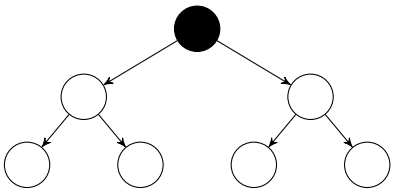
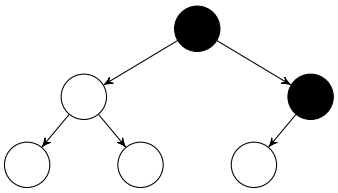
Tree Algorithms

1. (a)  (b) 

2. (a) Let $r(T)$ denote the root of tree T. Note the depth of node x in T is equal to the length of the path from $r(T)$ to x. Hence, $P(T) = \sum_{x \in T} d(x, T)$.
   For each node x in T_L, the path from $r(T)$ to x consists of the edge $r(T), r(T_L))$ and the path from $r(T_L)$ to x. The same reasoning applies for nodes x in T_R. Equivalently, we have

$$d(x,T) = \begin{cases} 0 & \text{if } x = r(T) \\ 1 + d(x, T_L) & \text{if } x \in T_L \\ 1 + d(x, T_R) & \text{if } x \in T_R \end{cases}$$

   Then,

$$\begin{aligned} \sum_{x \in T} d(x,T) &= d(r(T),T) + \sum_{x \in T_L} d(x,T) + \sum_{x \in T_R} d(x,T) \\ &= 0 + \sum_{x \in T_L} [1 + d(x,T_L)] + \sum_{x \in T_R} [1 + d(x,T_R)] \\ &= |T_L| + |T_R| + \sum_{x \in T_L} d(x,T_L) + \sum_{x \in T_R} d(x,T_R) \\ &= n - 1 + P(T_L) + P(T_R) \end{aligned}$$

   (b) Let T be a randomly built binary search tree with n vertices. Without loss of generality, we assume the n keys are $\{1, \ldots, n\}$.
   By definition, $P(n) = E_T[P(T)]$. Then,

$$\begin{aligned} P(n) &= E_T[P(T_L) + P(T_R) + n - 1] \\ &= n - 1 + E_T[P(T_L)] + E_T[P(T_R)] \end{aligned}$$

   Notice that

$$E_T[P(T_L)] = \sum_{i=1}^{n} E_T[P(T_L) | r(T) = i] \cdot \Pr(r(T) = i)$$

   .
   Since each element is equally likely to be the root of T, $\Pr(r(T) = i) = 1/n$ for all i. Conditioned on the event that element i is the root, T_L is a randomly built binary search tree on the first $i-1$ elements. To see this, assume we picked element i to the root. From the point of view of the left subtree, the elements $1, \ldots, i-1$ are inserted into the subtree in a random order, since these

elements are inserted into T in a random order and subsequently go into T_L in the same relative order. Hence, $E_T[P(T_L)|r(T) = i] = P(i-1)$. Putting these together, we get

$$E_T[P(T_L)] = \sum_{i=1}^{n} \frac{1}{n} P(i-1)$$

. Similarly, we get $E_T[P(T_R)] = \sum_{i=1}^{n} \frac{1}{n} P(n-i)$. Then,

$$\begin{aligned} P(n) &= n - 1 + E_T[P(T_L)] + E_T[P(T_R)] \\ &= n - 1 + \frac{1}{n} \sum_{i=1}^{n} [P(i-1) + P(n-i)] \\ &= n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} [P(i) + P(n-i-1)]. \end{aligned}$$

(c) This is the same recurrence that appears in quicksort.

(d) In words, our algorithm does the following:

- Constructs a randomly built binary search tree T by inserting given elements in a random order
- Traverses T in order to get a sorted list.

The first step takes $O(n \log(n))$-time in expectation. We observe that given the final state of tree T, we can compute the amount of work spent to construct T. To insert a node x at depth d, we traversed exactly the path from the root to the parent of x, at depth $d-1$, to insert it. Hence, we can upper bound the total work done to construct T by $O(P(T)) = O(n \log(n))$. The second step takes $O(n)$-time, so overall, the algorithm takes $O(n \log(n))$-time.

Graph Algorithms

---

1. (a) DFS: E, B, D, C, A; BFS: A, B, C, D, E.
   (b) DFS: E, B, C, D, A; BFS: A, B, C, D, E.

2. This solution only works on DAG's.

```
def dfs(G, start_v):
  for v in G.vertices:
    v.status = "unvisited"

  # Stack
  stack = []
  cur_time = 0
  stack.append(start_v)
  while not len(stack) == 0:
    # Peaks at the top vertex from stack
    u = stack.peak()
    if u.status == "done":
      stack.pop()
      continue

    if u.status == "unvisited":
      u.status = "in_progress"
      u.start_time = cur_time
      cur_time += 1

    all_done = True
    # Iterates through neighbors in reverse order
    for v in u.neighbors[::-1]:
      if v.status != "done":
        stack.append(v)
        all_done = False
    if all_done:
      u.status = "done"
      u.end_time = cur_time
      cur_time += 1
      stack.pop()
```

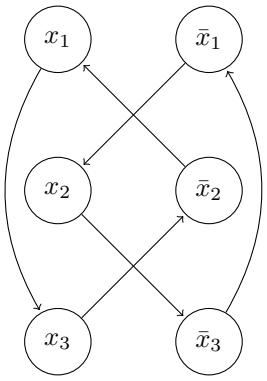
3. In words, our algorithm does the following:

   - Constructs a graph with a vertex v for each value $0, \ldots, T-1$.
   - For each $s_i \in \{s_1, s_2, \ldots, s_k\}$, add edges from vertex v to vertex $(v + s_i) \mod T$.
   - Start at vertex $(\sum_i s_i) \mod T$ and run BFS to find vertex 0, and return the length of the shortest path.

   This algorithm takes $O(T)$-time to construct the vertices, $O(kT)$-time to construct the edges, $O(k)$-time to compute the start vertex and $O(T + kT)$-time to BFS to find vertex 0.

4. (a)
   - $\bar{x}_1 \to x_2$, $\bar{x}_2 \to x_1$
   - $x_1 \to x_3$, $\bar{x}_3 \to \bar{x}_1$
   - $x_2 \to \bar{x}_3$, $x_3 \to \bar{x}_2$

   (b)



   (c) SCCs represent nodes that share the same state. If one node in an SCC is set to true, the implication graph forces all variables in that SCC to be set to true. If x and $\bar{x}$ were inside of the same SCC, this means that the original 2SAT problem has no solution because it is logically impossible for x to be both true and false, but the implication graph would require that.

   (d) First, to show that there is a corresponding component $\bar{C}$, we observe that if there is a cycle $a \to b \to \ldots \to z$ in C, we must have a corresponding cycle $\bar{z} \to \bar{y} \to \ldots \to \bar{a}$ in $\bar{C}$. Therefore, any cycle our component has must also exist somewhere else in the graph, which defines a corresponding component.

   To show that there are no edges between the component and its corresponding component, we use a proof by contradiction. If we assume that there is an edge between C and $\bar{C}$, C must contain some node that has an outward edge to $\bar{C}$, $m \to n$, where n is within $\bar{C}$. However, this also means that there is an edge $\bar{n} \to \bar{m}$, where $\bar{n} \in C$ and $\bar{m} \in \bar{C}$. Therefore, there are edges to go from $C \to \bar{C}$, and from $\bar{C} \to C$, which means that they are the same connected components, meaning that variables and their negations are within the same SCC, breaking our original assumption.

   (e) We go through the components in reverse-topological order. We set the nodes in the component to true, and also set the nodes in the corresponding component to false.