Asymptotic Notation

- A programs runtime across variable factors (language, diff pcs)

Runtime - how efficient a program runs

- Calculate programs runtime based on how many instructions the program has to perform based on size of programs input

Size of input = N

- How many steps performed relative to N

- Only care about dominate term - Don't care about constants

- Big theta (Θ) - if there is a loop count iterations for input

- example

```
function(n)
  count = 0
  while n/2 != 0
    count ++
    N = n/2
  return count
```

N	# of iterations
1	0
2	1
4	2
8	3
N	$\log_2 N$

$N = \log_2 N$

Drop constant =

$\Theta(\log n)$

Common Runtimes

- **Θ(1)**. This is *constant* runtime. This is the runtime when a program will always do the same thing regardless of the input. For instance, a program that only prints "hello, world" runs in `Θ(1)` because the program will always just print "hello, world".
- **Θ(log N)**. This is *logarithmic* runtime. You will see this runtime in search algorithms.
- **Θ(N)**. This is *linear* runtime. You will often see this when you have to iterate through an entire dataset.
- **Θ(N*logN)**. You will see this runtime in sorting algorithms.
- **Θ(N^2)**. This is an example of a *polynomial* runtime. You will see this runtime when you have to search through a two-dimensional dataset (like a matrix) or nested loops.
- **Θ(2^N)**. This is *exponential* runtime. You will often see this runtime in recursive algorithms (Don't worry if you don't know what that is yet!).

- **Θ(N!)**. This is *factorial* runtime. You will often see this runtime when you have to generate all of the different permutations of something. For instance, a program that generates all the different ways to order the letters "abcd" would run in this runtime.

Big Omega (Ω) and Big O

- Runtime of multiple cases

```
function(list):          # size n
    For value in list:
        if value is 12:
            Return true
    Return false
```

N	if first value is 12	if 12 Not in list
10	1	10
100	1	100
1000	1	1000
N	1	N

# of iterators

Best case we represent with Big Omega

$\Omega(1)$

↳ worst case we use Big O

$O(n)$

Adding Runtimes

Divide function into chunks

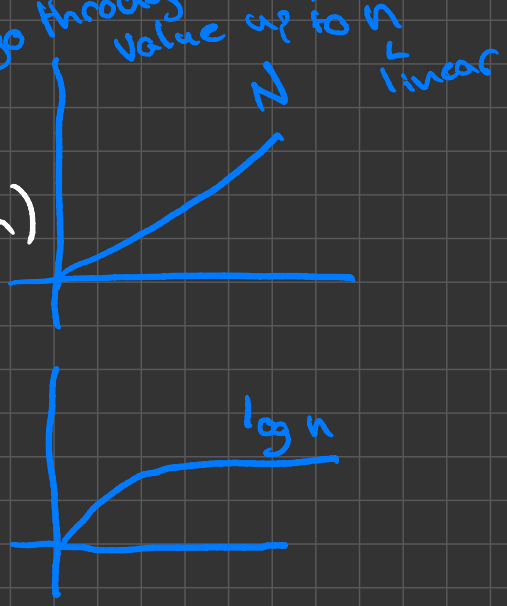
```
function(n)
```

- loop 1: print values 0 to N $\Theta(n)$
- loop 2: find lower bounded power of 2 $\Theta(\log n)$

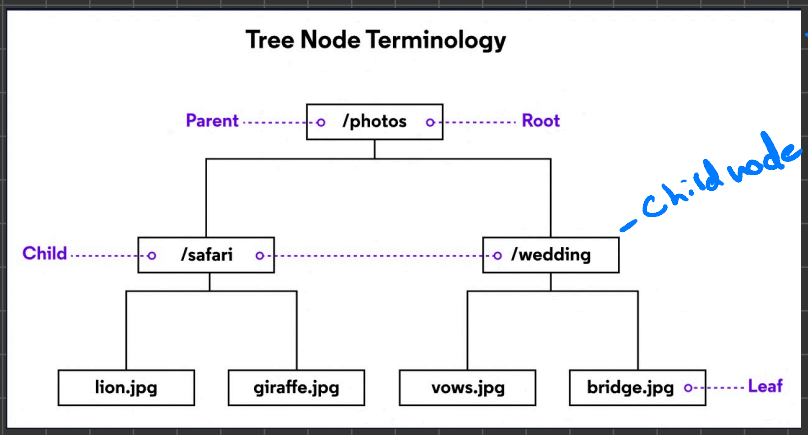
Runtime = $\Theta(n) + \Theta(\log n)$

- When we graph N grows faster making it the more dominant term.

Runtime = $\Theta(N)$



Trees - storing hierarchial data in nodes with a directed flow. Nodes store zero or more other tree nodes



-> Photos is the Root node

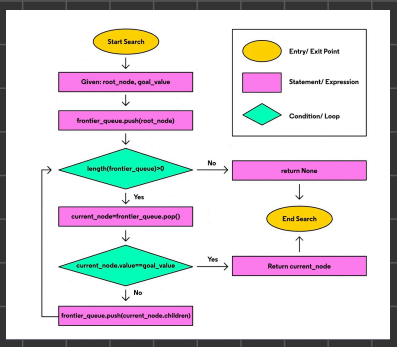
- Child node

- if two nodes share same parent they are siblings

- when a node has no children it is a leaf node

Tree Traversal Methods.

- breadth-first search - inspect every node on a level starting at top of the tree and then moving to the next level - Acts as queue
  - level by level search pattern. As nodes are searched children are added to queue. - FIFO



- Depth first search - Search deep into the branch and don't move until you've reached the end. - Acts as Stack

frontier - nodes to be searched next

- Begins at root node and explores until it reaches a leaf node then it travels back up the tree. Process cont until desired node is found
- Stack to store roots of unexplored subtrees

- Depth first has a complexity of O(n) n= nodes in tree

- found in many modern applications.

-Divide and Conquer Algorithms

- Recursively breaking down problem into smaller problems.
  - keep breaking down data set until each sub array contains 1 value
  - merge subarrays smaller # on left

- Start at middle value of dataset. If value we are searching is less use left side of array if it is greater use right

Follow $O(\log n)$ = 64 elements = 6 comparisons

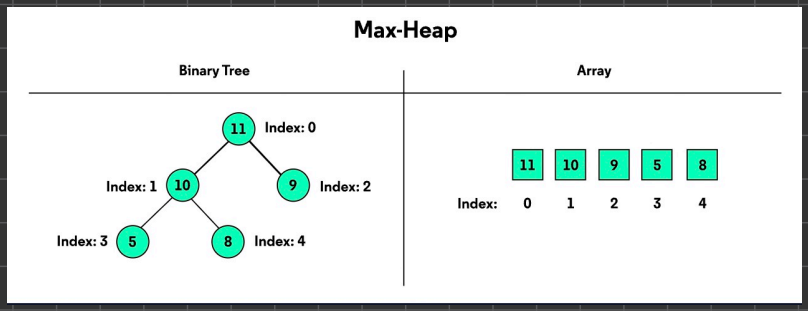
binary search.py

Heaps - maintain max or min value in dataset.

- used to create a priority queue

Max value tracking = Max-heap
min value tracking = Min-heap

- Binary tree with two qualities
  - Root = max value of data set
  - Every parent is greater then its children



- Tree index formula
  - left child: (index x 2) + 1
  - Right child: (index x 2) + 2
  - Parent: (index - 1) / 2

- Heapify -> Add a element to the end and start bumping it up.
  - Swap parent with child until structure is restored.

Heapsort - sorts arrays by inserting data into a heap data structure, and then repeatedly extracting the root of the heap. Time efficient due to O(n log n)

How algorithm works...

- Build a max-heap to store data from unsorted list
- Extract largest heap value and place it into a sorted list. <- heapify
- Replace heap root with last element of list then rebalance heap
- once max-heap is empty return sorted list

* Swap for last element because we know it has no children

Graphs - perfect for modeling networks

- composed of nodes or vertices that hold data

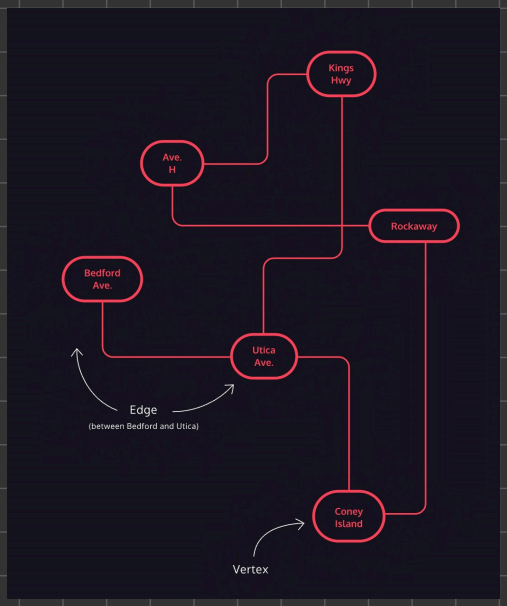
- Edges - connection between two vertices

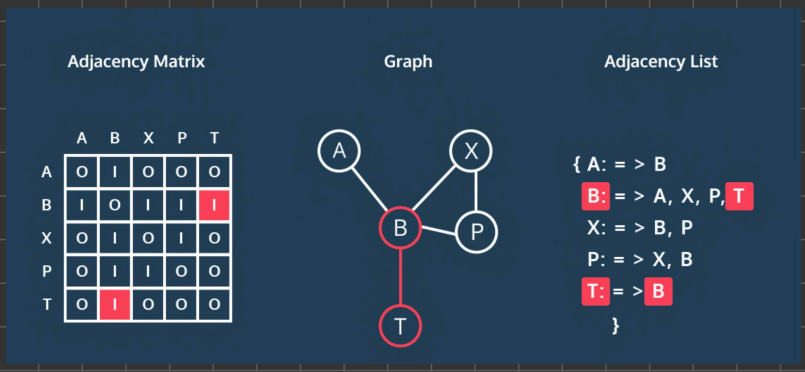
-A single node is a vertex

- Nodes are adjacent if edge directly connects them

- Weighted graph - edges have a # or cost associated with traveling between vertices

- directed graph - edges restrict direction of movement between vertices





—> 1 marks an edge. 0 = no edge

- `vertex` : A node in a graph.
- `edge` : A connection between two vertices.
- `adjacent` : When an edge exists between vertices.
- `path` : A sequence of one or more edges between vertices.
- `disconnected` : Graph where at least two vertices have no path connecting them.
- `weighted` : Graph where edges have an associated cost.
- `directed` : Graph where travel between vertices can be restricted to a single direction.
- `cycle` : A path which begins and ends at the same vertex.
- `adjacency matrix` : Graph representation where vertices are both the rows and the columns. Each cell represents a possible edge.
- `adjacency list` : Graph representation where each vertex has a list of all the vertices it shares an edge with.