General Notes

- The constant factor matters!
- Small numbers of trials = higher variability, therefore run a large number of trials to reduce variability
- Amortization: when the worst case happens rarely, but is necessary for the normal case to occur (eg allocating more memory for a static array after it's filled)

Optimizations

- Put the options most likely to evaluate as true earlier in the if/else chain to reduce the number of checks the computer does
- When moving an element in a list, instead of swapping each pair of elements, performing 2 operations for each i, we keep a variable for i and shift values down as needed, only inserting the element into position at the very end.
- Memoization can reduce the runtime of otherwise exponential (e.g. $O(2^n)$) recursive algorithms to linear by simply storing the results of previous function calls.

List and Dict operations

List operations

- A list is represented as an array; the largest costs come from growing beyond the current allocation size, or from inserting or deleting somewhere near the beginning
- Copy: O(n), Append: Worst case O(n) and Amortized O(1), Insert: O(n), Delete: O(n), length: O(1), min or max: O(n)
- "The worst case of this is amortized across some constant time operation so the total amount is still linear."
- Lists grow at full capacity, doubling it's size, and shrink at 1/4 capacity, halving it's size.

Dictionary operations

- Python uses a hashmap for its dictionaries; you can have collisions leading to worstcase O(n), but its usually O(1) w/ no collisions
- get: Avg O(1)/Worst O(n), set: Avg O(1)/Worst O(n), delete: Avg O(1)/Worst O(n).
- Dicts resize at 2/3 load.

Good Sorts

Mergesort

Divide array in half recursively, until it is down to 1 element. Merge array together like a zipper.

Time Complexity: $O(n \log n)$
Memory Complexity: $O(N)$

Top-down vs Bottom-up TL;DR

Top-down uses recursion: starts at **top** of tree and proceeds **downwards**. Bottom-up does not use recursion: starts at **bottom** of tree and iterates over pieces moving **upwards**.

Top-down

Traditional recursive approach

```
def mergesort(L):
  if len(L) <= 1:
    return
  mid = len(L) // 2
  left, right = L[:mid], L[mid:]
  mergesort(left)
  mergesort(right)
  temp = merge(left, right)
  for i in range(len(temp)):
    L[i] = temp[i]
def merge(left, right):
  L = []
  i = j = 0
  while i < len(left) or j < len(right):
    if i >= len(left):
      L.append(right[j])
      j += 1
    elif j >= len(right):
      L.append(left[i])
      i += 1
    else:
      if left[i] <= right[j]:
        L.append(left[i])
        i += 1
      else:
        L.append(right[j])
        j += 1
  return L
```

Bottom-up

Pass through array, merging as we go to double size of sorted subarrays. Keep performing the passes and merging subarrays, until you do a merge that encompasses the whole array. Generally more efficient than top-down, since recursive calls are expensive.

```
def mergesort2(L):
  if len(L) < 2:
    return L
  i = 1
  while i < len(L):
    j = 0
    while j < len(L):
      L[j:j+i*2] = merge(L[j:j+i], L[j+i:j+i*2])
      j += i*2
    i *= 2
  pass
```

Quicksort

1. Shuffle array to reduce impact of order on sorting speed
2. Pick first element of array as pivot
3. Create two sub arrays from remaining elements, one selecting those smaller, one selecting those larger. Put them on either side of the pivot
4. Recurse for each side of the pivot until everything is sorted.

```
def quicksort(L):
  copy = quicksort_copy(L)
  for i in range(len(L)):
    L[i] = copy[i]
def quicksort_copy(L):
  if len(L) < 2:
    return L
  pivot = L[0]
  left, right = [], []
  for num in L[1:]:
    if num < pivot:
      left.append(num)
    else:
      right.append(num)
  return quicksort_copy(left) + [pivot] +
    ↪ quicksort_copy(right)
```

- Fastest for disordered arrays, slowest for already sorted arrays
- Randomize array or select a random pivot to prevent worst case. (Best choice of a pivot is the median)
- **Best case**: The partitions are always of equal size : $\Omega(N \log N)$. Recurrence relation is $T(n) = 2T(n/2) + cn$.
- **Worst case**: One partition is always of size 0 (if the array is already sorted and we are picking pivots from the ends) : $O(N^2)$. Recurrence relation is $T(n) = T(n-1) + T(0) + cn$.
- **Average case**: 1.39 $N \log N \in \Theta(N \log N)$
- Uses less memory than merge sort. Space complexity $O(n)$

Heapsort

- Heaps/PQs are binary trees such that: (1). parents are always greater than both children, (2). tree is complete, (3). (assuming 0-indexing) the parent of node i is at array index $(i+1)//2$, the left child is at $2(i+1)-1$, and right at $2(i+1)$.
- To build a heap/PQ from a list, call `heapify()/sink_down()` in a loop, starting at the first non-leaf node (index `len(arr)//2-1`) to the root node. This step is $O(n)$. After this, you can call `extract_max/min()` repeatedly to obtain sorted list
- In regular usage the best, worst, and avg. case is $O(n \log n)$
- bottom up heapify is O(n) while top down is $O(nlog_n)$
- Along with storing the PQ data as an array, we can store the value/index pair in a dict, giving us amortized $O(\log n)$ `incr/decr_key()`

Graphs

BFS

- BFS is a graph traversal algorithm that visits nodes in breadth-first order using a queue data structure.
- It is useful for finding the shortest path between two nodes in an unweighted graph because it explores all nodes at a given distance from the starting node before moving on to nodes that are farther away.
- BFS has a time complexity of O(V+E), where V is the number of nodes and E is the number of edges in the graph.
- It can be implemented iteratively or recursively, but the iterative approach is typically preferred due to avoiding stack overflow errors on large graphs.
- BFS can also be used to detect cycles in a graph. If a node is visited twice during the BFS traversal, then there is a cycle in the graph.

```
def BFS(G, node1, node2):
  Q = deque([node1])
  marked = {node1 : True}
  for node in G.adj:
    if node != node1:
      marked[node] = False
  while len(Q) != 0:
    current_node = Q.popleft()
    for node in G.adj[current_node]:
      if node == node2:
        return True
      if not marked[node]:
        Q.append(node)
        marked[node] = True
  return False
```

DFS

1. Mark starting node as visited,
2. Go to next unmarked node in the current node's adjacent vertices,
3. Repeat.
4. If all adjacent nodes are marked, pop back up the stack and repeat with the next node.

- Useful for finding all vertices connected to one vertex, or finding a path between two.

Cycle detection

- Use depth-first search to traverse the graph and check for cycles by keeping track of visited nodes and their parent nodes.
- If the neighbor has been visited before and it is not the parent node of the current node, it means there is a cycle in the graph, so the function returns True.

```
def has_cycle(G):
  visited = set()
  for node in G.adj:
    if node not in visited:
      if has_cycle_helper(G, node, visited, None):
        return True
  return False
def has_cycle_helper(G, node, visited, parent):
  visited.add(node)
  for neighbor in G.adj[node]:
    if neighbor not in visited:
      if has_cycle_helper(G, neighbor, visited,
    ↪ node):
        return True
    elif neighbor != parent:
      return True
  return False
```

Checking connectedness

- BFS / DFS on all nodes, check if every other node is connected

```
def is_connected(G: Graph):
  # Return True if and only if there is a path
      ↪ between any two nodes in G
  for i in range(len(G.adj)):
    for j in range(i, len(G.adj)):
      if not DFS(G, i, j):
        return False
  return True
```

Trees

Binary Tree

A tree where nodes have at most two children. A complete binary tree requires that every leaf level be filled, all leaf elements lean left and if there is an "empty space" where there is no even node it must be a right child (ie nodes added to left child first).

```
class Node:
  def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None
class BST:
  def __init__(self):
    self.root = None
  def insert(self, data):
    if self.root == None:
      self.root = Node(data)
    else:
      self._insert(data, self.root)
  def _insert(self, data, cur_node):
    if data < cur_node.data:
      if cur_node.left == None:
        cur_node.left = Node(data)
      else:
        self._insert(data, cur_node.left)
    elif data >= cur_node.data:
      if cur_node.right == None:
        cur_node.right = Node(data)
      else:
        self._insert(data, cur_node.right)
  def height(self):
    if self.root == None:
      return 0
    else:
      return self._height(self.root, 0)
  def _height(self, cur_node, cur_height):
    if cur_node == None:
      return cur_height
    left_height = self._height(cur_node.left,
      ↪ cur_height + 1)
    right_height = self._height(cur_node.right,
      ↪ cur_height + 1)
    return max(left_height, right_height)
```

XC3 Trees

- if the root node of an XC3-Tree has i children, we say that XC3-Tree has degree i.
- Each child of the root node of an XC3-Tree is also an XC3-Tree.
- The ith child of the root node of an XC3-Tree has degree (i-2), if i > 2, and has degree 0 otherwise
- Finding height is O(logn)
- num of nodes in tree is fibinacci sequence offset by 2 (so i=1 is 2 i=2 is 3 etc.)

```
# value is root node value
# function is created so value of the node is the
      ↪ level that it is at
# Ex. root node = node value 0, level 1 = node
      ↪ value 1...
def create_XC3_tree(degree, value=0):
  if degree == 0:
    return Node(value)
  else:
    root = Node(value)
    for i in range(1, degree+1):
      if (i <= 2):
        # direct children with no children for i
      ↪ <= 2
        # single Node is appended to the root
        child = Node(value+1)
      else:
        # for direct children with children,
        # creates another XC3 tree with the
      ↪ required degree
        # append the sub-tree to root
        child_degree = i-2
        child = create_XC3_tree(child_degree,
      ↪ value+1)
      root.add_child(child)
    return root
```

More info in Lab 7 findings.

RBT

Self balancing binary tree. **Properties**

- root is black
- all NIL leaves r black
- all red nodes' children r black
- all paths from any node to their descendant leaves have the same num of black nodes
- new nodes start red

When you have a "problem child" (breaks rules), if the aunt is black then rotate, if it is red then you colour change. Time complexity for operations

- Insertion: $O(logn)$
- Deletion:

```
def rotate_right(self, tree):
  y = self.left
  self.left = y.right
  if y.right != None:
    y.right.parent = self
  y.parent = self.parent
  if self.parent == None:
    tree.root = y
  elif self.is_left_child():
    self.parent.left = y
  else:
    self.parent.right = y
  y.right = self
  self.parent = y
def rotate_left(self, tree):
  # see above, swap left and right dumbass <-
      ↪ @vraj
def fix(self, node):
  if node.parent == None:
    node.make_black()
  while node != None and node.parent != None and
      ↪ node.parent.is_red():
    if node.parent.is_left_child():
      uncle = node.get_uncle()
      if uncle != None and uncle.is_red():
        node.parent.make_black()
        uncle.make_black()
        node.parent.parent.make_red()
        node = node.parent.parent
      elif node.is_right_child():
        node = node.parent
        node.rotate_left(self)
      else:
        node.parent.make_black()
        node.parent.parent.make_red()
        node.parent.parent.rotate_right(self)
    else:
      uncle = node.get_uncle()
      if uncle != None and uncle.is_red():
        node.parent.make_black()
        uncle.make_black()
        node.parent.parent.make_red()
        node = node.parent.parent
      elif node.is_left_child():
        node = node.parent
        node.rotate_right(self)
      else:
        node.parent.make_black()
        node.parent.parent.make_red()
        node.parent.parent.rotate_left(self)
  self.root.make_black()
```

Shortest Path (Graphs)

Dijkstra's

- Does not work with negative weights because once a vertex is marked, the algorithm never develops this node again - it assumes the path with this node is the shortest
- When to use and why:
  - When there is no easily calculable heuristic
  - When all weights are guaranteed to be positive
  - When you don't know your destination
- Complexity: $O(V \cdot (\text{get next node}) + E \cdot (\text{update path}))$
  - For lists/dicts: $O(V^2 + E)$
  - For fib. heap: $O(V \log V + E)$
  - For fib. heap, dense graph: $O(V \log V + V^2) \approx O(V^2)$
  - For min. heap: $O(V \log V + E \log V)$
  - Multiply all terms by V for all-pairs ver.

```
def dijkstra(G,s,d):
  marked, dist = {}, {}
  Q = min_heap.MinHeap([])
  for i in range(G.number_of_nodes()):
    marked[i] = False
    dist[i] = float("inf")
    Q.insert(min_heap.Element(i, float("inf")))
  Q.decrease_key(s, 0)
  dist[s] = 0
  while not (Q.is_empty() or marked[d]):
    current_node = Q.extract_min().value
    marked[current_node] = True
    for neighbour in G.adj[current_node]:
      edge_weight = G.w(current_node, neighbour)
      if not marked[neighbour]:
        if dist[current_node] + edge_weight < dist
      ↪ [neighbour]:
          dist[neighbour] = dist[current_node] +
      ↪ edge_weight
          Q.decrease_key(neighbour, dist[neighbour
      ↪ ])
  return dist[d]
```

Bellman-Ford

- Works with negative weights, but does not work if there is a negative cycle
- When to use and why:
  - When there are negative weights
- Time complexity:
  - Regular: $O(EV)$
  - Complete graph (edge between every pair of vertices): $O(n^3)$
  - All-Pairs: $O(EV^2) = O(V^4)$

```
def bellman_ford(G, source):
  pred = {} #Predecessor dictionary
  dist = {} #Distance dictionary
  nodes = list(G.adj.keys())
  #Initialize distances
  for node in nodes:
    dist[node] = float("inf")
  dist[source] = 0
  #Meat of the algorithm
  for _ in range(G.number_of_nodes()):
    for node in nodes:
      for neighbour in G.adj[node]:
        if dist[neighbour] > dist[node] + G.w(node
      ↪ , neighbour):
          dist[neighbour] = dist[node] + G.w(node,
      ↪ neighbour)
          pred[neighbour] = node
  return dist
```

Floyd-Warshall

- All-pairs shortest path
- Works on negative weight graphs
- Beats Bellman-Ford for dense graphs, but loses for sparse graphs

```
def mystery(G):
  n = G.number_of_nodes()
  d = init_d(G)
  for k in range(n):
    for i in range(n):
      for j in range(n):k
```

```
      if d[i][j] > d[i][k] + d[k][j]:
        d[i][j] = d[i][k] + d[k][j]
  return d
def init_d(G):
  n = G.number_of_nodes()
  d = [[float("inf") for j in range(n)] for i in range(n)]
  for i in range(n):
    for j in range(n):
      if G.are_connected(i, j):
        d[i][j] = G.w(i, j)
    d[i][i] = 0
  return d
```

A*

```
def a_star(G, s, d, h):
  pred = {} # Predecessor dictionary
  dist = {} # Distance dictionary
  marked = {} # Marked dictionary
  Q = min_heap.MinHeap([])
  nodes = list(G.adj.keys())
  # Init to inf
  for node in nodes:
    Q.insert(min_heap.Element(node, float("inf")))
    dist[node] = float("inf")
    marked[node] = False
  # Set start distance to 0
  Q.decrease_key(s, 0)
  dist[s] = 0
  # Meat of the algorithm
  while not (Q.is_empty() or marked[d]):
    # extract the next minimum element and mark it
    current_element = Q.extract_min()
    current_node = current_element.value
    marked[current_node] = True
    # update keys by distance + heuristic
    for neighbour in G.adj[current_node]:
      # dont add the heuristic to the shortest path
      # do add it in the score of the min_heap
      if not marked[neighbour]:
        if dist[current_node] + G.w(current_node, neighbour) < dist[neighbour]:
          # add heuristic to weight in queue
          Q.decrease_key(neighbour, dist[current_node] + G.w(current_node, neighbour) + h.get(neighbour))
          dist[neighbour] = dist[current_node] + G.w(current_node, neighbour)
          # update the predecessor dictionary
          pred[neighbour] = current_node
  return (pred, dist[d])
```

Dynamic Programming

Why Use Dynamic Programming?

Dynamic programming splits a recursive problem into sub problems, and stores the results of these sub problems. This helps reduce the overall complexity of the function since the function does not need to waste time calculating something that has already been calculated.
Top Down Pros: 1. Solves fewer sub problems. 2. Only solves the problems it needs to. Cons: 1. Recursive in nature. 2. When solving problems, you're still solving the previous case until you get to a stored case or the base case
Bottom Up Pros: 1. Iterative in nature. 2. Quick lookup after problems have been solved. Cons: 1. Solves unnecessary subproblems. 2. You need to solve all the subproblems.

Subset Sum

s(i,t) means you are given a list of numbers i and want to see if u can sum them to a value t
Recursive method:

- $S(i, t) = S(i-1, t)$ or $S(i-1, t-n[i-1])$
- Case 1: You use $n[i-1]$ (where n is the list of numbers)
- Case 2: You don't use $n[i-1]$

Bottom-up method:

- $sp(i, j) = sp(i-1, j)$ or $sp(i-1, j-nums[i-1])$
- Space complexity: $\theta(nt) \rightarrow \theta(t)$ If all you care about is T or F, you can delete the row you are finished with every time you move to the next row, space complexity goes down to $O(t)$, where t is the length of the row
- Time complexity: $\theta(nt)$
- Iterative; solves all problems
- Bottom-up = you start from the base case (bottom) and build upwards to your solution

Top-down method:

- Same as recursive method, solves all problems you need to solve, recursion generally loses to iterative
- But for lists with a high max value, top-down beats bottom-up in time
- Top-down = you start with your solution and break it down into sub problems

```
def subset_sum_dynamic(numbers, target):
  sp = [[False for j in range(target + 1)] for i in range(len(numbers) + 1)]
  d = {}
  for i in range(len(numbers) + 1):
    sp[i][0] = True
  for i in range(1, len(numbers) + 1):
    for j in range(1, target + 1):
      if numbers[i - 1] > j:
        sp[i][j] = sp[i - 1][j]
        if sp[i - 1][j]:
          d[i, j] = ((i - 1, j), False)
      else:
        sp[i][j] = sp[i - 1][j] or sp[i - 1][j - numbers[i - 1]]
        if sp[i - 1][j]:h
          d[i, j] = ((i - 1, j), False)
        elif sp[i - 1][j - numbers[i - 1]]:
          d[i, j] = ((i - 1, j - numbers[i - 1]), True)
  if sp[len(numbers)][target]:
    print(recover_solution(d, numbers, target))
  return sp[len(numbers)][target]
def subset_sum_top_down(numbers, target):
  sp = {}
  for i in range(len(numbers) + 1):
    sp[(i,0)] = True
  for i in range(target + 1):
    sp[(0,i)] = i == 0
  top_down_aux(numbers, len(numbers), target, sp)
  print(len(sp))
  return sp[(len(numbers),target)]
def top_down_aux(numbers, i, j, sp):
  if numbers[i - 1] > j:
    if not (i - 1, j) in sp:
      top_down_aux(numbers, i - 1, j, sp)
    sp[(i, j)] = sp[(i - 1, j)]
  else:
    if not (i - 1, j) in sp:
      top_down_aux(numbers, i - 1, j, sp)
    if not (i - 1, j - numbers[i - 1]) in sp:
      top_down_aux(numbers, i - 1, j - numbers[i - 1], sp)
    sp[(i, j)] = sp[(i - 1, j)] or sp[(i - 1, j - numbers[i - 1])]
```

Splitting Strings

This problem uses a function to split a string into substrings of valid english words. Using the Trie data structure, an *add_word*() function is used to recursively build up words from a text file. It uses the *can_split*() function to split the strings to create substrings and to check if they are valid strings. Note that the DP approach for this problem results in an almost linear time complexity.

- Time complexity: $O(m)$ where m is the length of the longest word
- Space complexity: $O(mn)$ where n is the length of the string

```
def can_split_dynamic(s):
  sp = [True]
  d = {}
  for i in range(len(s)):
    b = False
    for j in range(i, max(i - 22, -1), -1):
      b = (sp[j] and t.check_word(s[j:i+1]))
      if b:
        d[i] = j
        break
    sp.append(b)
  return (sp,d)
```

Tries

```
class Trie:
  def __init__(self):
    self.is_word = False
    self.children = [None for _ in range(26)]
  def add_word(self, word):
    if word == "":
      self.is_word = True
    else:
      if self.children[letter_index(word[0])] == None:
        self.children[letter_index(word[0])] = Trie()
      self.children[letter_index(word[0])].add_word(word[1:])
  def check_word(self, word):
    if word == "":
      return self.is_word
    else:
      if self.children[letter_index(word[0])] == None:
        return False
      return self.children[letter_index(word[0])].check_word(word[1:])
  def get_height(self):
    heights = []
    for child in self.children:
      if child != None:
        heights.append(child.get_height())
    if heights == []:
      return 1
    return 1 + max(heights)
  def get_num_words(self):
    num_words = []
    for child in self.children:
      if child != None:
        num_words.append(child.get_num_words())
    if self.is_word:
      return 1 + sum(num_words)
    return sum(num_words)
  def create_random_word(self):
    index_list = []
    for i in range(26):
      if self.children[i] != None:
        index_list.append(i)
    if self.is_word:
      index_list.append(-1)
    j = random.randint(0, len(index_list) - 1)
    j = index_list[j]
    if j == -1:
      return ""
    return char_from_index(j) + self.children[j].create_random_word()
def letter_index(letter):
  return ord(letter) - 97
def char_from_index(i):
  return chr(i+97)
```

Egg Drop Problem

This problem is about determining which floor you can drop an egg from to see if the egg breaks. Below are the cases for the recursion:

- Case 1: The egg breaks. Lets say you have k eggs and you drop one and the egg breaks, then you have k-1 eggs and you have between 1 - n' floors to check, so sp(k, n) = sp(k-1, n-1)
- Case 2: The egg doesn't break. In this case, you search between n' to n, so we get sp(k,n) = sp(k, n-n')
- The overall recursion is max(sp(k-1, n'-1), sp(k, n-n')) + 1

The complexity of this problem is

- Time Complexity: O(F^2E) ?
- Space Complexity:

```
def min_drops(floors, eggs):
  opt_floors = {}
  L = sp = [[float("inf") for j in range(eggs+1)] for i in range(floors)]
  for i in range(floors):
    sp[i][1] = i+1
  for j in range(1, eggs+1):
    sp[0][j] = 1
  for i in range(1, floors):
    for j in range(2, eggs+1):
      current_min = float("inf")
      current_index = -1
      for n in range(1, i+1):
        if current_min >= max(sp[i-n][j], sp[n-1][j-1]) + 1:
          current_min = max(sp[i-n][j], sp[n-1][j-1]) + 1
          current_index = n
      sp[i][j] = current_min
      opt_floors[i,j] = current_index
  return sp[floors-1][eggs], opt_floors
```

dynamic problem ex: LCS

Given two strings str1 and str2, return the length of their longest common subsequence.

```
def lcs_recursive(str1, str2, m, n):
  if m == 0 or n == 0:
    return 0
  if str1[m-1] == str2[n-1]:
    return 1 + lcs_recursive(str1, str2, m-1, n-1);
  else:
    return max(lcs_recursive(str1, str2, m, n-1), lcs_recursive(str1, str2, m-1, n));
#memoizaiton cache
L = [[None]*(n+1) for i in range(m+1)]
def lcs_topdown(str1, str2, m, n):
 if m == 0 or n == 0:
  # Base case: LCS is 0 if either string has length 0.
  return 0
 if L[m][n] != -1:
  # If the length of the LCS for this pair of prefixes has already been computed
  return L[m][n]
 if str1[m-1] == str2[n-1]:
  # If the last characters of the strings match, include them in LCS.
  # Recursively call the function with the last character removed from each string.
  L[m][n] = 1 + lcs_topdown(str1, str2, m-1, n-1)
  return L[m][n]
 else:
  # If the last characters of the strings don't match, take max LCS by excluding last character of X or Y.
  L[m][n] = max(lcs_topdown(str1, str2, m, n-1), lcs_topdown(str1, str2, m-1, n))
  return L[m][n]
  def lcs_bottomup(str1 , str2):
  m = len(str1)
  n = len(str2)
  L = [[None]*(n+1) for i in range(m+1)]
  for i in range(m+1):
    for j in range(n+1):
      if i == 0 or j == 0 :
        L[i][j] = 0
      elif str1[i-1] == str2[j-1]:
        L[i][j] = L[i-1][j-1]+1
      else:
        L[i][j] = max(L[i-1][j] , L[i][j-1])
  return L[m][n]
```

Lab Takeaways

Lab 2

Covers the bad sorts (bubble, selection, insertion)

- Despite all the bad algorithms having the same worst-case performance, in the real world they performed significantly differently, with bubble sort being very bad and selection sort being the best (even though insertion sort should be theoretically faster)
- "Optimizing" an algorithm might make the runtime worse, if the optimization being performed ends up having more overhead than the non-optimized version
- Selection sort does not change with more or less disorder in the array, while insertion sort and bubble sort both perform better with a more ordered array.

Lab 3

Covers heap < merge < quick sort.

- In general, quicksort is fastest, with mergesort and heapsort trailing behind for one-off sorting.
- When arrays are near-sorted, the performance of quicksort falls off of a cliff
- Modifications to quicksort (dual quicksort) improve performance, up to a limit
- Bottom-up mergesort tends to be a good bit faster than top-down mergesort, as the recursive splitting step is removed
- Insertion sort is faster than mergesort and quicksort for very small lists (less than 10-15 elements in the list)

Lab 4

- As the proportion of edges increases, the probability of a cycle occurring in the graph also increased.
- The probability of all edges in a graph being connected follows a sigmoid curve

Lab 5

- The set of nodes in a minimum vertex cover and the set of nodes in a maximum independent set can be summed to equal the set of all nodes in a graph.

Lab 6

- Red-black trees have approximately half the height of a naive binary search tree, when working with random data.
- Binary search trees perform significantly worse with ordered data, versus random data, due to the lack of balancing. Red-black trees have relatively constant performance no matter the degree of disorder.
- If insertion speed is significantly more important than maintaining a perfectly balanced search tree, naive BSTs may be viable, but in most cases RBTs are preferred.

Lab 7

- Height of XC3-Tree can be determined from its degree i using the equation $h(i) = \lceil i/2 \rceil$, or by leaf node n where $h = \log_{\phi}(n)$
- Number of nodes in an XC3-Tree with degree i can be found using Fibonacci sequence where $nodes(i) = nodes(i-1) + nodes(i-2)$
- "Proof": Suppose $x \in \{nodes(i)\}, i \in \mathbb{N}$ and $\phi = \frac{1+\sqrt{5}}{2}$, the golden ratio. Then,

$$h(x) = \log_{\phi}(x) = \log_{\phi}\overbrace{(nodes(x-1) + nodes(x-2))}^{\text{Fibonacci Sequence}}$$

By the definition of logarithms, the above equality holds because we are dividing the Fibonacci sequence by ϕ, until it is equal to 1. Since the number of nodes in a XC3-Tree is the Fibonacci sequence–as derived at the beginning of this experiment–the division of each node by ϕ, by the definition of the golden ratio, represents the number of Fibonacci numbers we go through to get to the root node.
So, $\log_{\phi}(x)$ gives us the number of times we divided by ϕ to get to 1.
Since the number of times we divide by ϕ represents the number of Fibonacci numbers we go through to get first base case (which is the root node in the XC3 tree), The height of the tree must be $\log_{\phi}(n)$ for some leaf node n.
Now, since we have shown that the height of the XC3-tree is $\log_{\phi}(n)$, we can write that the time complexity of an XC3-tree is $O(\log_{\phi}(n))$ because of the change of base properties of the logarithm.

Final Lab

- Bellman-Ford's approximation algorithm performs significantly better than normal Bellman-Ford, while generally not being affected by the reduced number of relaxations
- Dijkstra's approximation algorithm performs poorly when the number of relaxations is small compared to the size and density of the graph
- Empirically testing A* on random data is hard, but using real-world data makes it easier
- Straight-line approximation is a good heuristic for A* on real-world data
- Use A* when we know the ending node, and have a good heuristic to work with.