Background
CSCI 432

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing

Running Time

What is it?

The total number of basic steps an algorithm executes to complete its task on input of size n.

Why use it?

It is a tool to compare algorithms that is agnostic to the machine running the algorithm.

How to use it?

Generally, big-O notation.

What?	Asymptotic upper bound on a function's growth
Why?	Simplifies comparisons by focusing on most dominant factors
How?	$O(g(n)) = \begin{Bmatrix} f(n): \exists\, c, n_0 > 0 \text{ such that} \\ 0 \leq f(n) \leq cg(n)\ \forall\ n \geq n_0 \end{Bmatrix}$

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)			
Running Time (Worst)			
Running Time (Best)			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)			
Running Time (Worst)			
Running Time (Best)			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$		
Running Time (Worst)			
Running Time (Best)			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$		
Running Time (Worst)	$O(n^2)$		
Running Time (Best)			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$		
Running Time (Worst)	$O(n^2)$		
Running Time (Best)	$O(1)$		

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	
Running Time (Worst)	$O(n^2)$		
Running Time (Best)	$O(1)$		

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	
Running Time (Best)	$O(1)$		

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	
Running Time (Best)	$O(1)$	$O(n \log n)$	

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	
Running Time (Best)	$O(1)$	$O(n \log n)$	

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

* with small modification

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$*

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$

What else could we use for comparison?

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$		

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓		

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
“Simplicity”			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
“Simplicity”	✓	-	-

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
“Simplicity”	✓	-	-
Data Durability			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_2(array a):
  a = a.sort
  ngth - 2:
  :
```

```
checkUnique_
  for int i
    for int
      if i
        retu
  return true
```

What if our input is an array of static sets:
[{3,6,4}, {65,3,2}, {12,1,9}, {3,6,4}]

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
“Simplicity”	✓	-	-
Data Durability			

Problem: Given an unsorted array, check if all elements are unique.

```
checkUnique_2(array a):
  a = a.sort
  ngth - 2:
  :
```

```
checkUnique_
  for int i
    for int
      if i
        retu
  return true
```

What if our input is an array of static sets:

[{3,6,4}, {65,3,2}, {12,1,9}, {3,6,4}]

Can't sort! Equality is defined, but not order.

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
"Simplicity"	✓	-	-
Data Durability	✓	✗	✓

Problem: Given an unsorted array
check if

What if our input is an array of Student objects:

```
Student {
  id
  name
  gpa
}
```

```
checkUni
  for in
    for
      if
  return true
```

```
th - 2:
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
“Simplicity”	✓	-	-
Data Durability			

Problem: Given an unsorted array
check if

What if our input is an array of Student objects:

```
Student {
  id
  name
  gpa
}
```

(generally) can't hash! Equality and order can be defined, but (generally) can't hash mutable objects.

```
checkUni
  for in
    for
      if
  return true
```

```
th - 2:
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
"Simplicity"	✓	-	-
Data Durability	✓	✓	✗

Problem: Given an unsorted array, check if all elements are unique.

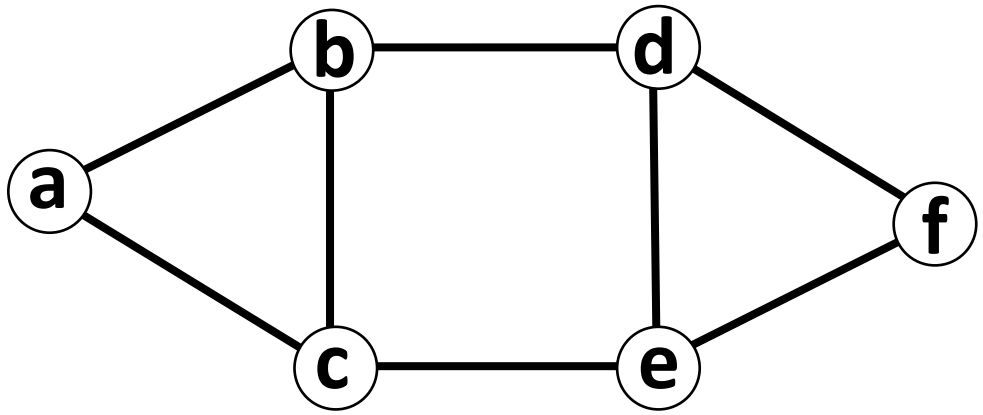
```
checkUnique_1(array a):
  for int i = 0 to a.length - 1:
    for int j = 0 to a.length - 1:
      if i != j && a[i] == a[j]:
        return false
  return true
```

```
checkUnique_2(array a):
  a = a.sort
  for int i = 0 to a.length - 2:
    if a[i] == a[i + 1]:
      return false
  return true
```

```
checkUnique_3(array a):
  h = new HashSet
  for e in a:
    h.add(e)
  if h.length == a.length
    return true
  return false
```

	Brute Force	Sorting	Hashing
Running Time (Expected)	$O(n^2)$	$O(n \log n)$	$O(n)$
Running Time (Worst)	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Running Time (Best)	$O(1)$	$O(n \log n)$	$O(1)$
Extra Space	$O(1)$	$O(1)$	$O(n)$
Preserves Input	✓	✗	✓
“Simplicity”	✓	-	-
Data Durability	✓	-	-

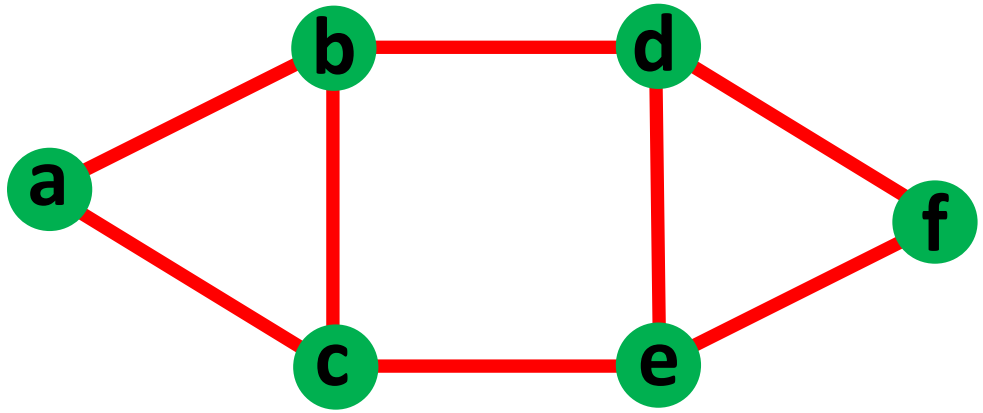
Graphs



Entity	Neighbors
a	b,c
b	a,c,d
c	a,b,e
d	b,e,f
e	c,d,f
f	d,e

Graphs are mathematical objects that represent connectivity relationships between entities.

Graphs



Edges

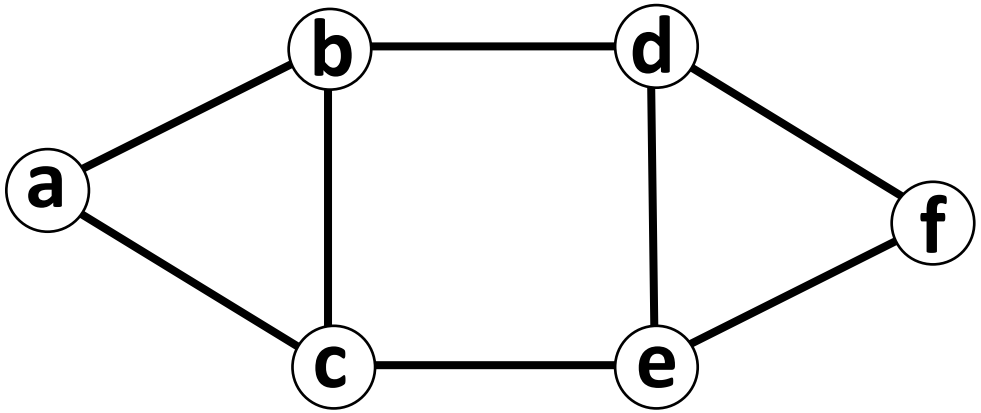
$$G = (V, E)$$

Vertices
(or Nodes)

Vertex	Neighbors
a	b,c
b	a,c,d
c	a,b,e
d	b,e,f
e	c,d,f
f	d,e

Graphs are mathematical objects that represent connectivity relationships between entities.

Graphs



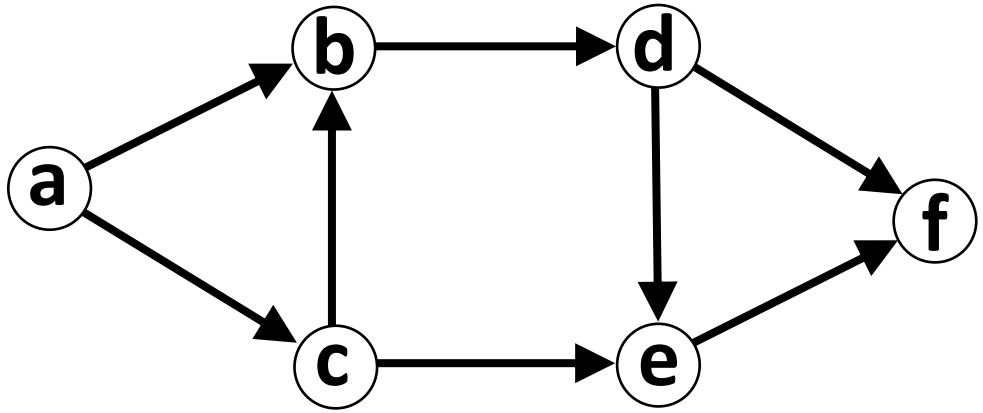
Edges

$$G = (V, E)$$

Vertices
(or Nodes)

- Edges can be undirected…

Graphs



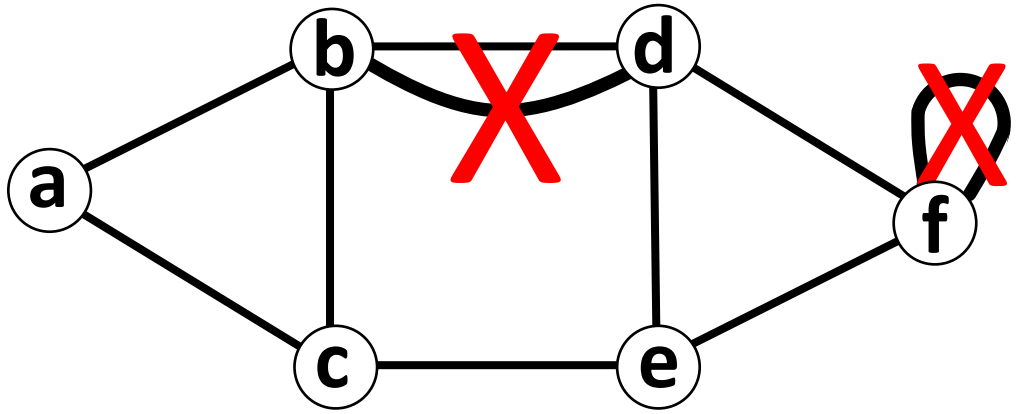
Edges

$$G = (V, E)$$

Vertices
(or Nodes)

- Edges can be undirected or directed.

Graphs



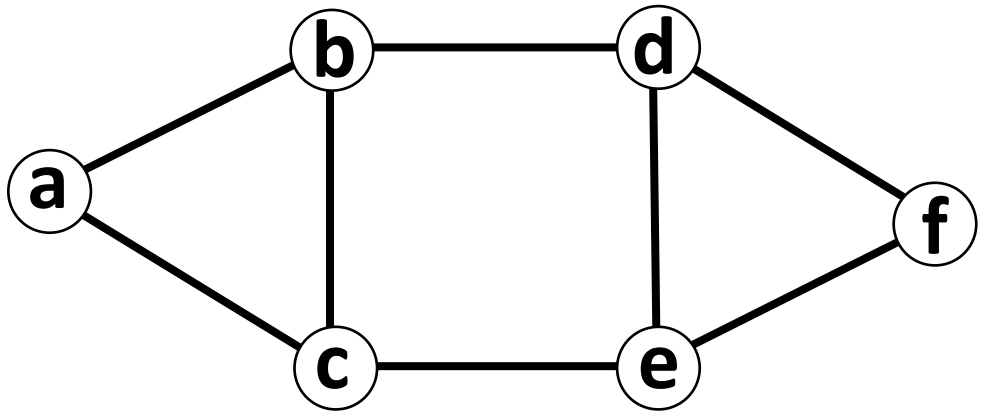
Edges

$$G = (V, E)$$

Vertices
(or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.

Graphs



Edges

$$G = (V, E)$$

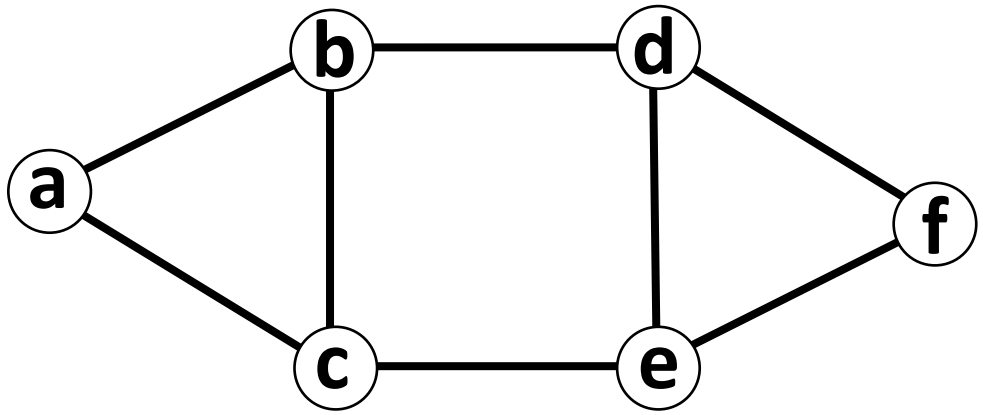
Vertices
(or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.

a,c,e,f ✓ **a,c,d,f** ✗

b,d ✓ **c,e,d,f,e** ✗

Graphs



Edges

$$G = (V, E)$$

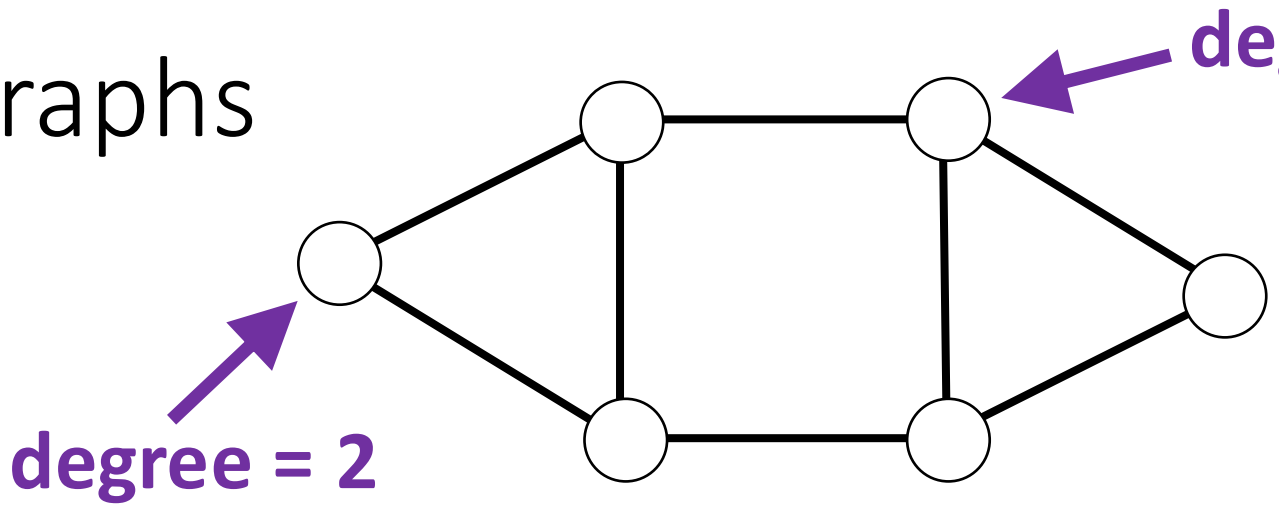
Vertices
(or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.
- Cycle = Sequence of vertices that start and end at same vertex.

c,b,d,e,c ✓ **a,c,e,f** ✗

(and usually with no other repeated vertices.)

Graphs



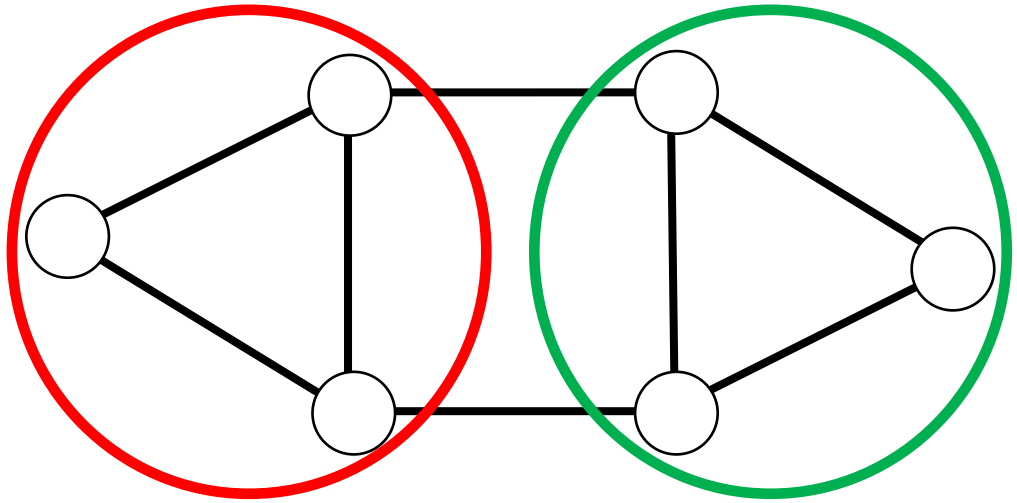
Edges

$$G = (V, E)$$

Vertices
(or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.
- Cycle = Sequence of vertices that start and end at same vertex.
- Degree of a vertex = $\deg(v)$ = # of edges touching it (undirected).

Graphs



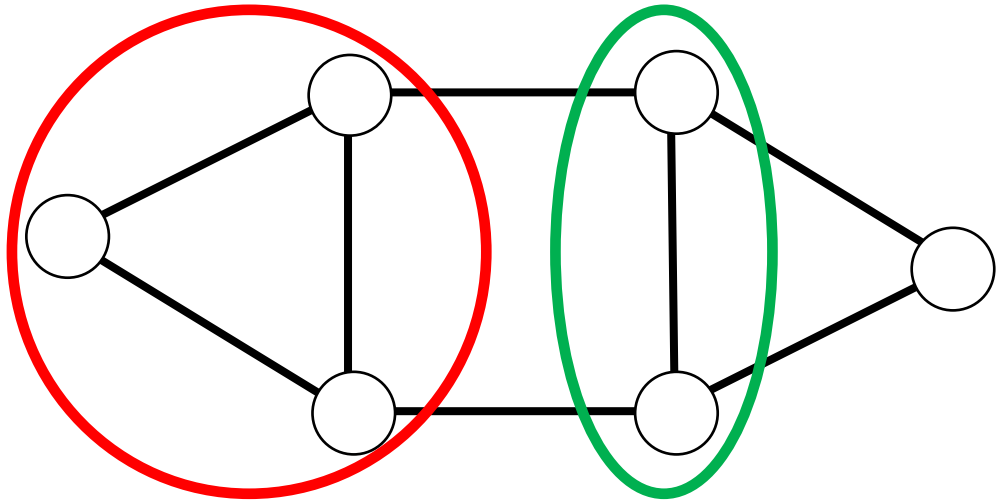
$$G = (V, E)$$

Edges

Vertices (or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.
- Cycle = Sequence of vertices that start and end at same vertex.
- Degree of a vertex = $\deg(v)$ = # of edges touching it (undirected).
- Cut = Partition of vertices into two disjoint subsets.

Graphs



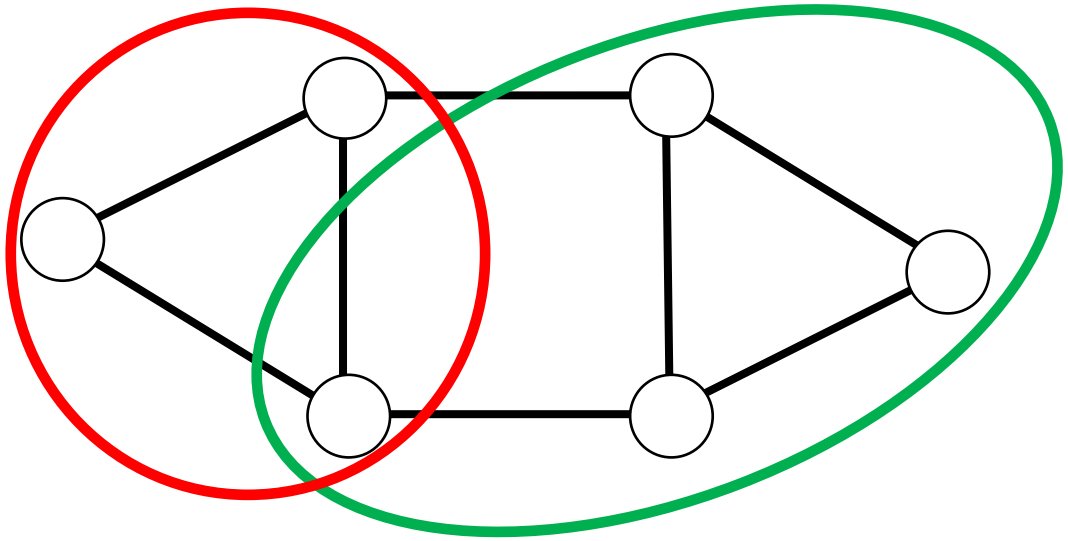
Edges

$$G = (V, E)$$

Vertices
(or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.
- Cycle = Sequence of vertices that start and end at same vertex.
- Degree of a vertex = $\deg(v)$ = # of edges touching it (undirected).
- Cut = Partition of vertices into two disjoint subsets.

Graphs



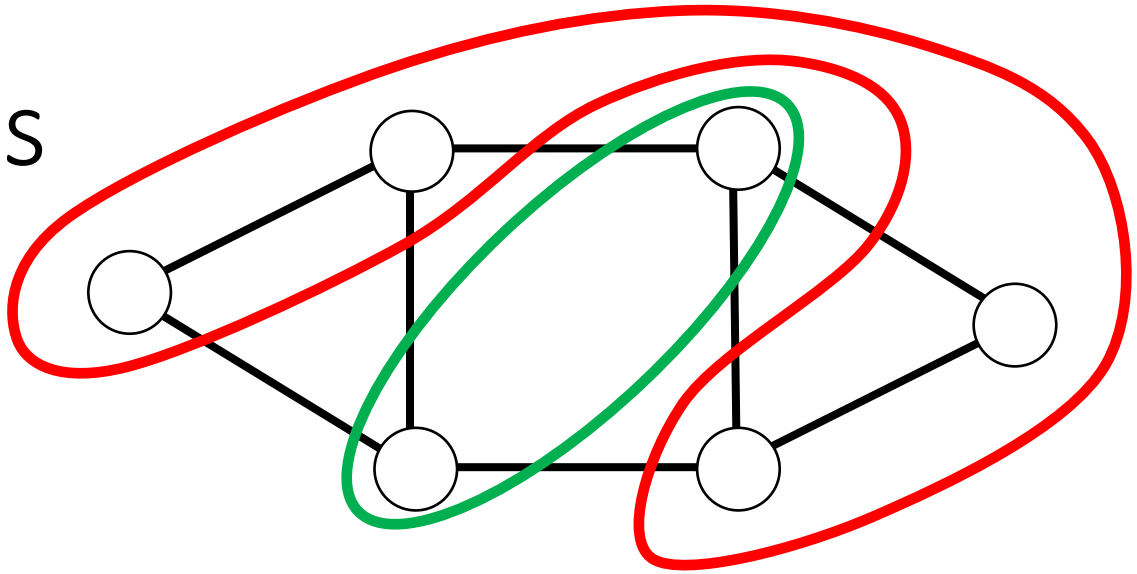
$$G = (V, E)$$

Edges

Vertices (or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.
- Cycle = Sequence of vertices that start and end at same vertex.
- Degree of a vertex = $\deg(v)$ = # of edges touching it (undirected).
- Cut = Partition of vertices into two disjoint subsets.

Graphs



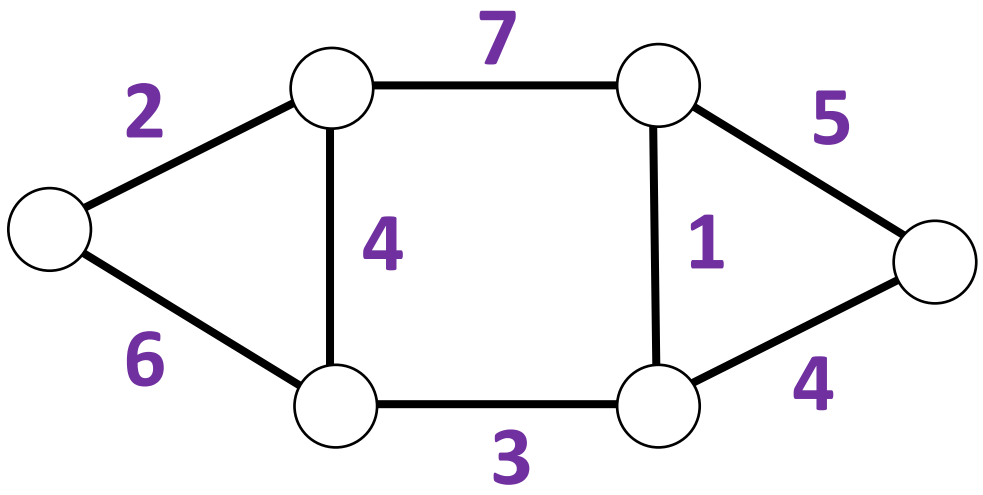
$$G = (V, E)$$

Edges

Vertices (or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.
- Cycle = Sequence of vertices that start and end at same vertex.
- Degree of a vertex = $\deg(v)$ = # of edges touching it (undirected).
- Cut = Partition of vertices into two disjoint subsets.

Graphs



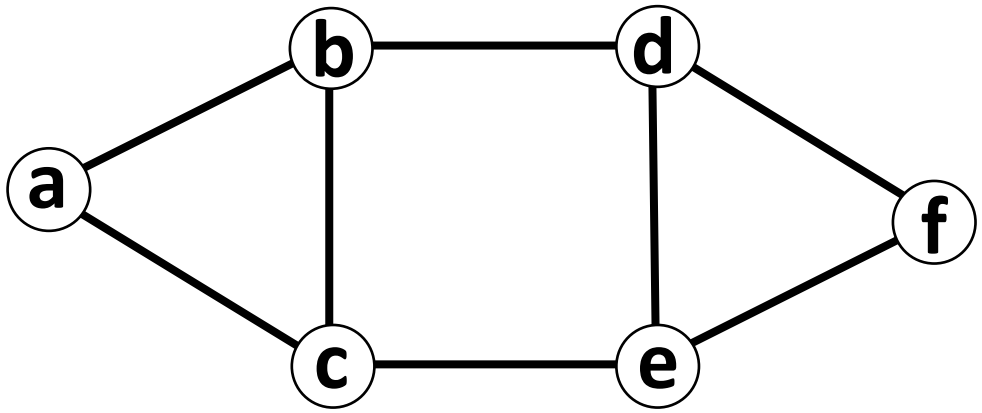
Edges

$$G = (V, E)$$

Vertices
(or Nodes)

- Edges can be directed or undirected.
- Simple graph = At most one edge between pair of vertices and no edges that start and end at same vertex.
- Path = Sequence of vertices connected by edges without loops.
- Cycle = Sequence of vertices that start and end at same vertex.
- Degree of a vertex = $\deg(v)$ = # of edges touching it (undirected).
- Cut = Partition of vertices into two disjoint subsets.
- Edges (or vertices) can be weighted (cost associated with using it).

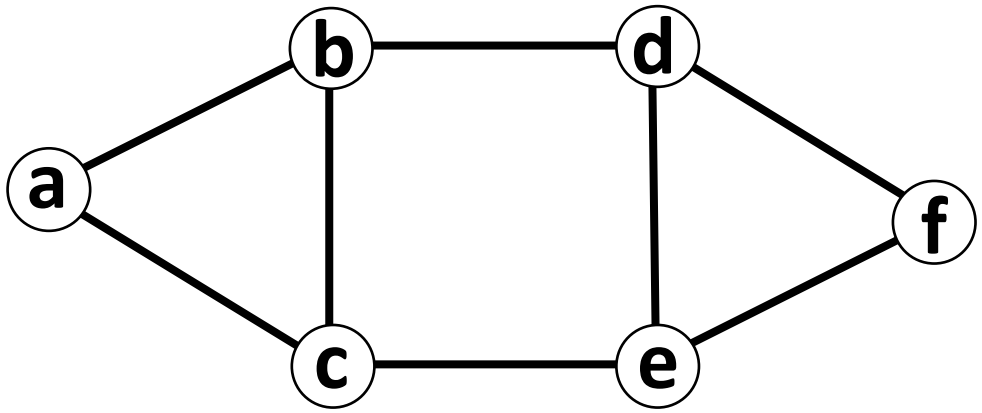
Graphs



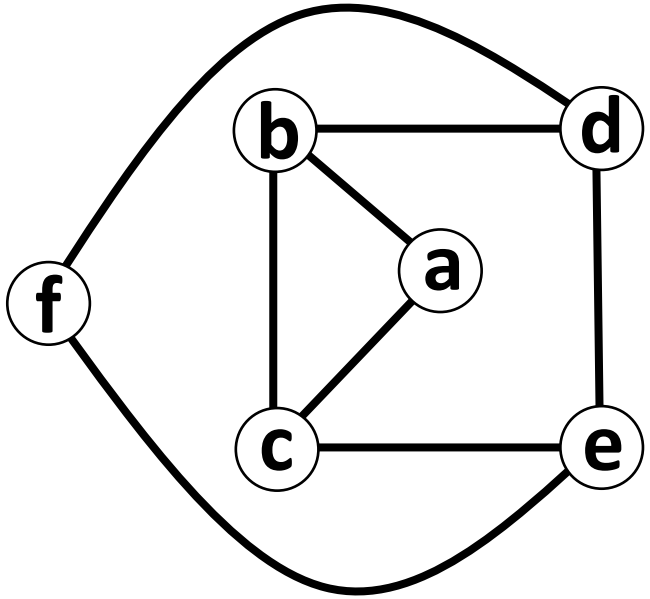
Vertex	Neighbors
a	b,c
b	a,c,d
c	a,b,e
d	b,e,f
e	c,d,f
f	d,e

Graphs are mathematical objects that represent connectivity relationships between entities.

Graphs



=



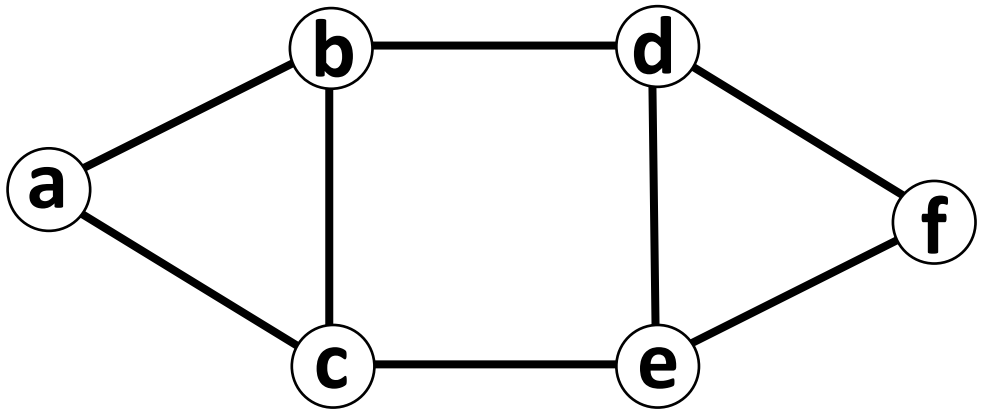
Vertex	Neighbors
a	b,c
b	a,c,d
c	a,b,e
d	b,e,f
e	c,d,f
f	d,e

=

Vertex	Neighbors
a	b,c
b	a,c,d
c	a,b,e
d	b,e,f
e	c,d,f
f	d,e

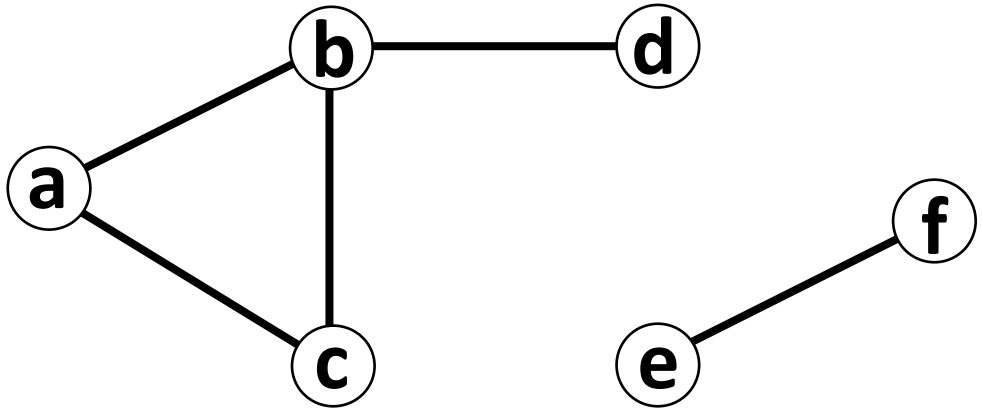
Topologically equivalent (i.e., same connectivity)

Special Graphs

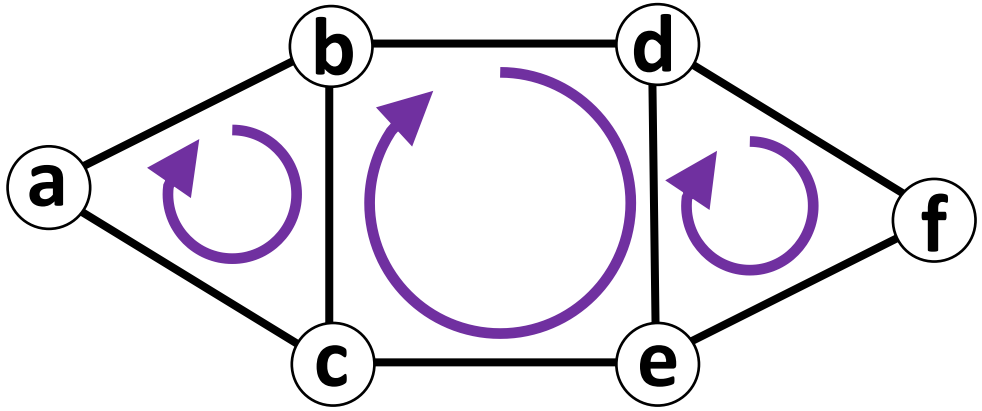


Vertices (or Nodes)
Edges
$\Big\}\ G = (V, E)$

- Connected Graph = Graph that has a path between every vertex pair.

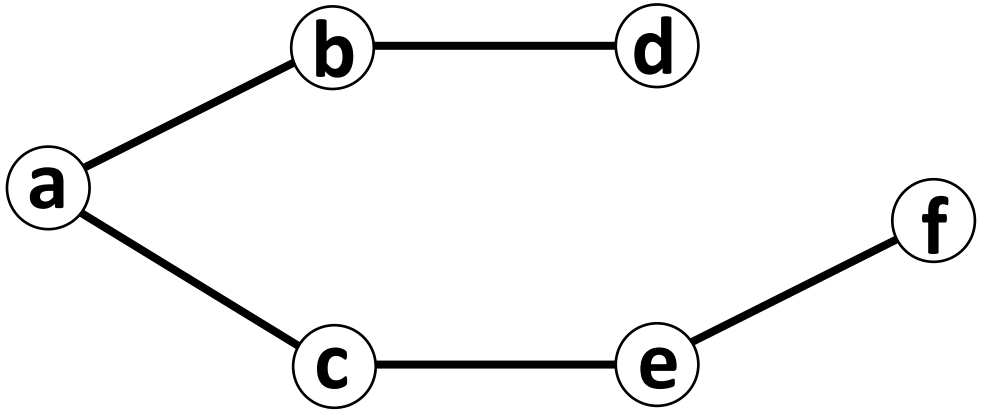


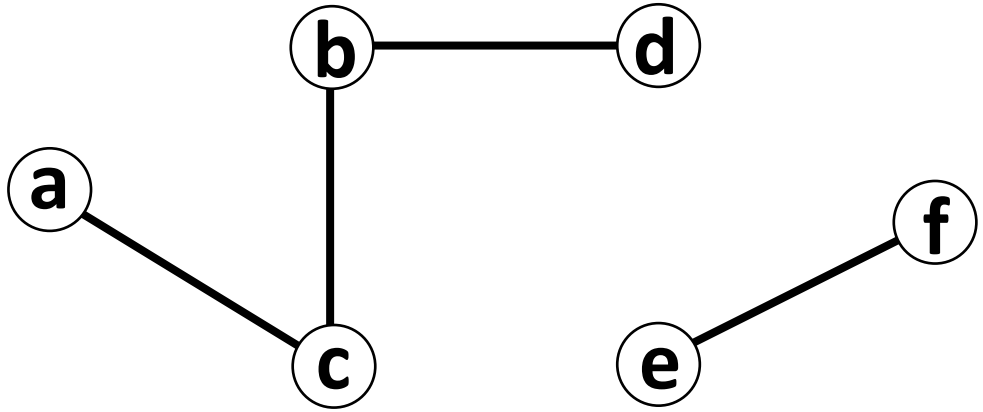
Special Graphs



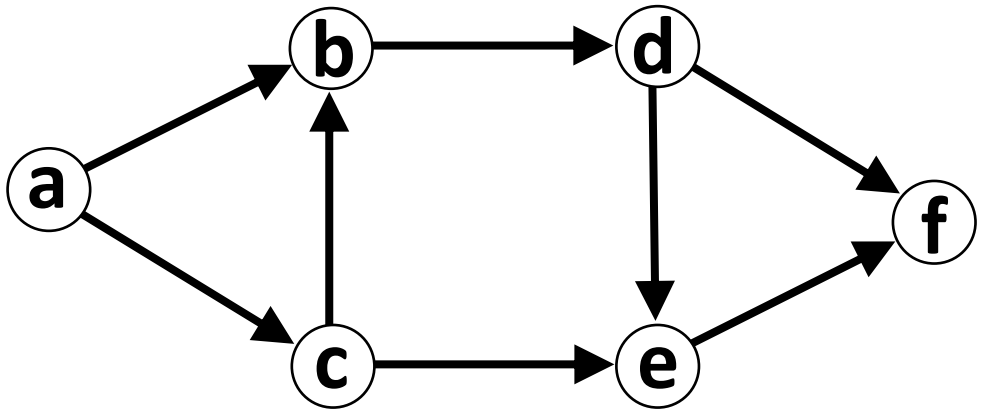
Vertices (or Nodes)
Edges
$\Big\}\ G = (V, E)$

- Connected Graph = Graph that has a path between every vertex pair.
- Acyclic Graph = Graph with no cycles.





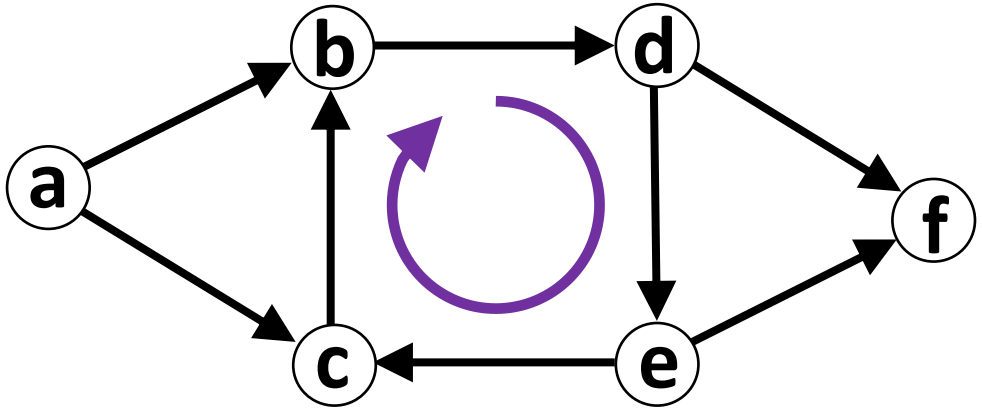
Special Graphs



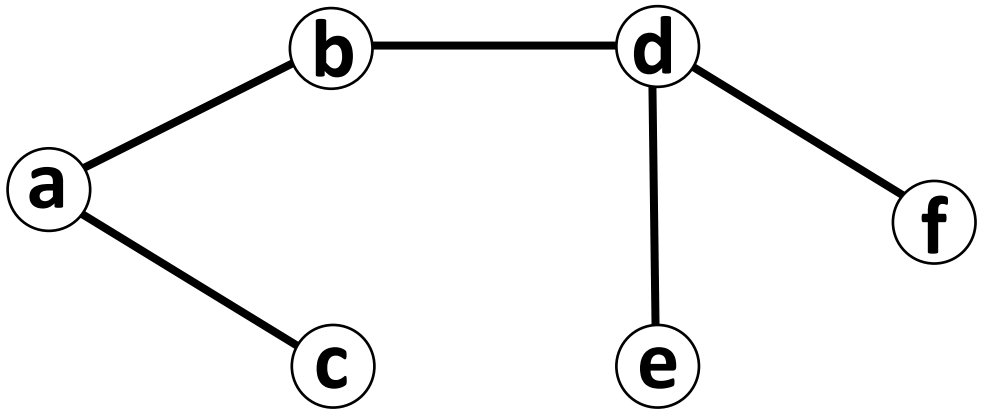
Vertices (or Nodes)
Edges

$$G = (V, E)$$

- Connected Graph = Graph that has a path between every vertex pair.
- Acyclic Graph = Graph with no cycles.
- Directed Acyclic Graph (DAG) = Directed graph with no cycles.

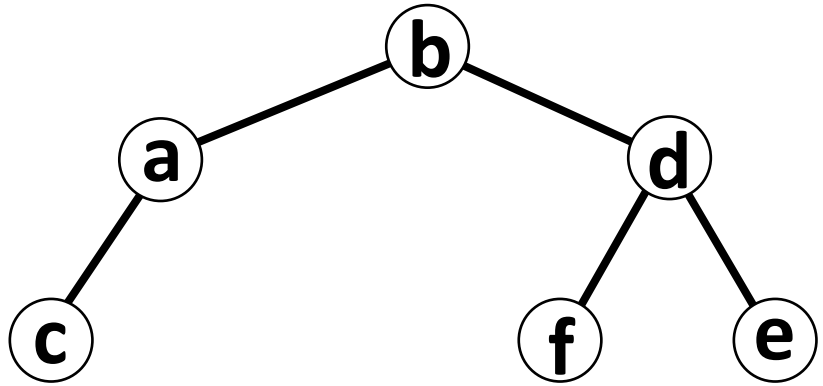


Special Graphs

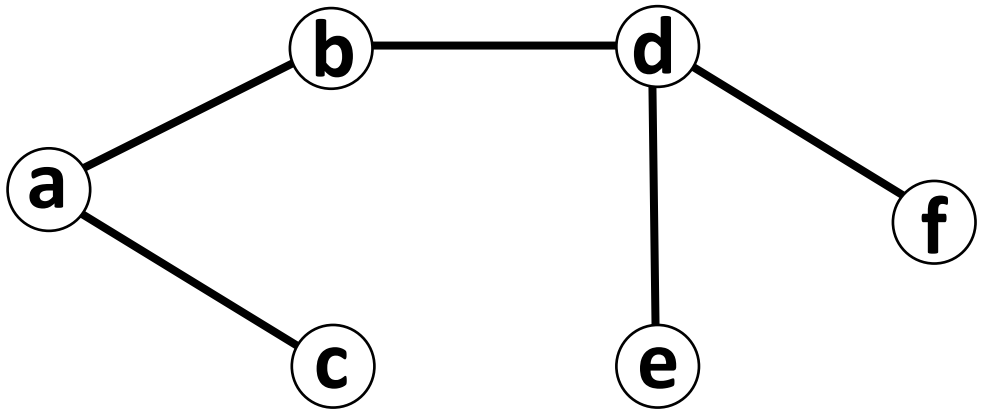


Vertices (or Nodes)
Edges
$\}$ $\boldsymbol{G = (V, E)}$

- Connected Graph = Graph that has a path between every vertex pair.
- Acyclic Graph = Graph with no cycles.
- Directed Acyclic Graph (DAG) = Directed graph with no cycles.
- Tree = Connected acyclic graph.

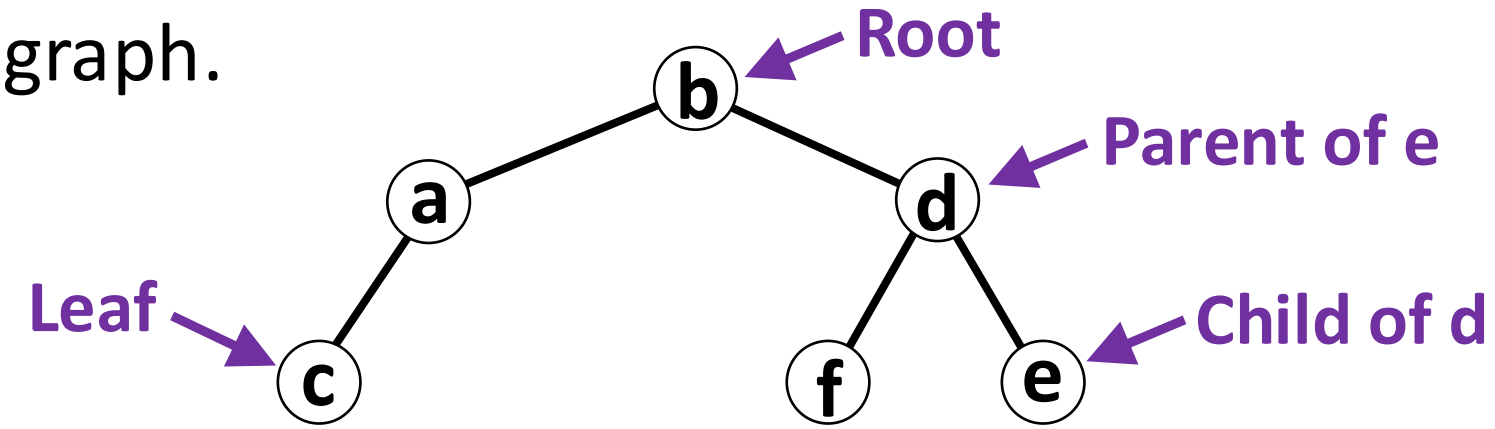


Special Graphs

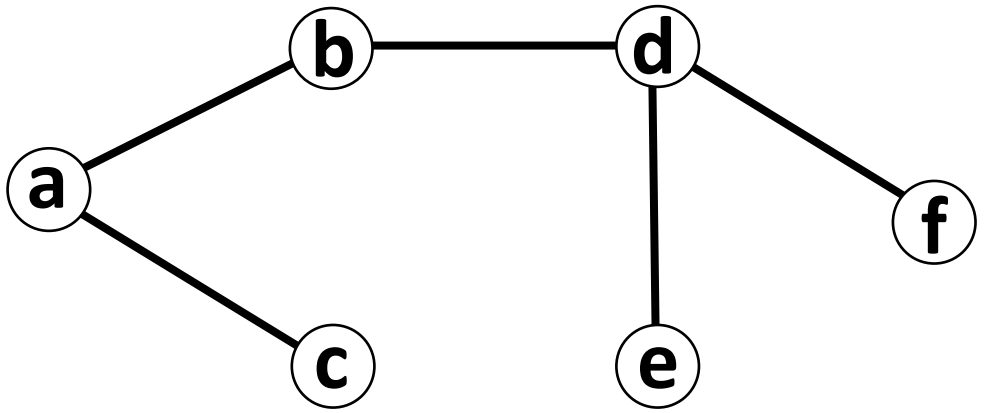


Vertices (or Nodes)
Edges
$\Big\}\ G = (V, E)$

- Connected Graph = Graph that has a path between every vertex pair.
- Acyclic Graph = Graph with no cycles.
- Directed Acyclic Graph (DAG) = Directed graph with no cycles.
- Tree = Connected acyclic graph.

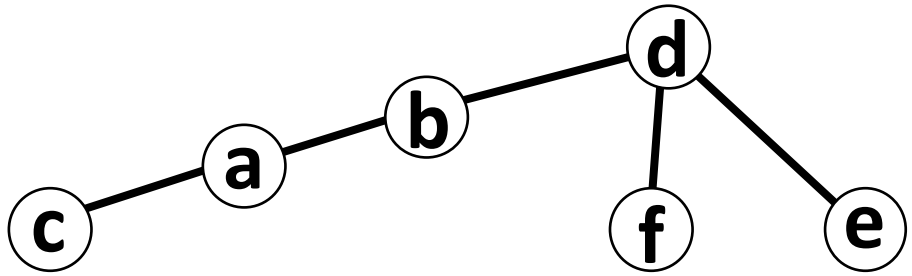


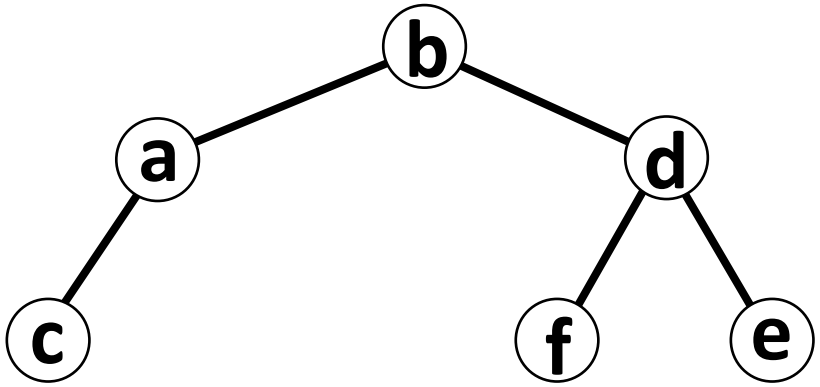
Special Graphs



Vertices (or Nodes)
Edges
$\Big\}\ G = (V, E)$

- Connected Graph = Graph that has a path between every vertex pair.
- Acyclic Graph = Graph with no cycles.
- Directed Acyclic Graph (DAG) = Directed graph with no cycles.
- Tree = Connected acyclic graph.





Topologically equivalent, but information may be lost…