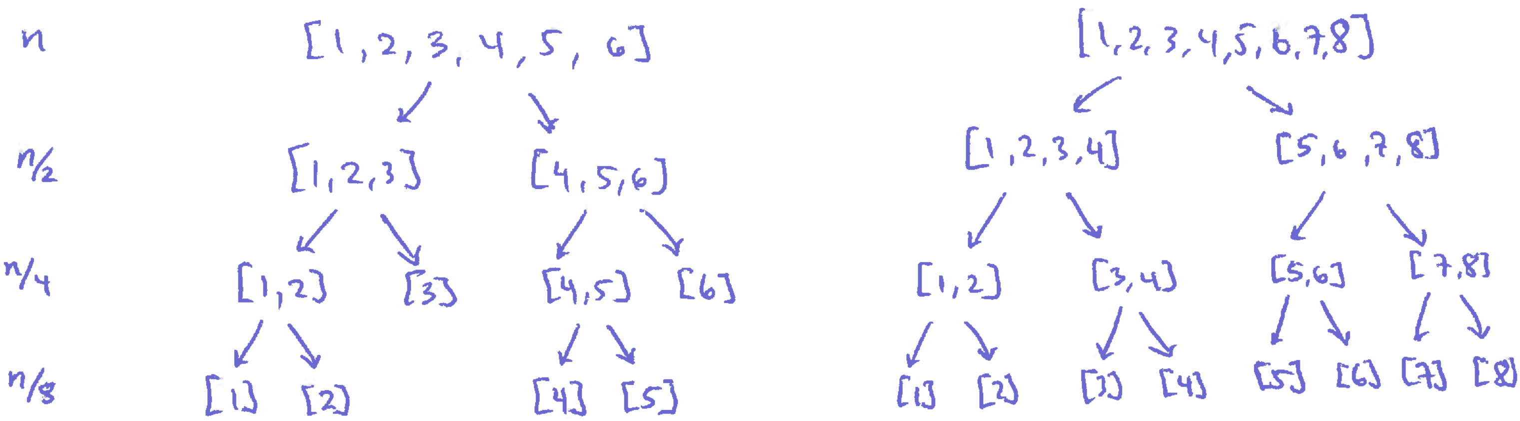
If $n = 2^m$, we have:

$$T(n) = m \cdot n - 2^m + 1 + 2^m \cdot T\left(\frac{n}{2^m}\right)$$
$$= mn - 2^m + 1 + 2^m$$
$$= mn + 1 = O(m\,n) = O(n \log n).$$

What about when n is <u>not</u> a power of 2? Many ways to think about this.

<u>Ex:</u> $n=6$ $n=8$



Since we're using big-O, we are only after upper-bounds. Thus let $m = \log_2 n$ and round up, so

$$n \leq 2^m$$

Then the runtime for lists of length n is

$$O(m \cdot 2^m) = O(n \log n).$$

To summarize

Selection Sort : $O(n^2)$
Bubble Sort : $O(n^2)$
Quick Sort : $O(n^2)$
Merge Sort : $O(n \log n)$

compare bubble and merge

40

Let's move away from sorting algorithms. They're a bit of a meme. There are many other sorting algorithms. Honourable mention:

<u>Bogo sort:</u> Randomly sort/shuffle until the list is in order. $O(n!) = O\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$.

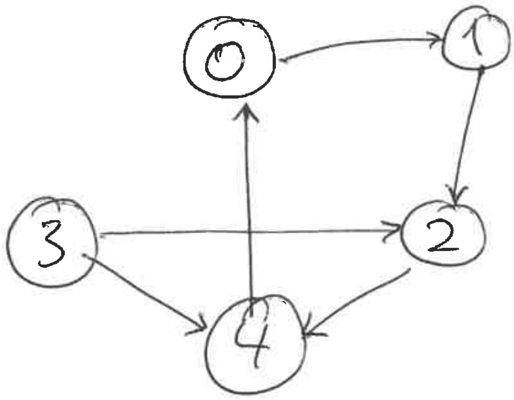
Graphs

Graphs are a fundamental data structure in computer science. And there are many algorithms that exploit the structure of a graph. ~~Two~~ Three examples:

1. Creating an AI player in chess.
2. Finding the nearest grocery store.
3. Spell checker.

Graphs are mathematical objects from combinatorics (i.e. discrete maths). We describe finite <u>directed graphs</u>. — this is what we mean by "graph". A graph is a set of <u>vertices</u> V and a set of <u>directed edges</u> E.

A vertex can be any object. For simplicity they will be non-neg. ints. Edges are pairs of vertices: (v_1, v_2). Visually:



Vertices:
$\{0, \ldots, 4\}$.
Edges:
$\{(0,1), (1,2), (2,4), (3,2), (3,4), (4,0)\}$

We do not allow for multiple edges between the same pair of vertices. Sometimes vertices are called <u>nodes</u>, and graphs called <u>networks</u>. Often we ~~wit~~ write: a graph is $G = (V, E)$.

In above 

, $2, 3$ are vertices
$(3,2)$ is the edge.
3 is the <u>source</u>,
2 is the <u>target</u>.

Sometimes the orientation is not important. Undirected graphs are modeled the same way, but an edge is now a 2-element <u>set</u> rather than tuple. Without additional context, a graph is usually undirected.

We won't have a use for loops: 

i.e. a vertex connected to itself — so we exclude them.

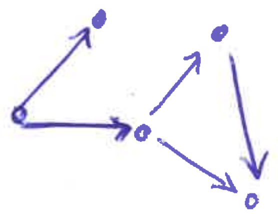
Path: a sequence of edges $e_1 e_2 \cdots e_n$ such that the target of e_i is the source of e_{i+1}.

Cycle: a path $e_1 \cdots e_n$ where the target of e_n is the source of e_1.

Connected graph: Every pair of vertices has a underlying undirected path between them.

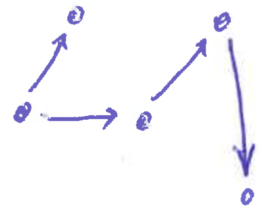
Tree: a connected graph whose underlying undirected graph has no cycles.

Ex:



Not a tree!

Ex:



Tree!

Tree are important examples of graphs with their own data structures and specialized algorithms.

We assume a graph has n vertices and m edges. It's convenient to assume vertices are $\{0, 1, \ldots, n-1\}$, but sometimes vertices are data themselves.

Some standard functions associated with graphs:

1. add_edge(i,j): add the edge (i,j) to E
2. remove_edge(i,j): remove the edge (i,j) from E
3. has_edge(i,j): check if (i,j) is in E
4. out_edges(i): get the list of vertices corresponding to outgoing edges at i.
5. in_edges(i): get the list of incoming edges at i.

We'll discuss some graph representations like we did with lists: linked lists and arrays.

Adjacency Matrix

The adjacency matrix of a graph is an $n \times n$ matrix of boolean values (or 0 and 1). If $A = (a_{ij})$ is the adjacency matrix of $G = (V, E)$, then

$$a_{ij} = \text{True} \iff (i,j) \text{ is in } E.$$

By the way, a matrix could be implemented as a list of lists, but it's often implemented as a (1-dimensional) array.

With an A.M., the first 3 functions are easily constant time (assuming getting a_{ij} is $O(1)$).