In-Class Quiz

- QuizCST370-S19_Quiz_5 on iLearn
- Password is jellyfish

CST 370 – ADVANCED ALGORITHMS

7. Best-, Worst-, & Average-Case Complexity Analysis

Objectives

You will be able to...

1. Informally define best-, worst-, and average-case complexities.
2. Identify edge cases that change the complexity of a given algorithm.
3. Informally define Big Omega and Big Theta notation.
4. Identify the formal definitions of Big Omega and Big Theta
5. Use Big O, Big Omega, and Big Theta notation to express the best-, worst-, and average-case performance of algorithms.

Calculating Big O

Time Complexity

Is the complexity always the same for all inputs?

Calculating Big O

Time Complexity

Is the complexity always the same for all inputs?

You guessed it...**no**. Computational complexity describes the relationship between input size and efficiency, but other attributes of input can affect its efficiency.

Calculating Big O

Time Complexity

```
int someFunc(int x) {
  int sum = 0;
  if (x % 100 == 0)
    sum += x;
  else if (x % 100 == 1)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {
    for (int i = 0; i < x; i++)
      sum += i*i;
  return sum;
}
```

Calculating Big O

Time Complexity

```
int someFunc(int x) {
  int sum = 0;                              O(1)
  if (x % 100 == 0)                         O(1)
    sum += x;                               O(1)
  else if (x % 100 == 1)                    O(1)
    for (int i = 0; i < x; i++)             O(n)
      for (int j = 0; j < x; j++)           O(n)
        sum += i*j;                         O(1)
  else {                                    O(1)
    for (int i = 0; i < x; i++)             O(n)
      sum += i*i;                           O(1)
  return sum;                               O(1)
}
```

Calculating Big O

Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                  O(1)
  if (x % 100 == 0)                             O(1)
    sum += x;
  else if (x % 100 == 1)                        O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {                                        O(n)
    for (int i = 0; i < x; i++)
      sum += i*i;
  return sum;                                   O(1)
}
```

Calculating Big O

Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                    O(1)
  if (x % 100 == 0)                               O(1)
    sum += x;
  else if (x % 100 == 1)                          O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {
    for (int i = 0; i < x; i++)                   O(n)
      sum += i*i;
  return sum;                                     O(1)
}
```

Only one of these three will be executed on each input.

Therefore, the Big O will vary for different inputs.

Calculating Big O

Best-Case Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                    O(1)
  if (x % 100 == 0)                               O(1)
    sum += x;
  else if (x % 100 == 1)                          O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {                                          O(n)
    for (int i = 0; i < x; i++)
      sum += i*i;
  return sum;                                     O(1)
}
```

Assuming we get to pick the input, what is the best time complexity we can achieve?

Calculating Big O

Worst-Case Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                  O(1)
  if (x % 100 == 0)                             O(1)
    sum += x;
  else if (x % 100 == 1)                        O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {                                        O(n)
    for (int i = 0; i < x; i++)
      sum += i*i;
  return sum;                                   O(1)
}
```

Assuming we get to pick the input, what is the worst time complexity we can achieve?

Calculating Big O

Average-Case Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                    O(1)
  if (x % 100 == 0)                               O(1)
    sum += x;
  else if (x % 100 == 1)                          O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {                                          O(n)
    for (int i = 0; i < x; i++)
      sum += i*i;
  return sum;                                     O(1)
}
```

Assuming we **don't** get to pick the input, what is the time complexity we should expect?

Calculating Big O

Average-Case Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                  O(1)
  if (x % 100 == 0)                             O(1)
    sum += x;
  else if (x % 100 == 1)                        O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {                                        O(n)
    for (int i = 0; i < x; i++)
      sum += i*i;
  return sum;                                   O(1)
}
```

What makes input “average”?

Calculating Big O

Average-Case Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                  O(1)
  if (x % 100 == 0)                             O(1)
    sum += x;
  else if (x % 100 == 1)                        O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {                                        O(n)
    for (int i = 0; i < x; i++)
      sum += i*i;
  return sum;                                   O(1)
}
```

What makes input “average”?

O((Best + Worst) / 2)?

Calculating Big O

Average-Case Time Complexity

```
int someFunc(int x) {
  int sum = 0;                                  O(1)
  if (x % 100 == 0)                             O(1)
    sum += x;
  else if (x % 100 == 1)                        O(n^2)
    for (int i = 0; i < x; i++)
      for (int j = 0; j < x; j++)
        sum += i*j;
  else {
    for (int i = 0; i < x; i++)                 O(n)
      sum += i*i;
  return sum;                                   O(1)
}
```

What makes input “average”?

~~O((Best + Worst) / 2)?~~

Complexity when given “average” inputs.

Calculating Big O

Average-Case Time Complexity

What makes input “average”?

- Input that has no special specifications (unlike the best and worst case input which we constructed to have certain properties)
- The majority of possible inputs
- The typical input for the problem
- No scientific or mathematical definition, but identifying it will become second nature with practice
- The average case is often the same as the best or worst case

Best-, Worst-, & Average-Case Complexities

Linear Search

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

Linear Search

Computational Complexity

Best	Average	Worst	Space

Best-, Worst-, & Average-Case Complexities

Linear Search: Best Case

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

Best-, Worst-, & Average-Case Complexities

Linear Search: Best Case

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

x is the first element (index 0 in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 8

Best-, Worst-, & Average-Case Complexities

Linear Search: Best Case O(1)

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

```
O(1) (the if statement evaluates to true on
the first iteration, i = 0, and returns early)
O(1)
O(1)

O(1) (but never executes)
```

x is the first element (index 0 in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 8

Best-, Worst-, & Average-Case Complexities

Linear Search: Worst Case

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

x is not found (not in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 11

Best-, Worst-, & Average-Case Complexities

Linear Search: Worst Case O(n)

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

```
O(n) (the if statement never evaluates to true
so all n iterations are executed)
O(1)
O(1)

O(1)
```

x is not found (not in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 11

Best-, Worst-, & Average-Case Complexities

Linear Search: Average Case

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

x is found somewhere in the middle

(in index `0<i<vec.size()-1` in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 6

Best-, Worst-, & Average-Case Complexities

Linear Search: Average Case O(n)

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {        O(n) (WHY?)
    if (vec[i] == x)                            O(1)
      return i;                                 O(1)
}
  return -1;                                    O(1) (but never executes)
}
```

x is found somewhere in the middle

(in index `0<i<vec.size()-1` in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 6

Best-, Worst-, & Average-Case Complexities

Linear Search: Average Case O(n)

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {        O(n) (WHY?)
    if (vec[i] == x)                            O(1)
      return i;                                 O(1)
}
  return -1;                                    O(1) (but never executes)
}
```

x is found somewhere in the middle (in index `0<i<vec.size()-1` in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 6

If x has equal probability of being found in any position, on average it is in position `n/2`, so the loop executes `n/2` times, which is just `O(n)` because we drop constants.

Best-, Worst-, & Average-Case Complexities

Linear Search: Average Case O(n)

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {        O(n) (WHY?)
    if (vec[i] == x)                            O(1)
      return i;                                 O(1)
}
  return -1;                                    O(1) (but never executes)
}
```

x is found somewhere in the middle (in index `0<i<vec.size()-1` in vec)

[8, 4, 3, 6, 1, 5, 10, 9], x = 6

Alternatively, think of x as being located in position `k * n` where `0 < k < 1`. Then the loop executes k * n times which again is just `O(n)` because we drop constants.

Best-, Worst-, & Average-Case Complexities

Linear Search: Space O(1)

```
int linearSearch(vector<int> &vec, int x) {
  for (int i = 0; i < vec.size(); i++) {        O(1)
    if (vec[i] == x)
      return i;
}
  return -1;
}
```

Linear Search

Computational Complexity

Best	Average	Worst	Space
O(1)	O(n)	O(n)	O(1)

Best-, Worst-, & Average-Case Complexities

Space Complexity

When we talk about space, we usually only give the worst-case. Why?

Best-, Worst-, & Average-Case Complexities

Space Complexity

When we talk about space, we usually only give the worst-case. Why? In practice, the best-, worst-, and average-case complexities are usually the same for space. In addition, as time is typically the more constrained resource, it usually suffices to ensure that we won't run out of memory in the worst case.

Best-, Worst-, & Average-Case Complexities

Binary Search

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)
     return -1;
   int m = (l + r) / 2;
   if (x < vec[m])
     return binarySearch(vec, x, l, m - 1);
   else if (x > vec[m])
     return binarySearch(vec, x, m + 1, r);
   else
     return m;
}
```

Best-, Worst-, & Average-Case Complexities

Binary Search: Best Case

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)
     return -1;
   int m = (l + r) / 2;
   if (x < vec[m])
     return binarySearch(vec, x, l, m - 1);
   else if (x > vec[m])
     return binarySearch(vec, x, m + 1, r);
   else
     return m;
}
```

x is the middle element

[1, 3, 4, 5, 6, 8, 9, 10], x = 5

Best-, Worst-, & Average-Case Complexities

Binary Search: Best Case O(1)

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)                                    O(1)
     return -1;                                  Never Executes
   int m = (l + r) / 2;                          O(1)
   if (x < vec[m])                               O(1)
     return binarySearch(vec, x, l, m - 1);      Never Executes
   else if (x > vec[m])                          O(1)
     return binarySearch(vec, x, m + 1, r);      Never Executes
   else                                          O(1) (Since x is the middle element x = vec[m]
     return m;                                   O(1) so no recursive calls are made)
}
```

x is the middle element

[1, 3, 4, 5, 6, 8, 9, 10], x = 5

Best-, Worst-, & Average-Case Complexities

Binary Search: Worst Case

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)
     return -1;
   int m = (l + r) / 2;
   if (x < vec[m])
     return binarySearch(vec, x, l, m - 1);
   else if (x > vec[m])
     return binarySearch(vec, x, m + 1, r);
   else
     return m;
}
```

x is not found (not in vec)

[1, 3, 4, 5, 6, 8, 9, 10], x = 11

Best-, Worst-, & Average-Case Complexities

Binary Search: Worst Case O(logn)

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)                                    O(1)
     return -1;                                  O(1)
   int m = (l + r) / 2;                          O(1)
   if (x < vec[m])                               O(1)
     return binarySearch(vec, x, l, m - 1);      What is the height of the recursive tree?
   else if (x > vec[m])                          O(1)
     return binarySearch(vec, x, m + 1, r);      What is the height of the recursive tree?
   else                                          O(1)
     return m;                                   Never Executes
}
```

x is not found (not in vec)

[1, 3, 4, 5, 6, 8, 9, 10], x = 11

Best-, Worst-, & Average-Case Complexities

Binary Search: Worst Case O(logn)

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)                                          O(1)
     return -1;                                        O(1)
   int m = (l + r) / 2;                                O(1)
   if (x < vec[m])                                     O(1)
     return binarySearch(vec, x, l, m - 1);            What is the height of the recursive tree?
   else if (x > vec[m])                                O(1)
     return binarySearch(vec, x, m + 1, r);            What is the height of the recursive tree?
   else                                                O(1)
     return m;                                         Never Executes
}
```

x is not found (not in vec)

[1, 3, 4, 5, 6, 8, 9, 10], x = 11

If x is not in vec, we recurse until `l > r` (the full max height of the recursive tree). Since we split the search space in half every recursive call, the height is `O(logn)`.

Best-, Worst-, & Average-Case Complexities

Binary Search: Average Case

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)
     return -1;
   int m = (l + r) / 2;
   if (x < vec[m])
     return binarySearch(vec, x, l, m - 1);
   else if (x > vec[m])
     return binarySearch(vec, x, m + 1, r);
   else
     return m;
}
```

x is found in an arbitrary index in vec

[1, 3, 4, 5, 6, 8, 9, 10], x = 9

Best-, Worst-, & Average-Case Complexities

Binary Search: Average Case O(logn)

```
int binarySearch(vector<int> &vec,                  O(1)
     int x, int l, int r) {                         O(1)
   if (l > r)                                       Never Executes
     return -1;                                     O(1)
   int m = (l + r) / 2;                             What is the height of the recursive tree?
   if (x < vec[m])                                  O(1)
     return binarySearch(vec, x, l, m - 1);         What is the height of the recursive tree?
   else if (x > vec[m])                             O(1)
     return binarySearch(vec, x, m + 1, r);         O(1)
   else
     return m;
}
```

x is found in an arbitrary index in vec

[1, 3, 4, 5, 6, 8, 9, 10], x = 9

Best-, Worst-, & Average-Case Complexities

Binary Search: Average Case O(logn)

```
int binarySearch(vector<int> &vec,                    O(1)
     int x, int l, int r) {                           O(1)
   if (l > r)                                         Never Executes
     return -1;                                       O(1)
   int m = (l + r) / 2;                               What is the height of the recursive tree?
   if (x < vec[m])                                    O(1)
     return binarySearch(vec, x, l, m - 1);           What is the height of the recursive tree?
   else if (x > vec[m])                               O(1)
     return binarySearch(vec, x, m + 1, r);           O(1)
   else
     return m;
}
```

x is found in an arbitrary index in vec

[1, 3, 4, 5, 6, 8, 9, 10], x = 9

If x is in vec, we recurse partway down the recursive tree. If x is equally likely to be found in any index, on average we will have to go halfway down the tree. This is `logn/2` operations which is still `O(logn)`.

Best-, Worst-, & Average-Case Complexities

Binary Search: Space O(logn)

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)
     return -1;
   int m = (l + r) / 2;                          O(1)
   if (x < vec[m])
     return binarySearch(vec, x, l, m - 1);
   else if (x > vec[m])
     return binarySearch(vec, x, m + 1, r);
   else
     return m;
}
```

```
  O(1) variables
+ O(logn) recursive stack
= O(logn)
```

Best-, Worst-, & Average-Case Complexities

Binary Search: Explicit Space O(1)

```
int binarySearch(vector<int> &vec,
     int x, int l, int r) {
   if (l > r)
     return -1;
   int m = (l + r) / 2;
   if (x < vec[m])
     return binarySearch(vec, x, l, m - 1);
   else if (x > vec[m])
     return binarySearch(vec, x, m + 1, r);
   else
     return m;
}
```

For recursive algorithms, I'm only going to care about explicit space complexity from now on.

- You shouldn't have to worry about implementing something recursively or iteratively in an interview.
- Every recursive algorithm can be rewritten iteratively (but it can get complicated).

Binary Search

Computational Complexity

Best	Average	Worst	Space
O(1)	O(logn)	O(logn)	O(1)

Now it’s your turn!

Work together to fill out the worksheet

- Your worksheet has on it all the algorithms we’ve talked about so far.
- Work together to fill out the time/space complexities for each algorithm.
- Use the notes sections to write down edge cases for the best/worst complexities.

Quick Sort

Computational Complexity

Best	Average	Worst	Space

Best-, Worst-, & Average-Case Complexities

Quick Sort

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {
      int pivot = partition(vec, l, r);
      quickSort(vec, l, pivot - 1);
      quickSort(vec, pivot + 1, r);
   }
}
```

Best-, Worst-, & Average-Case Complexities

Quick Sort: Best Case

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {
      int pivot = partition(vec, l, r);
      quickSort(vec, l, pivot - 1);
      quickSort(vec, pivot + 1, r);
   }
}
```

Partition always splits the array into two equal halves (pivots around the median element)

Best-, Worst-, & Average-Case Complexities

Quick Sort: Best Case O(nlogn)

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {                              O(1)
      int pivot = partition(vec, l, r);          O(n)
      quickSort(vec, l, pivot - 1);              What is the height of the recursive tree?
      quickSort(vec, pivot + 1, r);              What is the height of the recursive tree?
   }
}
```

Partition always splits the array into two equal halves (pivots around the median element)

Best-, Worst-, & Average-Case Complexities

Quick Sort: Best Case O(nlogn)

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {                                  O(1)
      int pivot = partition(vec, l, r);              O(n)
      quickSort(vec, l, pivot - 1);                  What is the height of the recursive tree?
      quickSort(vec, pivot + 1, r);                  What is the height of the recursive tree?
   }
}
```

Partition always splits the array into two equal halves (pivots around the median element)

If we split the array exactly in half every time until the resulting arrays have size 1, we will split it `logn` times (by definition) so each recursive tree will have height `logn`.

Best-, Worst-, & Average-Case Complexities

Quick Sort: Worst Case

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {
      int pivot = partition(vec, l, r);
      quickSort(vec, l, pivot - 1);
      quickSort(vec, pivot + 1, r);
   }
}
```

Partition always splits the array into
two pieces of lengths 0 and size – 1
(one half is always empty)

Best-, Worst-, & Average-Case Complexities

Quick Sort: Worst Case $O(n^2)$

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {                                  O(1)
      int pivot = partition(vec, l, r);              O(n)
      quickSort(vec, l, pivot - 1);                  What is the height of the recursive tree?
      quickSort(vec, pivot + 1, r);                  What is the height of the recursive tree?
   }
}
```

Partition always splits the array into two pieces of lengths 0 and size – 1 (one half is always empty)

Best-, Worst-, & Average-Case Complexities

Quick Sort: Worst Case $O(n^2)$

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {                                  O(1)
      int pivot = partition(vec, l, r);              O(n)
      quickSort(vec, l, pivot - 1);                  What is the height of the recursive tree?
      quickSort(vec, pivot + 1, r);                  What is the height of the recursive tree?
   }
}
```

Partition always splits the array into two pieces of lengths 0 and size – 1 (one half is always empty)

If we split the array in two so that one piece always has s size 0 until the resulting arrays have size 1, we only reduce the size of the array by 1 at every split. Therefore one recursive tree will have height 0 and the other will have height n (visualize a recursive tree that looks like a linked list).

Best-, Worst-, & Average-Case Complexities

Quick Sort: Average Case

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {
      int pivot = partition(vec, l, r);
      quickSort(vec, l, pivot - 1);
      quickSort(vec, pivot + 1, r);
   }
}
```

Partition splits the array into two non-zero length pieces

Best-, Worst-, & Average-Case Complexities

Quick Sort: Average Case O(nlgon)

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {                                  O(1)
      int pivot = partition(vec, l, r);              O(n)
      quickSort(vec, l, pivot - 1);                  What is the height of the recursive tree?
      quickSort(vec, pivot + 1, r);                  What is the height of the recursive tree?
   }
}
```

Partition splits the array into two non-zero length pieces

Best-, Worst-, & Average-Case Complexities

Quick Sort: Average Case O(nlgon)

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {                                  O(1)
      int pivot = partition(vec, l, r);              O(n)
      quickSort(vec, l, pivot - 1);                  What is the height of the recursive tree?
      quickSort(vec, pivot + 1, r);                  What is the height of the recursive tree?
   }
}
```

Partition splits the array into two non-zero length pieces

If we always partition the array around an arbitrary pivot, each element is equally likely to be on the left or right sides, so on average, each side ends up with on average `n/2` elements which makes the height on average `logn`.

Best-, Worst-, & Average-Case Complexities

Quick Sort: Space O(1)

```
void quickSort(vector<int> &vec, int l, int r) {
   if (r - l > 0) {
      int pivot = partition(vec, l, r);           O(1)
      quickSort(vec, l, pivot - 1);
      quickSort(vec, pivot + 1, r);
   }
}
```

```
 O(1) variables
+ O(logn) recursive stack
= O(logn)
```

Quick Sort

How to pick a pivot?

1. First element
2. Last element
3. Average element
4. Median element
5. Random element
6. Middle element
7. Median of first, last, and middle elements

Quick Sort

How to pick a pivot?

1. First element
2. Last element
3. Average element
4. Median element
5. Random element
6. Middle element
7. Median of first, last, and middle elements

Causes worst-case behavior (does not split the array) if array is sorted

Difficult/expensive to compute

What we did in class

Even better than what we did in class

Quick Sort

Computational Complexity

Best	Average	Worst	Space
O(nlogn)	O(nlogn)	$O(n^2)$	O(1)

Big O Notation

Informal Definition

`O(n)`: Read this as “Big O of n.”

Big O is an *approximation* that tells us about an algorithm’s *worst-case scenario* use of time or space.

Big Ω Notation

Informal Definition

Ω(n): Read this as “Big Omega of n.” Assume n is the input size.
If this is confusing, think of Ω(n) time as meaning an algorithm runs in *approximately* n operations *at least*.
Similarly, Ω(n) space means an algorithm consumes *approximately* n units of memory *at least*.

Big Ω is an *approximation* that tells us about an algorithm’s *best-case scenario* use of time or space.

Big O Notation

Formal Definition

Mathematically, Big-O is **the least** (loose) upper bound on the number of operations or amount of memory used by an algorithm.

Big Ω Notation

Formal Definition

Mathematically, Big-Ω is **the greatest** (loose) lower bound on the number of operations or amount of memory used by an algorithm.

Big O & Big Ω Notation

Usage

Instead of saying an algorithm is best-case $O(n)$ and worst-case $O(n^2)$, we can instead say it is $O(n^2)$ and $\Omega(n)$. These statements are equivalent and both acceptable.

Big O & Big Ω Notation

Usage

Instead of saying an algorithm is best-case O(n) and worst-case $O(n^2)$, we can instead say it is $O(n^2)$ and Ω(n). These statements are equivalent and both acceptable.

Note: As you have seen, Big-O is sometimes qualified with "x-case" to refer to a case other than the worst-case (the formal definition). Big-Ω is never used to refer to a case other than the best-case. Remember, if you see Big-O without the "x-case" qualifier you should infer worst-case.

Big θ Notation

Informal Definition

`θ(n):` Read this as “Big Theta of n.” Assume n is the input size.

If this is confusing, think of θ(n) time as meaning an algorithm *always* runs in *approximately* n operations.

Similarly, θ(n) space means an algorithm *always* consumes *approximately* n units of memory.

Big θ Notation

Informal Definition

`θ(n):` Read this as “Big Theta of n.” Assume n is the input size.

If this is confusing, think of θ(n) time as meaning an algorithm *always* runs in *approximately* n operations.

Similarly, θ(n) space means an algorithm *always* consumes *approximately* n units of memory.

Big θ Notation

Formal Definition

Mathematically, Big-θ is the asymptotic tight bound on the number of operations or amount of memory used by an algorithm.

Note: An algorithm is $\theta(n)$ if and only if it is both $O(n)$ and $\Omega(n)$. This means not all algorithms have a Big-θ representation. If an algorithm has a Big-θ representation, it tells us the worst- and best- (and therefore average-) cases are the same. You cannot *use Big-θ to refer to the average case otherwise.*

Big O Notation

Common Complexity Classes Compared

Retro video about sorting (1980)

https://youtu.be/gv0JUEqaAXo?t=125

Why do we care?

Efficient algorithms are important.

https://www.mercurial-scm.org/repo/hg/rev/0b03454abae7

Using mercurial (source control) at Facebook was slow because of an inefficient algorithm in mercurial.

So we fixed it!

Why was it slow in the first place?

Working code is more important.

"Premature Optimization is the root of all evil"

~Donald Knuth

Facebook's codebase was larger than the input size mercurial expected.

Break!

10-minute break

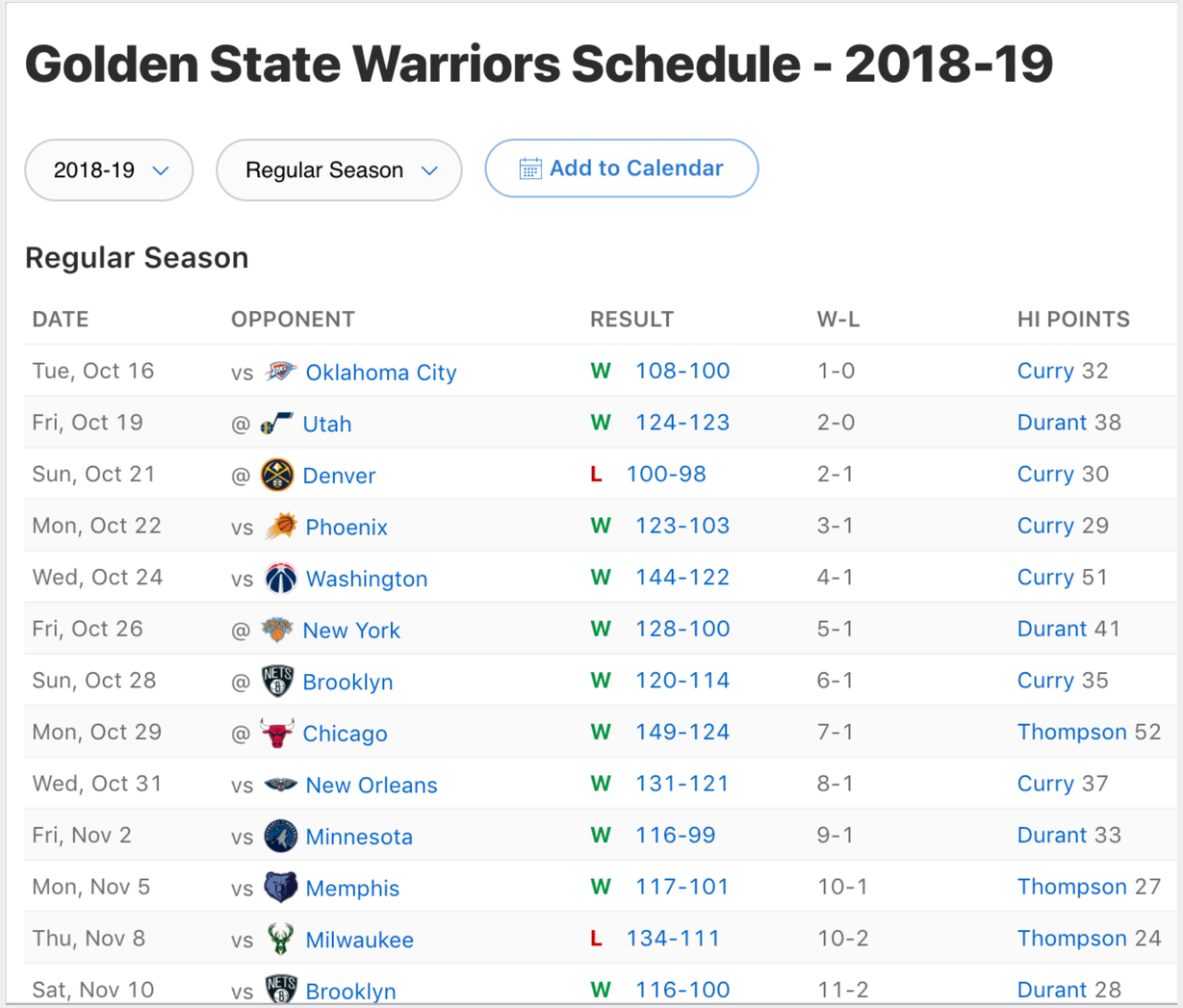
This wraps up the Computational Complexity unit. We're going to start talking about new data structures and get back into coding next!

What do these have in common?

- A Map of the continental US

What do these have in common?

- A Map of the continental US
- An NBA team schedule



Golden State Warriors Schedule - 2018-19

2018-19 | Regular Season | Add to Calendar

Regular Season

DATE	OPPONENT	RESULT	W-L	HI POINTS
Tue, Oct 16	vs Oklahoma City	W 108-100	1-0	Curry 32
Fri, Oct 19	@ Utah	W 124-123	2-0	Durant 38
Sun, Oct 21	@ Denver	L 100-98	2-1	Curry 30
Mon, Oct 22	vs Phoenix	W 123-103	3-1	Curry 29
Wed, Oct 24	vs Washington	W 144-122	4-1	Curry 51
Fri, Oct 26	@ New York	W 128-100	5-1	Durant 41
Sun, Oct 28	@ Brooklyn	W 120-114	6-1	Curry 35
Mon, Oct 29	@ Chicago	W 149-124	7-1	Thompson 52
Wed, Oct 31	vs New Orleans	W 131-121	8-1	Curry 37
Fri, Nov 2	vs Minnesota	W 116-99	9-1	Durant 33
Mon, Nov 5	vs Memphis	W 117-101	10-1	Thompson 27
Thu, Nov 8	vs Milwaukee	L 134-111	10-2	Thompson 24
Sat, Nov 10	vs Brooklyn	W 116-100	11-2	Durant 28

What do these have in common?

- A Map of the continental US
- An NBA team schedule
- A college course schedule

Cohort 5

Semester	Course	Title	Institution	Requirement
FALL	CSS 1	Introduction to Computer Science & Programming Fundamentals	Hartnell	Major
	ENG 1A	College Composition & Reading	Hartnell	A2
	MAT 3A	Analytic Geometry & Calculus I	Hartnell	B4 / Major
	PHL 10	Ethics	Hartnell	C2
	CST 286	Physics of Computing	CSUMB	B1 & B3
WINTER	BIO 42	Human Biology	Hartnell	E
SPRING	ENG 2	Critical Thinking & Writing	Hartnell	A3
	MUS 5	Ethnic Music in the U.S.	Hartnell	C1
	CSS 2A	Object Oriented Programming	Hartnell	Major
	CSS 7	Discrete Structures	Hartnell	Major
	CST 205	Multimedia Design & Programming	CSUMB	Major
SUMMER	COM 1	Introduction to Public Speaking	Hartnell	A1
	POL 1	American Political Institutions	Hartnell	US 2 & 3
FALL	ENG 1B	College Literature & Composition	Hartnell	C2
	CSS 2B	Data Structures & Algorithms	Hartnell	Major
	CSS 3	Computer Architecture & Assembly Language Programming	Hartnell	Major
	BIO 48/48L	Environmental Science / Environmental Science Lab	Hartnell	B2 & B3
	MATH 270	Mathematics for Computing	CSUMB	Major
WINTER	HIS 17B	United States History B (Civil War-Present)	Hartnell	US1
SPRING	CST 300	Major ProSeminar	CSUMB	Major
	CST 338	Software Design	CSUMB	Major
	CST 336	Internet Programming	CSUMB	Major
	CST 370	Design & Analysis of Algorithms	CSUMB	Major
SUMMER		***Left open for internship experience***		
FALL	CST 363	Introduction to Database Systems	CSUMB	Major
	CST 334	Operating Systems	CSUMB	Major
	CST 462S	Computer Science and Community	CSUMB	Major / UDGE

What do these have in common?

They can be modeled with graphs!

- A Map of the continental US.
- An NBA team schedule.
- Course schedule.

Graphs and Graph algorithms we'll be talking about in this unit can be used to create these maps, schedules, and solve other problems!

CST 370 – ADVANCED ALGORITHMS

7.2 Introduction to Graphs

Objectives

You will be able to...

1. Informally and formally define a graph.
2. Define and identify properties of a graph (cyclic/acyclic, directed/undirected, connected/unconnected).
3. Represent a graph using adjacency list notation.

Graphs

Informal Definition

A graph is a data structure that models connections between different items.

- Countries are connected by borders
- Teams are connected by games played against each other
- Courses are connected by pre-requisites

Do Now

Representing Friendships

Draw a diagram to represent your social network (i.e., who you are friends with, who your friends are friends with, who your friends' friends are friends with...)

Graphs

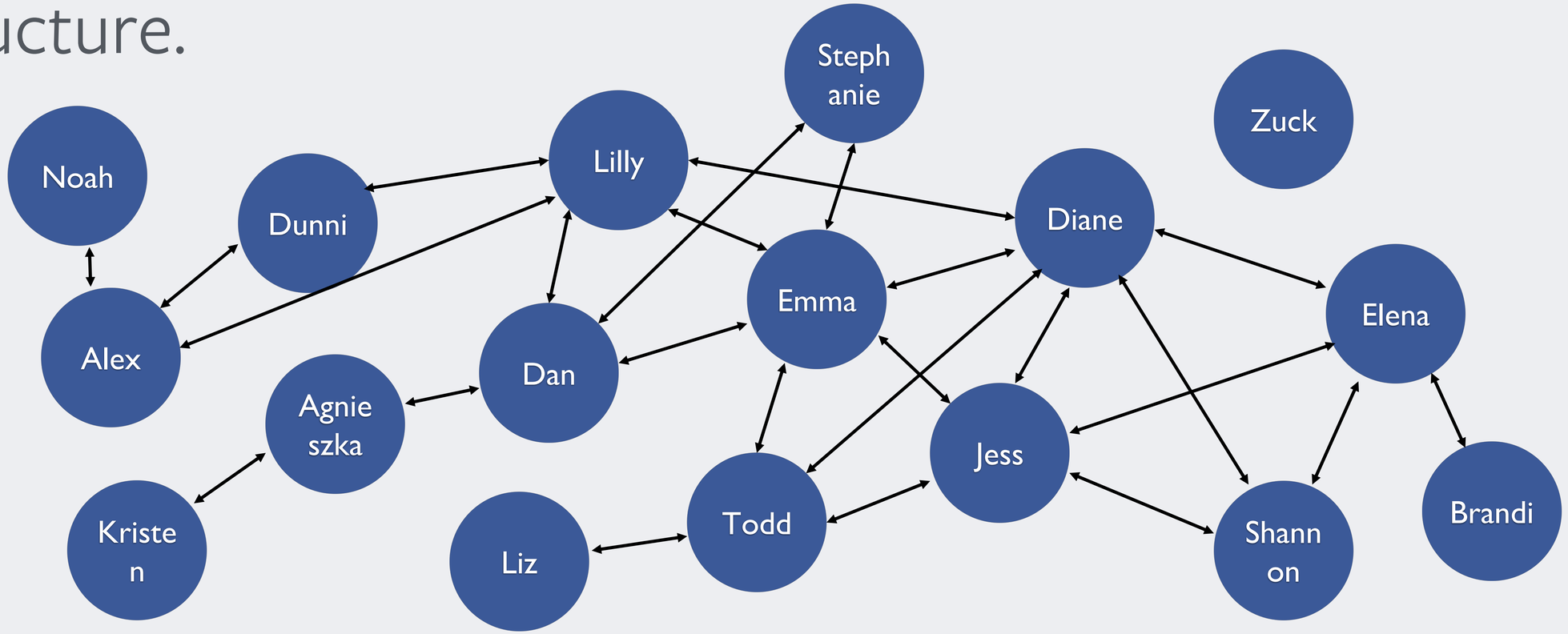
Use Case

Suppose we want to represent friendships in a data structure.

Graphs

Use Case

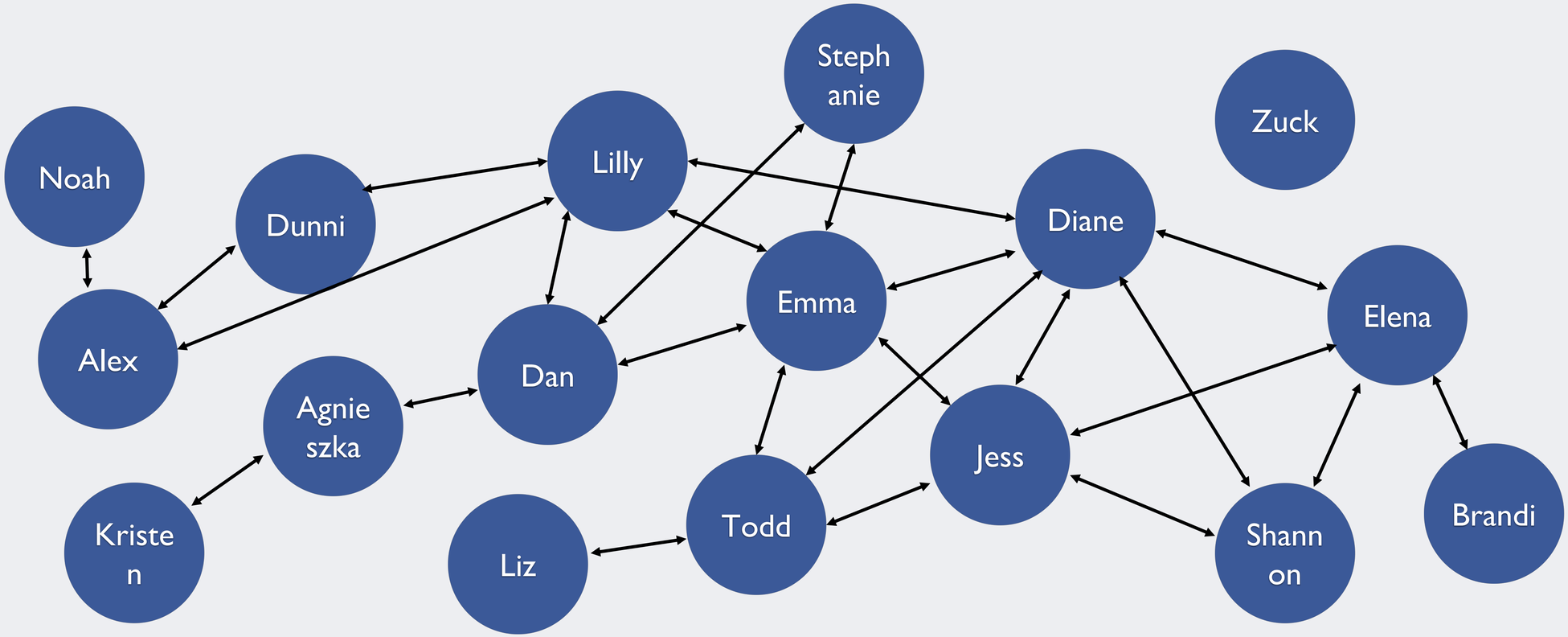
Suppose we want to represent friendships in a data structure.



Graphs

Properties

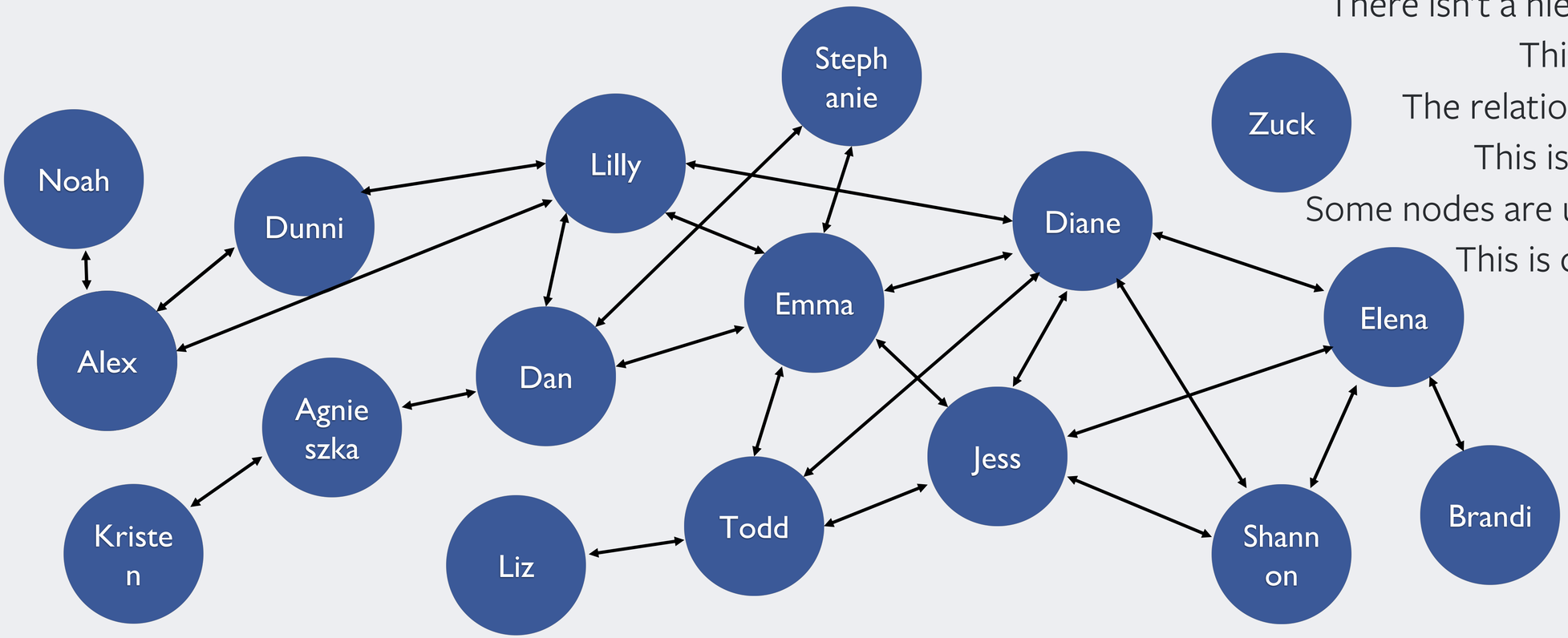
This looks a little like an n-ary tree, except...



Graphs

Properties

This looks a little like an n-ary tree, except...

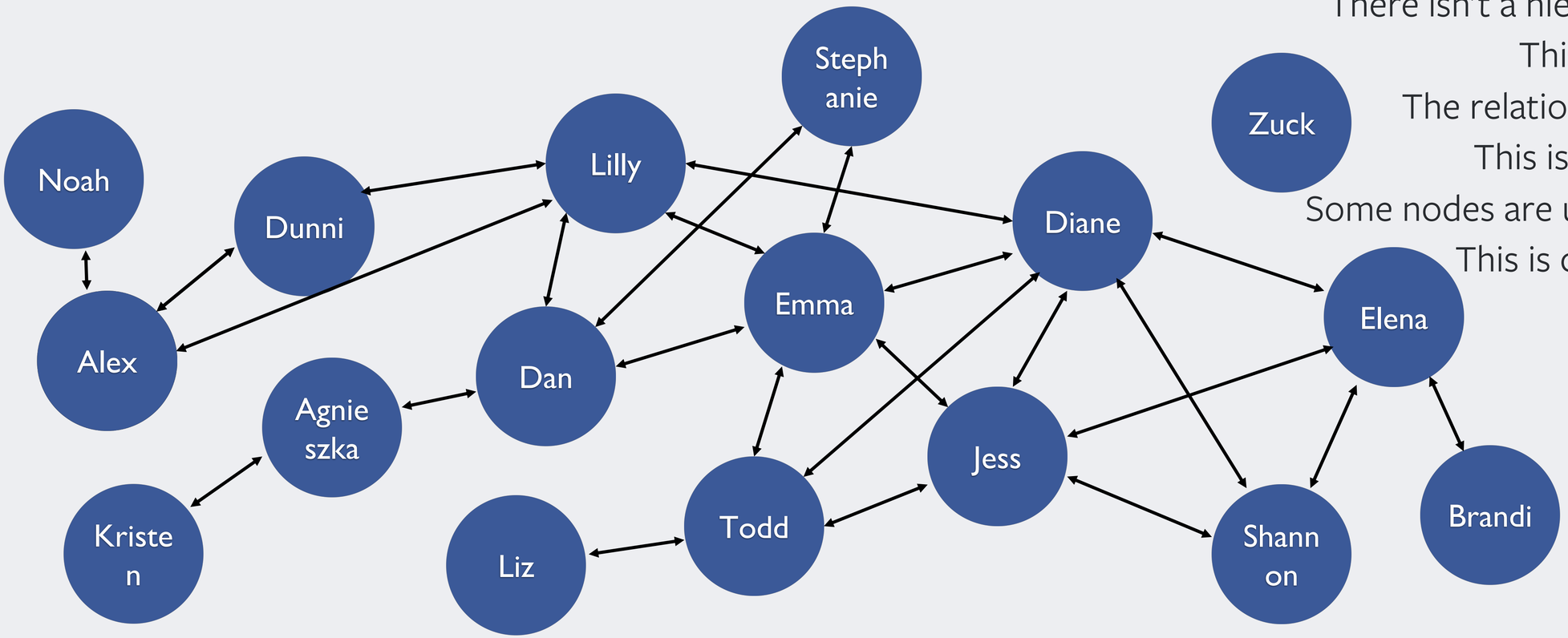


There isn't a hierarchy – you can go in circles
This is called **cyclic**.
The relationships are bi-directional.
This is called **undirected**.
Some nodes are unreachable from other nodes.
This is called **unconnected**.

Graphs

Properties

This looks a little like an n-ary tree, except...



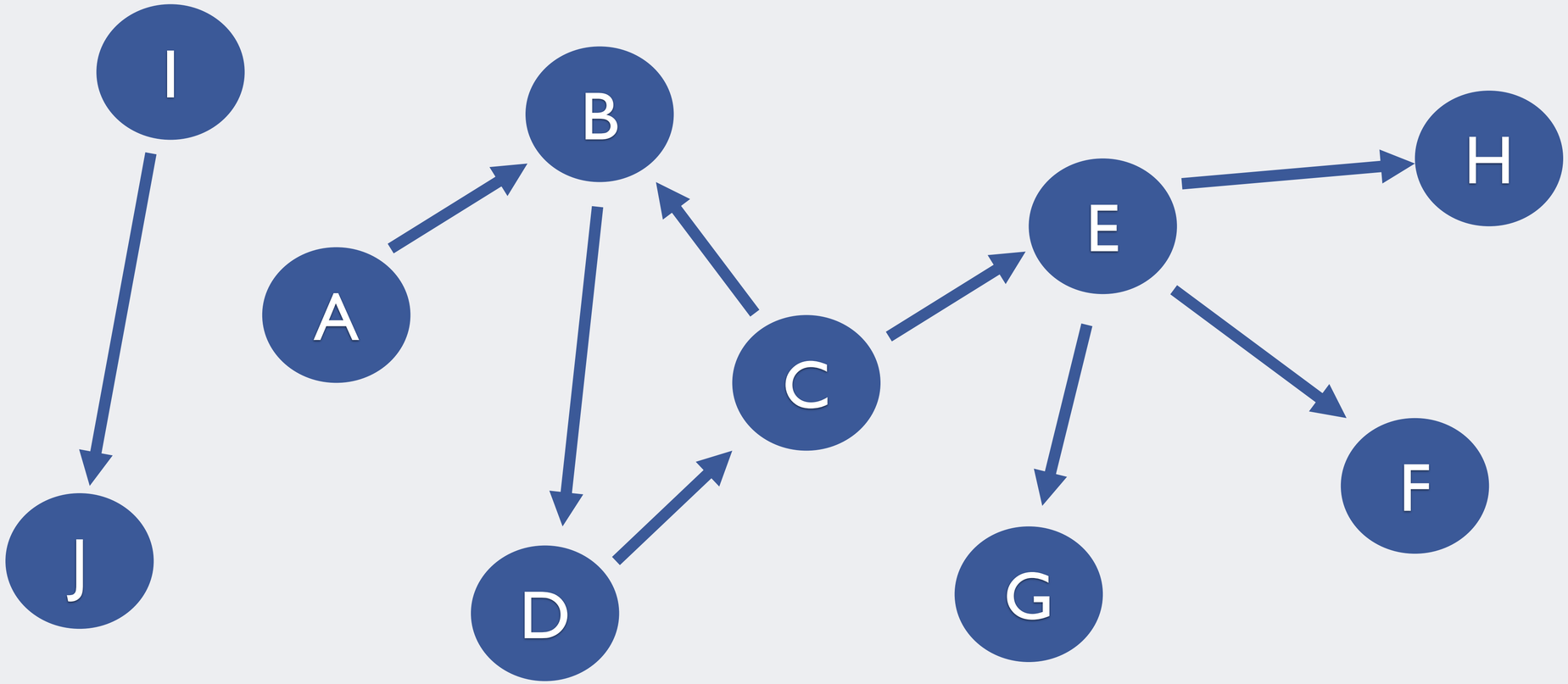
There isn't a hierarchy – you can go in circles
This is called **cyclic**.
The relationships are bi-directional.
This is called **undirected**.
Some nodes are unreachable from other nodes.
This is called **unconnected**.

Graphs

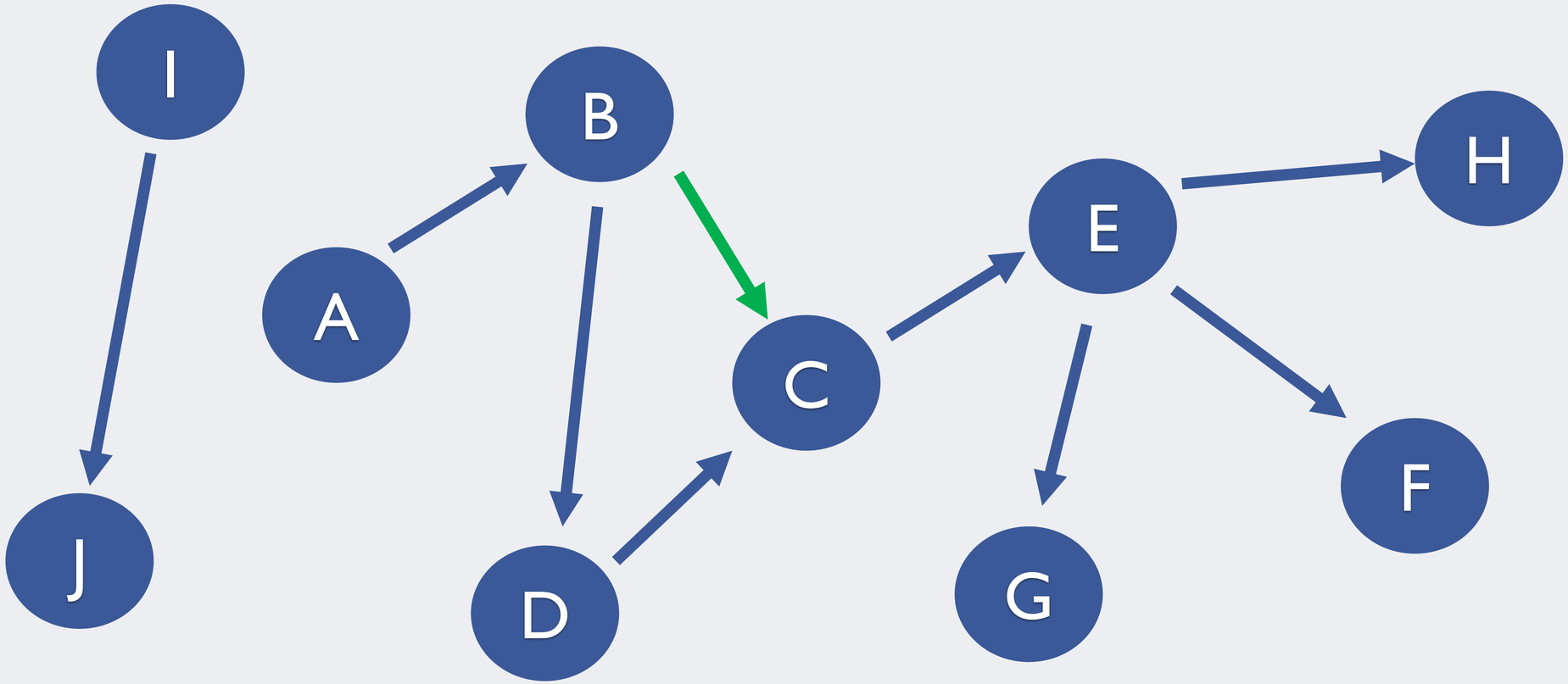
Properties

- Graphs may be **acyclic** or **cyclic**.
- Graphs may be **directed** or **undirected**.
- Graphs may be **connected** or **unconnected**.

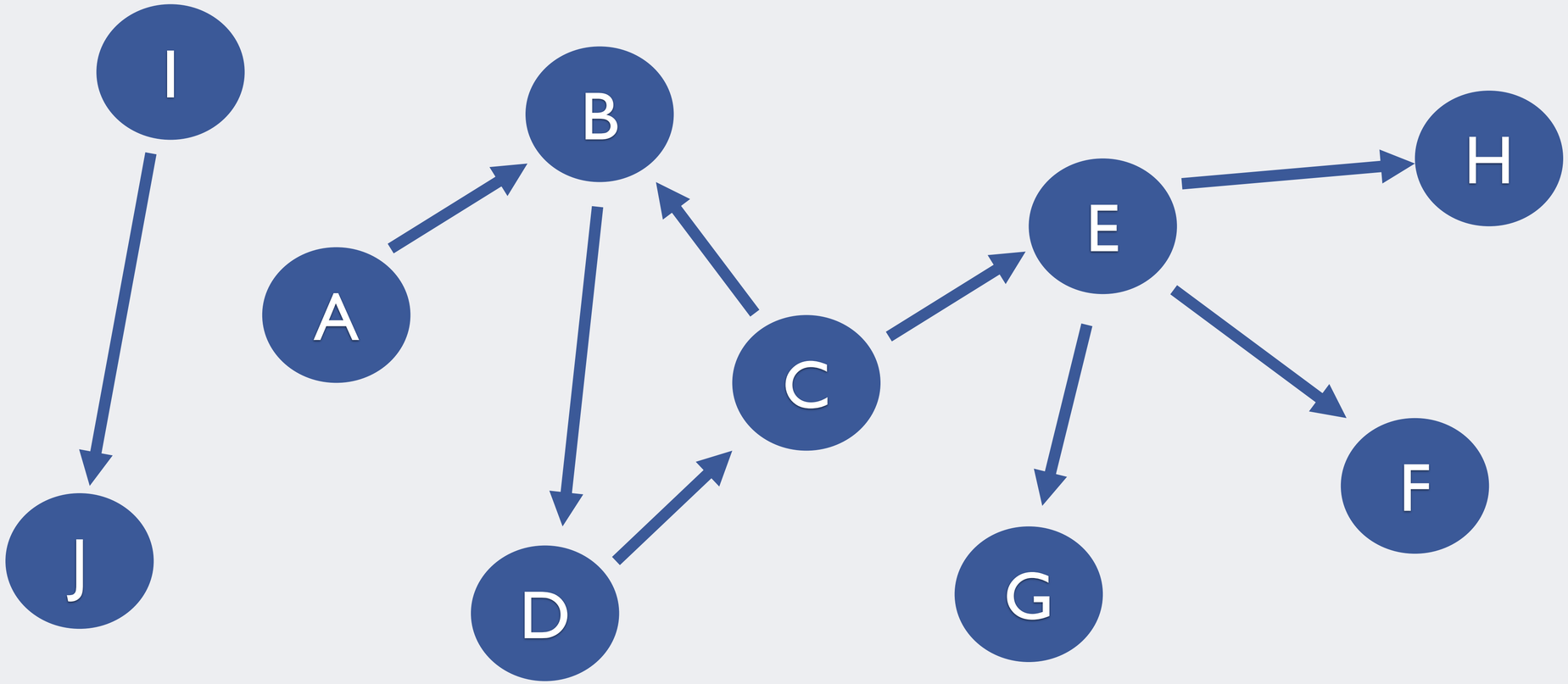
Cyclic Graph



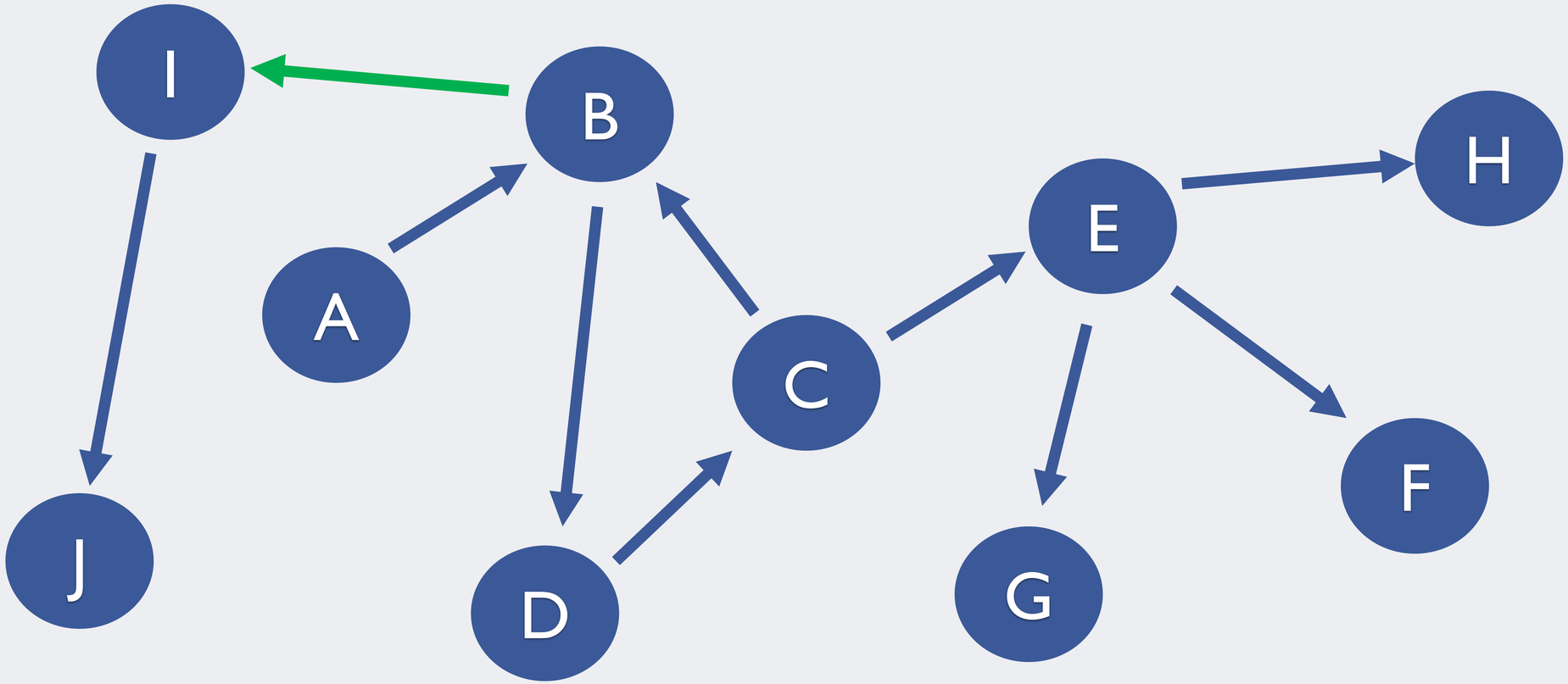
Acyclic Graph



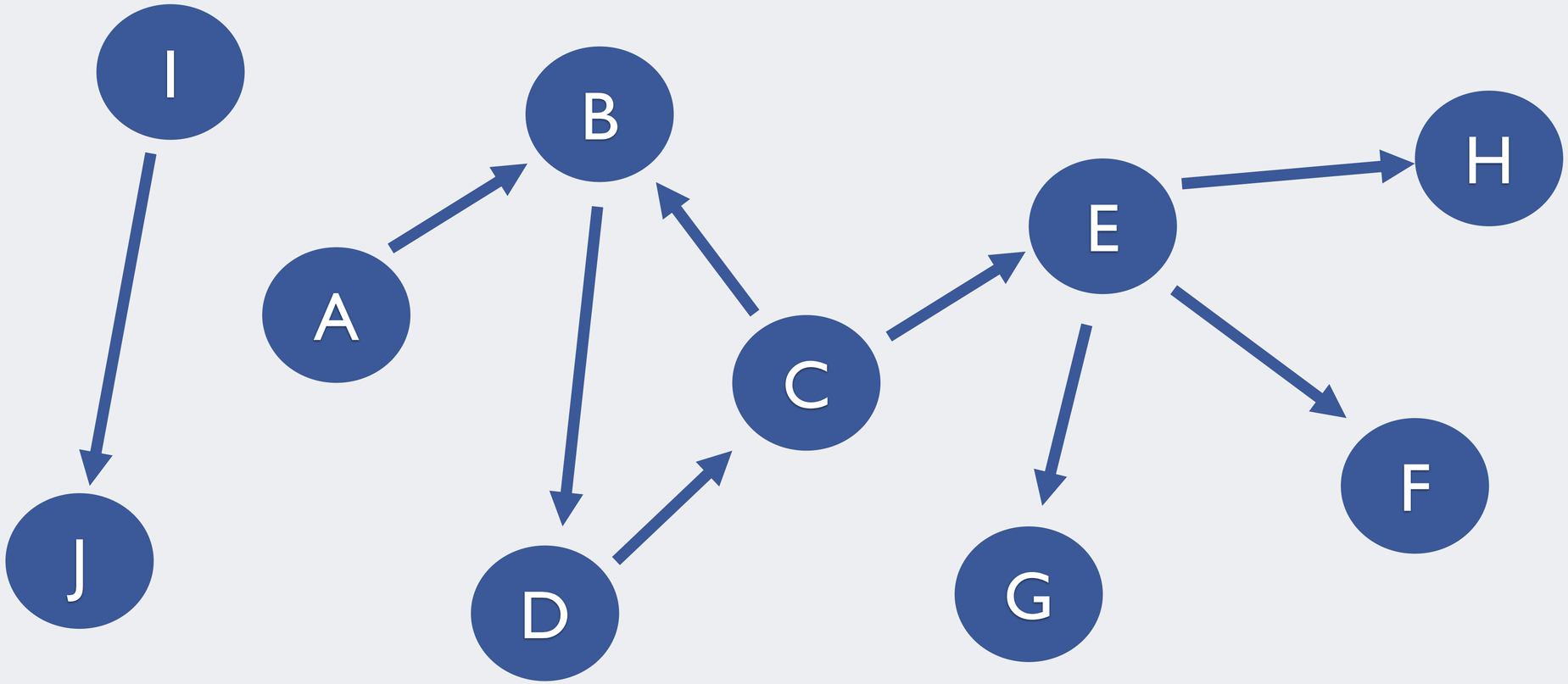
Unconnected Graph



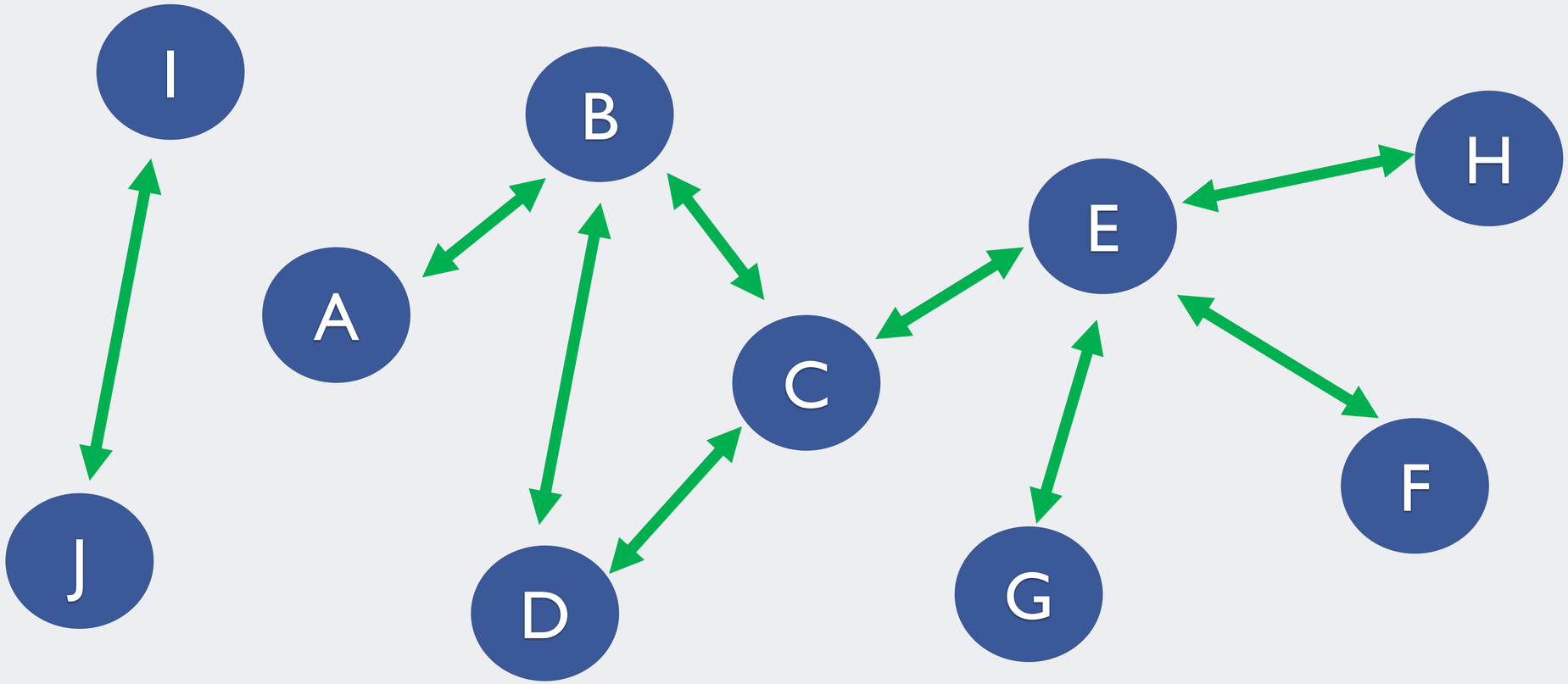
Connected Graph



Directed Graph



Undirected Graph



Graphs

Informal Definition

A tree is just a specific type of graph — a *connected acyclic directed* graph!

Graphs

Informal Definition

A tree is just a specific type of graph — a *connected acyclic directed* graph!

Therefore an informal definition for **a graph is a generalization of a tree** where each of these properties (connected, cyclic, directed) is optional.

Graphs

Formal Definition

But if graphs can have any combination of these properties, what do all graphs have in common?

Graphs

Formal Definition

But if graphs can have any combination of these properties, what do all graphs have in common?

Formally, **a graph is a set of vertices and a set of edges between two vertices**.

Graphs

Formal Definition

But if graphs can have any combination of these properties, what do all graphs have in common?

Formally, **a graph is a set of vertices and a set of edges between two vertices**.

Vertices (singular: *vertex*) is just the term for nodes in graphs.

Edges are the relationships or connections between nodes.

Graphs

Formal Definition

But if graphs can have any combination of these properties, what do all graphs have in common?

Formally, **a graph is a set of vertices and a set of edges between two vertices**.

Vertices (singular: *vertex*) is just the term for nodes in graphs.

Edges are the relationships or connections between nodes.

We say two vertices are *adjacent* if there is an edge between them.

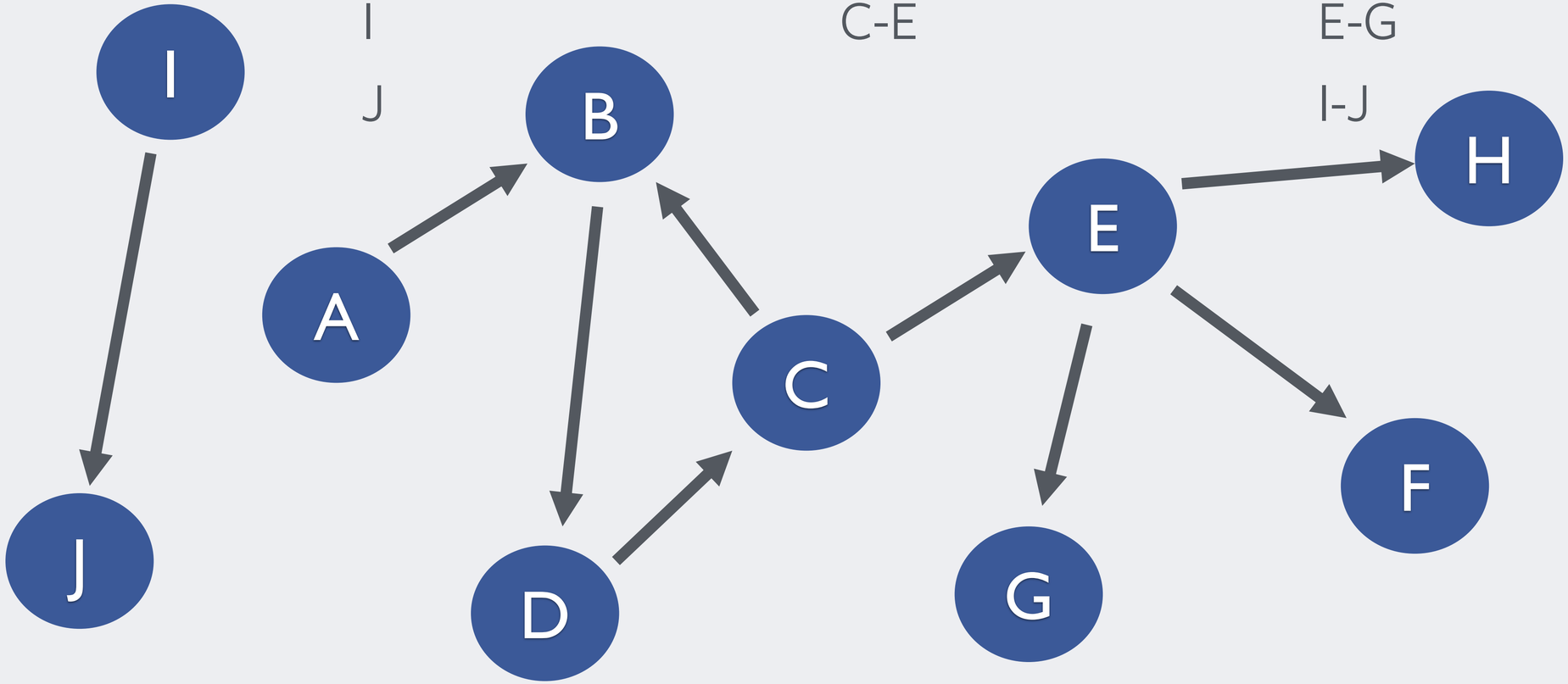
We call all the vertices adjacent to a vertex v the *neighbors* of v.

10 Vertices:

A
B
C
D
E
F
G
H
I
J

9 Edges:

A-B
B-D
C-B
C-E
D-C
E-F
E-H
E-G
I-J



Graphs

Representations

Recall how we represented a binary tree using a Node struct:

```
struct TreeNode {
  string data;
  TreeNode *left;
  TreeNode *right;
};
```

Graphs

Representations

We can also represent a graph using a Node struct as we did a binary tree, storing an array of neighbors instead of left and right children.

```
struct GraphNode {
  string data;
  vector<GraphNode *> neighbors;
};
```

Graphs

Representations

Whereas we represented an entire tree by a single node – its root – we represent a graph by an array of all its nodes:

```
vector<GraphNode *> vertices
```

Graphs

Representations

Whereas we represented an entire tree by a single node – its root – we represent a graph by an array of all its nodes:

```
vector<GraphNode *> vertices
```

This is called an **adjacency list** representation of a graph.

Graphs

Representations

Whereas we represented an entire tree by a single node – its root – we represent a graph by an array of all its nodes:

```
vector<GraphNode *> vertices
```

This is called an **adjacency list** representation of a graph.

Why can we not just use a root node?

Graphs

Representations

Whereas we represented an entire tree by a single node – its root – we represent a graph by an array of all its nodes:

```
vector<GraphNode *> vertices
```

This is called an **adjacency list** representation of a graph.

Why can we not just use a root node?

In an unconnected graph, we will not be able to reach all nodes in the graph from any single starting point!

Graphs

Use Case

We saw an example of using a graph to represent a social network (friendships); more generally, graphs are useful for representing relationships between entities:

- Geographic locations (flight finding, driving/walking directions, air traffic routes, etc.)
- Transportation systems (design, directions, etc.)
- Recommendation engines
- Page rank algorithms
- Network routing
- AI for game playing
- Matching problems (i.e., organ donors to patients)
- Scheduling problems (i.e., MLB season calendar)
- Clustering/classification problems (i.e., diseases, species, consumer markets, text for natural language processing)
- Robotics