Trees and Intro to Counting
A Special Type of Graph: Trees

```
program
  int id(main) () {} code
  instruction code
    id(cout) ...
    instruction
      return 0;
```

```
Order    Family    Genus    Species
Caninae  Felidae   Panthera  Panthera pardus
Canidae  Mustelidae Taxidea  taxus
          Lutrae    Lutra lutra
          Canidae   Canis latrans
                      Canis lupus
```

```
0   x   o   x
X tries to maximize score
-1  x   o   x
-1  o   o   o   o   o   o   -1
O tries to minimize score
```

```
Automaton
Example Inputs
```

```
100
19
35
17
3
25
1
2
7
```
A **rooted tree** is a connected directed acyclic graph in which one vertex has been designated the root, which has no incoming edges, and every other vertex has exactly one incoming edge.

Rosen p. 747-749
(Rooted) Trees: definitions

A **rooted tree** is a connected directed acyclic graph in which one vertex has been designated the root, which has no incoming edges, and every other vertex has exactly one incoming edge.

Special case of DAGs from last class. Note that each vertex in middle has *exactly one* incoming edge from layer above. Edges are directed *away from* the root.

Rosen p. 747-749
Which of the following directed graphs are trees (with root indicated in green)?

A. 

B. 

C. 

D.
(Rooted) Trees: definitions

Rosen p. 747-749
If vertex $v$ is not the root, it has exactly one incoming edge, which is from its parent, $p(v)$.

**Height** of vertex $v$ is given by the recurrence:

$$h(v) = h(p(v)) + 1 \quad \text{if } v \text{ is not the root, and }$$

$$h(r) = 0$$
(Rooted) Trees: definitions

Height of vertex \( v \): \( h(v) = h(p(v)) + 1 \) \( \text{if v is not the root, and} \)
\( h(r) = 0 \)

What is the height of the red vertex?

A. 0  
B. 1  
C. 2  
D. 3  
E. None of the above.
(Rooted) Trees: definitions

**Height** of vertex $v$: $h(v) = h(p(v)) + 1$ \textit{if $v$ is not the root, and} $h(r) = 0$

**Height** of tree is maximum height of a vertex in the tree.

*Rosen p. 753*
A binary tree is a rooted tree where every (internal) vertex has no more than 2 children.

How many leaves does a binary tree of height 3 have?

A. 2  
B. 3  
C. 6  
D. 8  
E. None of the above.
A binary tree is a rooted tree where every (internal) vertex has no more than 2 children.

How many leaves does a binary tree of height 3 have?
A. 2
B. 3
C. 6
D. 8
E. None of the above.

*See Theorem 5 for proof of upper bound*
A full binary tree is a rooted tree where every internal vertex has exactly 2 children.

Which of the following are full binary trees?

A. 

B. 

C. 

D.
A full binary tree is a rooted tree where every internal vertex has exactly 2 children.

**At most** how many vertices are there in a full binary tree of height $h$?

A. $\Theta(h)$  
B. $\Theta(2^h)$  
C. $\Theta(h^2)$  
D. $\Theta(\log h)$

Max number of vertices when tree is balanced
A **full** binary tree is a rooted tree where every internal vertex has exactly 2 children.

*Key insight: number of vertices doubles on each level.*

\[
1 + 2 + 4 + 8 + \ldots + 2^h = 2^{h+1} - 1 \quad \text{i.e.} \quad \Theta(2^h)
\]

If \( n \) is number of vertices:

\[
n = 2^{h+1} - 1
\]

so

\[
h = \log(n+1) - 1 \quad \text{i.e.} \quad \Theta(\log n)
\]
Relating height and number of vertices:

$log(n+1) - 1 \leq h \leq ___$

This is what we just proved.

How do we prove?

What tree with $n$ vertices has the greatest possible height?
Relating height and number of vertices:

log(n+1) – 1 \leq h \leq n-1

This is what we just proved.

How do we prove?

What tree with n vertices has the greatest possible height?
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

**Implementation**

Each vertex is an object with the fields:

- `p = parent`
- `lc = left child`
- `rc = right child`
- `value`

When is `p` null?

- A. If we have an error in our implementation.
- B. When the value is 0.
- C. When the vertex is a leaf node.
- D. When the vertex is the root node.
- E. None of the above.
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

**Implementation**

Each vertex is an object with the fields:

- `p` = parent
- `lc` = left child
- `rc` = right child
- `value`

When is `lc` null?

- A. If we have an error in our implementation.
- B. When the value is 0.
- C. When the vertex is a leaf node.
- D. When the vertex is the root node.
- E. None of the above.
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

For each vertex $v$:
- If $x$ is in subtree rooted at $lc(v)$, $\text{value}(x) \leq \text{value}(v)$.
- If $x$ is in the subtree rooted at $rc(v)$, $\text{value}(x) \geq \text{value}(v)$.
Binary Search Trees

• Facilitate binary search (must **maintain sorted order** of data)
• Dynamic

How would you search for "orange?"
Binary Search Trees

- Facilitate binary search (must **maintain sorted order** of data)
- Dynamic

To search for target T in a binary search tree.

1. Compare T to value(r) where r is the root.
2. If T = value(r), done 😊.
3. If T < value(r), search recursively starting at lc(r).
4. If T > value(r), search recursively starting at rc(r).
Binary Search Trees

• Facilitate binary search (must maintain sorted order of data)
• Dynamic

To search for target T in a binary search tree.

1. Compare T to value(r) where r is the root.
2. If T = value(r), done 😊.
3. If T < value(r), search recursively starting at lc(r).
4. If T > value(r), search recursively starting at rc(r).

How long does this take?
Binary Search Trees

• Facilitate binary search (must **maintain sorted order** of data)
• Dynamic

To search for target T in a binary search tree.

1. Compare T to value(r) where r is the root.
2. If T = value(r), done 😊.
3. If T < value(r), search recursively starting at lc(r).
4. If T > value(r), search recursively starting at rc(r).

**How long does this take?**

Constant time at each level. Number of levels is height+1.
Binary Search Trees

- Facilitate binary search (must **maintain sorted order** of data)
- Dynamic

To search for target T in a binary search tree.

1. Compare T to value(r) where r is the root.
2. If T = value(r), done 😊.
3. If T < value(r), search recursively starting at lc(r).
4. If T > value(r), search recursively starting at rc(r).

How long does this take? Time proportional to height!
An unrooted tree is a connected undirected graph with no cycles.
Theorem: An undirected graph is an unrooted tree if and only if it contains all the edges of some rooted tree.

What does this mean?

(1) If we replace all directed edges in a rooted tree with undirected edges, the result will be an unrooted tree.

(2) There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.
Equivalence between rooted and unrooted trees

**Goal (1):** If we replace all directed edges in a rooted tree with undirected edges, the result will be an unrooted tree.

**What do we need to prove?**

A. The resulting undirected graph will be connected.
B. The resulting undirected graph will be undirected.
C. The resulting undirected graph will not have cycles.
D. All of the above.
**Goal (1):** If we replace all directed edges in a rooted tree with undirected edges, the result will be an **unrooted tree**.

**SubGoal (1a):** this resulting graph is connected, i.e. between any two vertices $u$ and $v$ there is a path in the graph.

*Idea: To find path between purple and orange, follow parents of purple all the way to root, then follow its children down to orange.*
**Goal (1):** If we replace all directed edges in a rooted tree with undirected edges, the result will be an **unrooted tree**.

**SubGoal (1b):** this resulting graph has no cycles.

**Idea:** Towards a contradiction, assume there is a cycle and consider the simplest cycle (with no repeated vertices).

*Start at vertex at highest level in the cycle. Next step must go to a child node, etc. Can never go up to higher level again because vertices in rooted tree only have one incoming edge.*
Equivalence between rooted and unrooted trees

Goal (2): There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

Idea: finding right directions for edges will be similar to finding topological sort of a DAG.
Equivalence between rooted and unrooted trees

**Goal (2):** There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

**SubGoal (2a):** Any unrooted tree with at least two vertices has a vertex of degree exactly 1.

**Proof:** Towards a contradiction, assume that all n vertices have degree 0 or \( \geq 2 \). Since a tree is connected, eliminate the case of degree-0 vertices. **Goal:** construct a cycle to arrive at a contradiction.

Start at any vertex \( u_0 \).
Pick \( u_{i+1} \) so that it is adjacent to \( u_i \) but is **not** \( u_{i-1} \). **Why?**

Get \( u_0, u_1, \ldots, u_n \). By Pigeonhole Principle, must repeat. **Cycle!**
**Goal (2):** There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

**SubGoal (2b):** If $T$ is unrooted tree and $v$ has degree 1 in $T$, then $T\{v\}$ is unrooted tree.

**Proof:** To check that $T\{v\}$ is unrooted tree, confirm

* $T\{v\}$ is connected and

* $T\{v\}$ does not have a cycle.
Equivalence between rooted and unrooted trees

**Goal (2):** There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

**SubGoal (2b):** If $T$ is an unrooted tree and $v$ has degree 1 in $T$, then $T - \{v\}$ is an unrooted tree.

**Proof:** To check that $T - \{v\}$ is an unrooted tree, confirm

* $T - \{v\}$ is connected and

* $T - \{v\}$ does not have a cycle.
Goal (2): There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

Using the subgoals to achieve the goal:

Root($T$: unrooted tree with $n$ nodes)
1. If $n=1$, let the only vertex $v$ be the root, set $h(v):=0$, and return.
2. Find a vertex $v$ of degree 1 in $T$, and let $u$ be its only neighbor.
3. Root($T\{v\}$).
4. Set $p(v):=u$ and $h(v):=h(u)+1$. 

Recursion!
Counting

1, 2, 3, 4, ...
What comes next?

A. 4
B. 8
C. 5
D. 6
E. None of the above.
What do we mean by counting?

How many arrangements or combinations of objects are there of a given form?

How many of these have a certain property?
Why is counting important?

For computer scientists:

- **Hardware**: How many ways are there to arrange components on a chip?
- **Algorithms**: How long is this loop going to take? How many times does it run?
- **Security**: How many passwords are there?
- **Memory**: How many bits of memory should be allocated to store an object?
In some video games, each player can create a character with custom facial features.

How many distinct characters are possible?
In some video games, each player can create a character with custom facial features. How many distinct characters are possible?

Considering only these 12 hairstyles and 8 hair colors, how many different characters are possible?

A. \(8 + 12 = 20\)
B. \(8 \times 12 = 96\)
C. \(8^{12} = 68719476736\)
D. \(12^{8} = 429981696\)
E. None of the above
For any sets, A and B: \(|A \times B| = |A| \cdot |B|\)

In our example:

A = \{ hair styles \} \quad |A| = 12
B = \{ hair colors \} \quad |B| = 8

\(A \times B = \{ (s, c) : s \text{ is a hair style and } c \text{ is a hair color} \}\)

\(|A \times B| = \text{ the number of possible pairs of hair styles & hair colors}
\quad = \text{ the number of different ways to specify a character}\)
For any sets, $A$ and $B$: $|A \times B| = |A| \cdot |B|$

More generally:

Suppose that a procedure can be broken down into a sequence of two tasks. If there are $n_1$ ways to do the first task and for each of these ways of doing the first task, there are $n_2$ ways to do the second task, then there are $n_1 \cdot n_2$ ways to do the procedure.
For any sets, $A$ and $B$: $|A \times B| = |A| \cdot |B|$

More generally:

To count the number of pairs of objects:
* Count the number of choices for selecting the first object.
* Count the number of choices for selecting the second object.
* Multiply these two counts.

CAUTION: this will only work if the *number* of choices for the second object doesn't depend on which first object we choose.
Other than the 96 possible custom Miis, a player can choose one of 10 preset characters.

How many different characters can be chosen?

A. 96
B. 10
C. 106
D. 960
E. None of the above.
For any **disjoint** sets, A and B: $|A \cup B| = |A| + |B|$

**In our example:**

$A = \{ \text{custom characters} \} \quad |A| = 96$

$B = \{ \text{preset characters} \} \quad |B| = 10$

$A \cup B = \{ m : m \text{ is a character that is either custom or preset} \}$

$|A \cup B| = \text{the number of possible characters}$
For any disjoint sets, A and B: $|A \cup B| = |A| + |B|$

More generally:

If a task can be done either in one of $n_1$ ways or in one of $n_2$ ways, where none of the set of $n_1$ ways is the same as any of the set of $n_2$ ways, then there are $n_1 + n_2$ ways to do the task.
For any **disjoint** sets, A and B: \(|A \cup B| = |A| + |B|\)

**More generally:**

To count the number of objects with a given property:
* Divide the set of objects into mutually exclusive (disjoint/nonoverlapping) groups.
* Count each group separately.
* Add up these counts.
Select which method lets us count the number of length n binary strings.

A. The product rule.
B. The sum rule.
C. Either rule works.
D. Neither rule works.
Select which method lets us count the number of length n binary strings.

A. The product rule. Select first bit, then second, then third …
B. The sum rule. \{0\ldots\} \cup \{1\ldots\} gives recurrence \(N(n) = 2N(n-1)\), \(N(0)=1\)
C. Either rule works.
D. Neither rule works.
Memory: storing length $n$ binary strings

How many binary strings of length $n$ are there?

How many bits does it take to store a length $n$ binary string?
Memory: storing length $n$ binary strings

How many binary strings of length $n$ are there? $2^n$

How many bits does it take to store a length $n$ binary string? $n$

**General principle:** number of bits to store an object is

$$\left\lceil \log_2(\text{number of objects}) \right\rceil$$

Why the ceiling function?
**Scenario:** We want to store a user’s state of residence in our database. How many bits of memory do we need to allocate?

A. 2  
B. 5  
C. 6  
D. 10  
E. 50
Scenario: We want to store a non-negative integer that has at most n digits. How many bits of memory do we need to allocate?

A. n
B. $2^n$
C. $10^n$
D. $n \cdot \log_{10} 10$
E. $n \cdot \log_{10} 2$
At an ice cream parlor, you can choose to have your ice cream in a bowl, cake cone, or sugar cone. There are 20 different flavors available.

How many single-scoop creations are possible?

A. 20  
B. 23  
C. 60  
D. 120  
E. None of the above.
At an ice cream parlor, you can choose to have your ice cream in a bowl, cake cone, or sugar cone. There are 20 different flavors available.

You can convert your single-scoop of ice cream to a sundae. Sundaes come with your choice of caramel or hot-fudge. Whipped cream and a cherry are options. How many desserts are possible?

A. $20 \cdot 3 \cdot 2 \cdot 2$
B. $20 \cdot 3 \cdot 2 \cdot 2 \cdot 2$
C. $20 \cdot 3 + 20 \cdot 3 \cdot 2 \cdot 2$
D. $20 \cdot 3 + 20 \cdot 3 \cdot 2 \cdot 2 \cdot 2$
E. None of the above.
A scheduling problem

In one request, four jobs arrive to a server: J1, J2, J3, J4.

The server starts each job right away, splitting resources among all active ones.

Different jobs take different amounts of time to finish.

**How many possible finishing orders are there?**

A. $4^4$
B. $4 + 4 + 4 + 4$
C. $4 \times 4$
D. None of the above.
A scheduling problem

In one request, four jobs arrive to a server: J1, J2, J3, J4.

The server starts each job right away, splitting resources among all active ones.

Different jobs take different amounts of time to finish.

How many possible finishing orders are there?

Product rule analysis
• 4 options for which job finishes first.
• Once pick that job, 3 options for which job finishes second.
• Once pick those two, 2 options for which job finishes third.
• Once pick first three jobs, only 1 remains.

Which options are available will depend on first choice; but the number of options will be the same.

(4)(3)(2)(1) = 4! = 24
Permutation:

rearrangement / ordering of n distinct objects so that each object appears exactly once

Theorem 1: The number of permutations of n objects is

\[ n! = n(n-1)(n-2) \ldots (3)(2)(1) \]

Convention: 0! = 1

Rosen p. 407
Planning a trip to

New York
Chicago
Baltimore
Los Angeles
San Diego
Minneapolis
Seattle

Must start in New York and end in Seattle.

How many ways can the trip be arranged?

A. $7!$
B. $2^{7}$
C. None of the above.
Traveling salesperson

Planning a trip to

New York
Chicago
Baltimore
Los Angeles
San Diego
Minneapolis
Seattle

Must start in New York and end in Seattle.
Must also visit Los Angeles immediately after San Diego.

How many ways can the trip be arranged now?
Traveling salesperson

Planning a trip to

New York
Chicago
Baltimore
Los Angeles
San Diego
Minneapolis
Seattle

Must start in New York and end in Seattle.
Must also visit Los Angeles immediately after San Diego.

How many ways can the trip be arranged now?

Treat LA & SD as a single stop.

(1)(4!)(1) = 24 arrangements.
Traveling salesperson

Planning a trip to

New York
Chicago
Baltimore
Los Angeles
San Diego
Minneapolis
Seattle

Must start in New York and end in Seattle.
Must also visit Los Angeles and San Diego immediately after each other (in any order).

How many ways can the trip be arranged now?
Planning a trip to

New York
Chicago
Baltimore
Los Angeles
San Diego
Minneapolis
Seattle

Must **start in New York** and **end in Seattle**.
Must also **visit Los Angeles** and **San Diego** immediately after each other (in any order).

*How many ways can the trip be arranged now?*

Break into two disjoint cases:

Case 1: LA before SD  
24 arrangements

Case 2: SD before LA  
24 arrangements
Traveling salesperson

Planning a trip to

New York  Chicago  Baltimore  Los Angeles  San Diego  Minneapolis  Seattle

Must start in New York and end in Seattle. Must also visit Los Angeles and San Diego immediately after each other (in any order).

How many ways can the trip be arranged now?

Realistically, choose order of visiting cities based on distance... we wouldn't go to Los Angeles, then Minneapolis, then San Diego, then New York, then Seattle, then Chicago, etc.
Is there an order of visiting the cities that stops at each city exactly once and minimizes the distance traveled?

<table>
<thead>
<tr>
<th></th>
<th>NY</th>
<th>Chicago</th>
<th>Balt.</th>
<th>LA</th>
<th>SD</th>
<th>Minn.</th>
<th>Seattle</th>
</tr>
</thead>
<tbody>
<tr>
<td>NY</td>
<td>0</td>
<td>800</td>
<td>200</td>
<td>2800</td>
<td>2800</td>
<td>1200</td>
<td>2900</td>
</tr>
<tr>
<td>Chicago</td>
<td>800</td>
<td>0</td>
<td>700</td>
<td>2000</td>
<td>2100</td>
<td>400</td>
<td>2000</td>
</tr>
<tr>
<td>Balt.</td>
<td>200</td>
<td>700</td>
<td>0</td>
<td>2600</td>
<td>2600</td>
<td>1100</td>
<td>2700</td>
</tr>
<tr>
<td>LA</td>
<td>2800</td>
<td>2000</td>
<td>2600</td>
<td>0</td>
<td>100</td>
<td>1900</td>
<td>1100</td>
</tr>
<tr>
<td>SD</td>
<td>2800</td>
<td>2100</td>
<td>2600</td>
<td>100</td>
<td>0</td>
<td>2000</td>
<td>1300</td>
</tr>
<tr>
<td>Minn.</td>
<td>1200</td>
<td>400</td>
<td>1100</td>
<td>1900</td>
<td>2000</td>
<td>0</td>
<td>1700</td>
</tr>
</tbody>
</table>
Traveling salesperson

Planning a trip to

New York
Chicago
Baltimore
Los Angeles
San Diego
Minneapolis
Seattle

Is there an order of visiting the cities that stops at each city exactly once and minimizes the distance traveled?

Want a Hamiltonian tour
Developing an algorithm which, given a set of cities and distances between them, computes a shortest distance path between all of them is **NP-hard** (considered intractable, very hard).

**Is there any algorithm for this question?**

A. No, it's not possible.
B. Yes, it's just very slow.
C. ?
Traveling salesperson

Exhaustive search algorithm

List all possible orderings of the cities.
For each ordering, compute the distance traveled.
Choose the ordering with minimum distance.

How long does this take?

Want a Hamiltonian tour
Traveling salesperson

Exhaustive search algorithm: given $n$ cities and distances between them.

List all possible orderings of the cities. For each ordering, compute the distance traveled. Choose the ordering with minimum distance. $O(\text{number of orderings})$

How long does this take?

Want a Hamiltonian tour
Traveling salesperson

Exhaustive search algorithm: given \( n \) cities and distances between them.

List all possible orderings of the cities. For each ordering, compute the distance traveled. Choose the ordering with minimum distance. \( \text{O(number of orderings)} \)

How long does this take?

A. \( O(n) \)
B. \( O(n^2) \)
C. \( O(n^n) \)
D. \( O(n!) \)
E. None of the above.
Traveling salesperson

Exhaustive search algorithm: given \( n \) cities and distances between them.

List all possible orderings of the cities.
For each ordering, compute the distance traveled.
Choose the ordering with minimum distance.

How long does this take?

A. \( \mathcal{O}(n) \)
B. \( \mathcal{O}(n^2) \)
C. \( \mathcal{O}(n^n) \)
D. \( \mathcal{O}(n!) \)
E. None of the above.

\( 2^n < n! < n^n \) for large \( n \)

Moral: counting gives upper bound on algorithm runtime.
Announcements

HW5 Due
Tuesday
10pm

Office Hours

Mine are today 10-11, Saturday 1-3, CSE 4204.

Lots more on the course calendar!