Cheat Sheet / Study Notes - Efficient Algorithms

Paul

February 17, 2025

1 General Theory

1.1 Asymptotic Notation

The set of functions that asymptotically grow not faster than $g(n)$ are:

$$\mathcal{O}(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c, n_0 \in \mathbb{R}^+ \text{ such that } \forall n \geq n_0 : \boldsymbol{f(n) \leq c \cdot g(n)}\}$$

The set of functions that asymptotically grow not slower than $g(n)$ are:

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c, n_0 \in \mathbb{R}^+ \text{ such that } \forall n \geq n_0 : \boldsymbol{f(n) \geq c \cdot g(n)}\}$$

The set of functions that asymptotically grow at the same rate as $g(n)$ are:

$$\Theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n))$$

1.2 Power Series

The geometric series is defined as:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ for } |x| < 1$$

For any constant c we have:

$$\sum_{i=0}^{\infty} (c \cdot x)^i = \frac{1}{1-c \cdot x} \text{ for } |c \cdot x| < 1$$

The exponential function is defined as:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

And for a constant c we have:

$$e^{c \cdot x} = \sum_{i=0}^{\infty} \frac{(c \cdot x)^i}{i!}$$

2 Recursion

2.1 Master Theorem

Let $a \geq 1$, $b > 1$, $\epsilon > 0$ then

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

then:

- If $f(n) = \mathcal{O}\left(n^{\log_b(a) - \boldsymbol{\epsilon}}\right)$ then $T(n) = \Theta\left(n^{\log_b a}\right)$
- If $f(n) = \Theta\left(n^{\log_b(a)} \boldsymbol{(\log n)^k}\right)$ then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$
- If $f(n) = \Omega\left(n^{\log_b(a) + \boldsymbol{\epsilon}}\right)$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and sufficiently large n then $T(n) = \Theta(f(n))$

2.2 Proof by Induction for Recursion

To show that $f(n) = \Theta(g(n))$ we perform the following algorithm:

- Guess that $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$
- Prove that $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$ by induction
- The inductive hypothesis is that $T(n) = \mathcal{O}(g(n))$, i.e. $T(n) \leq c \cdot g(n)$ for some constant c and sufficiently large n
- Analogously for $\Omega(g(n))$

2.3 Linear Homogeneous Recurrence Relations

Given the recursion $a_n = \sum_{i=1}^{k} c_i a_{n-i}$ we can solve it by finding the characteristic polynomial $p(x) = x^k - \sum_{i=1}^{k} c_i x^{k-i}$ and then solving it for the roots $x_1, x_2, \ldots, x_k$. The general solution is then $a_n = \sum_{i=1}^{k} \alpha_i x_i^n$ where the α_i are determined by the initial conditions. For roots x_i with multiplicity m_i we have to include terms of the form $n^j x_i^n$ for $j = 0, 1, \ldots, m_i - 1$ in the general solution. Example:

$$\begin{aligned} a_n &= 3a_{n-2} - 2a_{n-3} \\ \mathcal{X}(\lambda) &= \lambda^3 - 3\lambda + 2 \\ &= (\lambda + 1)(\lambda + 1)(\lambda - 2) \end{aligned}$$

With the initial conditions $a_0 = 3, a_1 = 2, a_2 = 1$ we get:

$$a_n = \alpha(-1)^n + n \cdot \beta(-1)^n + \gamma 2^n$$

Solving for the constants:

$$\begin{aligned} \alpha + 0 + \gamma &= 3 & &\Rightarrow \alpha = 3 - \gamma \\ -\alpha - \beta + 2\gamma &= 2 & &\Rightarrow \beta = 3\gamma - 5 \\ \alpha + 2\beta + 4\gamma &= 11 & &\Rightarrow \gamma = 2 \end{aligned}$$

And therefore the solution is:

$$\begin{aligned} a_n &= (-1)^n + n(-1)^n + 2 \cdot 2^n \\ &= (-1)^n (1 + n) + 2^{n+1} \end{aligned}$$

2.4 Inhomogeneous Recurrence Relations

Given the recursion $a_n = \sum_{i=1}^{k} c_i a_{n-i} + f(n)$ we can solve it by transforming it into a homogeneous recursion. To do this we iteratively substitute the recursion into itself until we get a homogeneous recursion. For example:

$$\begin{aligned} a_n &= a_{n-1} + n^2 \\ a_{n-1} &= a_{n-2} + (n-1)^2 = a_{n-2} + n^2 - 2n + 1 \end{aligned}$$

Adding and subtracting the two equations we get:

$$\begin{aligned} a_n &= 2a_{n-1} + n^2 - a_{n-1} \\ &= 2a_{n-1} + n^2 - \left(a_{n-2} + n^2 - 2n + 1\right) \\ &= 2a_{n-1} - a_{n-2} + 2n - 1 \end{aligned}$$

Similarly for a_{n-1}

$$\begin{aligned}
a_{n-1} &= a_{n-2} + (n-1)^2 \\
&= a_{n-2} + n^2 - 2n + 1 \\
a_{n-2} &= a_{n-3} + (n-2)^2 \\
&= a_{n-3} + n^2 - 4n + 4 \\
a_{n-1} &= 2a_{n-2} + \left(n^2 - 2n + 1\right) - \left(a_{n-3} + n^2 - 4n + 4\right) \\
&= 2a_{n-2} - a_{n-3} + 2n - 3
\end{aligned}$$

And going back to the original equation:

$$\begin{aligned}
a_n &= 3a_{n-1} - a_{n-2} + 2n - 1 - (2a_{n-2} - a_{n-3} + 2n - 3) \\
&= 3a_{n-1} - 3a_{n-2} + a_{n-3} + 2
\end{aligned}$$

We have to repeat this process until we get a homogeneous equation.

$$a_{n-1} = 3a_{n-2} - 3a_{n-3} + a_{n-4} + 2$$

And finally:

$$\begin{aligned}
a_n &= 4a_{n-1} - 3a_{n-2} + a_{n-3} + 2 - a_{n-1} \\
&= 4a_{n-1} - 3a_{n-2} + a_{n-3} + 2 - (3a_{n-2} - 3a_{n-3} + a_{n-4} + 2) \\
&= 4a_{n-1} - 6a_{n-2} + 4a_{n-3} - a_{n-4}
\end{aligned}$$

2.5 Generating Functions

A generating function is a formal power series that encodes the sequence of coefficients of a sequence. For example the generating function of the Fibonacci sequence $a_n = a_{n-1} + a_{n-2}$ is:

$$A(z) = \sum_{n=0}^{\infty} a_n z^n = \frac{z}{1 - z - z^2}$$

where the coefficients of the series are the Fibonacci numbers.

We can solve recursion equations with generating functions by performing algebraic operations on the generating functions. For example:

$$\begin{aligned}
a_n &= a_{n-1} + 2^{n-1} \text{ with } a_0 = 2 \\
A(z) &= \sum_{n=0}^{\infty} a_n z^n \\
&= a_0 + \sum_{n=1}^{\infty} a_n z^n \\
&= 2 + \sum_{n=1}^{\infty} \left(a_{n-1} + 2^{n-1}\right) z^n \\
&= 2 + \sum_{n=1}^{\infty} a_{n-1} z^n + \sum_{n=1}^{\infty} 2^{n-1} z^n \\
&= 2 + z \sum_{n=1}^{\infty} a_{n-1} z^{n-1} + z \sum_{n=1}^{\infty} 2^{n-1} z^{n-1} \\
&= 2 + z \sum_{n=0}^{\infty} a_n z^n + z \sum_{n=0}^{\infty} 2^n z^n \\
&= 2 + zA(z) + z \sum_{n=0}^{\infty} 2^n z^n
\end{aligned}$$

And therefore $(1+z)A(z) = 2 + z\sum_{n=0}^{\infty} 2^n z^n$, which we can solve for $A(z)$:

$$\begin{aligned}(1+z)A(z) &= 2 + z\sum_{n=0}^{\infty}(2z)^n \\ &= 2 + z\frac{1}{1-2z} \\ &= \frac{2-3z}{1-2z}\end{aligned}$$

And therefore:

$$\begin{aligned}A(z) &= \frac{2-3z}{(1-z)(1-2z)} \\ &= \frac{(1-z)+(1-2z)}{(1-z)(1-2z)} \\ &= \frac{1}{1-z} + \frac{1}{1-2z} \\ &= \sum_{n=0}^{\infty} z^n + \sum_{n=0}^{\infty} 2^n z^n \\ &= \sum_{n=0}^{\infty} \left(1+2^n\right) z^n\end{aligned}$$

And therefore $a_n = 1 + 2^n$.

3 Tree Structures

3.1 Binary Search Tree

A binary search tree is a binary tree where each node has a key and the keys are ordered such that for each node, all keys in the left subtree are less than the key of the node and all keys in the right subtree are greater than the key of the node.

Biary Search Tree 1 TREE-SEARCH(x, k): Go left if the key is less than the current node, go right if the key is greater than the current node. The average case time complexity is $\mathcal{O}(\log n)$ and the worst case time complexity is $\mathcal{O}(n)$.

```
if x = NULL or k = key[x] then
    return x
end if
if k < key[x] then
    return TREE-SEARCH(left[x], k)
else
    return TREE-SEARCH(right[x], k)
end if
```

Biary Search Tree 2 TREEMIN(x): Go left until you reach the leftmost node. The time complexity is $\mathcal{O}(\log n)$ with a worst case time complexity of $\mathcal{O}(n)$.

```
if left[x] = NULL or x = NULL then
    return x
end if
return TREEMIN(left[x])
```

3.2 Red-Black Trees

A red-black tree is a binary search tree with the following properties:

Biary Search Tree 3 TreeSucc(x): If the right subtree is not empty, return the minimum of the right subtree. Otherwise, go up the tree until you reach a node that is the left child of its parent. The time complexity is $\mathcal{O}(\log n)$ with a worst case time complexity of $\mathcal{O}(n)$.

```
if right[x] ≠ NULL then
    return TreeMin(right[x])
end if
y ← p[x]
while y ≠ NULL and x = right[y] do
    x ← y
    y ← p[y]
end while
return y
```

Biary Search Tree 4 TreeInsert(x, z): Insert a node into the binary search tree. The time complexity is $\mathcal{O}(\log n)$ with a worst case time complexity of $\mathcal{O}(n)$. The idea is to traverse the tree until we reach a leaf node and then insert the new node there.

```
y ← NULL
x ← root[T]
while x ≠ NULL do
    y ← x
    if key[z] < key[x] then
        x ← left[x]
    else
        x ← right[x]
    end if
end while
p[z] = y
if y = NULL then
    root[T] = z
else if key[z] < key[y] then
    left[y] = z
else
    right[y] = z
end if
```

Biary Search Tree 5 DELETE(x, z): Delete a node from the binary search tree. The time complexity is $\mathcal{O}(\log n)$ with a worst case time complexity of $\mathcal{O}(n)$. If the node has no children or only one child, we can simply remove the node. If the node has two children, we can replace it with its successor.

```
if left[z] = NULL or right[z] = NULL then
    y ← z
else
    y ← TREESUCC(z)
end if
if left[y] ≠ NULL then
    x ← left[y]
else
    x ← right[y]
end if
if x ≠ NULL then
    p[x] = p[y]
end if
if p[y] = NULL then
    root[T] = x
else if y = left[p[y]] then
    left[p[y]] = x
else
    right[p[y]] = x
end if
if y ≠ z then
    key[z] = key[y]
end if
return y
```

- The height is at most $\mathcal{O}(\log n)$
- The black-height of a node is the number of black nodes on the path from the node to the leaves, not counting the node itself. The black-height of the tree is the black-height of the root.
- A red-black tree of black-height h has at least $2^h - 1$ internal nodes.
- The root is black
- Every leaf is black
- For each node x, all simple paths from x to descendant leaves have the same number of black nodes
- Every red node has two black children
- Properties:
  - Every node is either red or black
  - The root is always black
  - Every leaf (NIL) is black
  - If a node is red, then both its children are black (no consecutive red nodes)
  - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes (black height)
- Insert operation:
  - Insert the node as in a regular BST
  - Color the new node red
  - Fix violations of Red-Black properties:
    - Case 1: Uncle is red - Recolor parent, uncle, and grandparent

        * Case 2: Uncle is black (triangle) - Rotate parent
        * Case 3: Uncle is black (line) - Rotate grandparent and recolor
    - Ensure root remains black
- Delete operation:
    - Perform standard BST deletion
    - If deleted node is black, fix double-black violation:
        * Case 1: Sibling is red - Recolor and rotate
        * Case 2: Sibling is black with black children - Recolor sibling
        * Case 3: Sibling is black with near child red - Rotate sibling's child and recolor
        * Case 4: Sibling is black with far child red - Rotate sibling and recolor
- Search operation:
    - Standard BST search, no rebalancing needed
    - Time complexity is $\mathcal{O}(\log n)$ due to guaranteed height balance

3.3 Splay Trees

A splay tree is a self-adjusting binary search tree with the property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up, and deletion in $\mathcal{O}(\log n)$ amortized time. The idea is to move the accessed node to the root of the tree by performing a sequence of rotations.

Splay Tree 6 SPLAY(X): Splay the node x to the root of the tree. The time complexity is $\mathcal{O}(\log n)$ amortized.

```
while x ≠ root do
    if parent(x) = root then
        ZIG(x) right rotation (or left rotation)
    else if x is left child and parent(x) is left child or x is right child and parent(x) is right child then
        ZIGZIG(x) right rotation grandparent, right rotation parent (or left-left rotation)
    else
        ZIGZAG(x) left rotation parent, right rotation grandparent (or right-left rotation)
    end if
end while
```

- For each operation (insert, search, delete), splay the node to the root of the tree
- If the node is not found, splay the last accessed node
- Delete operation:
    - Search for x
    - Splay(x)
    - Search for predecessor y of x. By definition this is the largest element in the left subtree of x
    - splay (y) on left subtree of x. By definition this will result in a left subtree with no right child
    - Join the right subtree of x as the child of y
- Insert operation:
    - Search for x
    - y is the last accessed node, which is either the predecessor or successor of x
    - splay(y)
    - insert x as the root of the tree. If y is the predecessor of x it becomes the left child, otherwise it becomes the right child.
- Search operation:
    - Search for x
    - splay(x), or the last accessed node if x is not found

4 Hashing

4.1 Hash Functions

- hash function: $h : U \to \{0, 1, \ldots, n-1\}$
- Universe U of keys and a table T of size n
- The probability of a collision in a uniform hash function when hashing m keys into a table of size n is $1 - e^{-\frac{m(m-1)}{2n}}$
- There are two types of dealing with collisions:
  - open addressing: when a collision occurs, the algorithm tries to find another slot in the table
  - chaining: when a collision occurs, the algorithm stores the key in a linked list
- A class of hash functions $\mathcal{H} = \{h : U \to T\}$ is called universal if
  - $\forall_{x \neq y} x, y \in U : |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{|T|}$
  - i.e. the probability of a collision is at most $\frac{1}{n}$

4.2 Open Addressing

We add a function f to the hash function h to get the probing sequence $h(k, i)$ where i is the number of probes.

- Linear Probing: $h(k, i) = (h'(k) + i) \mod n$
- Quadratic Probing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod n$
- Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod n$

4.3 Cuckoo Hashing

Worst case guarantee of $O(1)$ for lookups, insertions and deletions. Every key is guaranteed to be either in table one or table two as a direct hit.

Insertion steps:

- Idea is to eject the residing key, which in turn rejects the key in the alternate table until we find a nill position
- If $T_1 [h_1 (x)] = x$ or $T_2 [h_2 (x)] = x$ then return
- Otherwise do for a maximum of n steps:
  - exchange x and $T_1 [h_1 (x)]$. If $x = null$ return
  - exchange x and $T_2 [h_2 (x)]$. If $x = null$ return
- If it does not terminate in n steps, change the hash function and rehash everything
- Try inserting with the new hash tables

5 Heaps

5.1 Priority Queue

Dijkstra's Algorithm (Single-Source Shortest Path) The algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph. For a vertex set V and edge set E:

Initialize:
  $d[s] = 0$ (source distance)
  $d[v] = \infty$ (all other vertices)
  $Q = V$ (priority queue)

While $Q \neq \emptyset$:
  $u = \text{extract\_min}(Q)$
  For each neighbor v of u :
    If $d[v] > d[u] + w(u, v)$:
      $d[v] = d[u] + w(u, v)$

Time complexity: $O((|V| + |E|) \log |V|)$ with binary heap

Prim's Algorithm (Minimum Spanning Tree) Builds a minimum spanning tree by selecting edges with minimum weight that connect a vertex in the tree to a vertex outside the tree. For a graph $G = (V, E)$:

Initialize:
  $\text{key}[s] = 0$ (source vertex)
  $\text{key}[v] = \infty$ (all other vertices)
  $\text{parent}[v] = \text{NIL}$ (for all vertices)

While priority queue $Q \neq \emptyset$:
  $u = \text{extract\_min}(Q)$
  For each $v \in \text{Adj}[u]$:
    If $v \in Q$ and $w(u, v) < \text{key}[v]$:
      $\text{parent}[v] = u$
      $\text{key}[v] = w(u, v)$

Time complexity: $O(|E| \log |V|)$ with binary heap

Key differences: 1. Dijkstra's maintains distances from source: $d[v]$ 2. Prim's maintains weights of edges: key$[v]$ 3. Dijkstra's finds shortest paths from a source 4. Prim's builds a minimum spanning tree

The MST weight is given by:

$$MST_{weight} = \sum_{v \in V \setminus \{s\}} w(v, \text{parent}[v])$$

5.2 Binary-Heap

Complete binary tree, except for the last level with the property that the parent node is always smaller than their children.

- By definition the minimum element is the root
- The maximum element is one of the leaves
- Insertion of x:
  - Insert x as leftmost leave element.
  - Exchange x with parent untill the min-property is restored
- Deletion of x:
  - Replace x with the rightmost leave element y

- Fix min property of y

Let $A[1, \cdots, n]$ be an array representing a binary heap. Then:

- The parent of element i can be found in $A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$
- The left child of element i can be found at in $A[2i+1]$
- The right child of element i can be found at in $A[2i+2]$

5.3 Binomial Heap

5.4 Fibonacci Heap

- collection of min-heap trees
- root level has a doubly linked circular list
- root level has a min pointer to the minimum element. Since each tree is a min-heap, the min pointer is the minimum of all trees
- We keep track of wether the nodes are marked or not

Insert

- Insert a new node with key k into the root list
- If the new node is smaller than the current min, update the min pointer

Delete-Min

- Remove the minimum element (pointed to by min pointer)
- Add all its children to the root list
- Consolidate the root list:
  - Combine trees of the same degree until no two trees have the same degree
  - Create a temporary array indexed by tree degree
  - For each tree in root list, combine with existing tree of same degree if any
  - When combining, make the larger root a child of the smaller root
- Update the min pointer by finding minimum among remaining roots

Delete

- Decrease the key to negative infinity
- Cut the node from its parent and add it to the root list
- If parent is not root, mark it
- If parent was already marked:
  - Cut parent from its parent and add to root list
  - Continue cascading cuts up the tree until reaching either:
    - An unmarked node (mark it and stop)
    - A root (stop)
- Perform Delete-Min operation to remove the node with negative infinity key

Decrease-Key

- Decrease the key of the node to the new value
- If new key is smaller than parent's key:
  - Cut the node from its parent and add to root list
  - Perform cascading cuts on the parent as in Delete operation
- If new key is smaller than current minimum, update min pointer

6 Flows and Cuts

6.1 Minimum Cut

Let $G = (V, E)$ be a flow network with source s and sink t. A cut (S, T) is a partition of the vertex set V such that $s \in S$ and $t \in T$. The capacity of the cut is defined as:

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

6.2 Maximum Flow

Ford-Fulkerson Algorithm