Revision Notes for CO202 Algorithms II

Spring 2018

1 Order of Growth

- **Asymptotic bound**: $f(n)$ is order $\Theta(g(n))$ if there is $c_1, c_2, n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
- **Asymptotic upper bound**: $f(n)$ is order $O(g(n))$ if there is $c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- **Asymptotic lower bound**: $f(n)$ is order $\Omega(g(n))$ if there is $c, n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

Note: not average / best / worst case.

2 Divide and Conquer

1. **Divide** the problem into smaller sub-problems.
2. **Conquer** sub-problems by solving recursively (recursive case). If size is small enough, can be solved trivially (base case).
3. **Combine** solutions to sub-problems into final solution.

Recurrences

Equations that describe functions in terms of its value on smaller inputs. Assuming:

1. Trivial problems $n \leq c$ solved in constant time.
2. Division yields a sub-problems, each of size $1/b$.
3. Divide takes time $D(n)$ and combine $C(n)$.

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Solving Recurrences

1. **Substitution**: guess and use induction to prove. (Guess $O(f(n))$ and then prove $T(n) \leq cf(n)$). For the induction:
   (a) **Use constants** wherever necessary.
   (b) **Use strong induction**. Try assuming it holds for n/b.
   (c) Choose **any valid base case**.
   (d) If stuck, **strengthen inductive hypothesis**. Subtracting lower order terms can help (e.g. prove $T(n) \leq cn^2 - kn$ instead of $T(n) \leq cn^2$).
2. **Recursion Tree**: Convert recurrence into a tree whose nodes are costs at different levels.
   (a) Substitute directly into the tree.
   (b) Remember that $k^{\log_c n} = n^{\log_c k}$ and $\sum_{k=0}^{n} c^k = \frac{1-c^n}{1-c}$.
3. **Simple Master Theorem**: For $T(n) = aT(n/b) + \Theta(n^d)$:
   (a) If $d < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
   (b) If $d = \log_b a$, then $T(n) = \Theta\left(n^d \lg n\right)$.
   (c) If $d > \log_b a$, then $T(n) = \Theta\left(n^d\right)$.
4. **Generic Master Theorem**: For $T(n) = aT(n/b) + f(n)$, if for some constant $\epsilon > 0$:
   (a) $f(n) = O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
   (b) $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$.
   (c) $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$, then $T(n) = \Theta(f(n))$.

   For (c) the **regularity condition** must hold: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficielntly large n.

Example: Merge Sort

```
MERGE-SORT(A, p, r):
    if p < r:                        # more than one item?
        q = floor((p + r)/2)         # divide array
        MERGE-SORT(A, p, q)          # conquer 1st subarray
        MERGE-SORT(A, q + 1, r)      # conquer 2nd subarray
        MERGE(A, p, q, r)            # combine subarrays
```

The **combine** step:

```
MERGE(A, p, q, r):
n1 = q - p + 1                       # length of 1st subarray
n2 = r - q                           # length of 2nd subarray
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1:
```

```
    L[i] = A[p + i - 1]           # copy values to 1st array
for j = 1 to n2:
    R[j] = A[q + j]               # copy values to 2nd array
L[n1 + 1] = inf                   # set sentinel
R[n2 + 1] = inf                   # set sentinel
i = 1
j = 1
for k = p to r:                   # merge subarrays
    if L[i] <= R[j]:
        A[k] = L[i]
        i = i + 1
    else:
        A[k] = R[j]
        j = j + 1
```

Takes time:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) & n > 1 \end{cases} = \Theta(n \lg n)$$

3 Dynamic Programming

- Combines solutions to overlapping sub-problems.
- Saves its answer in a table to avoid re-computation.

Requirements

- **Optimal substructure**: Optimal solution contains optimal solution to sub-problems.
- **Overlapping sub-problems**: Solution combines solutions to overlapping sub-problems.

Developing a DP Algorithm

1. Characterise the structure of an optimal solution.
2. Recursively define the value of an optimal solution: **test your definition very carefully**.
3. Compute the value of an optimal solution, in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Top-Down with Memoisation vs. Bottom-Up

- Bottom-up is more efficient by a constant factor because there is no overhead for recursive calls.
- Bottom-up may benefit from optimal memory access.
- Top-down can avoid computing solutions of sub-problems that are not required.
- Top-down is 'more natural'.

Recording the Solution Keep an additional array to record which sub-problem was used.

Example 1: Rod Cutting

- A rod of length i is worth p_i.
- The maximum revenue, $r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$.

```
#### Top-Down with Memoisation
MEMOIZED-CUT-ROD(p, n):
    let r[0..n] be a new array
    for i = 0 to n:
        r[i] = -inf
    return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(n, p, r):
    if r[n] >= 0
        return r[n]
    if n == 0:
        q = 0
    else:
        q = -inf
        for q = 1 to n
            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)
    r[n] = q
    return q

#### Bottom-Up
BOTTOM-UP-CUT-ROD(p, n):
    let r[0..n] be a new array
    r[0] = 0
    for j = 1 to n:
        q = -inf
        for i = 1 to j:
            q = max(q, p[i] + r[j - i])
        r[j] = q
    return r[n]
```

Example 2: Longest Common Subsequence

Step 1: Find optimal structure. For $X = \langle x_1, \ldots, x_m \rangle$, $Y = \langle y_1, \ldots, y_n \rangle$ and $Z = \langle z_1, \ldots, z_k \rangle$, where Z is the LCS of X and Y:

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is LCS of X and Y_{n-1}.

Step 2: Define recursive solution. Where $l(i,j)$ is the length of an LCS of sequences $X_{i..m}$ and $Y_{j..n}$:

$$l(i,j) = \begin{cases} 0 & \text{if } i = m \text{ or } j = n \text{ (base case)} \\ l(i-1, j-1) + 1 & \text{if } i < m,\ j < n \text{ and } x_i = y_j \text{ (case 1)} \\ \max \begin{cases} l(i-1,j) & \text{(case 2)} \\ l(i, j-1) & \text{(case 3)} \end{cases} & \text{if } i < m,\ j < n \text{ and } x_i \neq y_j \text{)} \end{cases}$$

Steps 3 and 4: Compute value and construct solution.

Example 3: Levenshtein Distance Where $d(i,j)$ is the edit distance of sequences $X_{i..m}$ and $Y_{j..n}$:

$$d(i,j) = \begin{cases} \max(i,j) & \text{if } i = 0 \text{ or } j = 0 \text{ (base case)} \\ \min \begin{cases} d(i-1,j) + 1 & \text{(delete)} \\ d(i, j-1) + 1 & \text{(insert)} \\ d(i-1, j-1) & \text{(no-op)} \end{cases} & \text{if } i < m,\ j < n \text{ and } x_i = y_j \\ \min \begin{cases} d(i-1,j) + 1 & \text{(delete)} \\ d(i, j-1) + 1 & \text{(insert)} \\ d(i-1, j-1) + 1 & \text{(replace)} \end{cases} & \text{if } i < m,\ j < n \text{ and } x_i \neq y_j \end{cases}$$

4 Greedy Algorithms

- Applied to optimisation problems.
- When there is a choice, always make the choice that looks best at the moment (don't look ahead).

Requirements

1. **Optimal substructure**: Optimal solution contains optimal solutions to sub-problems.
2. **Greedy-choice property**: Globally optimal solution obtained through locally optimal choices.

Developing a Greedy Algorithm

1. Demonstrate optimal substructure.
2. Cast the problem as one in which making a choice only leaves one sub-problem to solve.
3. Prove the optimality of the solution when making greedy choices.

Example 1: Activity Selection Problem Given a set of proposed activities, $S = \{a_1, a_2, \ldots, a_n\}$, each a_i having a start time s_i and finish time f_i, which all use the same resource, find maximum number of mutually compatible activities.

1. **Greedy choice**: choose the activity a_k with earliest finish time.
2. Solve the sub-problem with $S_k = \{a_i \in S \text{ such that } s_i \geq f_k\}$.

```
# Assuming activities are ordered by finish time
ACTIVITY-SELECTOR(s, f):
    n = len(s)
    A = [s[0]]
    k = 0
    for m = 1 to n - 1:
        if s[m] >= f[k]:
            A = A + [s[m]]
            k = m
    return A
```

Example 2: Fractional Knapsack Problem **Greedy choice**: choose item with maximum value per unit weight.

5 Randomised Algorithms

Strategies

1. Randomise the **input** (e.g. random permutations — hiring problem).
2. Randomise the **computation** (e.g. random choices — quicksort / BST insert).

Benefits

1. Help avoid pathologic inputs.
2. Yield good expected running time.
3. Allow dealing with large input domains.

Example 1: Hiring Problem Randomise the input to reduce the chance of worst-case.

Example 2: Quicksort

1. **Divide**: partition the array `A[p..r]` into `A[p..q-1]` and `A[q+1..r]` with all elements less than or equal to and greater than or equal to `A[q]` respectively.
2. **Conquer**: sort the two subarrays recursively using quicksort.

Running time depends on whether partitioning is balanced or unbalanced:

- **Balanced**: runs asymptotically as fast as merge sort.

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$$

- **Unbalanced**: can run asymptotically as fast as insertion sort.

$$T(n) = T(n-1) + T(0) + \Theta(n) = \Theta(n^2)$$

Possible solution:

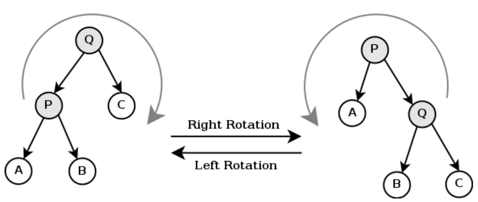
- Select pivot using random sampling (or median of 3 randomly selected samples).

Example 3: Binary Search Trees

- **BST property**: For every node y in the left subtree of x, y.key $\leq x$.key and every node z in the right subtree of x, y.key $\geq z$.key.
- Insert and search have complexity $O(h)$ where h is the height of the tree.

Expected height of randomly built BST with n keys is $O(\lg n)$. Try to achieve by:

- Randomly permute input (if known in advance).
- Recursively choose to insert at the root (by using **rotations**) with probability $1/$size or as a leaf.



Example 4: Skip Lists

- Maintain a hierarchy of linked sublists.
- Good for fast **search of ordered sequence**.

Not feasible to maintain ideal skip-list as we need to reorganise after each insert. Solution:

- An element in layer i appears with some probability in layer $i+1$.

Example 5: Find a Zero Bit From an array of 0 and 1 bits. Possible **generate and test** algorithms:

- **Las Vegas Algorithm**: choose an index randomly until you find a 0.
  - Always correct but unbounded resources.
- **Monte Carlo Algorithm**: choose up to k random indices attempting to find a 0.
  - Not always correct but bounded resources.

6 Visualising Algorithms

Provide **understanding** and **insights** into characteristics and support **debugging**.

6.1 Sorting

Visualisation Methods

- Animations.
- Weave visualisations.
- Leaning lines.

6.2 Random Sampling

Uniform Sampling

```
UNIFORM-SAMPLE(width, height):
    x = RANDOM-REAL(0, 1) * width
    y = RANDOM-REAL(0, 1) * height
    return (x, y)
```

Best-Candidate Sampling

```
BEST-CANDIDATE-SAMPLE(width, height, samples, n):
    best_candidate = (0, 0)
    best_distance  = 0
    for i = 1 to n:
        c = UNIFORM-SAMPLE(width, height)
        d = DISTANCE(FIND-CLOSEST(samples, c), c)
        if d > best_distance
            best_distance = d
            best_candidate = c
return best_candidate
```

Poisson-Disc Sampling

- Use a grid (each to contain a maximum of one sample) to speed up searches.
- Select a first sample randomly and make it **active**.
- Choose an **active** sample and generate up k points between radius r and $2r$ from the sample.
  - If a point is ever found further than r from all other samples (check using the grid), choose it and make it **active**.
  - Otherwise, make this point not **active**.

Visualisation Methods

- Histograms (x, y coordinates or minimum distances).
- Scatter plots.
- Voronoi diagrams.

6.3 Random Shuffling

Want to find **unbiased** algorithm (every permutation equally likely).

Fisher-Yates Shuffling

```
FISHER-YATES-SHUFFLE(A):
for i = n down to 2:
    j = RANDOM(1, i)    # make sure i is included
    SWAP(A[i], A[j])
```

Visualisation Methods

- Leaning lines.
- Swap matrices.

7 String Matching

For text $T[1..n]$, attempt to find a pattern $P[1..m]$ with $m \leq n$.

Definitions

- P occurs in T with **shift** s if $T[s+j] = P[j]$ for $1 \leq j \leq m$.
- $w \sqsubset x$ if w is a prefix of x.
- $w \sqsupset x$ if w is a suffix of x.
- The prefix $P[1..k]$ of a pattern $P[1..m]$ is denoted P_k.

If x and y are suffixes of z:

- If $|x| \leq |y|$, then x is a suffix of y.
- If $|x| \geq |y|$, then y is a suffix of x.
- If $|x| = |y|$, then $x = y$.

7.1 Knuth-Morris-Pratt Matching

- **Key Idea**: Go through T character by character, use a prefix function

$$\pi[q] = \max(k \text{ such that } k < q \text{ and } P_k \text{ is a suffix of } P_q)$$

Example:

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

- PREFIX-FUNCTION takes $O(m)$ and KMP-MATCHER $O(n)$ time.

```
KMP-MATCHER(T,P)
n = T.length
m = P.length
π = PREFIX-FUNCTION(P)
q = 0
for i = 1 to n
    while q > 0 and P[q+1] ≠ T[i]
        q = π[q]
    if P[q+1] == T[i]
        q = q+1
    if q == m
        PRINT(i-m)
        q = π[q]
```

```
PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k+1] ≠ P[q]
        k = π[k]
    if P[k+1] == P[q]
        k = k+1
    π[q] = k
return π
```

7.2 ~~Boyer-Moor Matching~~

- Start matching from the right of P. When matching breaks, shift P as far right as possible, maximising skipped comparisons.
- **Bad character rule**: Current character (β) in T doesn't match current character in P.
  - If β not in P, shift P so that it skips β.
  - If β is in P, shift P so it aligns that the right-most β in P.

For $P =$ ababaca:

c	a	b	c
bcr $[c]$	7	4	6

- **Good suffix rule**: A suffix of P has been matched in T up to a bad character in T.
  - If the match occurs somewhere else in P, shift to align it with the right-most occurrence in P.
  - If the match occurs nowhere else in the pattern, skip the pattern.
  - If the prefix of P matches the suffix of the match, align the prefix of P with the suffix of the match.

i	1	2	3	4	5	6	7	8
$P[i]$	a	b	a	b	a	c	a	ϵ
gsr $[i]$	6	6	6	6	6	2	1	/

i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ϵ
gsr $[i]$	5	5	2	5	1	/

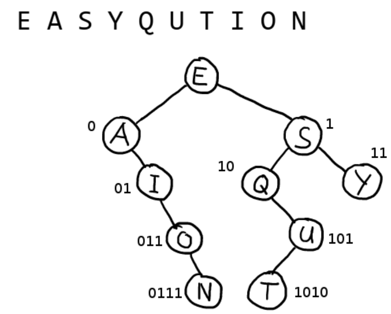
- BCR-TABLE and GSR-TABLE take $O(m)$ and BM-MATCHER $O(mn)$ time.

8 ~~Radix Search~~

Examine keys one piece at a time, rather than a full comparison.

Digital Search Trees Exactly the same as BSTs except left/right branching is not based on full comparisons, but selected bits of keys.

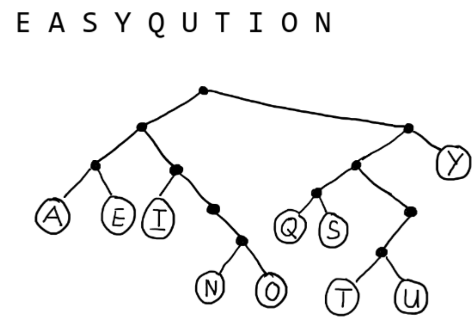
E	00101
A	00001
S	10011
Y	11001
Q	10001
U	10101
T	10100
I	01001
O	01111
N	01110



- Still compares whole keys.
- Maximum depth is the maximum key length (b).
- Keys with similar prefixes degrade performance.

Binary Search Tries Keys are only stored at the leaf nodes.

E	00101
A	00001
S	10011
Y	11001
Q	10001
U	10101
T	10100
I	01001
O	01111
N	01110



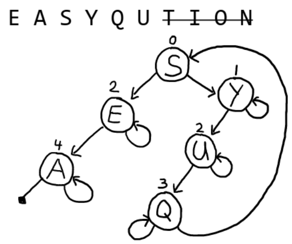
- Don't compare the full key until a leaf is reached.
- For n keys, average search requires $\lg n$ comparisons and b in worst case.
- One-way branching creates extra unnecessary nodes in the trie.
- Two different types of nodes means implementation is complex.

Patricia Tries

- Each node includes the index of the bit to be tested.

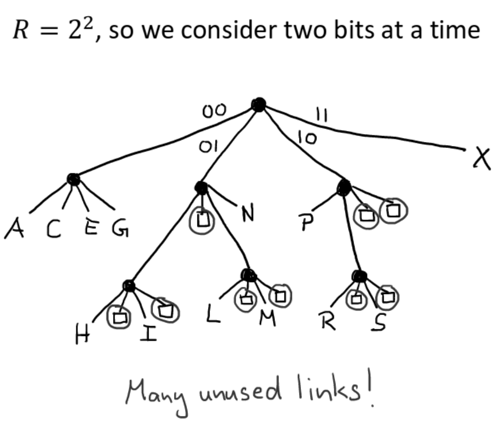
- Uses pointers up the tree instead of NULLs.
- **Search**: Follow pointers until they point at the **same level** or **up** the trie. Then check this node.

E	00101
A	00001
S	10011
Y	11001
Q	10001
U	10101
T	10100
I	01001
O	01111
N	01110



- Only has n nodes.
- Only requires about $\lg n$ bit comparisons.

Multiway Search Tries Examine r bits at a time, using nodes with $R = 2^r$ links.



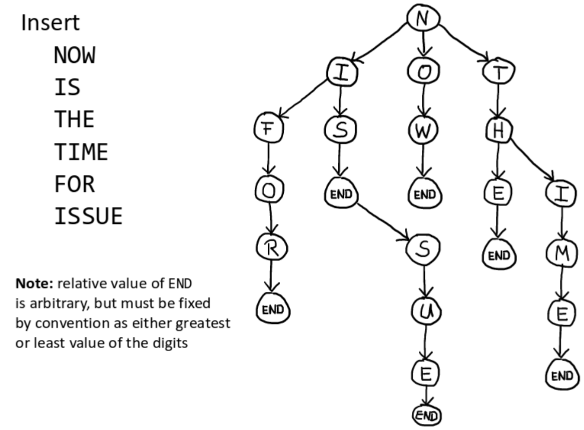
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100

- Requires $O(\log_R n)$ comparisons.
- Increased number of links leads to wasted space.

Existence Tries Special purpose multiway trie whose keys are data (i.e. a set).

- Keys are distinct and no key is prefix of another.
- Keys are of fixed length or have a termination digit.

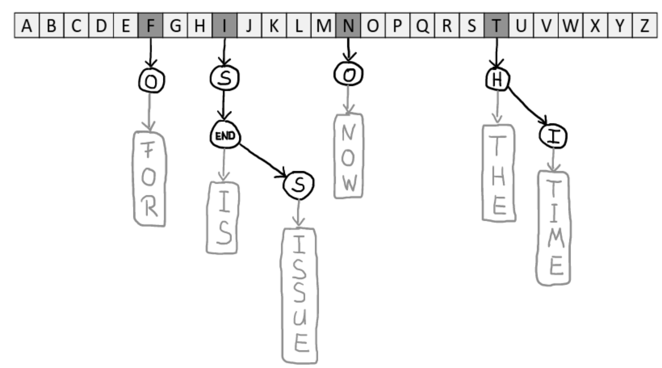
Ternary Search Tries Each node has three links (less than, equal to, greater than).



- Works well for non-randomness in key structure (e.g. URLs).
- Search misses tend to be efficient.
- Good for suffix trees (index into shared text).

Optimised TSTs

- **Wide head**: create table of R TSTs.
- **Compact tails**: store keys at leaves.



9 Graph Algorithms

- A graph $G = (V, E)$ can be represented as an adjacency-list or matrix.

- A tree:
  - Has $V-1$ edges and no cycles.
  - Has $V-1$ edges and is connected.
  - Is connected, but removing any edge disconnects it.
  - Is acyclic, but adding any edge creates a cycle.
  - Exactly one simple path connects any pair of vertices.

Breadth First Search Uses a queue, builds breadth-first trees, computes shortest paths.

Depth First Search Uses natural recursion, produces parenthesis structure.

- Edges classified as **tree**, **back** (point up the tree), **forward** (point down the tree) or **cross**.

Parenthesis Theorem For any two vertices u and v, one holds:

1. $[u.d, u.f]$ and $[v.d, v.f]$ are completely disjoint; neither u nor v is descendant of the other in the depth-first forest.
2. $[u.d, u.f]$ is entirely contained within $[v.d, v.f]$; u is a descendant of v in a depth-first tree.
3. $[u.d, u.f]$ and $[v.d, v.f]$ are completely disjoint; v is a descendant of u in a depth-first tree.

Topological Sort Sort vertices by finish time of DFS.

9.1 Spanning Tree

1. A tree T.
2. Includes all vertices V of $G = (V, E)$.
3. $T \subseteq E$.

Kruskal Algorithm Choose the edge of lowest weight as long as it doesn't create a cycle.

9.2 Shortest Path

Dijkstra Algorithm Choose the vertex with minimum distance, record predecessors.

- Doesn't work for negative weights.
- Time complexity: $O(|V| \log |V| + |E|)$ using min-priority queue.

Bellmann-Ford Algorithm Esentially Djisktra's algorithm applied n times (n is the number of nodes).

- Need to check for negative-weight cycles at end.
- Time complexity: $O(|V| |E|)$.

Floyd-Warshall Algorithm

$$D_{ij}^{(k)} = \begin{cases} w_{ij} & k = 0 \\ \min\left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right) & k \geq 1 \end{cases}$$

- Time complexity: $O\left(|V|^3\right)$.

Johnson's Algorithm Better for sparse graphs.

1. Add a new node q connected to all other nodes with a zero-weight edge.
2. Use Bellman-Ford, starting from q to get a minimum weight $h(v)$ for each node.
3. Reweight the edges in the original graph — $w'(u, v) = w(u, v) + h(u) - h(v)$.
4. Use Dijsktra algorithm.

9.3 Maximum Flow

- Remove anti-parallel edges using **auxiliary vertices**.
- Deal with multiples sources and sinks by introducing **supersources** / **supersinks**.

Constraints

- **Capacity constraint**: for all edges, flow must be less weight.
- **Flow conservation**: for all non-source/sink vertices, flow in = flow out.

Ford-Fulkerson Method While there exists an **augmenting path** p in the **residual network**, augment flow f along p.

Cuts If f is the maximum flow G $|f| = c(S, T)$ for some minimum cut (S, T) in G.