General Notes

CS, ML and Stats

Patrick Daly

Contents

1 Computer Science

1.1 Algorithms

DFS Time: O(n), Space: O(n)
Solution exists far away.

Recursive

```
def dfs(node):
  if node:
    # do stuff if pre-order
    if node.left:
      dfs(node.left)
    # do stuff if in-order
    if node.right:
      dfs(node.right)
    # do stuff if post-order
```

Iterative

```
def dfs(node): # if bst, may need to swap search left/right
    visited = set()
    stack = [node]
    while stack:
        current = stack.pop(-1)
        print(current.val)
        if current not in visited:
            visited.add(current)
        if current.left and current.left not in visited:
            stack.append(current.left)
        if current.right and current.right not in visited:
            stack.append(current.right)
    return visited
```

BFS Time: O(n), Space(n)
Solution exists nearby.

Iterative

```
def bfs(node):
```

```
    stack = [node]
    while stack:
        current = stack.pop(-1)
        if current.left:
            stack.append(current.left)
        if current.right:
            stack.append(current.right)
```

Mergesort Time: O(nlogn), Space: O(n)

```
def mergesort(array, start, end):
    if start < end:
        mid = (start+end) // 2
        mergesort(array, start, mid)
        mergesort(array, mid+1, end)
        merge(array, start, mid, end)
```

```
def merge(array, start, mid, end):
    left = array[start: mid+1]
    right = array[mid+1: end+1]
    i, j, k = 0, 0, start
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            array[k] = left[i]
            i += 1
        else:
            array[k] = right[j]
            j += 1
        k += 1
    if j == len(right):
        array[k: end+1] = left[i:]
```

```
def binarysearch(array, val, low, high):
    if high < low:
        # can also return high or low index
        return 'Not found!'
    mid = (low + high) // 2
    if array[mid] > val:
        return binarysearch(array, val, low, mid-1)
```

```
    elif array[mid] < val:
        return binarysearch(array, val, mid+1, high)
    return mid
```

Greatest Common Divisor (GCD) Time: ?, Space: ?
Euclid: Time: O(ln^2min(a, b)), Space: O(1)

```
def gcd(a, b):
    return gcd(b, a % b) if b else a
```

Stein: Time: 60% faster than Euclid in theory, similar in practice Space: O(1)

```
def stein(a, b):
    # binary gcd
    if a == b: return a
    if a == 0: return v
    if b == 0: return a
    if ~a & 1: # a is even
        if b & 1: # b is odd
            return stein(a >> 1, v)
        else: # b is even
            return stein(a >> 1, b >> 1) << 1
    else:
        if ~b & 1: # b is even
            return stein(a, b >> 1)
        else: # b is odd
            if a > b:
                return stein((a-b) >> 1, b)
            else:
                return stein(u, (b-a) >> 1)
```

Fibonacci Time: O(n), Space: O(1)
Iterative

```
def fib_iterative(n):
    if n == 0: return 0
    if n == 1: return 1
    first, seecond, fibn = 0, 1, 0
    for i in range(2, n+1):
```

```
        fib_n = fib_n2 + fib_n1
        fib_n = fib_n2
        fib_n2 = fib_n
    return fibn
```

Recursive

```
def fib_recursive(n, dp={0: 0, 1: 1}):
    if n in dp:
        return dp[n]
    return fib_recursive(n-2, dp) + fib_recursive(n-1, dp)
```

Longest Increasing Subsequence Time: O(nlogn), Space: O(n)

```
def lis(nums):
    tails = [0]*len(nums)
    maxlen = 0
    for num in nums:
        start, end = 0, maxlen
        while start != end:
            mid = (start+end)//2
            if tails[mid] < num:
                start = mid + 1
            else:
                end = mid
        tails[start] = num
        maxlen = max(maxlen, start+1)
    return maxlen
```

Topological Sort Time: (V + E), Space: O(V)

Identifying a linear ordering of vertices such that if the graph G contains an edge (u, v), then u appears before v in the ordering. Often used in identifying depency graphs or sources in a event chain. Multiple solutions may exist.

```
def topo_sort(G):
    def dfs(u):
        visited.add(u)
        for v in G[u]:
            if v not in visited:
              dfs(v)
```

```
7           order.insert(0, u)
8       order = list()
9       visited = set()
10      for u in G:
11          if u not in visited:
12              dfs(u)
13      return order
```

Strongly Connected Components Time: O(V + E), Space: O(V + E)

Return all subgraphs for which each node is reachable from all other nodes in the subgraph. Tarjan and Pearce algorithms reduce the space requirement by 1-2x but they're more complicated to remember.

```
1   def transpose(G):
2       Gt = defaultdict(set)
3       for u in G:
4           for v in G[u]:
5
6   def scc_kosaraju(G):
7       def visit(u, add_to_stack):
8           visited[u] = True
9           if not add_to_stack:
10              sccs[-1].append(u)
11          for v in G[u]:
12              if not visited[v]:
13                  visit(v, add_to_stack)
14          if add_to_stack:
15              stack.append(v)
16      sccs = [[]]
17      visited = dict.fromkeys(G.keys(), False)
18      stack = list()
19      for u in G:
20          if not visited[u]:
21              visit(u, add_to_stack=True)
22      G = transpose(G)
23      visited = dict.fromkeys(G.keys(), False)
24      while stack:
25          u = stack.pop()
26          if not visited[u]:
27              visit(u, add_to_stack=False)
```

```
28                 if stack:
29                     sccs.append([])
30         return sccs
```

1.2 Data Sctructures

1.3 Linux

Common Commands

1. grep
2. awk
3. xargs
4. find
5. cut

Software & Packages

1. Vim
2. Tmux
3. Ranger
4. Autojump
5. Tldr
6. Jq
7. Ccat

2 Machine Learning

2.1 Supervised

2.1.1 Ordinary Least Squares (OLS)

2.1.2 Generalized Linear Model(GLM)

2.1.3 Logistic Regression

2.1.4 Linear Discriminant Analysis

2.1.5 Support Vector Machines

2.1.6 K-Nearest Neighbors

2.1.7 Gaussian Process

2.1.8 K-Nearest Neighbors

2.1.9 Decision Trees

2.1.10 Random Forest

2.1.11 Gaussian Process

2.1.12 Naive Bayes

2.1.13 Kalman Filter? dunno where this should go yet... maybe estimation section?

2.2 Unsupervised

2.2.1 Gaussan Mixture Models

2.2.2 K-Means

2.2.3 Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

2.2.4 Spectral Clustering

2.2.5 Hierarchical Clustering

2.2.6 Factor Analysis

2.2.7 Independent Component Analysis

2.2.8 Principal Component Analysis

2.2.9 Non-Negative Matrix Factorization (NMF)

2.2.10 Latent Dirichlet Allocation (LDA)

2.2.11 Outliear Detection?

3 Deep Learning

3.1 Convolutional Networks

3.2 Recurrent Networks