute-force algorithm is acceptable.

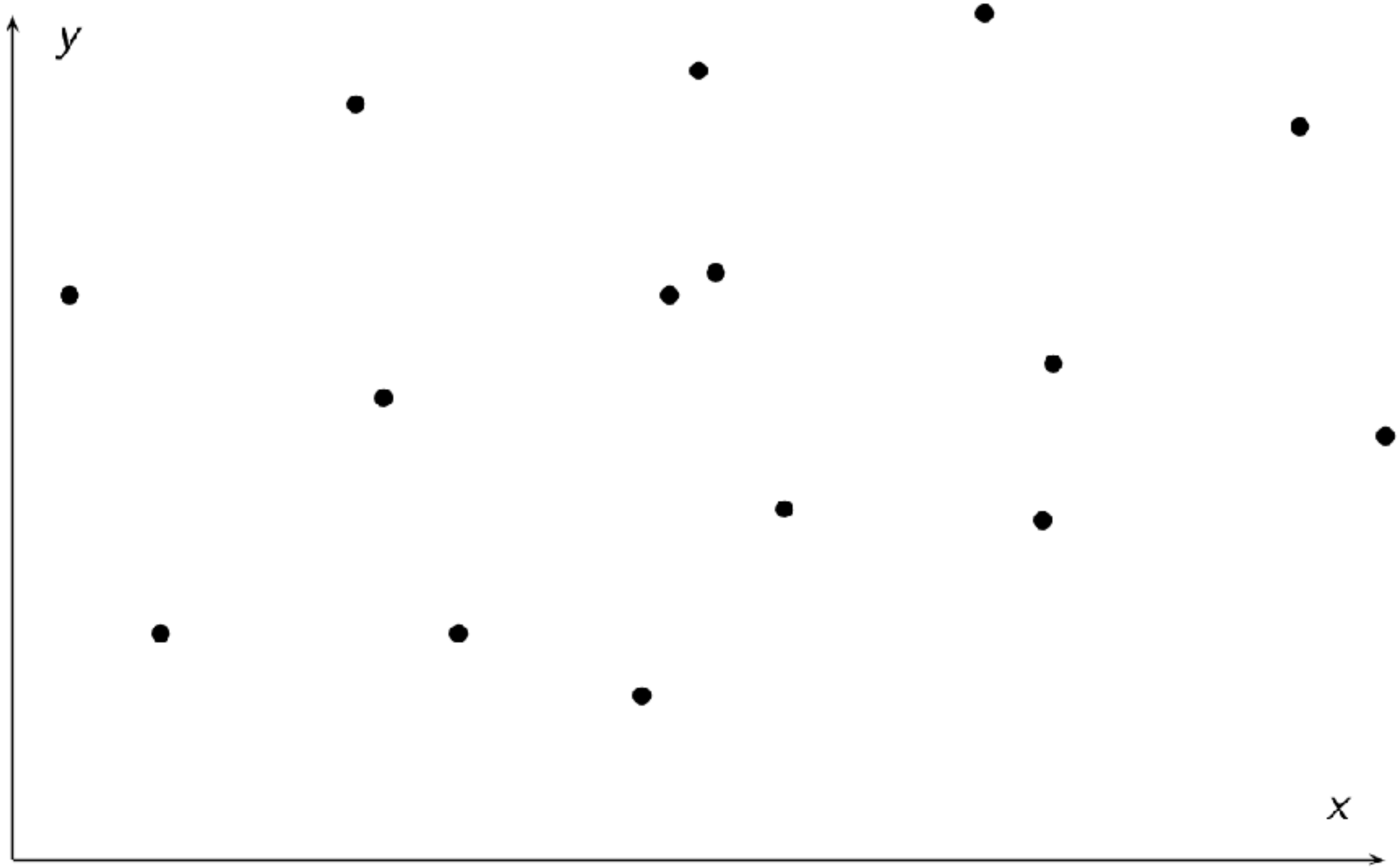
Brute Force Geometric Algorithms: Closest Pair

- **Problem:** Given n points is k-dimensional space, find a pair of points with minimal separating Euclidean distance.

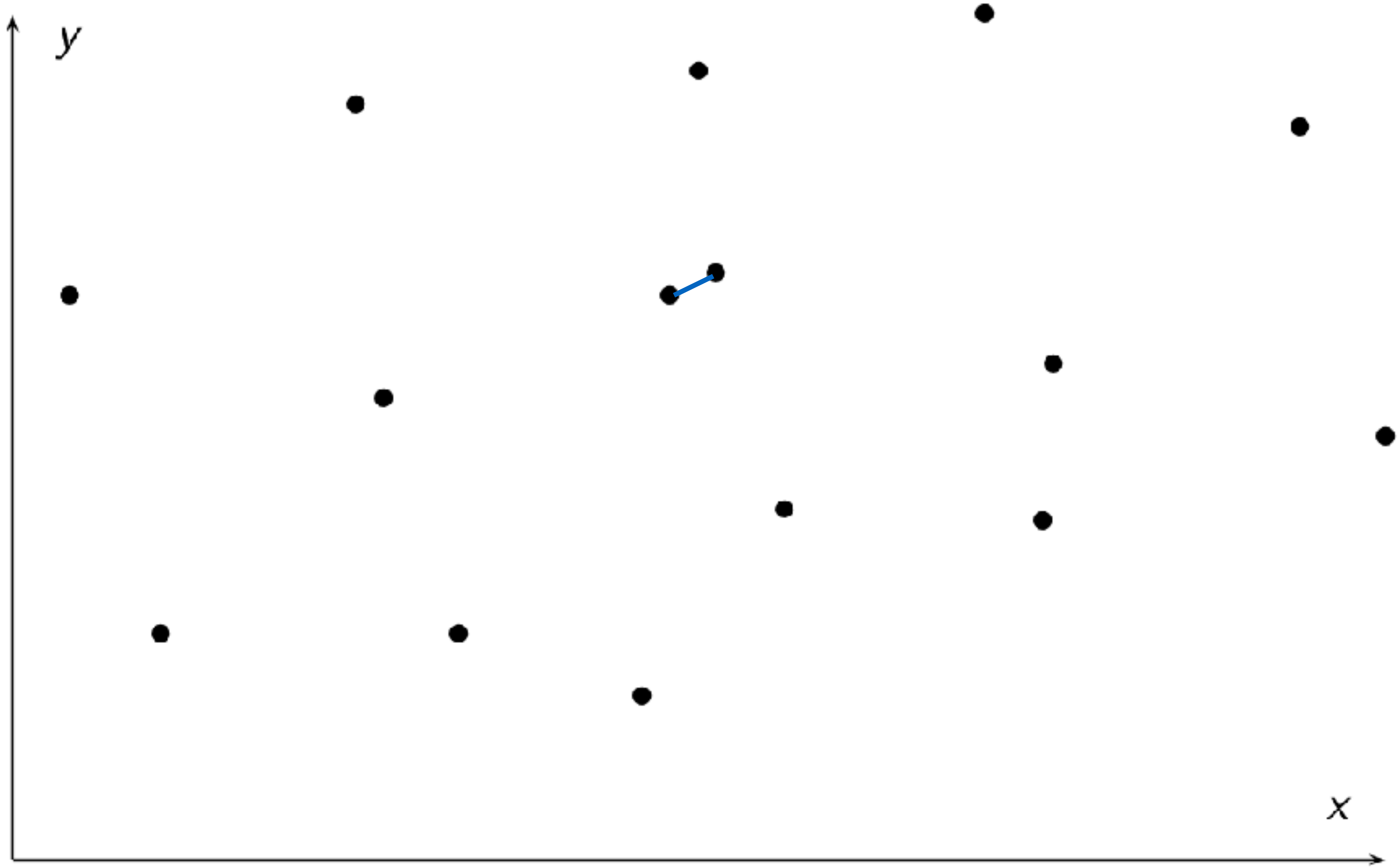
- The brute force approach considers each pair in turn (except that once it has found the distance from x to y, it does not need to consider the distance from y to x).

- For simplicity, we look at the 2-dimensional case, the points being $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$.

Closest Pair Problem (2D)



Closest Pair Problem (2D)

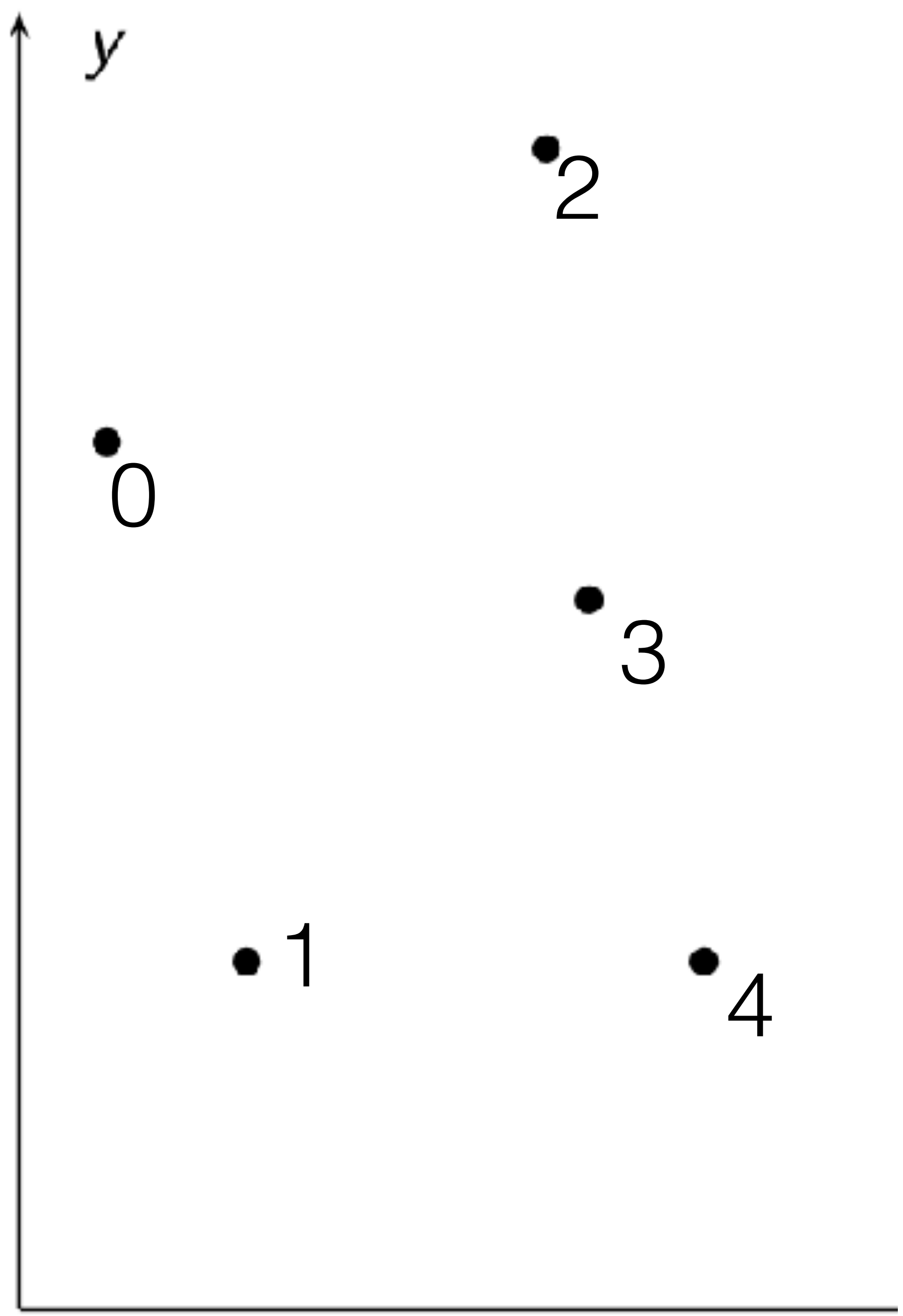


Brute Force Closest Pair

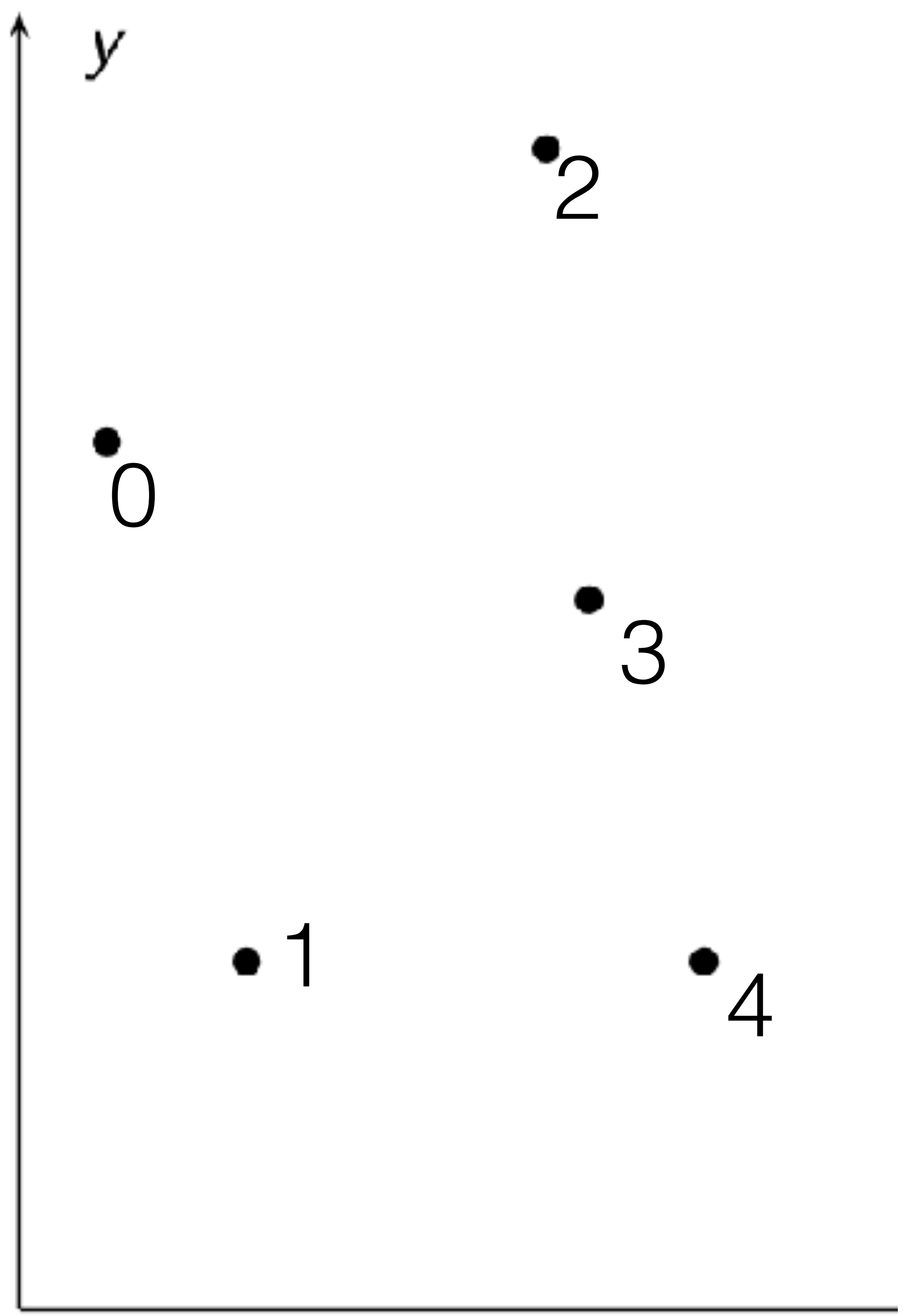
Try all combinations (x_i, y_i) and (x_j, y_j) with $i < j$:

$min \leftarrow \infty$
for $i \leftarrow 0$ to $n-2$ **do**
  for $j \leftarrow i+1$ to $n-1$ **do**
    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ ▷ Distance for this pair
    if $d < min$ **then**
      $min \leftarrow d$ ▷ Smallest distance so far
      $p_1 \leftarrow i$ ▷ Remember this (i,j) combination
      $p_2 \leftarrow j$
return p_1, p_2

Closest Pair Problem (2D)

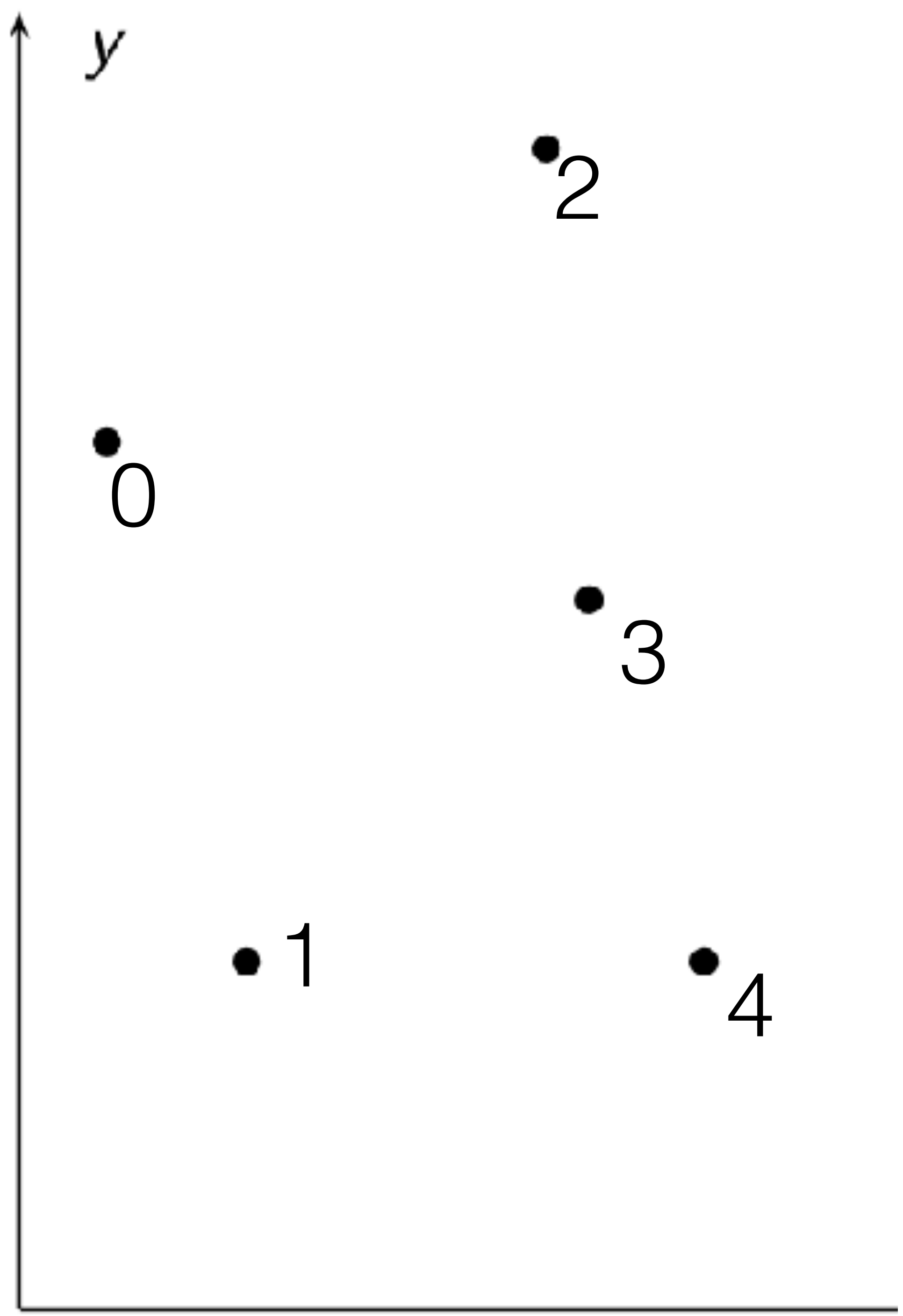


Closest Pair Problem (2D)



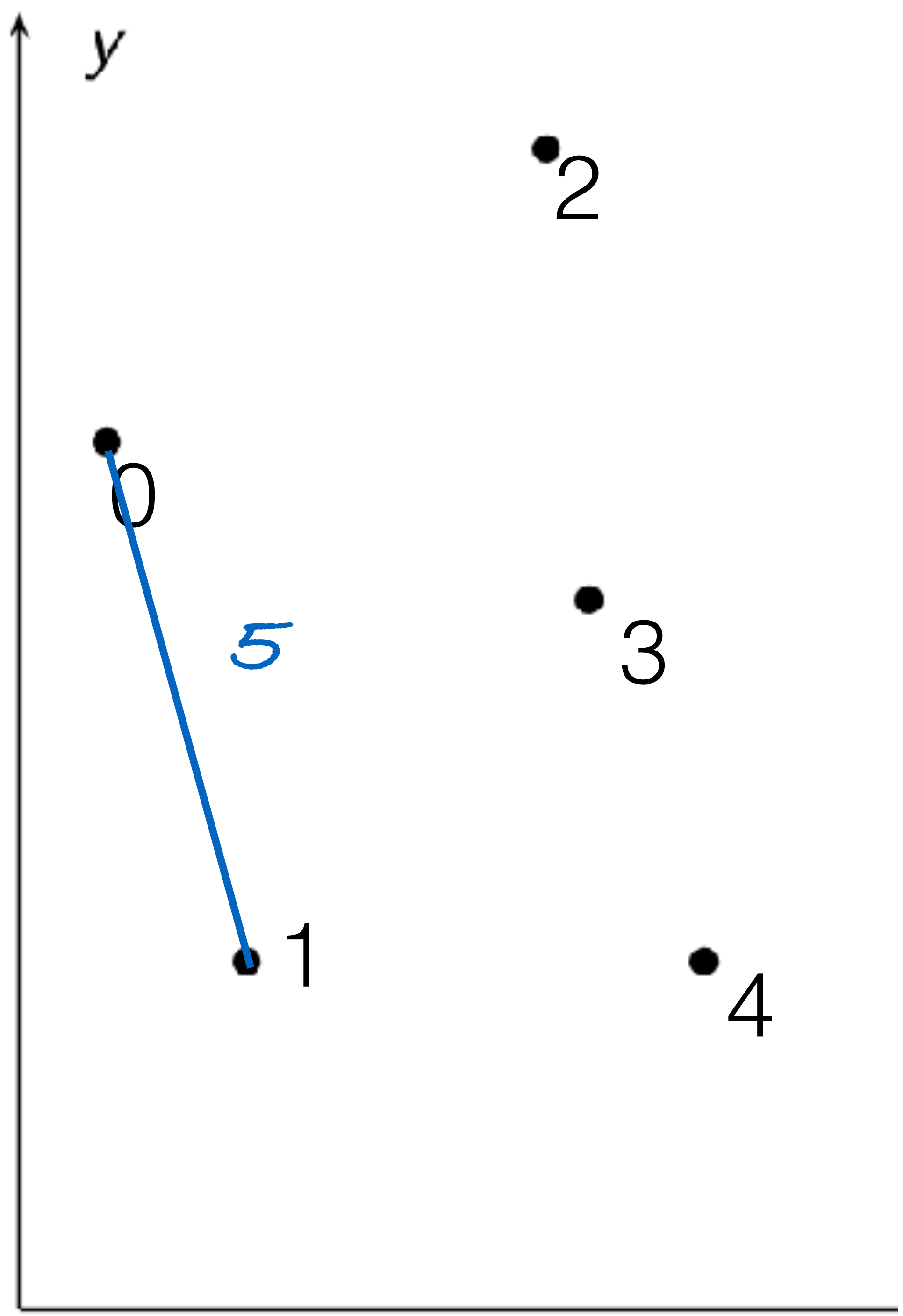
i=0

Closest Pair Problem (2D)



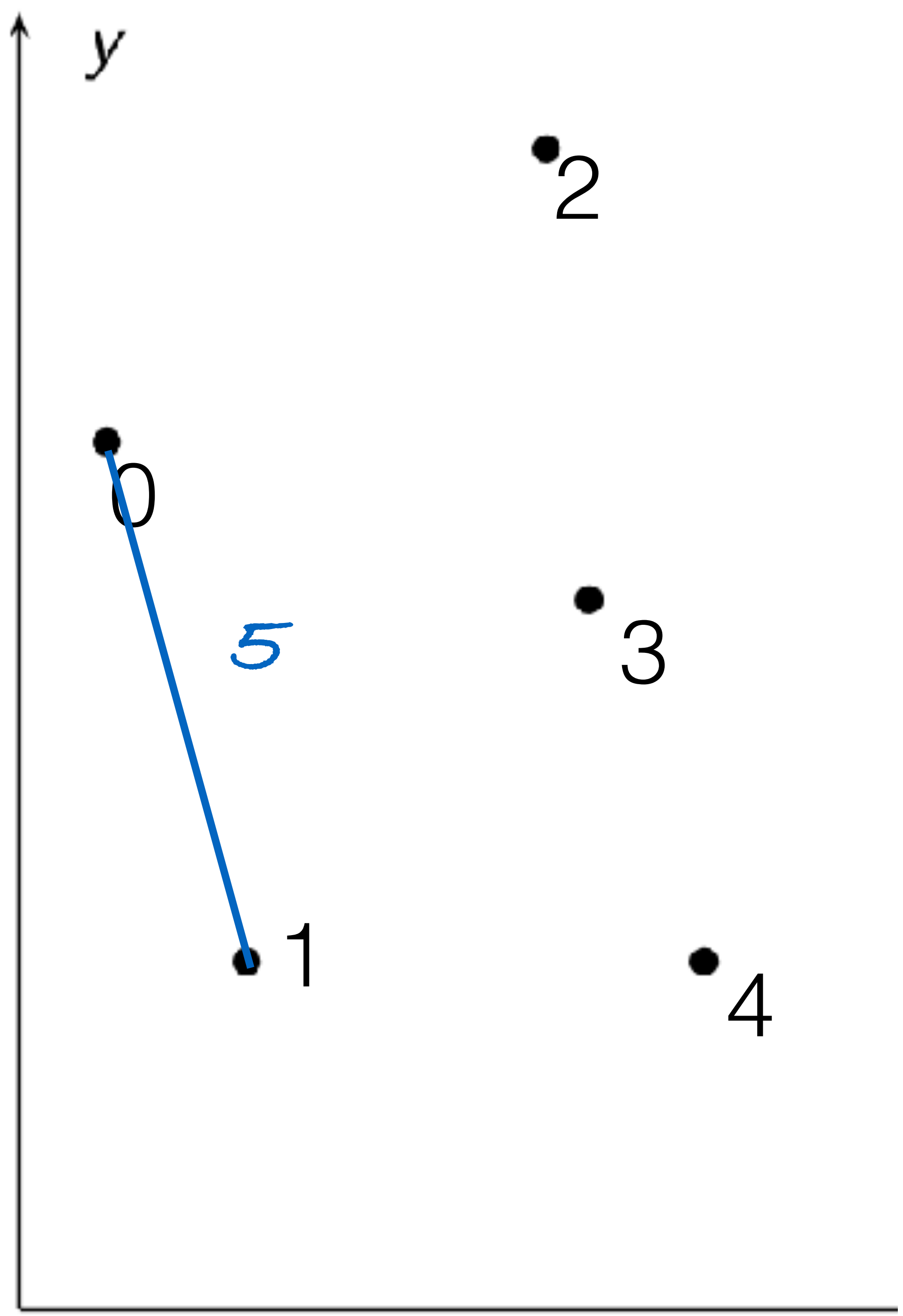
i=0 min=∞

Closest Pair Problem (2D)



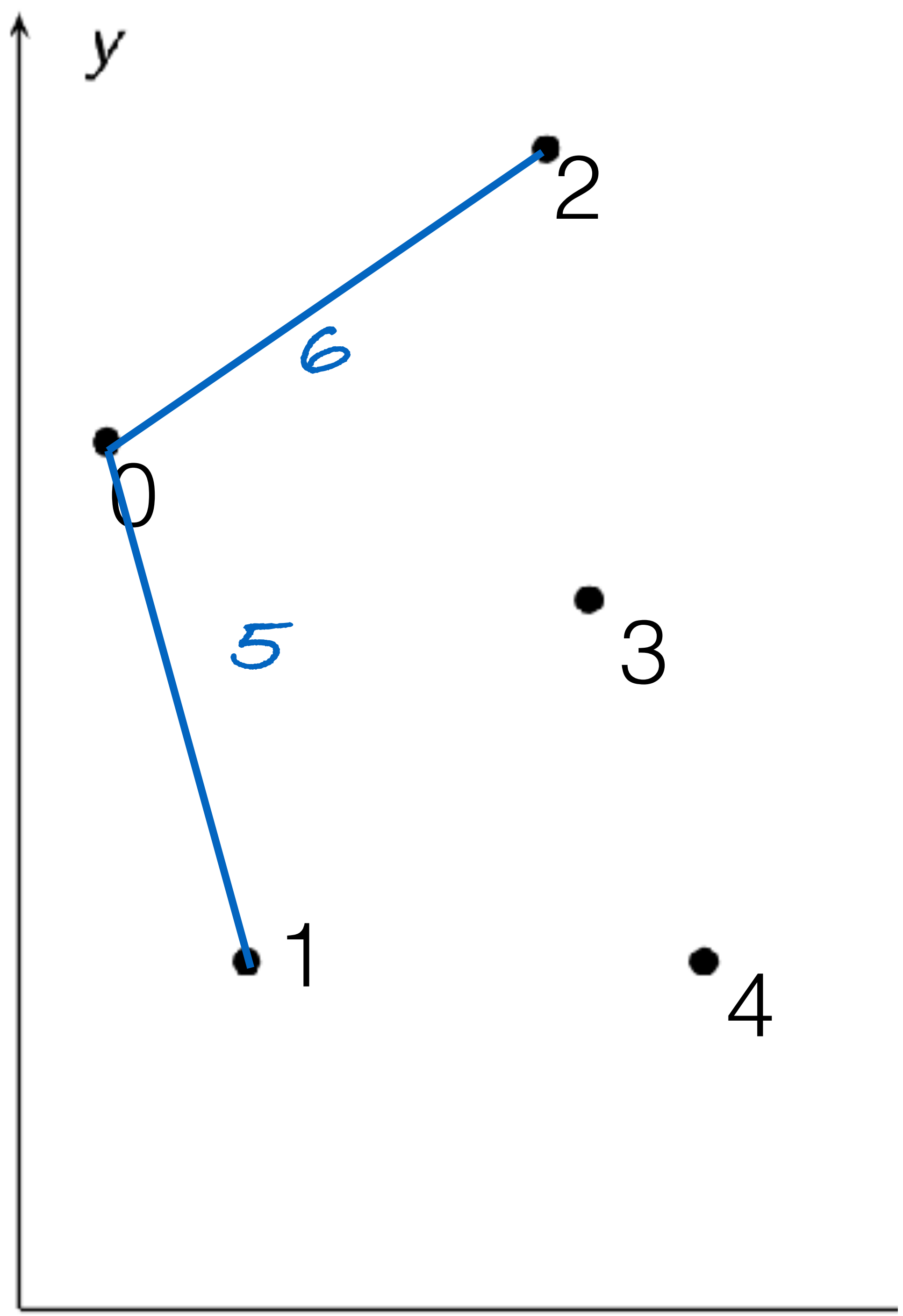
i=0 min=∞

Closest Pair Problem (2D)



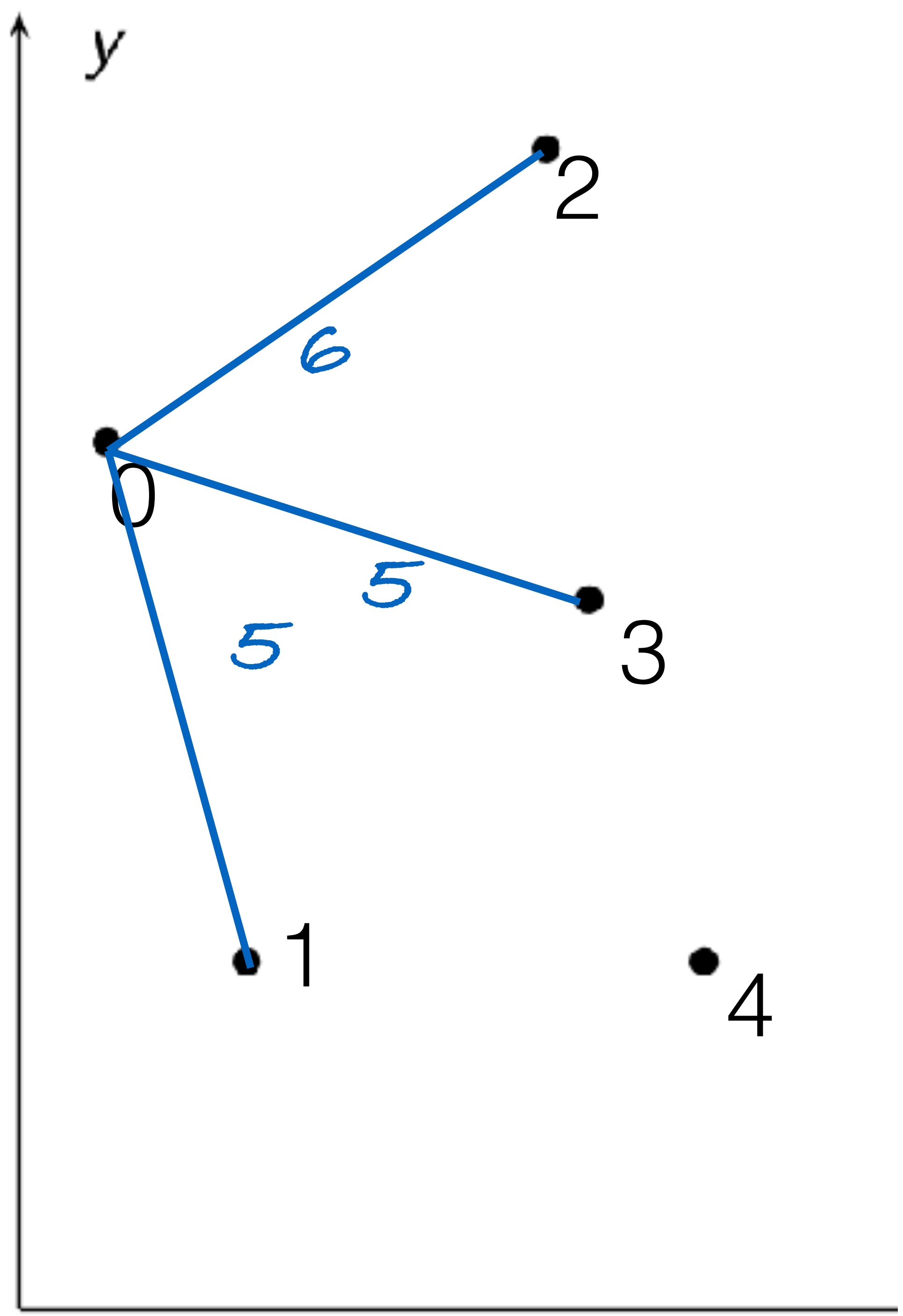
i=0 min=5

Closest Pair Problem (2D)



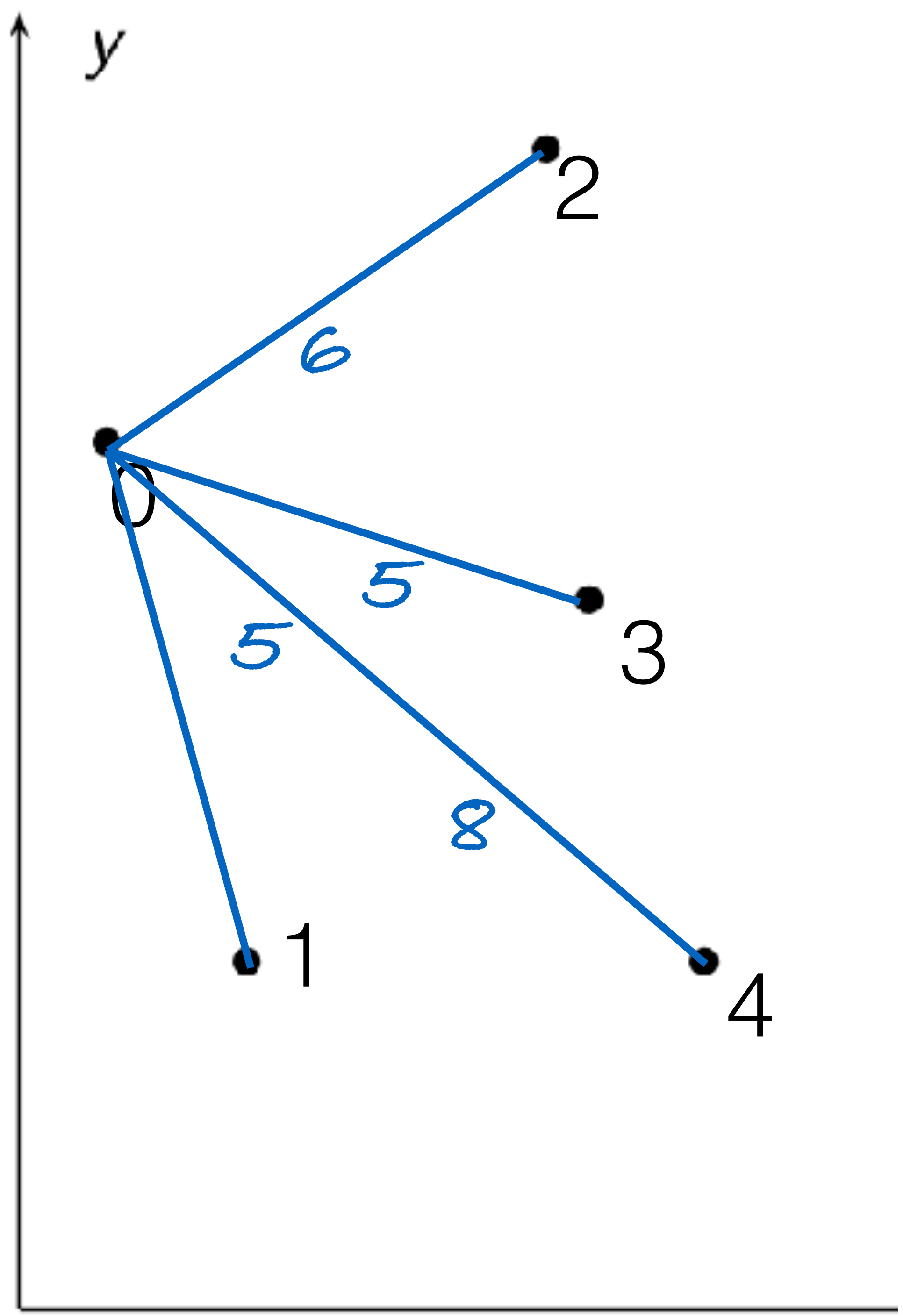
i=0 min=5

Closest Pair Problem (2D)



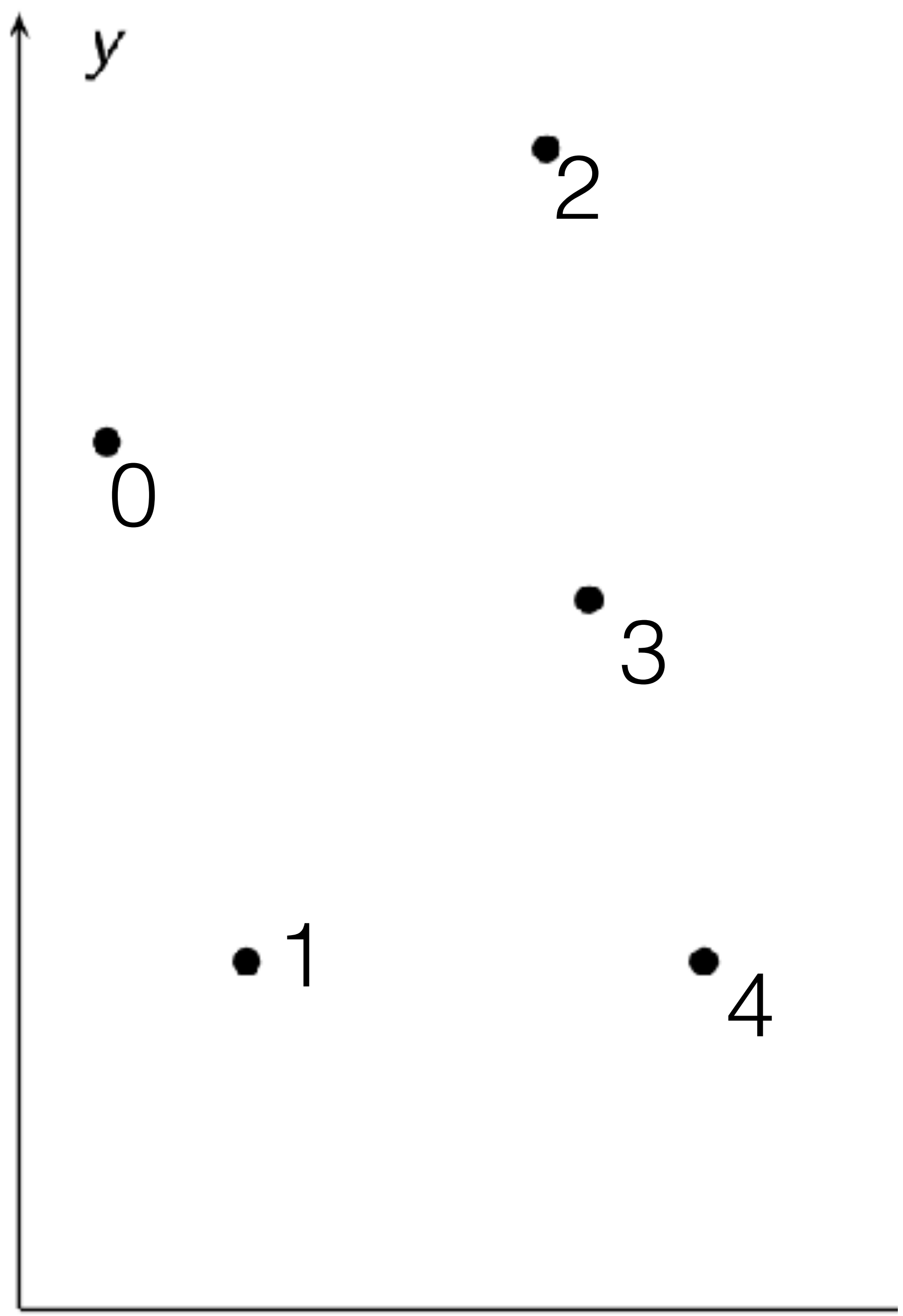
i=0 min=5

Closest Pair Problem (2D)



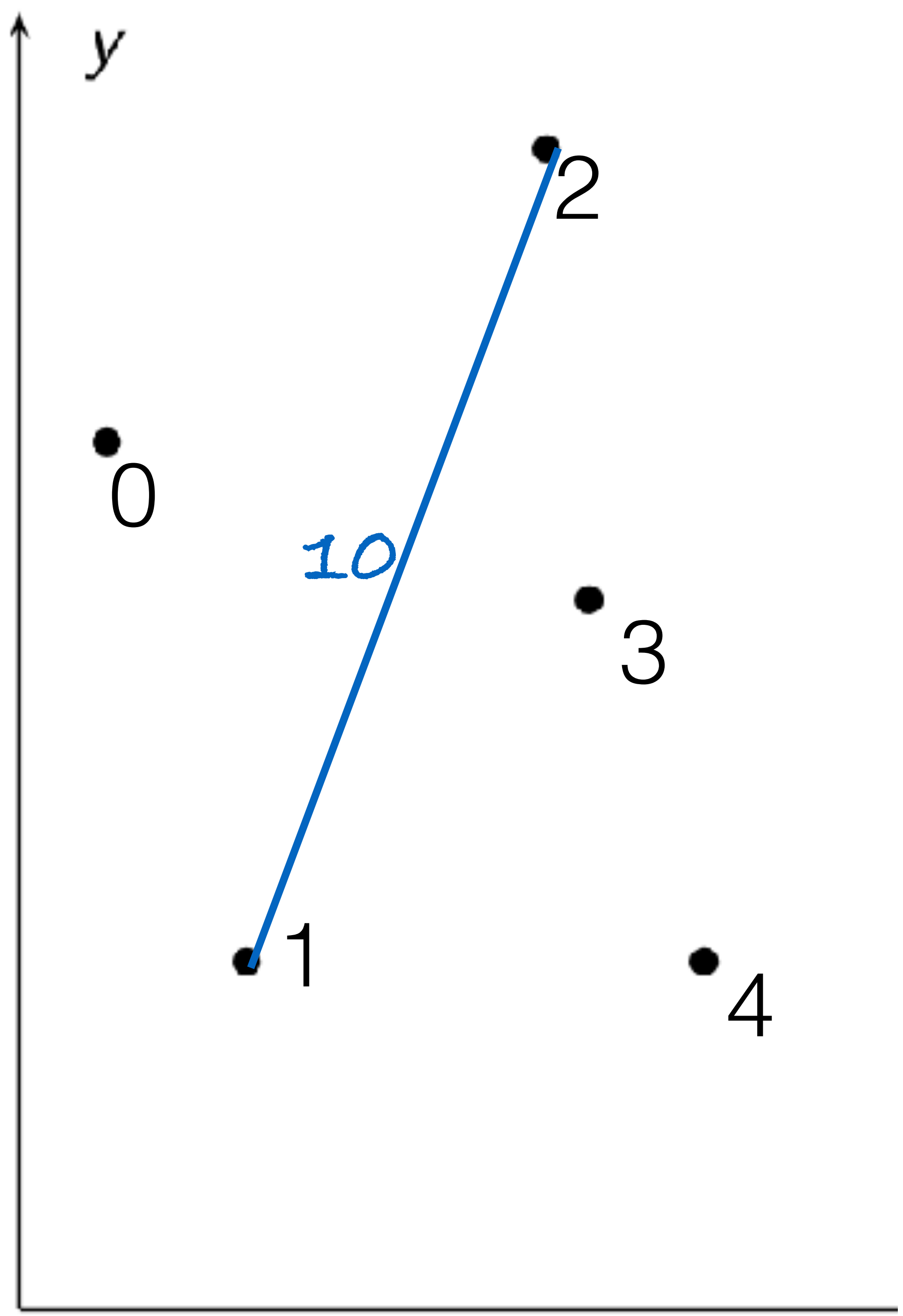
i=0   min=5

Closest Pair Problem (2D)



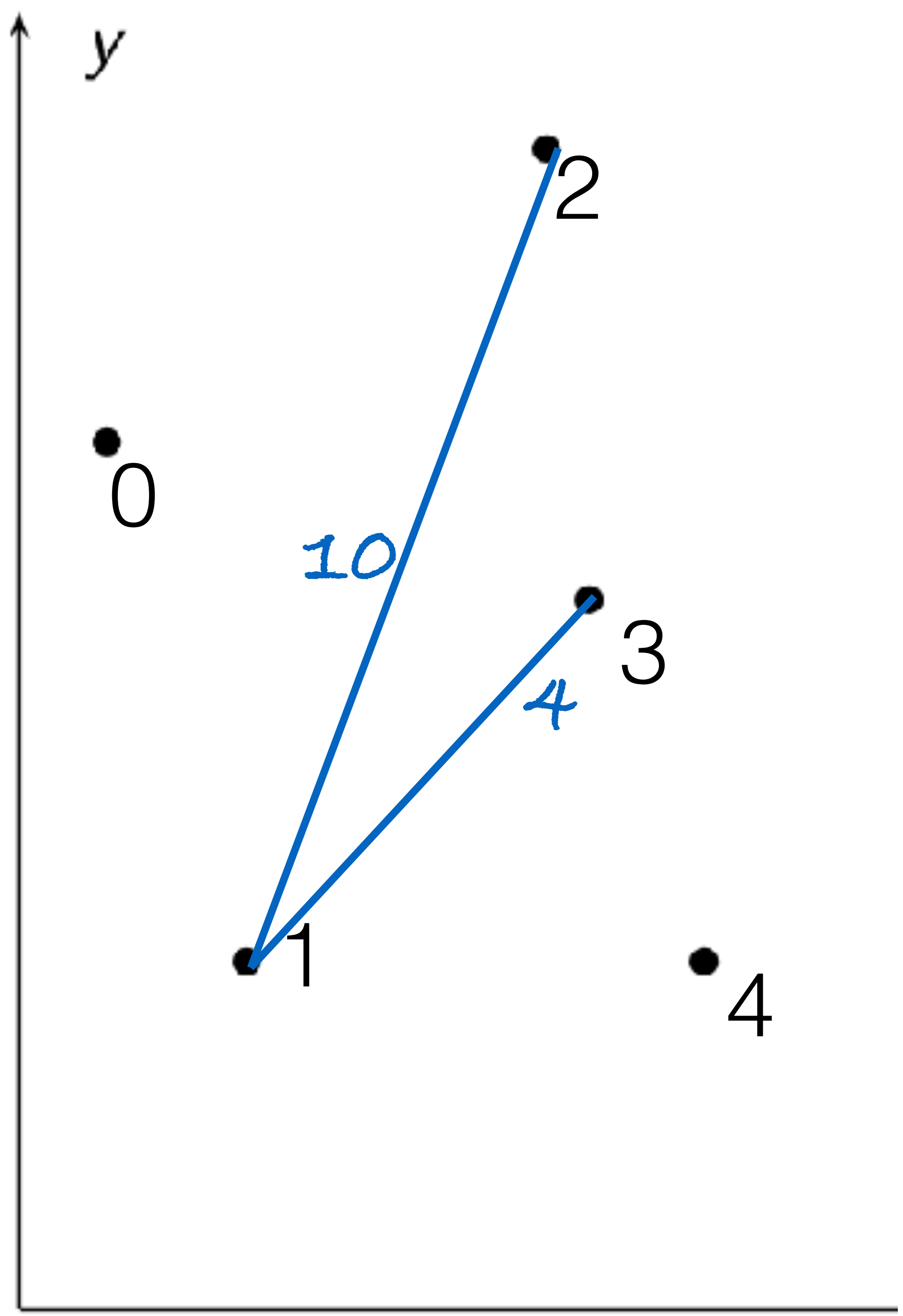
i=1 min=5

Closest Pair Problem (2D)



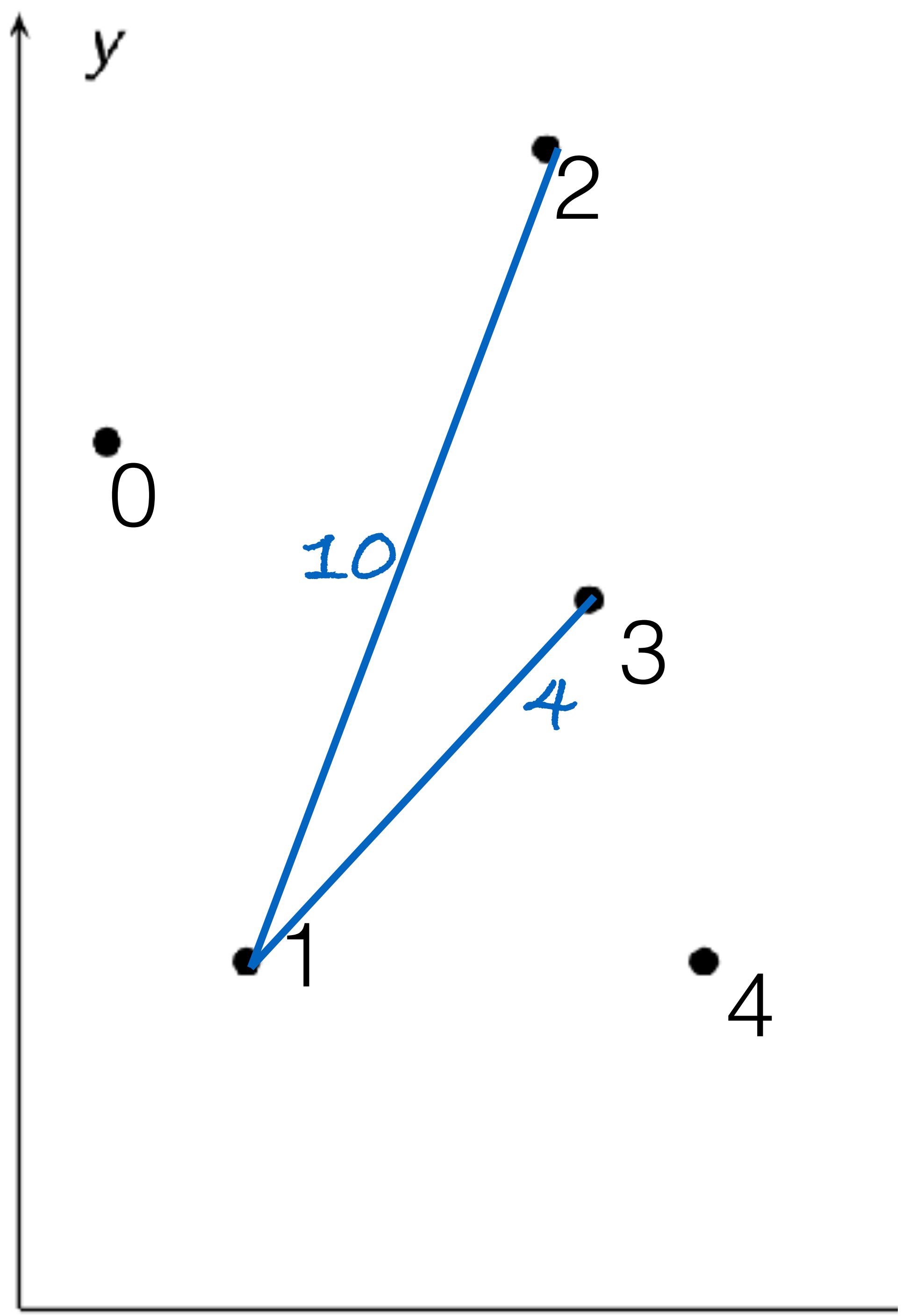
i=1 min=5

Closest Pair Problem (2D)



i=1 min=5

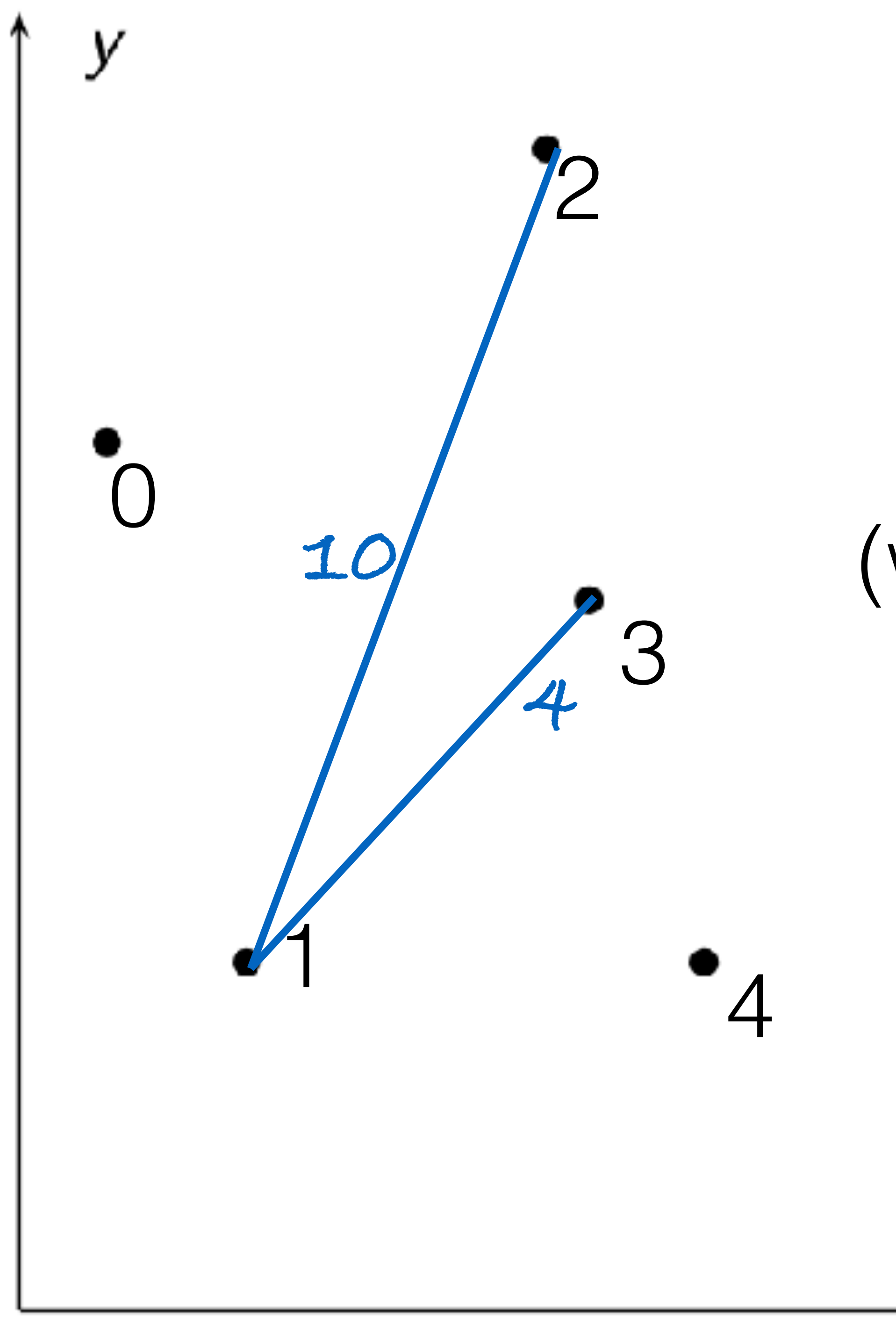
Closest Pair Problem (2D)



i=1 min=4

Closest Pair Problem (2D)

THE UNIVERSITY OF MELBOURNE



i=1 min=4

and so on until
i=3 and j=4
(which will find the minimum here)

Analysing Brute Force Closest Pair Algorithm

- Not hard to see that the algorithm is
- Note, however, that we can speed up the algorithm considerably, by utilising the monotonicity of the square root function
- Does that contradict the claim?
- Later we will see a clever divide and conquer approach leads to a $\Theta(n \log n)$ algorithm

Analysing Brute Force Closest Pair Algorithm

- Not hard to see that the algorithm is $\Theta(n^2)$
- Note, however, that we can speed up the algorithm considerably, by utilising the monotonicity of the square root function
- Does that contradict the claim?
- Later we will see a clever divide and conquer approach leads to a $\Theta(n \log n)$ algorithm

Analysing Brute Force Closest Pair Algorithm

- Not hard to see that the algorithm is $\Theta(n^2)$
- Note, however, that we can speed up the algorithm considerably, by utilising the monotonicity of the square root function $a < b \implies \sqrt{a} < \sqrt{b}$
- Does that contradict the claim?
- Later we will see a clever divide and conquer approach leads to a $\Theta(n \log n)$ algorithm

Analysing Brute Force Closest Pair Algorithm

- Not hard to see that the algorithm is $\Theta(n^2)$
- Note, however, that we can speed up the algorithm considerably, by utilising the monotonicity of the square root function $a < b \implies \sqrt{a} < \sqrt{b}$
- Does that contradict the $\Theta(n^2)$ claim?
- Later we will see a clever divide and conquer approach leads to a $\Theta(n \log n)$ algorithm

Brute Force Summary

THE UNIVERSITY OF MELBOURNE

- Simple, easy to program, widely applicable.
- Standard approach for small tasks.
- Reasonable algorithms for some problems.
- **But:** Generally inefficient—does not scale well.
- Use brute force for prototyping, or when it is known that input remains small.

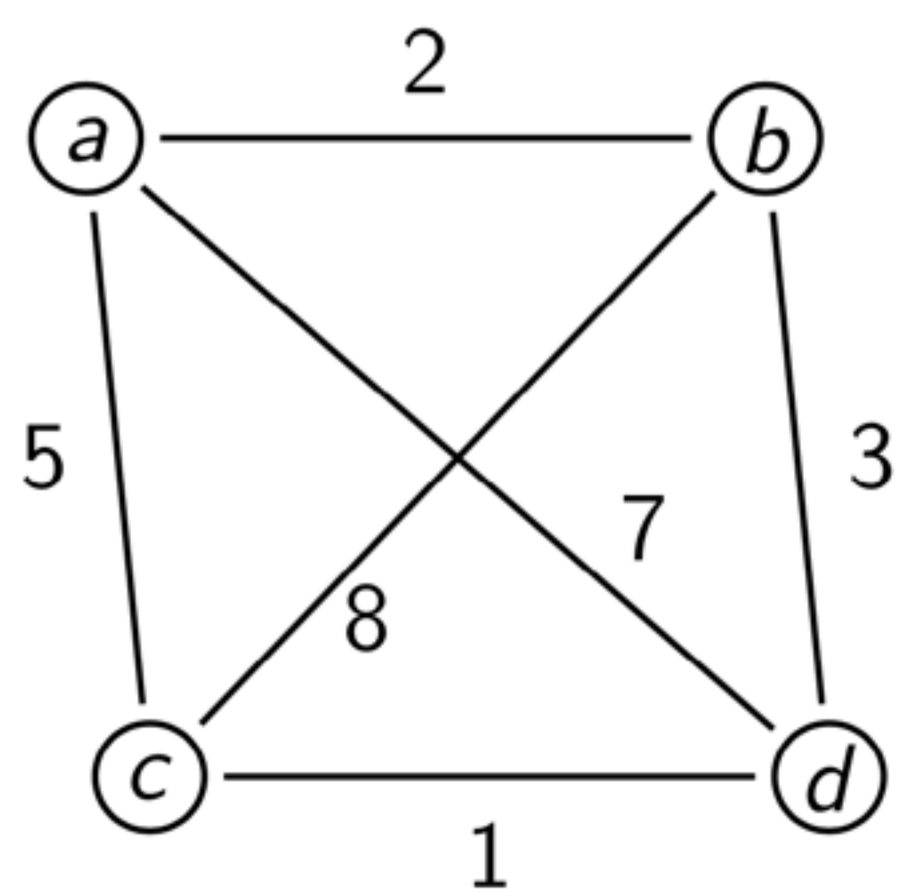
Exhaustive Search

- Problem type:
  - Combinatorial decision or optimization problems
  - Search for an element with a particular property
  - Domain grows exponentially, for example all permutations
- The brute-force approach—generate and test:
  - Systematically construct all possible solutions
  - Evaluate each, keeping track of the best so far
  - When all potential solutions have been examined, return the best found

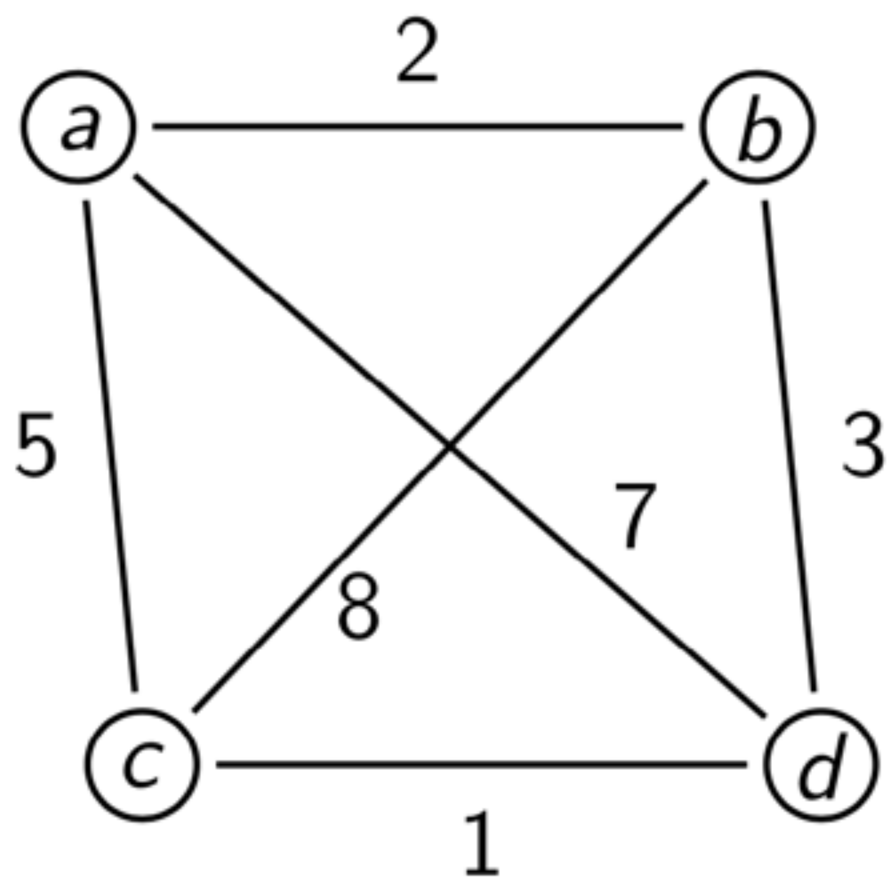
Example 1: Travelling Salesperson (TSP)

THE UNIVERSITY OF MELBOURNE

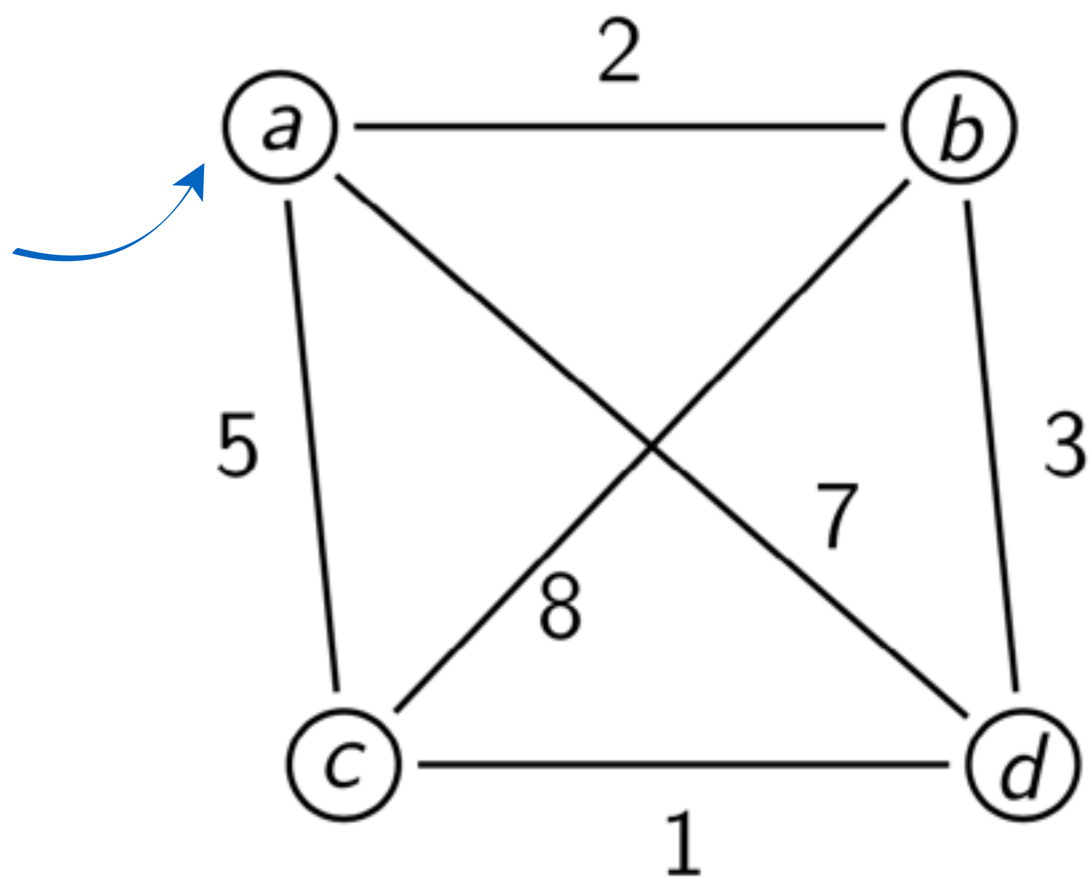
- Find the shortest **tour** (visiting each node exactly once before returning to the start) in a *weighted, undirected* graph.



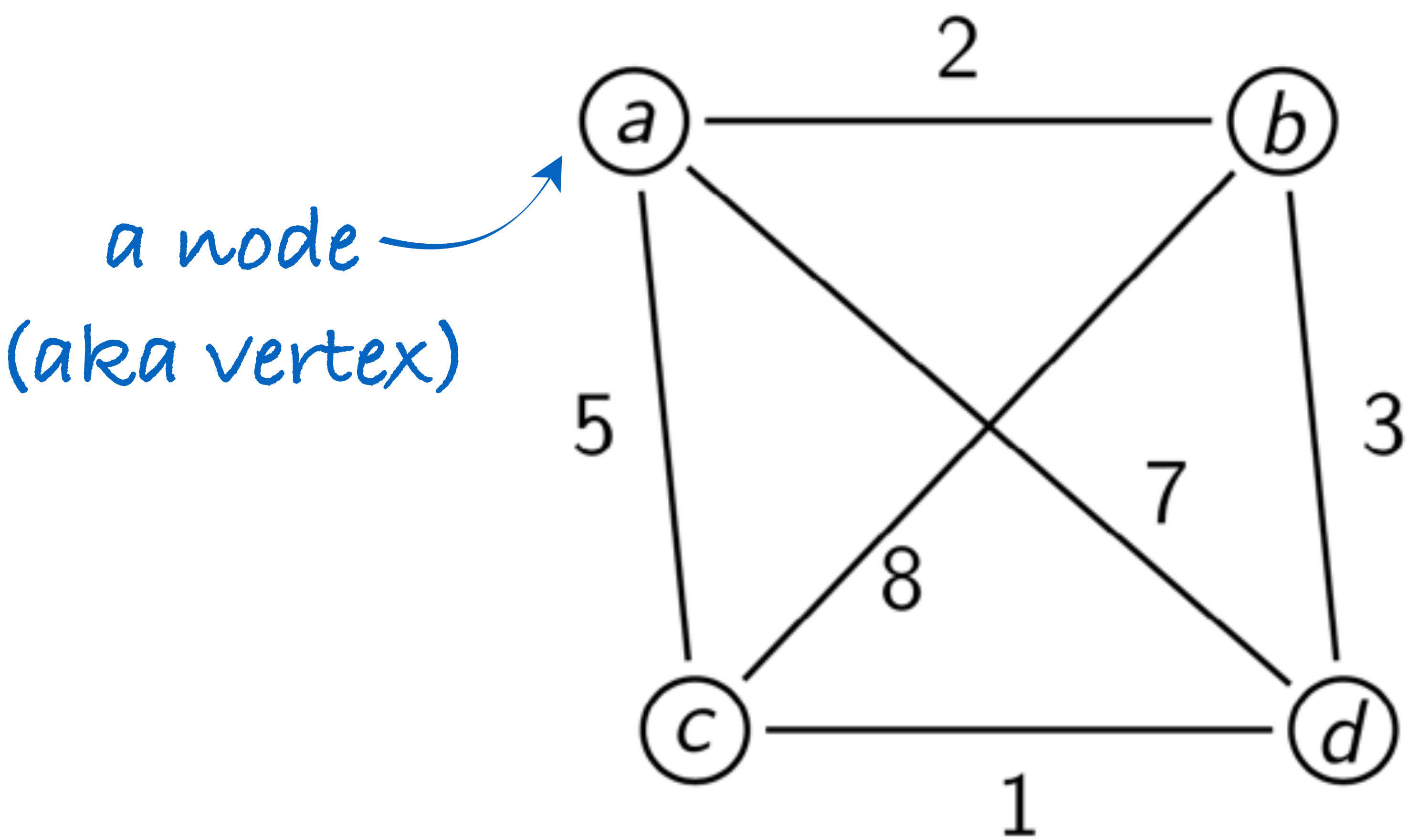
Side Note: Graph Concepts



Side Note: Graph Concepts

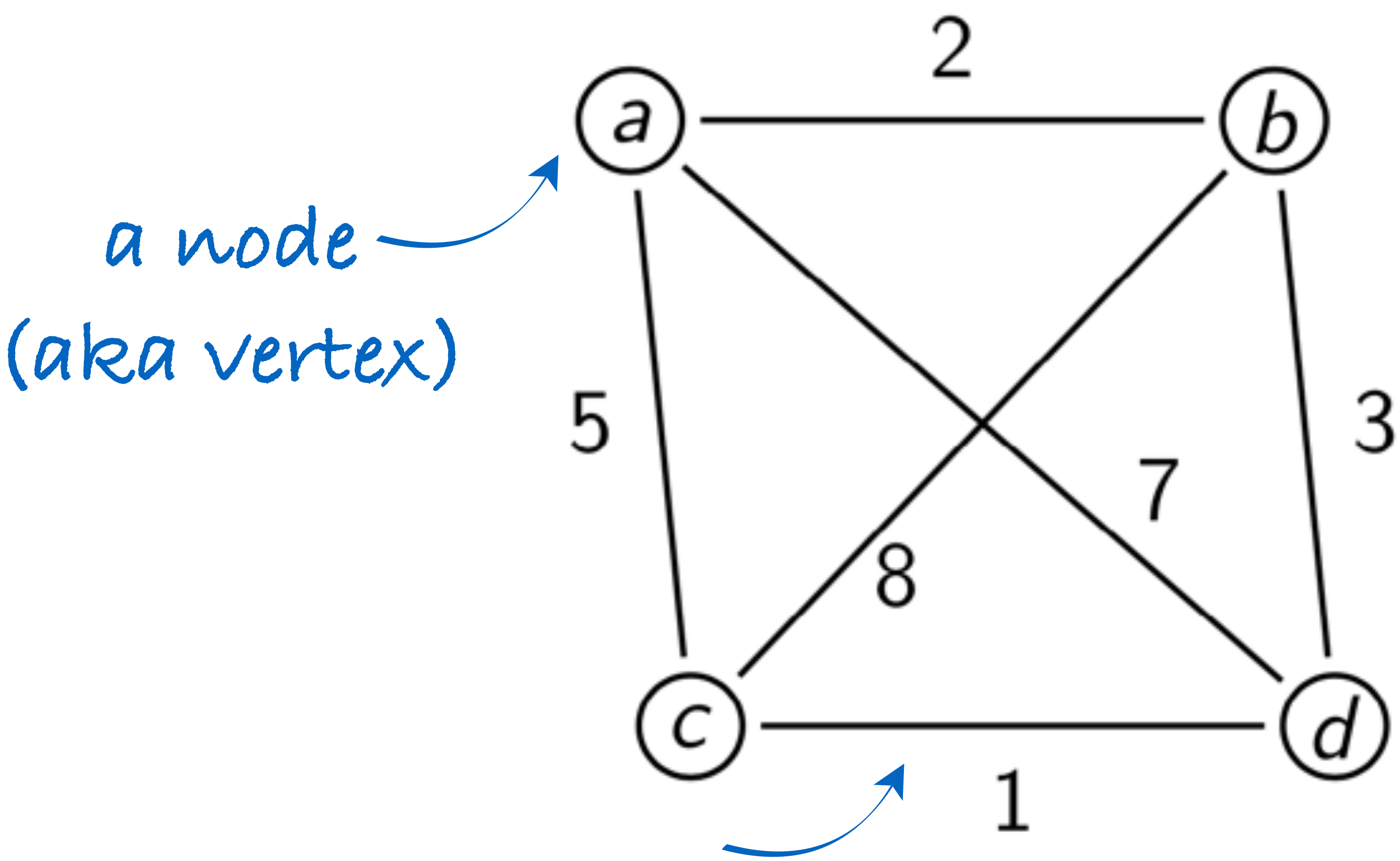


Side Note: Graph Concepts

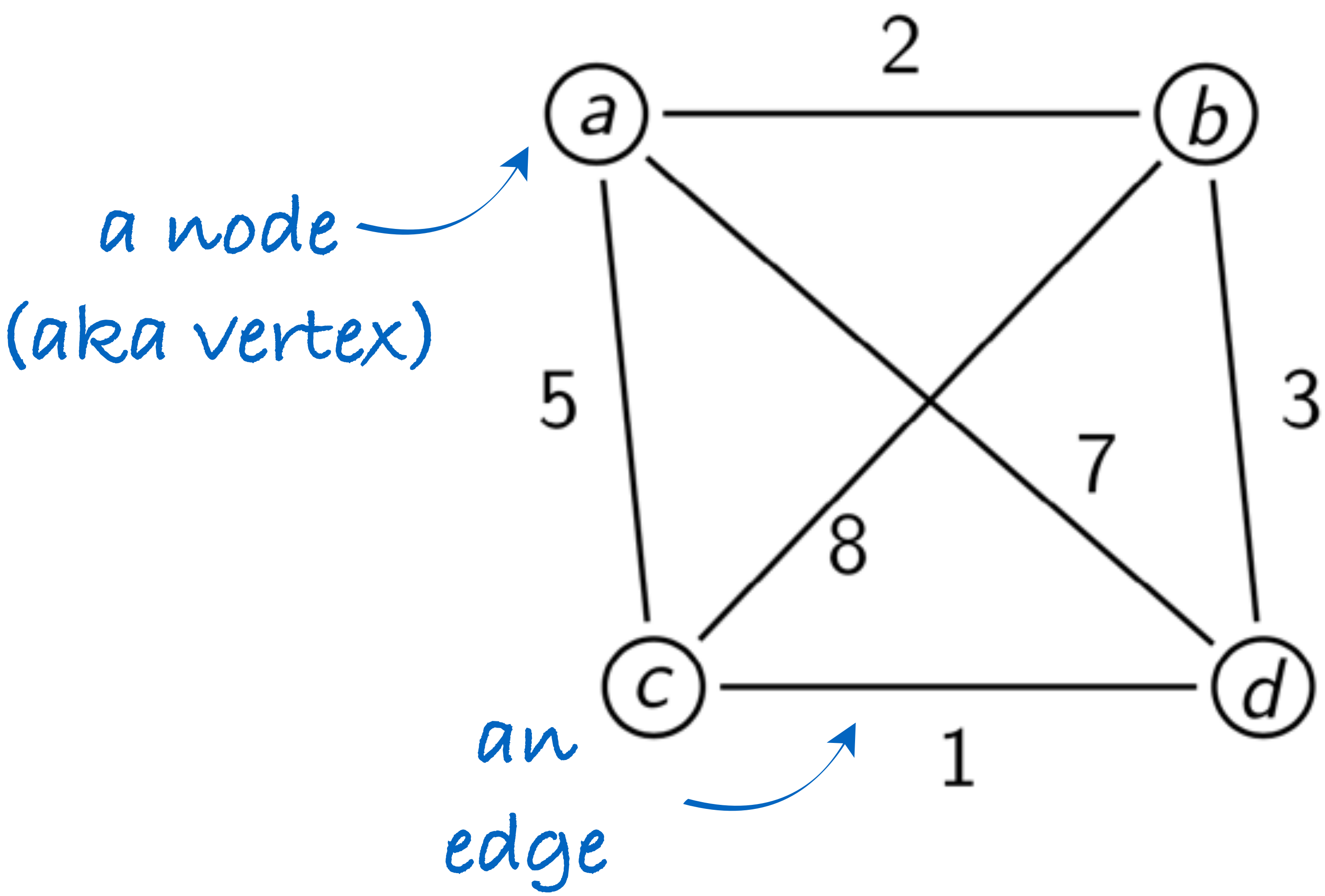


Side Note: Graph Concepts

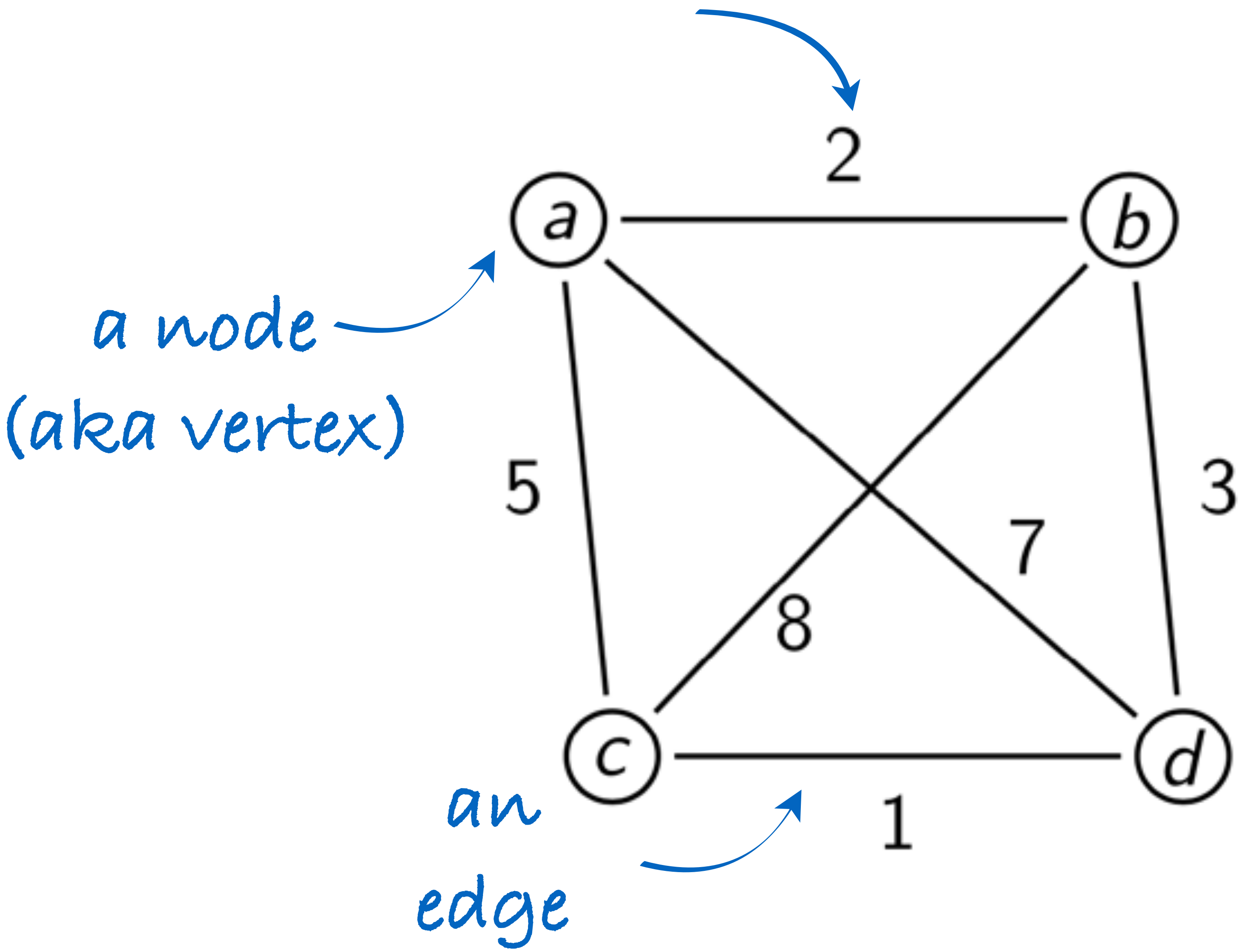
THE UNIVERSITY OF MELBOURNE



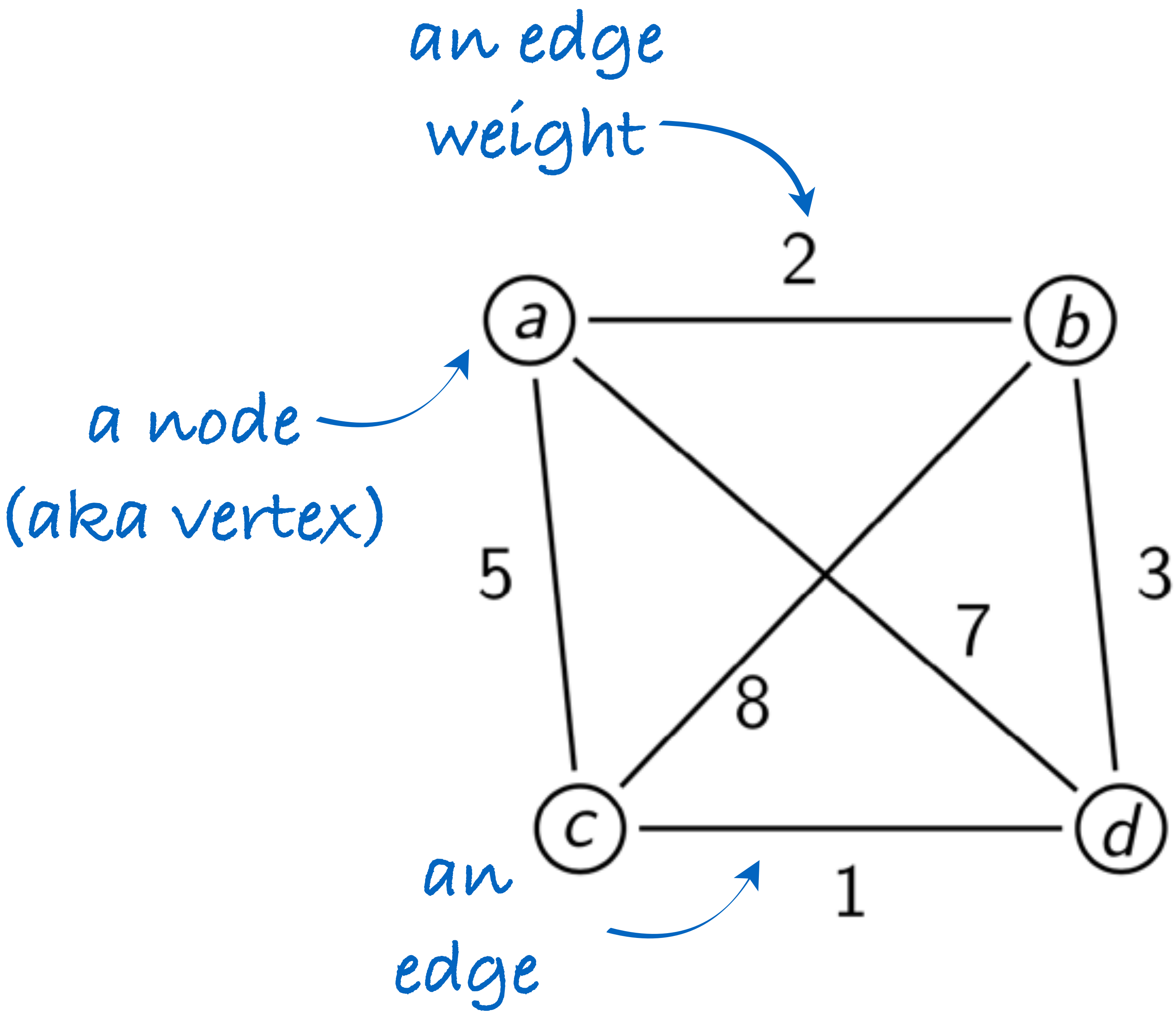
Side Note: Graph Concepts



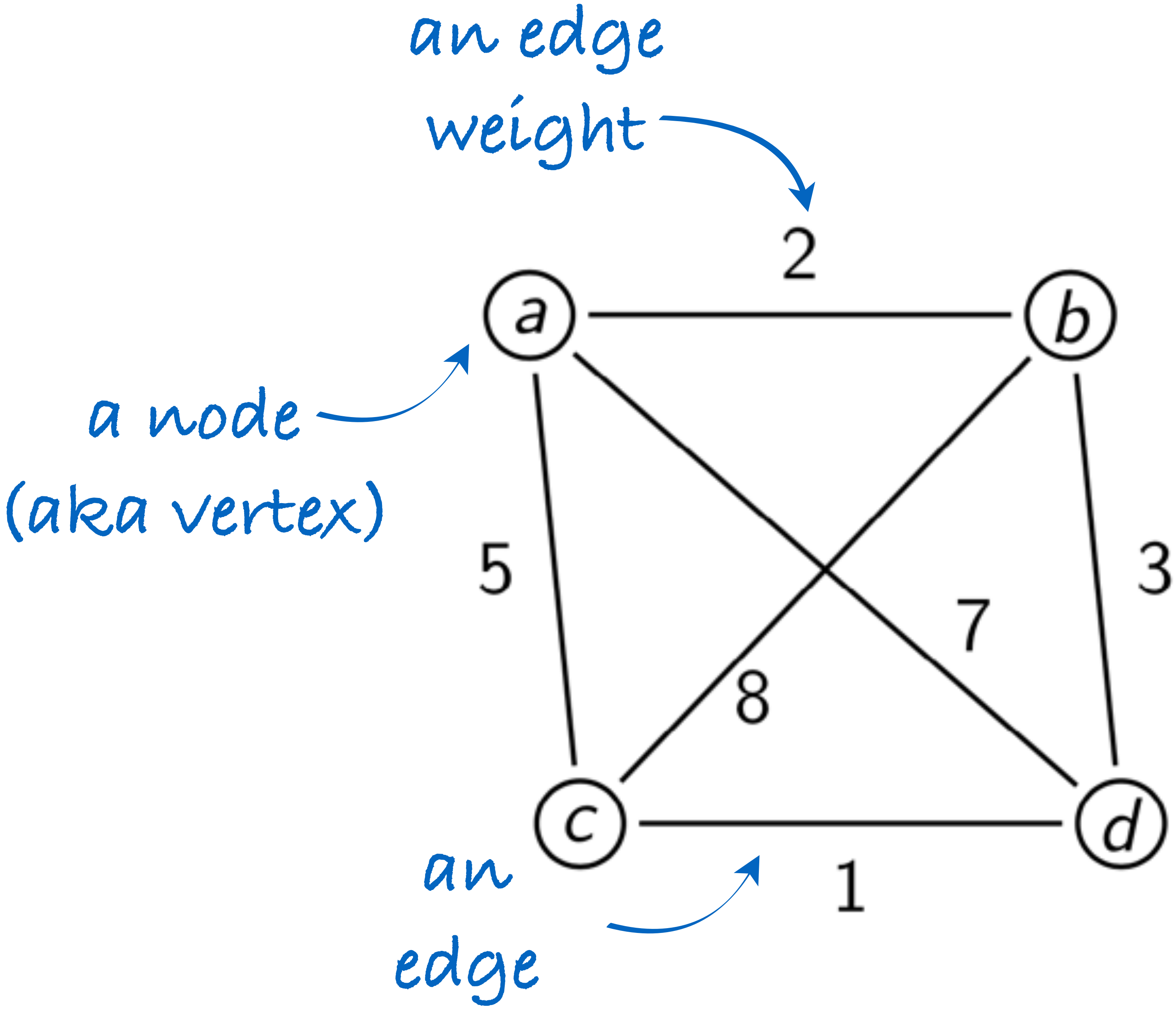
Side Note: Graph Concepts



Side Note: Graph Concepts



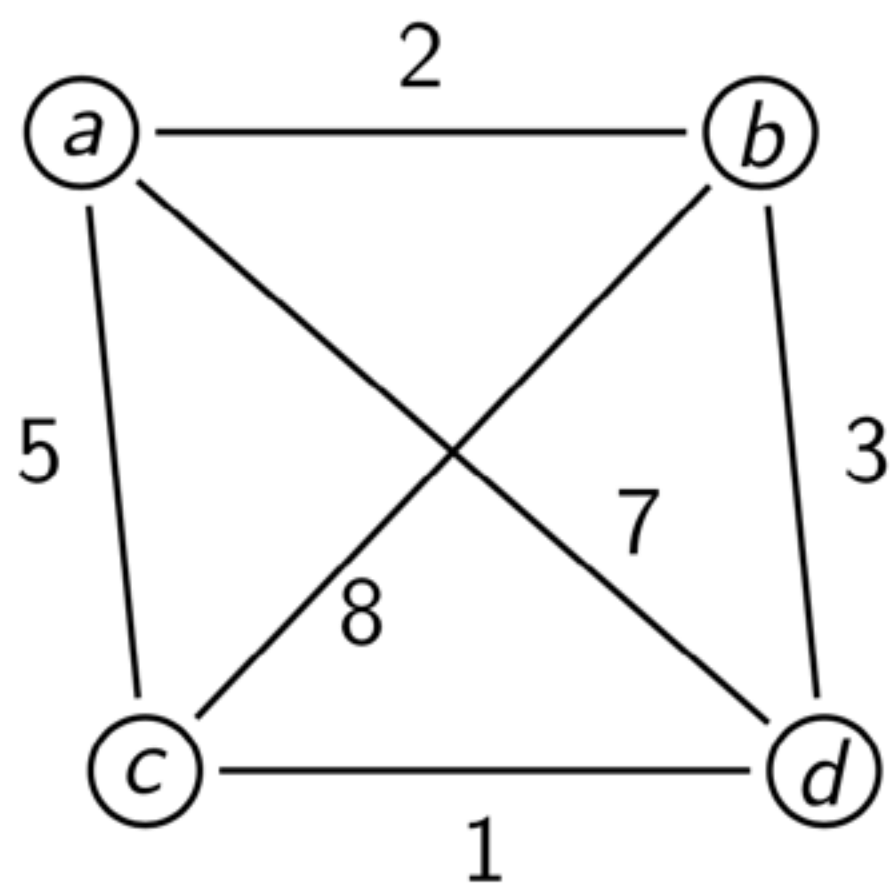
Side Note: Graph Concepts



This graph is <u>undirected</u> since edges do not have directions associated with them.

Example 1: Travelling Salesperson (TSP)

- Find the shortest **tour** (visiting each node exactly once before returning to the start) in a *weighted, undirected* graph.



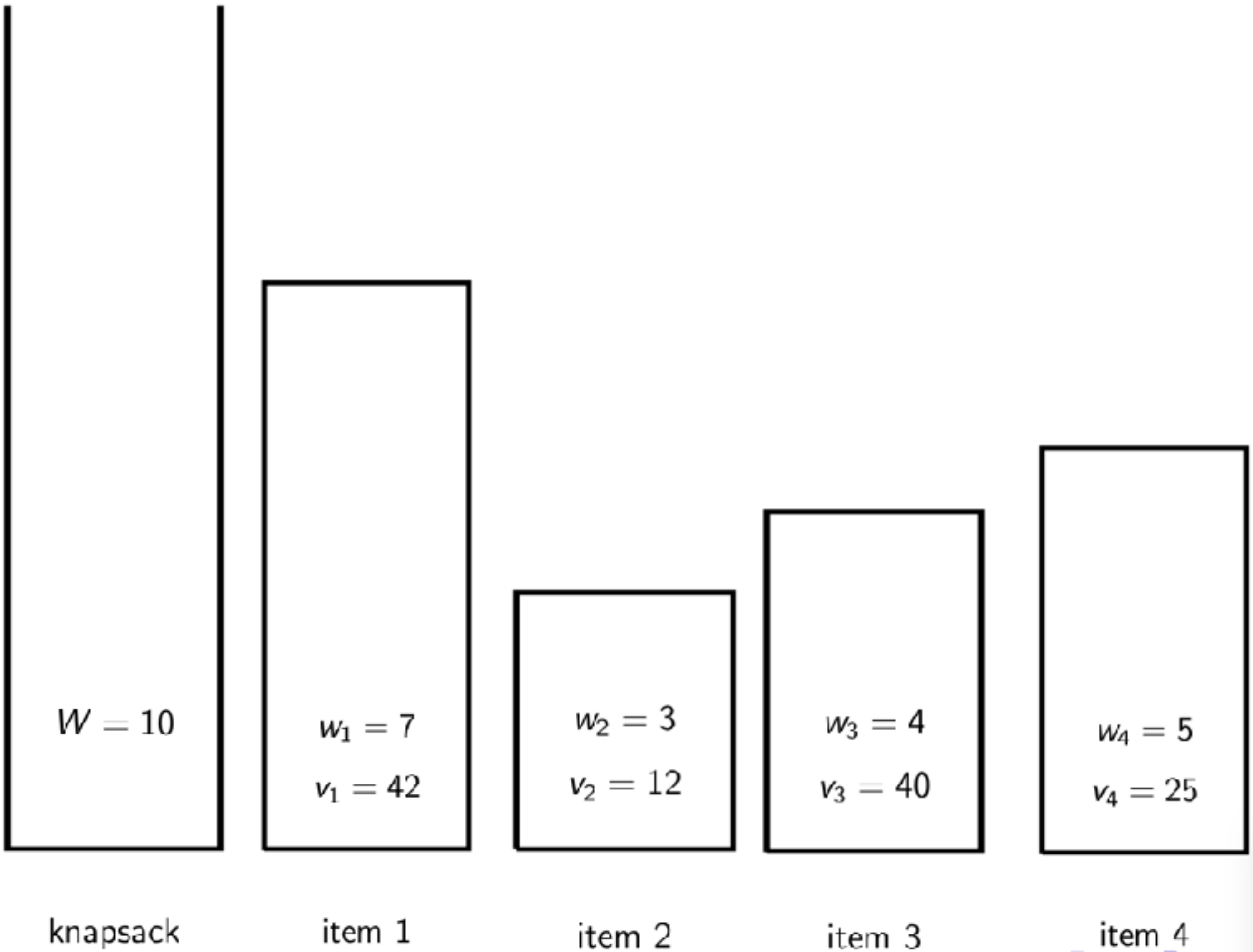
Tours

$a - b - c - d - a$	:	18
$a - b - d - c - a$	:	11
$a - c - b - d - a$	:	23
$a - c - d - b - a$	:	11
$a - d - b - c - a$	:	23
$a - d - c - b - a$	:	18

Example 2: Knapsack

- Given *n* items with
  - Weights: $w_1, w_2, \ldots, w_n$
  - Values: $v_1, v_2, \ldots, v_n$
  - Knapsack of capacity W
- find the most valuable selection of items whose combined weight does not exceed W

Knapsack Example



Knapsack: Example

Set	Weight	Value
$\emptyset$	0	0
$\{1\}$	7	42
$\{2\}$	3	12
$\{3\}$	4	40
$\{4\}$	5	25
$\{1,2\}$	10	54
$\{1,3\}$	11	NF
$\{1,4\}$	12	NF

Set	Weight	Value
$\{2,3\}$	7	52
$\{2,4\}$	8	37
$\{3,4\}$	9	65
$\{1,2,3\}$	14	NF
$\{1,2,4\}$	15	NF
$\{1,3,4\}$	16	NF
$\{2,3,4\}$	12	NF
$\{1,2,3,4\}$	19	NF

NF means “not feasible”: exhausts the capacity of the knapsack.

Later we’ll find a better algorithm based on **dynamic programming**

Comments on Exhaustive Search

- Exhaustive search algorithms have acceptable running times **only for very small instances.**

- In many cases there are better alternatives, for example, Eulerian tours, shortest paths, minimum spanning trees, . . .

- But for some problems, it is **known** that there is essentially no better alternative.

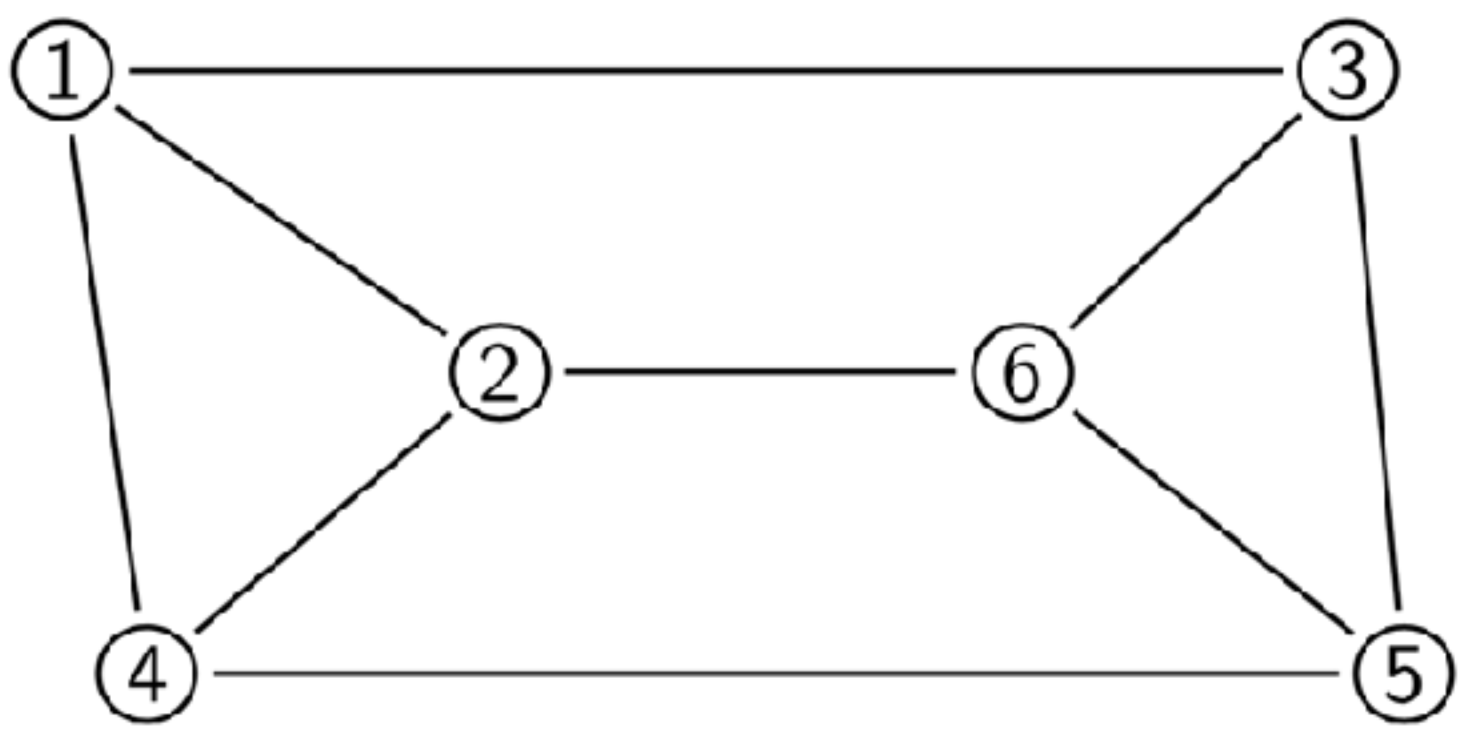
- For a large class of important problems, it **appears** that there is no better alternative, but we have no proof either way.

Hamiltonian Tours

- The Hamiltonian tour problem is this:

- In a given undirected graph, is there a simple tour (a path that visits each **node** exactly once, except it returns to the starting node)?

- Is there a Hamiltonian tour of this graph?

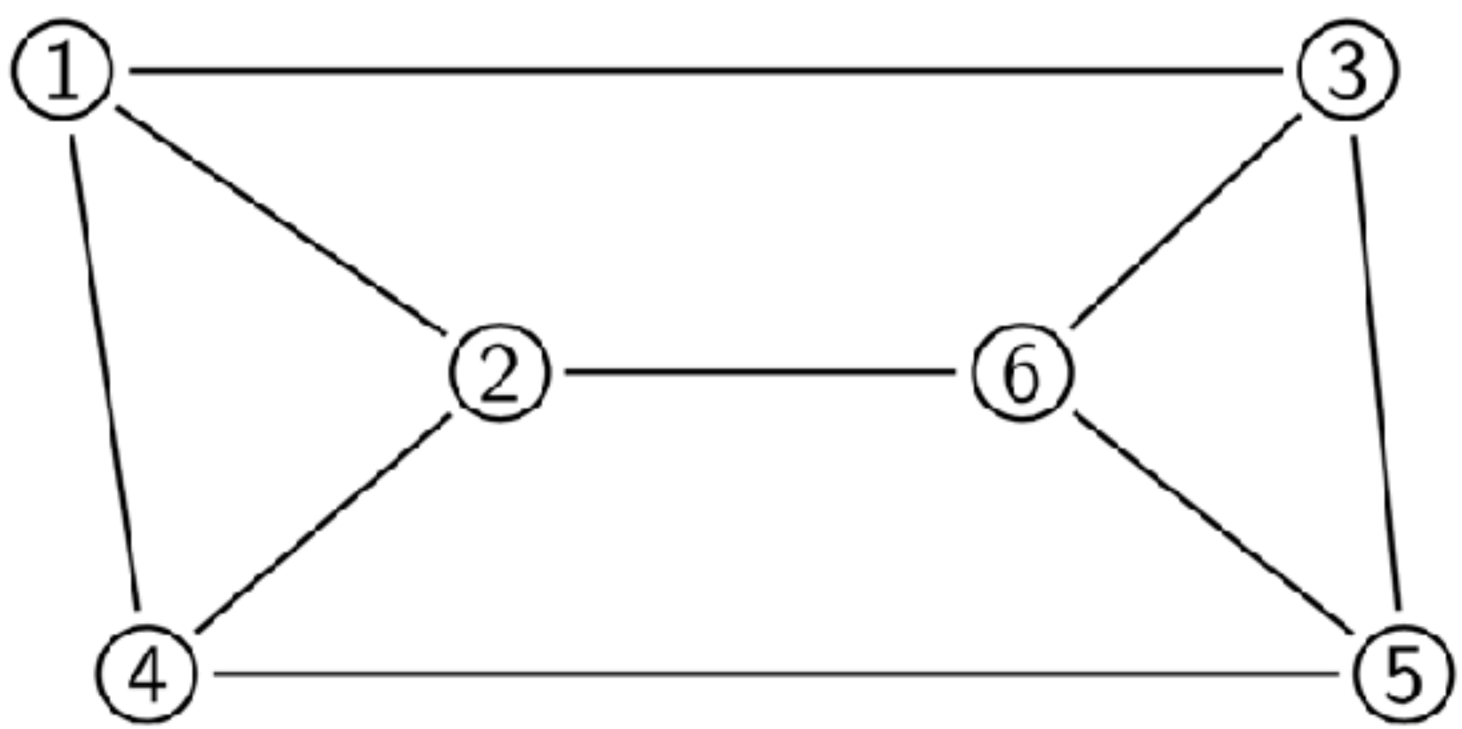


Eulerian Tours

- The Eulerian tour problem is this:

- In a given undirected graph, is there a path which visits each **edge** exactly once?

- Is there a Eulerian tour of this graph?



Hard and Easy Problems

- Recall that by a **problem** we usually mean a parametric problem: an infinite family of problem "instances".
- Sometimes our intuition about the difficulty of problems is not very reliable. The Hamiltonian Tour problem and the Eulerian Tour problem look very similar, but one is hard and the other is easy. We shall see more examples of this phenomenon later.
- For many important **optimization** problems we do not know of solutions that are essentially better than exhaustive search (a whole raft of **NP-complete** problems, including TSP, knapsack).
- In those cases we may look for **approximation algorithms** that are fast and still find solutions that are reasonably close to the optimal.
- We plan to return to this idea in Week 12.

Next Up

- **Recursion** as a problem solving technique