CSE 250
Lecture 37

Final Review

Day 1

Logarithms

Logarithms (refresher)

- Let $a, b, c, n > 0$
- Exponent rule: $\log(n^a) = a\log(n)$
- Product rule: $\log(an) = \log(a) + \log(n)$
- Division rule: $\log(\frac{n}{a}) = \log(n) - \log(a)$
- Change of base from b to c: $\log_b(n) = \frac{\log_c(n)}{\log_c(b)}$
  - Base changes are only a constant factor off
- Log/Exponent are inverses: $b^{\log_b(n)} = \log_b(b^n) = n$

Asymptotic Analysis

Growth Functions

A growth function must be a non-decreasing function of the form

$$f : \mathbb{Z}^+ \cup \{0\} \rightarrow \mathbb{R}^+$$

f is a function from ...

… to …

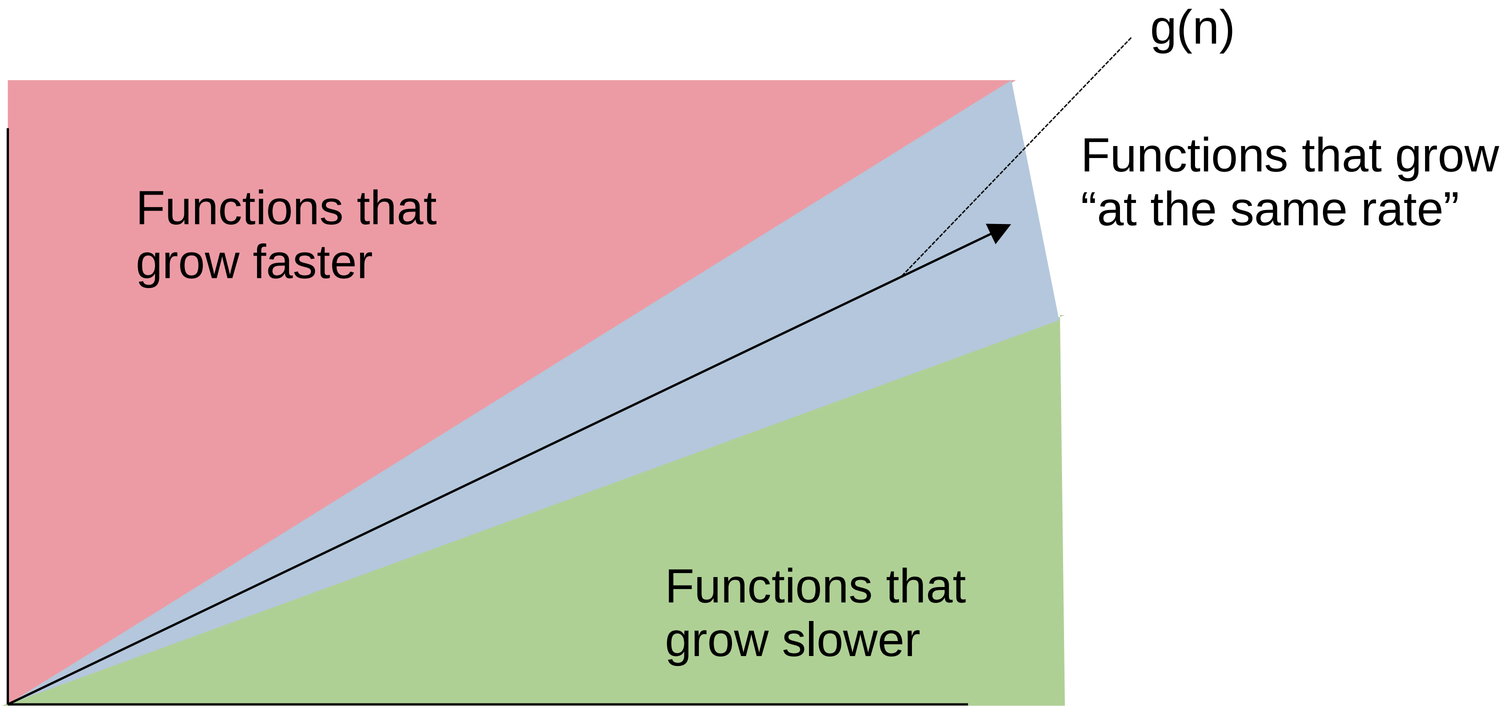
$$\mathbb{Z}^+ \cup \{0\} = \{0, 1, 2, 3, \ldots\}$$

(non-negative integers)

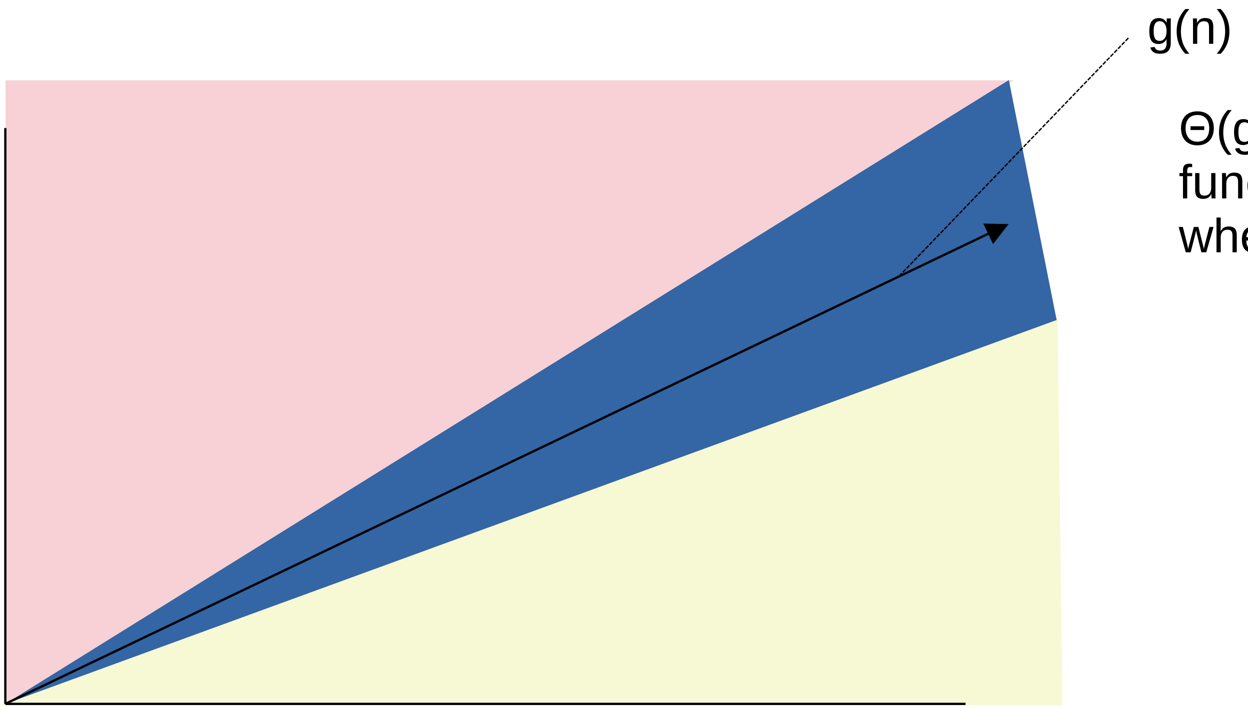
$$\mathbb{R}^+ = \{ \, x \mid x \in \mathbb{R}, x > 0 \, \}$$

(positive real numbers)

Classify Functions by their Scaling

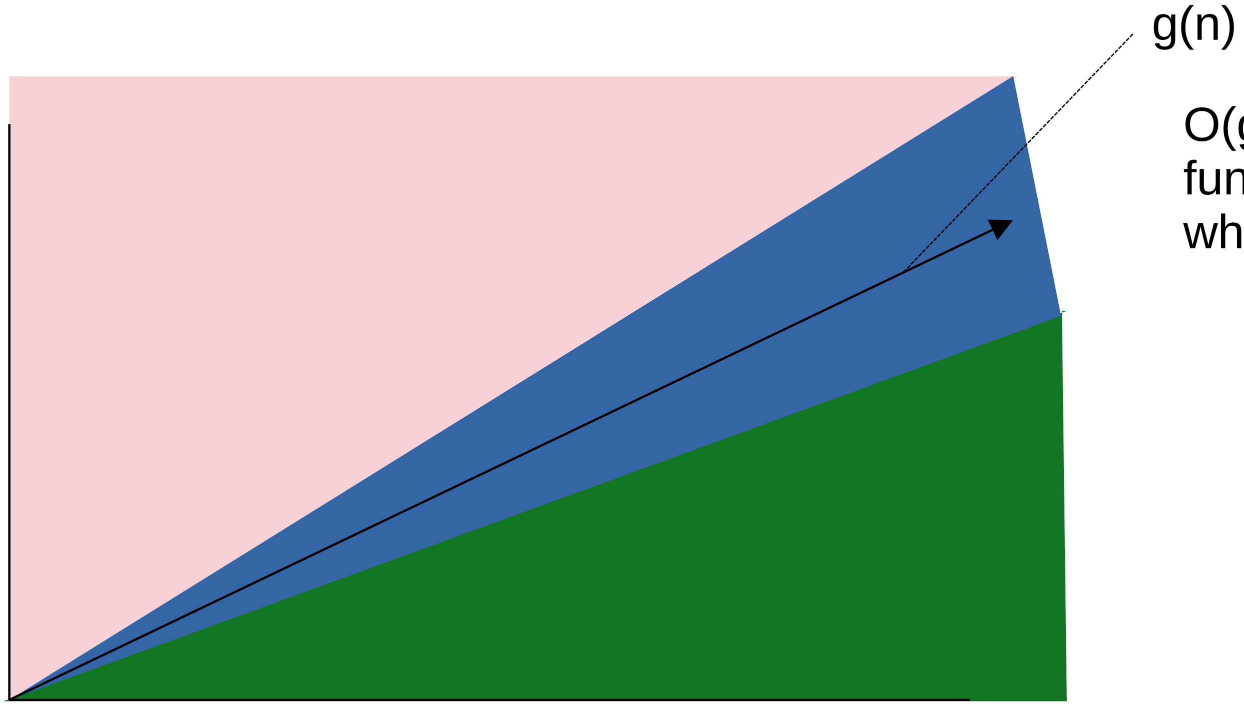


Big-Θ



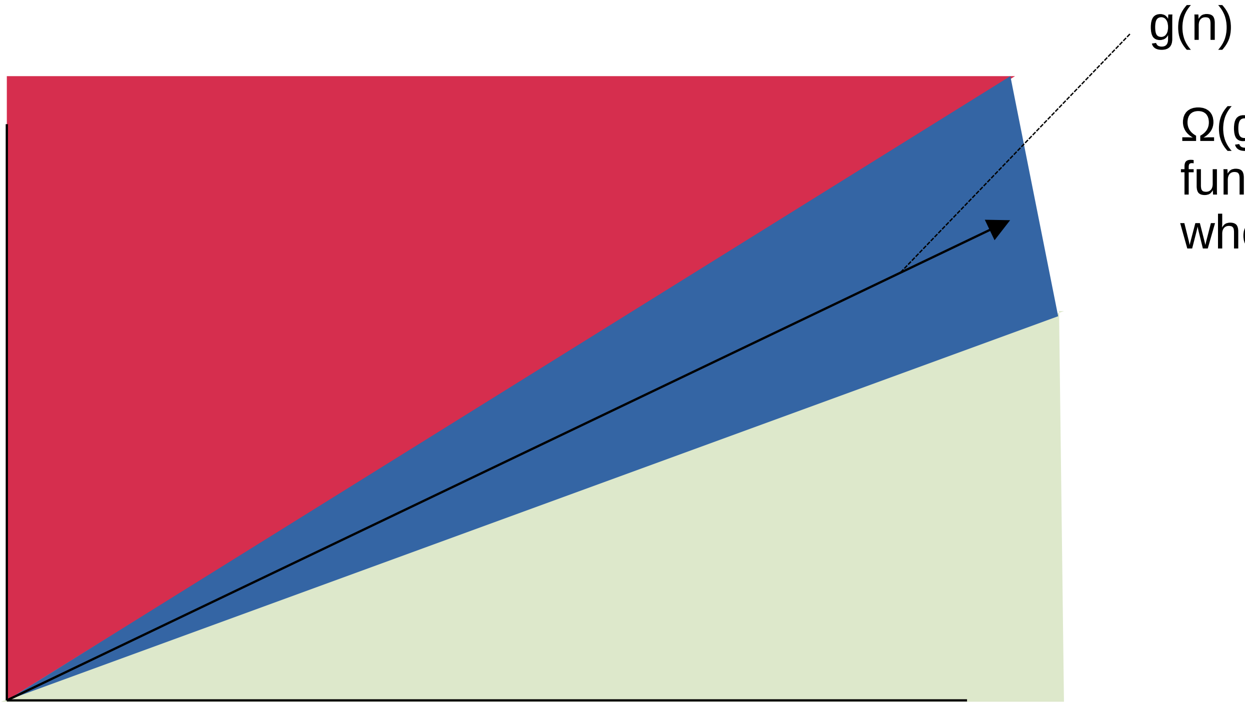
Θ(g) is the set of functions where f “=” g

Big-O



O(g) is the set of functions
where f “≤” g

Big-Ω



Ω(g) is the set of functions
where f "≥" g

Types of Bounds

- [no qualifier] **Runtime**: The guaranteed runtime of the function
  - O(g(n)): The algorithm never runs slower than c·g(n)
  - Ω(g(n)): The algorithm never runs faster than c·g(n)
  - Θ(g(n)): The algorithm always runs within [a·g(n), b·g(n)]
- **Amortized Runtime**: Guaranteed per-call runtime over n calls
  - O(g(n)): n invocations of the algorithm take at most c·n·g(n)
- **Expected Runtime**: 'Typical' runtime without guarantees
  - O(g(n)): The algorithm usually takes no more than c·g(n)
    - … but it's random, it could take longer if you're unlucky.

Runtime Terminology

- "Worst-case" runtime
  - The O() runtime of the function
- "Tight" runtime
  - A bound (O or Ω) with no better bound of the same type.
    - Remember that $n = O(n^2)$ *(although it's not tight)*
  - A Θ bound is always tight.

Big-O

- Big-O (big oh) is an upper-bound on functions

  for any two functions $f, g : \mathbb{Z}^+ \cup \{0\} \to \mathbb{R}^+$

$$f(n) \in O(g(n)) \text{ iff } \exists c, n_0 \text{ s.t. } \forall n \geq n_0, f(n) \leq cg(n)$$

$f(n)$ is lower than $g(n)$ scaled by c

$O(g(n))$ is a set of functions

… where for every n bigger than n_0

and $f(n)$ is in it if (and only if)

There's some constant c and some "low" n value n_0

Big-Ω

- Big-Ω (big omega) is a lower-bound on functions

  for any two functions $f, g : \mathbb{Z}^+ \cup \{0\} \rightarrow \mathbb{R}^+$

$$f(n) \in \Omega(g(n)) \text{ iff } \exists c, n_0 \text{ s.t. } \forall n \geq n_0, f(n) \geq cg(n)$$

$\Omega(g(n))$ is a set of functions

and $f(n)$ is in it if (and only if)

There's some constant c
and some "low" n value n_0

… where for every n
bigger than n_0

$f(n)$ is greater than $g(n)$
scaled by c

Big-Θ

- Big-Θ (big theta) is a joint bound on functions

  for any two functions $f, g : \mathbb{Z}^+ \cup \{0\} \rightarrow \mathbb{R}^+$

$$f(n) \in \Theta(g(n)) \text{ iff } [f(n) \in O(g(n))] \wedge [f(n) \in \Omega(g(n))]$$

$\Theta(g(n))$ is a set of functions

and $f(n)$ is in it if (and only if)

$f(n)$ is upper-bounded by $g(n)$

and $f(n)$ is also lower-bounded by $g(n)$

Dominant Terms

$$\text{exponential} \gg \text{polynomial} \gg \log \gg \text{constant}$$

Common Runtimes

- **Constant Time**: $\Theta(1)$
  - e.g., $T(n) = c$ (for some constant $c > 0$)
- **Logarithmic Time**: $\Theta(\log(n))$
  - e.g., $T(n) = c\log(n)$ (for some constant $c > 0$)
- **Linear Time**: $\Theta(n)$
  - e.g., $T(n) = c_1 n + c_0$ (for some constants c_1, c_0 where $c_1 > 0$)
- **Quadratic Time**: $\Theta(n^2)$
  - e.g., $T(n) = c_2 n^2 + c_1 n + c_0$
- **Polynomial Time**: $\Theta(n^k)$ (for some $k \in \mathbb{Z}^+$)
  - e.g., $T(n) = c_k n^k + \ldots + c_2 n^2 + c_1 n + c_0$
- **Exponential Time**: $\Theta(c^n)$ (for some $c > 0$)

Indexing into a Linked List

- Runtime to retrieve the **i**th element is linear in **i**
  - $O(i)$ is a tight bound: $i \leq O(i)$
    - $O(i^2)$ is a bound; $i \leq O(i^2)$ (but not a tight one)
  - $\Omega(i)$ is a tight bound: $i \geq \Omega(i)$
  - Since the runtime is $O(i)$ and $\Omega(i)$, it is also $\Theta(i)$

Appending to an ArrayBuffer

- Runtime is either constant [typical case] **OR** linear [if resizing]
  - $O(n)$ is a tight bound: $1 \leq O(n)$, $n \leq O(n)$
  - $\Omega(1)$ is a tight bound: $1 \geq \Omega(1)$, $n \geq \Omega(1)$
  - There is no Θ bound (the tight O bound $\neq$ the tight Ω bound)
- Runtime of n appends is provably $O(n)$ (and $\Theta(n)$, $\Omega(n)$)
  - Amortized runtime of $^{O(n)}/_{n} = O(1)$

Θ(i)

- **Observation**
  - The only time when tight bounds O(f) ≠ Ω(f) is when f is
    - ...defined by cases.
      - as in appending to an array buffer
    - ...has variable runtimes
      - e.g., indexing into a linked list is O(n), but Θ(i)

Quick Sort

- Each level of splits takes O(n) total runtime
  - Typically, each split will cut the input array in (nearly) half
    - Will need log(n) levels of splits
  - **No guarantees**: Unlikely, but might accidentally always pick the lowest value as a pivot for each split.
    - Might need as many as n levels of splits
  - **Runtime**: $O(n^2)$
  - **Expected Runtime**: $O(n \cdot \log(n))$

Sequences

Immutable Sequence ADTs

- `apply(`**`idx`**`: Int): A`
  - Get the element (of type A) at position **`idx`**.
- `iterator: Iterator[A]`
  - Get access to <u>view</u> all elements in the seq, in order, once.
- `length: Int`
  - Count the number of elements in the seq.

Mutable Sequence ADTs

- `apply(`**`idx`**`: Int): A`
  - Get the element (of type A) at position **`idx`**.
- `iterator: Iterator[A]`
  - Get access to <u>view</u> all elements in the seq, in order, once.
- `length: Int`
  - Count the number of elements in the seq.
- `insert(`**`idx`**`: Int, `**`elem`**`: A): Unit`
  - Insert the element at position **`idx`** with the value **`elem`**.
- `remove(`**`idx`**`: Int): Unit`
  - Remove the element at position **`idx`**.

Runtime Cost for Appends

- $T(n)$ = insert cost + reserve cost = $\Theta(n) + \Theta(n) = \Theta(n)$
- Append runtime is **Amortized** $O(1)$
  - Runtime for <u>one</u> append is $O(n)$
  - Runtime for <u>n</u> appends is $\Theta(n)$
- "Amortized" describes runtime over the long run.
  - reserve is only called log(n) times (very infrequently)
  - Not quite the same as the "average" case
    - Average case is the <u>expected runtime</u> over any input
    - Here, $\Theta(n)$ <u>is the runtime</u>.

Amortized → Upfront costs paid off over time

Overview

Function	Array	LL by Index	LL by Pointer
apply	Θ(1)	Θ(i)	Θ(1)
update	Θ(1)	Θ(i)	Θ(1)
insert	O(n)	Θ(i)	Θ(1)
remove	O(n)	Θ(i)	Θ(1)
append	Amortized O(1)	Θ(1)	Θ(1)

Bubble Sort for Mutable Sequences

```
def sort(seq: mutable.Seq[Int]): Unit =
{
  val n = seq.length
  for(i ← n – 2 to 0 by -1; j ← i to n)
  {
    if(seq(j+1) < seq(j))
    {
      val temp = seq(j+1)
      seq(j+1) = seq(j)
      seq(j) = temp
    }
  }
}
```

Is the runtime $T(n) = \Theta(n^2)$?

- **What is the cost of seq(j+1) < seq(j)?**
- **What is the cost of each seq(k)?**

Bubble Sort for Immutable Sequences

```
def sort(seq: Seq[Int]): Seq[Int] =
{
  val newSeq = seq.toArray
  val n = seq.length
  for(i ← n – 2 to 0 by -1; j ← 0 to i)
  {
    if(newSeq(j+1) < newSeq(j))
    {
      val temp = seq(j+1)
      seq(j+1) = seq(j)
      seq(j) = temp
    }
  }
  return newSeq.toList
}
```

Is the runtime $T(n) = \Theta(n^2)$?

- **What is the cost of seq.toArray?**
- **What is the cost of newSeq.toList?**

Searching Sequences

```
def indexOf[T](seq: Seq[T], value: T, from: Int): Int = {
  for(i ← from 0 until seq.length) {
    if(seq(i).equals(value)) { return i }
  }
  return -1
}
```

Expected runtime is T(n) = Θ(n)

```
def count[T](seq: Seq[T], value: T): Int = {
  var count = 0; var i = indexOf(seq, value, 0)
  while(i != -1) {
    count += 1; indexOf(seq, value, i+1)
  }
  return count
}
```

Expected runtime is T(n) = Θ(n)

Recursion

Fibonacci Sequence Runtime

The runtime of a recursive function is easiest to represent with a recurrence relation

```
def fib(n: Int) = {
    if(n == 0 || n == 1) { n }
    else { fib(n-1) + fib(n-2) }
}
```

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & \textbf{otherwise} \end{cases}$$

(this specific recurrence has a closed form, but ask on Piazza)

Factorial

```
def fact(n: Int): Long = {
    if(n <= 0) { 1 }
   else { n * fact(n-1) }
}
```

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n \leq 0 \\ T(n-1) + \Theta(1) & \textbf{otherwise} \end{cases}$$

What is the closed form?

How much space is used?

Tail-Recursive Factorial

```
def fact(n: Int): Long = {
   if(n <= 0) { 1 }
   else { n * fact(n-1) }
}
```

```
def fact(n: Int): Long = {
   var total = 1l
   for(i ← 1 to n) {
      total *= i
   }
   return total
}
```

The compiler can (sometimes) figure this out on its own!

Divide and Conquer

- Recursive Solutions
  - Solve a problem building from solution(s) to smaller versions of the same problem.
- The Divide and Conquer Strategy
  - **Divide** problem into smaller subproblem(s)
  - **Conquer** subproblem(s) by solving recursively
  - **Combine** solutions to subproblem(s) into final solution

Divide and Conquer

- Towers of Hanoi
  - n = 1: Move disk directly
  - n > 1: Solve n-1 subproblem 2 times (Conquer)
- Factorial
  - n = 0: 1
  - n >0:
    - Compute (n-1)! (Conquer)
    - Multiply by n (Merge)

No real “divide” step in any of these examples.

Merge Sort

- If the sequence has 1 or 0 values: Done!
- If n > 1
  - Divide: “Split” the sequence in half
  - Conquer: Sort each half independently
  - Combine: Merge halves together

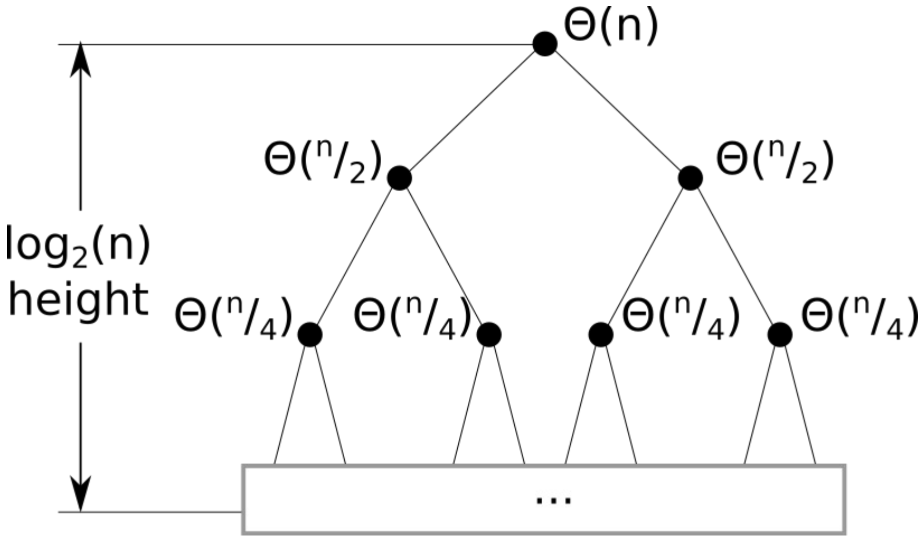
Merge Sort Analysis

- Suppose data is a sequence of size n
  - Assume n is a power of 2 to simplify analysis
- Divide: “Split” the sequence in half   D(n) = Θ(n)
- Conquer: Sort left and right halves   a = 2, b = 2, c = 1
- Combine: Merge sorted halves together   C(n) = Θ(n)

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n) + \Theta(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n) & \textbf{otherwise} \end{cases}$$

Merge Sort: Recursion Tree

There are $\log(n)$ levels in the tree



At level i, there are 2^i tasks, each with runtime $\Theta\left(\frac{n}{2^i}\right)$

$$T(N) = \sum_{i=0}^{\log(n)} \sum_{j=1}^{2^i} \Theta\left(\frac{n}{2^i}\right)$$

$$= \sum_{i=0}^{\log(n)} (2^i - 1 + 1)\Theta\left(\frac{n}{2^i}\right)$$

$$= \sum_{i=0}^{\log(n)} 2^i \Theta\left(\frac{n}{2^i}\right)$$

$$= \sum_{i=0}^{\log(n)} \Theta(n)$$

$$= (\log(n) - 0 + 1)\Theta(n)$$

$$= \Theta(n)\log(n) + \Theta(n)$$

$$= \Theta(n \log(n))$$

Merge Sort: Inductive Analysis

- Base Case: n = 1

$$T(n) = \Theta(1) = c'$$

- True for any $n_0 > 1$, $c > c'$

Merge Sort: Inductive Analysis

- Inductive step for step n > 1: assume for all m < n
  - $T(m) = c \cdot m \log(m)$
- Now use that to show $T(n) = c \cdot n \log(n)$

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + \Theta(n) \\
&\leq 2(c\frac{n}{2}\log(\frac{n}{2})) + \Theta(n) \\
&= cn\log(n) - cn\log(2) + \Theta(n) \\
&\leq cn\log(n) - cn + \Theta(n) \\
&= cn\log(n) - cn + dn \text{ (for some constant } d > 0) \\
&\leq cn\log(n) \text{ (as long as } c \geq d)
\end{aligned}
$$

Stacks and Queues

Stacks vs Queues

Stack

- `push(item)`
  - Insert at end of list
- `pop`
  - Remove from **end** of list
- `top`
  - Retrieve **end** of list

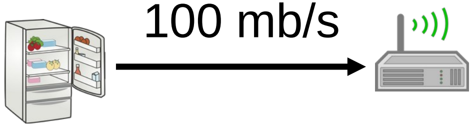
Queue

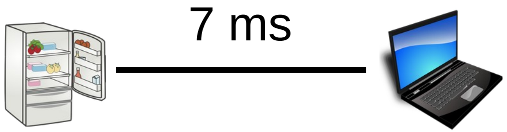
- `enqueue(item)`
  - Insert at end of list
- `dequeue`
  - Remove from **front** of list
- `front`
  - Retrieve **front** of list

Graphs

Edge Types

- Directed Edge
  - Ordered pair of vertices (u, v)
  - origin (u) → destination (v)
  - e.g., transmit bandwidth
- Undirected Edge
  - Unordered pair of vertices (u, v)
  - e.g., round-trip latency
- Directed Graph: All edges are directed
- Undirected Graph: All edges are undirected

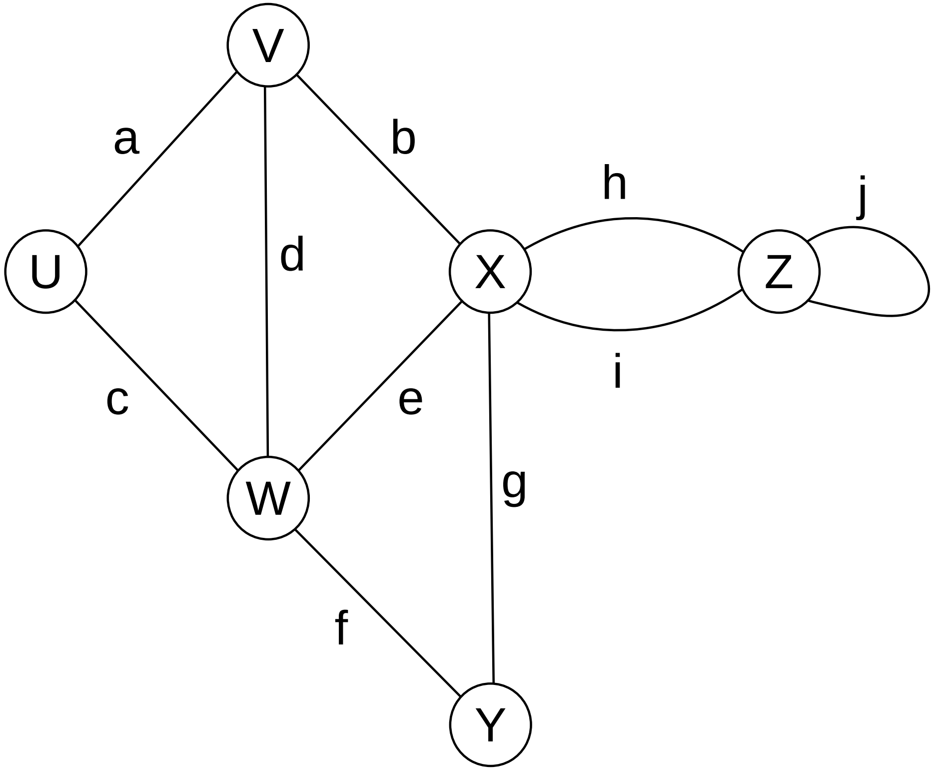




openclipart.org

Terminology

- **Endpoints** (end-vertices) of an edge
  - U, V are the endpoints of a
- Edges **incident** on a vertex
  - a, b, d are incident on V
- **Adjacent** Vertices
  - U, V are adjacent
- **Degree** of a vertex (# of incident edges)
  - X has degree 5
- **Parallel Edges**
  - h, i are parallel
- **Self-Loop**
  - j is a self-loop
- **Simple Graph**
  - A graph without parallel edges or self-loops



Edge List Summary

- addEdge, addVertex: **O(1)**
- removeEdge: **O(1)**
- removeVertex: **O(1) + O(vertex.incidentEdges)**
- vertex.outEdges, vertex.inEdges, vertex.incidentEdges: **O(m)**
  - (total cost to visit all out/in/incident edges)
- vertex.edgeTo: **O(m)**
- **Space Used**: **O(n+m)**

Add an Adjacency List

```
class DirectedGraphV3[LV, LE]
{
  def addEdge(orig: Vertex, dest: Vertex, label: LE): Edge =
  {
    val edge = new Edge(label)
    edge._listNode = edges.append(edge)
    orig._outEdges.append(edge)
    dest._inEdges.append(edge)
    return edge
  }
  class Vertex(_label: LV){
    val _outEdges: LinkedList[Edge]
    val _inEdges: LinkedList[Edge]
    // …
  }
}
```

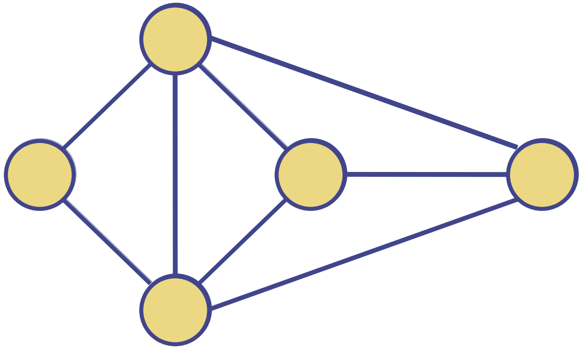
Adjacency List Summary

- addEdge, addVertex: **O(1)**
- removeEdge: **O(1)**
- removeVertex: **O(deg(vertex))**
- vertex.outEdges: **O(|outEdges|)** to visit all outEdges
  - Same for vertex.inEdges, vertex.incidentEdges
- vertex.edgeTo: **O(|outEdges|)**
- **Space Used**: **O(n+m)**

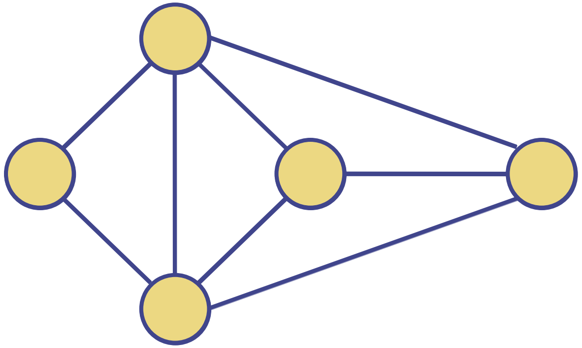
A few more terms...

- A subgraph **S** of a graph **G** is a graph where
  - **S**'s vertices are a subset of **G**'s vertices
  - **S**'s edges are a subset of **G**'s edges
- A spanning subgraph of **G** is a subgraph that contains all of **G**'s vertices

Subgraph



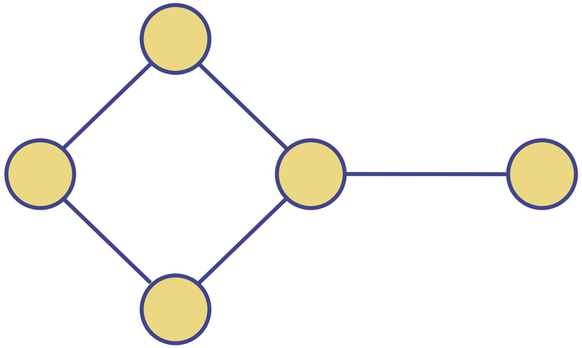
Spanning Subgraph



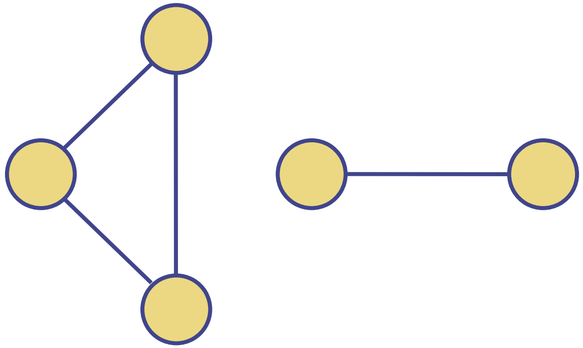
A few more terms…

- A graph is connected if there is a path between every pair of vertices.
- A connected component is a maximal connected subgraph of **G**.
  - Maximal means you can't add any new vertex without breaking the property.
  - Any subset of **G**'s edges that connects the subgraph is fine.

Connected Graph



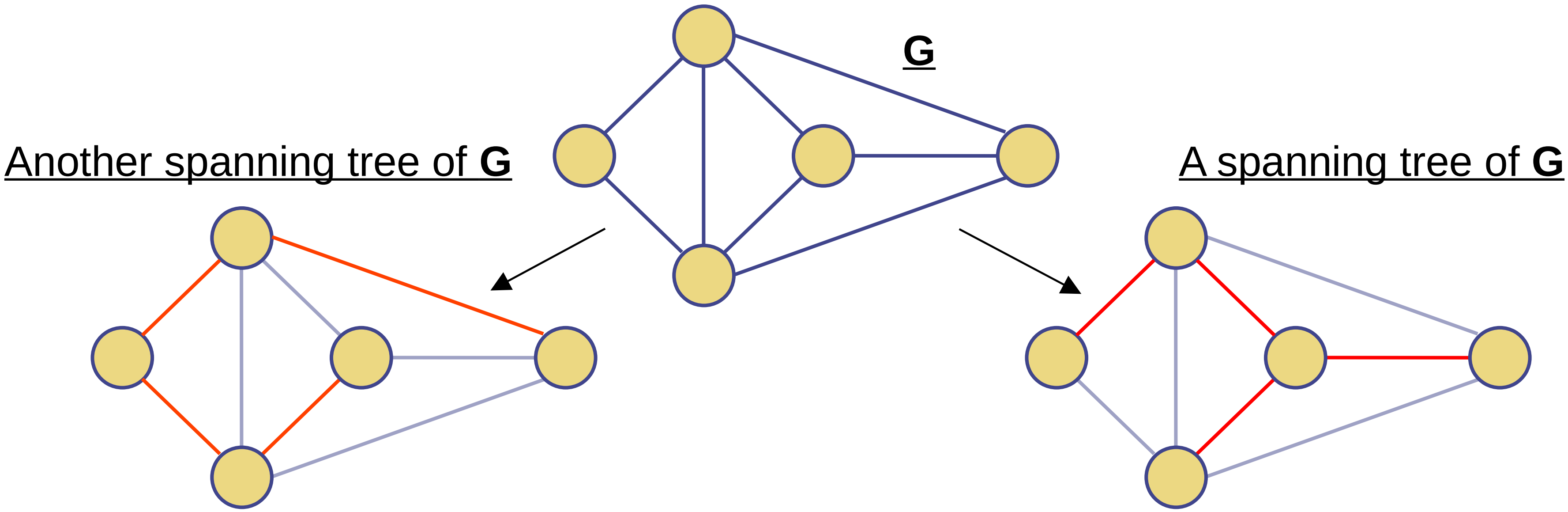
Disconnected Graph



(2 connected components)

A few more terms...

- A spanning tree of a connected graph is a spanning subgraph that is a tree.
  - not unique unless the graph is a tree.



Recall…

- Searching the maze with a Stack **Depth-First Search**
  - Try out every path, one at a time…
  - … repeatedly backtrack and try another
- Searching the maze with a Queue **Breadth-First Search**
  - Try out every path in parallel…
  - … repeatedly pick a path and expand it by one step

Depth-First Search

- DFS Marking Vertices UNVISITED: $O(|\text{vertices}|)$
- DFS Marking Edges UNVISITED: $O(|\text{edges}|)$
- DFS Vertex Loop: $O(|\text{vertices}|)$
- All Calls to DFSOne: $O(\sum_v 1 + deg(v))$

$$= O(|\text{vertices}| + |\text{edges}|))$$

$$O(|\text{vertices}| + |\text{edges}|)$$

Breadth-First Search

- Primary Goals
  - Visit every vertex in the graph **in increasing order of distance from the starting vertex**
  - Construct a spanning tree for every connected component
    - Side effect: Compute connected components
    - Side effect: Compute paths between pairs of vertices
    - Side effect: Determine if the graph is connected
    - Side effect: Identify cycles
    - Side effect: **Identify shortest paths to the starting vertex**
  - Complete in time O(|vertices|+|edges|)
  - Complete with memory overhead **O(|vertices|)**

Breadth-First Search

- BFS Marking Vertices UNVISITED: $O(|\text{vertices}|)$
- BFS Marking Edges UNVISITED: $O(|\text{edges}|)$
- BFS Vertex Loop: $O(|\text{vertices}|)$
- All connected components: $O(\sum_v 1 + deg(v))$ $= O(|\text{vertices}| + |\text{edges}|))$

$$O(|\text{vertices}| + |\text{edges}|)$$

DFS vs BFS

Application	DFS	BFS
Spanning Trees		
Connected Components		
Paths/Connectivity		
Cycles		
Shortest Paths		
Articulation Points		