Cheat Sheet / Study Notes - Efficient Algorithms

Paul

February 14, 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 $f(n) = \Theta\left(n^{\log_b(a) \cdot \boldsymbol{\log^k n}}\right)$ 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 Data 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

Insertion

- Insert the node z as in a binary search tree
- Color the node red
- Fix the tree by rotating and recoloring the nodes
- Case I: Red uncle
  - Recolor parent $p[z]$ and uncle $u[z]$ to black
  - Recolor grandparent $g[z]$ to red (unless it's root)
  - Move z pointer to grandparent and repeat if needed
  - This case preserves black height but may propagate red violation up
- Case II: Black uncle and z is left child:
  - Recolor parent $p[z]$ to black

  - Recolor grandparent $g[z]$ to red
  - Right rotate around grandparent
  - This case terminates - no further violations possible
- Case III: Black uncle and z is right child:
  - Left rotate around parent $p[z]$
  - Convert to Case II by making z point to old parent
  - Apply Case II fix
  - This requires two rotations but also terminates

I'll help create comprehensive notes for red-black tree deletion:

- Delete node y as in a binary search tree using transplant
- If y is black, fixing is needed since black height was reduced
- Let x be the node that moved into y's position
- Extra black is given to x (making it "double black" if already black)
- Cases to fix double black:
- Case I: Red sibling w:
  - Color sibling w black
  - Color parent $p[x]$ red
  - Left rotate at parent $p[x]$
  - New sibling is now black - converts to cases 2, 3, or 4
- Case II: Black sibling w with both black children:
  - Color sibling w red
  - Move double black up to parent $p[x]$
  - Repeat if needed (may propagate to root)
  - If reached red node, just color it black and done
- Case III: Black sibling w, red left child, black right child:
  - Color w's left child black
  - Color w red
  - Right rotate at w
  - Converts to Case IV
- Case IV: Black sibling w with red right child:
  - Color w same as parent $p[x]$
  - Color parent $p[x]$ black
  - Color w's right child black
  - Left rotate at parent $p[x]$
  - Remove extra black from x
  - This case terminates - all properties restored

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
```

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.

- 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