Algorithms and Datastructures HS24 / cs.shivi.io

1 Searching

- **Linear Search:** Iterate over every element until element is found.
- **Binary Search:** (Pre: Sorted List) Check if element is at middle. Otherwise rescope to upper or lower half and repeat.

2 Sorting

2.1 Bubble Sort

Invariant: After each n-th pass, the n-th largest element is at its correct position.

- **Single Pass:** Iterate through list and swap one element with it's next if it's larger. This "bubbles" the largest element to the back.
- **Repeated Passes:** Do $n-1$ single passes to bring the largest $n-1$ elements in it's correct order (which automatically puts the first element in it's correct place too.)

Runtime: $O(n^2)$

2.2 Selection Sort

Invariant: After each n-th pass, the n-th largest element is at its correct position.

- **Single Pass:** In each pass find the maximum element in the *unsorted* array and swap it with the last element in that *unsorted* array.
- **Repeated Passes:** Do single passes for the array from $1...i$, where $i = n, ..., 1$.

Runtime: $O(n^2)$

2.3 Insertion Sort

Invariant: $I(j) : A[1...j]$ is sorted

- **Single Pass:** Pick element right of the sorted array. Push it into it's correct position (using binary search) in the sorted array.
- **Repeated Passes:** Do the single pass $n-1$ times.

Runtime: $O(n^2)$

2.4 Merge Sort

Invariant: $I(l, r) : A[l...\text{mid}] \wedge A[\text{mid}+1...r]$ are both sorted

- **Divide:** Recursively split array into two halfs until each array has one element (bas case).
- **Merge:** Convert the two sorted halfs into one sorted array.

```
def mergeSort(A, l, r):
  if l < r:
    mid = (l+r)//2
    mergeSort(A, l, mid)
    mergeSort(A, mid+1, r)
    merge(A, l, mid, r)
def merge(A, l, mid, r):
  # merge sorted subarrays
  # copy remaining elements from left/right part
```

Runtime: $O(n \log n)$

2.5 Quick Sort

- **Divide:** Pick pivot (*how can influence the runtime!*). Partition the array (`[smaller..., pivot, ...bigger]`). The pivot is at it's correct position.
- **Conquer:** Recursively apply the divide process for the left and right subarrays until they have one element. That's it!

Runtime: Worst pick for pivot: $O(n^2)$, Average pick: $O(n \log n)$

2.6 Heap Sort

We'll look at the heap DS in the DS part.

```
def heapsort(A):
  H = MaxHeap(A)
  for i in range(n, -1, -1):
    A[i] = H.extractMax()
```

Runtime: $O(n \log n)$

3 Data Structures

3.1 ADT List

Supports: `append(val)`, `get(i-th)`, `insert(val, idx)`, ... Can be implemented by: 1) Array (fast retr., slow insert/del) 2) Linked List (fast insert/del, slow retr.)

3.2 (Max) Heap

We store our heap as a binary tree with the **"heap condition: val child $\leq$ val parent $\forall$ nodes"**.

We need to implement two operations 1) `.extractMax()` 2) `MaxHeap()`.

3.2.1 Extract Max

The idea is to simply pop the root node. Now we have two binary heap trees. We take the node at the bottom right of our binary tree and put it at the roots position. Then since our heap condition holds for the two subtree, we can swap our substitute element with it's children until the heap condition is satisfied.

```
def extractMax(self):
  m = self.root
  self.root = self.lastElement()
  del self.lastElement
  el = self.root

  while True:
    largestChild = el.l if el.l > el.r else el.r
    if largestChild > el:
      self.swap(el, largestChild)
    else:
      break

  return m
```

3.2.2 Create Heap

The idea here is similar. We `insert()` our new element at the bottom right. Then we swap it with it's parent until the heap condition is satisfied.

```
def insert(self, node):
  el = self.pushEmptyBottomRight(node)
  while el.parent and (el > el.parent):
    self.swap(el, el.parent)

def createHeap(A):
  H = MaxHeap()
  for v in A:
    H.insert(v)
```

3.3 Binary Trees

Conditions: 1) $\forall n(\forall x((x \text{ is left } n \Rightarrow x \leq n) \wedge (x \text{ is right } n \Rightarrow x \geq n)))$. 2) No duplicates (optional, but we'll stick to it)

- **Searching ($O(\log n)$):** Think binary search...
- **Insertion ($O(\log n)$):** Think insertion sort... (if key is found then don't do anything)
- **Deletion ($O(\log n)$):** 1) Locate Node 2) ...
  - Case 2a (zero or one child): Connect substitute node with child
  - Case 2b (two children): Find the smallest key in the right subtree (inorder succesor) and replace the current node with that node. Then (recursively) delete the inorder succesor.

3.4 AVL Binary Trees

Same as binary tree but with the **AVL Property:** $\forall n(|h_l - h_r| \leq 1)$.

The idea is that when we insert a node we "repair" our AVL tree if it violates the AVL condition. We have four cases:

3.5 Union Find

The **Union-Find** data structure tracks a collection of disjoint sets and supports two main operations:

- **Find**: Determine which set an element is in.
- **Union**: Merge two sets into one.

In this implementation, we will:

- Use a **representative array** (`rep[n]`) where `rep[n]` is the parent of node `n`.

- **No path compression**: The parent pointer will not be optimized to directly point to the root.
- **Union by size**: We merge the smaller set into the larger one by overwriting the parent pointer.

Operations:

1. **Find**: Returns the representative of the set containing element `x`.
   - It directly accesses `rep[x]` to get the parent.
2. **Union**: Merges the sets containing `x` and `y` by overwriting the parent of the smaller set with the larger set's root.

```python
class UnionFind:
  def __init__(self, n):
      self.rep = list(range(n))
      self.members = [[i] for i in range(n)]

  def find(self, x):
      return self.rep[x]

  def union(self, x, y):
      rootX = self.find(x)
      rootY = self.find(y)

      if rootX != rootY:
          # smaller set to be merged into larger set
                  if  len(self.members[rootX])  < len(self.members[rootY]):
              for elem in self.members[rootX]:
                  self.rep[elem] = rootY
              self.members[rootY].extend(self.members[rootX])
              self.members[rootX] = []
          else:
              # ... same but switch X, Y ...
```

Runtime: Find $O(1)$, Union: $O(n)$

4 Dynamic Programming

The idea of DP is to solve problems by breaking them down into smaller subproblems and solving either iteratively (top-down or bottom-up) or recursively (memoization).

4.1 SRTBOT Framework and Tips

SRTBOT Framework

1. **Subproblem:** Define smaller problems.
2. **Recurrence:** Express solution using subproblems.
3. **Topological Order:** Process subproblems in correct order.
4. **Base Case:** Handle trivial cases.
5. **Original Problem:** Express original as subproblems.
6. **Time Complexity:** Analyze subproblem space and computation.

Good Subproblems

- Prefixes: `X[:i]` $\rightarrow O(n)$
- Suffixes: `X[i:]` $\rightarrow O(n)$
- Substrings: `X[i,j]` $\rightarrow O(n^2)$
- **Avoid:** Subsequences $\rightarrow O(2^n)$

For multiple sequences: Multiply subproblem space (cross product).

Constraints:

- Add constraints to subproblems, e.g., start/end at `i`.
- Original problem may require $O(n)$ instead of $O(1)$ if constraint affects recurrence.

Multiple Parties (e.g., Games):

- Add player as a parameter:
  - Maximize your "score" while the opponent minimizes it.
- Use **min-max recursion** with pruning.

Problem Types:

1. **Min/Max:** Optimize values (e.g., longest path, score).
2. **Boolean:** Yes/No problems (e.g., reachability).
3. **Count:** Count all valid solutions (e.g., total paths).

Key Notes:

- Subproblem expansion often includes **min/max**, but only final value matters.
- For games: Alternate max/min in recurrence.
- Example: `dp[i][j][player]` $\rightarrow$ maximize/minimize scores per turn.

4.2 Jump Game II

Problem: Given an array of non-zero numbers representing the maximum jump length from that position, determine the minimum jumps required to reach the last element.

Solution:

We use a greedy 1D approach. The idea is to "bucket" how far we get using n steps. We use the previous bucket to see how far we could maximally jump from that bucket ($\max(\{i + v_i \mid i \in$ prev bucket$\})$), this gives us the range for the current bucket.

Invariant: $M[k] =$ maximal idx reachable with k jumps

Runtime: $O(n)$

4.3 Longest Common Subsequence

Problem: Given two strings A, B find the length of the longest common subsequence (= only allowed to remove chars).

Solution:

This becomes a 2D problem since we have the following possibilities:

- `A[0] == B[0]`: `return 1+dp(1, 1)`
- `A[0] != B[0]`: Return the max of 1) removing the first char 2) removing the second char. The third option of removing both chars is simply a result of the first two.

This can be modelled as a 2D table.

Runtime: $O(n^2)$

4.4 Edit Distance

Problem: Given two strings A, B find the number of operations to convert $A \rightarrow B$. Operations are: 1) Insertion 2) Deletion 3) Substitution

Solution:

This too becomes a 2D problem. We have the possibilities:

- `1+ED(i-1, j)`: Delete A_i
- `1+ED(i, j-1)`: Insert B_j
- `(A[i] != B[j])+ED(i-1,j-1)`: Equal or Substitution

Now we simply need to pick the minimal value of these three.

Our bases cases are: 1) `ED(0,0) = 0` 2) `ED(i, 0) = i` since we need to insert i times and 3) `ED(0, j) = j` for the same reasons.

This can again be modelled as a 2D table.

Runtime: $O(n \cdot m)$, where $n =$ length of A, $m =$ length of B.

4.5 Subset Sum

Problem: Given a set of n non-negative integers A and a target sum T, determine if a subset of the given numbers can add up precisely to the target.

Solution:

Our subproblem becomes $\text{SS}(i, s) :=$ can the last i elements of the set $A[i:]$ sum up to s? Then our recurrence becomes 1) Either include current element 2) or exclude the current element. Hence $\text{SS}(i, s) = \text{OR}\{\text{SS}(i+1, s), \text{SS}(i+1, s - A_i)\}$. Now we simply need to calculate $\text{SS}(i, s)$ from $i = n, \ldots, 1$, we can't put a constraint on how s should evolve, but we know that $s = [0, T]$.

Runtime: $O(n \cdot T)$ (pseudopolynomial)

4.6 Knapsack

Problem: Given a set of n items A, each with weight w_i and profit p_i, and a knapsack with weight capacity W, output the most optimal collection of items which fit the constraints.

Solution:

Our subproblem becomes $\text{KS}(i, w) :=$ max profit using subset $A[i:s]$ not exceeding weight w. Our subproblem becomes $\text{KS}(i, w) = \max\{\text{KS}(i+1, w), \text{KS}(i+1, w - w_i) + p_i\}$.

Runtime: $O(n \cdot W)$.

4.7 Longest Increasing Subsequence

Problem: Given a sequence A of unique numbers. We are looking for the length of the longest strictly increasing subsequence (i.e by deleting some or no elements).

Solution:

Our subproblem becomes $\text{LIS}(i) :=$ Lenght of LIS which starts and includes A_i. Our recurrence becomes $\text{LIS}(i) = \max\{1 + \text{LIS}(j) \mid j \in [i+1, n] \wedge A_i < A_j\}$.

Runtime: $O(n^2)$

5 Graph Theory

A graph G is a set of vertices (nodes) and edges (connections) $G = (V, E)$. The edges can be directed or undirected, making the graph directed or undirected.

5.1 Terminology and Properties

- **Walk:** A sequence of connected nodes (repetition allowed)

- **Path:** A walk where no node is repeated except possibly the start and end node.
- **Cycle:** A path that starts and ends at the same node.
- **Eulerian:**
  - **Walk:** A walk traversing every edge exactly once.
  - **Path:** An Eulerian walk which is a path.
  - **Cycle:** An Eulerian path which is a cycle.
- **Hamiltonian:**
  - **Walk:** A walk that visits every node exactly once.
  - **Path:** An Hamiltonian walk which is a path.
  - **Cycle:** An Hamiltonian path which is a cycle.

Sum of degrees: Undirected: $\sum_{v \in V} \deg(v) = 2\ |E|$, Directed: $\sum_{v \in V} \deg_{\text{in}}(v) + \deg_{\text{out}}(v) = 2\ |E|$.

5.2 Eulerian

Existence:

- **Cycle:** For *undirected* iff 1) All nodes haven even degree 2) Graph is connected
- **Path:** For *undirected* iff 1) Exactly two nodes have odd degrees 2) Graph is connected.
- **Cycle:** For *directed* iff 1) In and Out degree are same 2) Graph is strongly connected
- **Path:** For *directed* iff 1) Exactly one node has $\deg_{\text{out}} - \deg_{\text{in}} = 1$ (start) 2) Exactly one node has $\deg_{\text{in}} - \deg_{\text{out}} = 1$ (end) 3) All other nodes have equal in and out degrees 4) Graph is connected

Construction (Undirected, Cycle or Path):

1. Check Eulerian properties (gives you info about what to expect)
2. For cycle: Start at any node, For path: Start at odd-degree node
3. Randomly walk around until stumbling onto start node. This creates a subcycle.
4. Find a node with unused edges on the current cycle. Run the previos step to generate a new subcycle. Merge that subcycle with our current subcycle.
5. Repeat until no edges remain. Output the merged cycle.

Runtime: $O(|E|)$

5.3 Hamiltonian

No straightforward existence criteria, no efficient algorithm.

5.4 Topological Sort

Linearly order nodes of a DAG such that for every edge `u -> v`. The idea is:

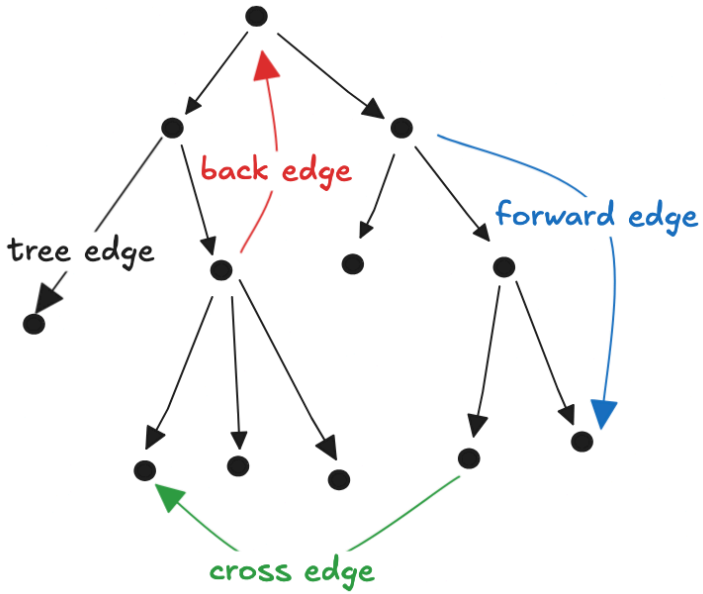
1. Use **DFS**. Append a node to the result **after (!) visiting all successors** (postorder).
2. For any unvisited nodes run step 1.
3. Reverse the result for topological order.

```python
def visit(u, topo_order):
  u.mark()
  for v in u.succesor():
    if not v.isMarked():
      visit(v, topo_order)
  topo_order.append(u)

def toposort(G):
  topo_order = []
  for u in G.nodes():
    if not u.isMarked():
      visit(u, topo_order)
  return topo_order.reverse()
```

Tip: To check for cycles you can maintain a "marked" array where $0 =$ unvisited, $1 =$ visited, $2 =$ subtree visited. You can check for already visited nodes in the relaxation step.

Runtime: Matrix: $O(n^2)$, List: $O(n+m)$



6 Shortest Path Algorithms

6.1 Single Source Shortest Path

6.1.1 No Weights: BFS

Use **BFS** for unweighted graphs to find the shortest path from a source to all other nodes. The idea is:

1. **Start from the source node**: Make a distances array. All other nodes are at ∞.
2. **Explore neighbors**: Use a **queue** (FIFO) to explore each node's neighbors and relax them.

```python
def bfs(source, graph):
  dist = {node: float('inf') for node in graph}
  dist[source] = 0
  queue = deque([source])

  while queue:
      u = queue.popleft()
      for v in u.neighbors():
          if dist[v] == float('inf'):
              dist[v] = dist[u] + 1
              queue.append(v)
  return dist
```

Hacky Tip: If a queue is not available/allowed in Codeexpert, use a large array with two pointers (start, end), to make an ADT Queue.

Runtime: $O(n+m)$.

6.1.2 Weighted DAG: Toposort + Iterative Relaxation

Use **Toposort + Iterative Relaxation** if you are dealing with a weighted DAG. Negative weights are allowed since we don't have cycles.

```python
def algo(source, graph):
  order = toposort(graph)
  dist = [...]
  for u in order:
    for v in u.succesor():
      relax(u, v)
```

Runtime: $O(n+m)$

6.1.3 Non Negative: Dijkstra

Dijkstra's algorithm works by **greedily** expanding the closest nodes and ensuring that once a node's shortest distance is found, it is never updated.

1. **Start at the source**: Initialize the source node's distance as 0 and all other nodes as infinity.
2. **Use a priority queue (min-heap)** to always select the node with the **smallest known distance**.
3. **Relax edges**: For each selected node, update the distances to its neighbors. If a shorter path is found, update the neighbor's distance.
4. **Repeat**: Continue selecting the next closest node until all nodes have been visited.

```python
def dijkstra(source, graph):
  distances = [...]
  pq = MinHeap([(0, source)])
  while not pq.empty():
    u = pq.extractMin() # pop
    for v in u.successors()
      dist = distances[u] + weight(u, v)
      if dist < distances[v]:
        distances[v] = dist
        pq.add((dist, v))
  return distances
```

Runtime: Binary Heap: $O((n+m)\log n)$, Fibonacci Heap: $O(n \log n + m)$

6.1.4 Any Graph: Bellman-Ford

The **Bellman-Ford algorithm** works by relaxing the edges $(\forall u, \forall v (u \to v))$ iteratively $n-1$ times. If the n-th time we are able to relax, we have a negative weight cycle.

```python
def bellmanford(source, graph):
  distances = [...]
  for i in range(n-1):
    for u in graph:
      for v in u.succesor():
        relax(u, v)

  for u in graph:
    for v in u.succesor():
      if relax(u, v):
        return -1

  return distances
```

Runtime: $O(n \cdot m)$

6.2 All Source Shortest Paths

The **All-Pairs Shortest Paths (APSP)** problem finds the shortest paths between **all pairs of nodes** in a graph.

6.2.1 Floyd-Warshall

The **Floyd-Warshall algorithm** solves the APSP problem using dynamic programming. It works by iteratively considering each node as an intermediate step in paths between all pairs of nodes.

1. **Initialize distances**: $u \to v$ is set to $w(u \to v)$ if there is an edge, to 0 if $u \to u$, and ∞ otherwise.
2. **Dynamic Programming**: $D_{i,j} = \min\{D_{i,j}, D_{i,k} + D_{k,j} \mid \forall k \in V\}$

```python
def floyd_warshall(graph):
  dist = [...]
  for k in range(len(graph)):
    for i in range(len(graph)):
      for j in range(len(graph)):
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
  return dist
```

Negative cycles: To check for negative cycles, check if $D(v \to v) < 0 \forall v \in V$ after all iterations.

Runtime: $O(n^3)$, best for dense graphs.

6.2.2 Johnson

Johnson's algorithm solves the APSP problem efficiently for sparse graphs.

1. **Reweight edges**: Add a new node q connected to every node v with edge weight 0. Use **Bellman-Ford** from q to compute a potential function $h(v)$ for all nodes v. Reweight each edge (u, v) as: $w'(u, v) = w(u, v) + h(u) - h(v)$ This ensures all edge weights $w'(u, v) g = 0$.
2. **Run Dijkstra**: For each node $sinV$, run **Dijkstra's algorithm** to find shortest paths using $w'(u, v)$.
3. **Adjust distances**: Convert distances back to the original weights: $d(u, v) = d'(u, v) - h(u) + h(v)$

```python
def johnson(graph):
  graph.add_node('q')
  for node in graph.nodes:
    graph.add_edge('q', node, 0)

  h = bellman_ford(graph, 'q')
  if h is None:
    return "Negative weight cycle detected"

  for (u, v, w) in graph.edges:
    graph.update_edge(u, v, w + h[u] - h[v])

  distances = {}
  for node in graph.nodes:
    distances[node] = dijkstra(graph, node)

  for u in graph.nodes:
    for v in graph.nodes:
      # beware: unreachable means oo +- x = ??
      distances[u][v] = distances[u][v] - h[u] + h[v]

  return distances
```

Runtime: Reweighting (Bellman-Ford): $O(n \cdot m)$, Dijkstra for each node: $O(n \cdot m + n^2 \log n)$ (using Fibonacci Heap), Total: $O(n \cdot m + n^2 \log n)$

7 Minimum Spanning Trees

A **Minimum Spanning Tree (MST)** of a connected, weighted graph is a subset of the edges that connects all the vertices without any cycles and with the minimum possible total edge weight.

7.1 Boruvka

Boruvka's algorithm finds an MST by repeatedly adding the cheapest edge from each component to the MST. It works in the following steps:

1. **Initialization**: Start with each node as its own component.
2. **Find minimum edge**: For each component, find the cheapest edge connecting it to any other component.
3. **Merge components**: Add these edges to the MST and merge the components.
4. **Repeat**: Continue until only one component remains (the MST).

I suggest you don't implement this in code, there are better options.

Runtime: $O((n + m) \log n)$

7.2 Prim

Prim's algorithm finds an MST by starting from a random node and repeatedly adding the minimum weight edge that connects a node inside the MST to a node outside.

1. **Initialization**: Start with an arbitrary node as the MST.
2. **Select the smallest edge**: From the nodes in the MST, choose the smallest edge that connects to a node outside the MST.
3. **Add the edge**: Add the chosen edge and the new node to the MST.
4. **Repeat**: Continue this process until all nodes are included in the MST.

Here too, I suggest you don't implement this in code, there are better options.

Runtime: $O((n + m) \log n)$

7.3 Kruskal

Kruskal's algorithm finds an MST by sorting all edges in the graph by their weight and then adding edges one by one, ensuring no cycles are formed. It uses a **union-find** data structure to keep track of which nodes are in the same component.

```python
def kruskal(graph):
  mst = []
  # Sort edges by weight
    edges = sorted(graph.edges, key=lambda edge: edge[2])
  uf = UnionFind(graph.nodes)

  for u, v, weight in edges:
      if uf.find(u) != uf.find(v):
          mst.append((u, v, weight))
          uf.union(u, v)

  return mst
```

Runtime: Sorting: $O(m \log m) = O(m \log n)$, Union-Find: $O(\alpha(n)) = O(n)$, Total $O(m \log n)$

8 Runtime Analysis

8.1 Limits

Indeterminate Forms:

$\frac{0}{0}, \frac{\infty}{\infty}, \infty - \infty, 0 \cdot \infty, 1^\infty, \infty^0, 0^0$

L'Hôpital's Rule:

Applies for indeterminate cases. $\lim_{x \to c} \frac{f}{g} = \lim_{x \to c} \frac{f'}{g'}$.

Tips:

1. Factorize expressions or rationalize numerators/denominators.
2. Look for dominant terms (e.g., highest power of x in polynomials).
3. For $\infty - \infty$, rewrite terms with a common denominator or use substitutions.

8.2 Runtime Complexity Rankings

$1 < \log n < \sqrt{n} < n < n \log n = \log(n!) < n^2 < n^3 < \ldots < 2^n < n! < n^n$