Summary

Sean's Algorithm Cheatsheet

> Take notes to forget. by Sean

https://seangone.gitbooks.io/sean-s-cheatsheet/content/

Algorithms

Algorithm Basics

- Backtracking

Time & Space Complexity Analysis

Physical Data Structures

- **Array**
  - `implement ArrayList`
  - Unsorted Array
    - **Sorting Algortihms**
    - `Quick sort`
    - `Merge sort`
    - `Comparsion`
  - Sorted Array
    - **Binary Search**
    - `Quick Select` - k-th smallest
  - **Two Pointers**
  - Char Array
    - **String**
    - `Remove EmptySpace in String`

        - `implementEncoding and Decoding`
- **Linked List**
    - `Reverse a Linked List`

Logical Data Structure

- `implementEncoding and Decoding`
- **Binary Tree**
    - Binary Search Tree
        - red black tree
        - `isBST`
        - `Insert/Delete in BST`
    - `check complete`
    - `pre/in/post order traversal` - recursive and iterative style
    - `N-ary tree serialize/deserialize`
    - `Sibiling jump in tree`
    - `Lowest Common Ancestor`
    - `Path Sum`
        - `Maximum Path Sum Binary Tree I`
- **Graph**
    - `Connectivity`
- **Queue & Stack**
- **Heap**
    - `Top K`
- **Hash Table**
    - `implement HashMap`
    - `implement LRUCache`
    - `implement HashCode()`
- **Trie tree**
- Segmentation tree

Other Problems

- Dynamic Programming
    - `Minimum Cuts For Palindromes`
- `Skyline`

Java Coding

- `StringBuilder`
- `implement iterator`
- Thread safe - Mutex synchronized
  - `Lock`

OOD

- `Parking Lot`
- `BlackJack`
- `Factory pattern`
- `Singleton`

System Design

- `In-memory File System`

Algorithm Basics

问题分析思路

- 使用**循环**拆成重复的子问题 - `loop`
- 使用**分类讨论**拆分多种情况/分支 - `if-else`
- 使用**分治法(divide and conquer)**拆成**可继续拆分的子问题**直到**最小情况(base case)** - `recursion / iteration`

Time & Space Complexity Analysis

分析方法

- **展开recursion tree进行分析**
  - **分析每个节点花的时间、空间。**
  - **分析每层花的总时间、空间。**
- **注意分析Worst Case**
- 分析空间时，注意**分析Call Stack占用的最大内存大小**
- 针对Tree问题
  - 不要针对树本身分析，要画出recursion tree
  - Eg. isIdentical 四叉树 in Class 03/24
- Amortized Time Complexity - 平均用时分析方法
  - 一次大量耗时的Action为之后很多次Action节省时间。
  - 注意：Amortized time仍然可能有worst case， Amortized分析只针对不适合对算法的单次操作进行分析。

常见的优化方向

- Space
  - **in-place** - 利用input的空间，减少额外空间的使用
- Time
  - **Dynamic Programming**
    - 记录子问题的Solution，方便解决相同子问题时利用
  - 减少内存分配释放次数。
    - Java的堆内存垃圾回收机制，回收内存需要时间开销。所以过多创建局部变量是开销更大的。
    - 减少局部堆内存分配，比如在merge sort中，可以只在初始化时创建一次临时变量，减少开销。
- 代码可读性
  - 尽可能避免使用global varible。 使用stateful的写法，尽量使用参数传递，避免使用field。

```
// 1) stateful
int[] helper;
public void mergeSort(int[] array){
  mergeSort(array, 0, array.length - 1);
}

// 2) stateless
public void mergetSort(int[] array){
  int[] helper = new int[array.length];
  mergeSort(array, helper, 0, array.length - 1);
}
```

Divide and Conquer - 分治法

适用问题

- 问题缩小到一定规模可以很容易的被解决，子问题相互独立。
- 问题可以分解为若干个规模较小的同类型子问题，即该问题具有**最优子结构**性质
- 利用该问题分解出的子问题的解可以合并为该问题的解。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

基本问题

- 是很多算法的基础（快速排序、归并排序）。
- 二叉树相关问题 - 使用二叉树的数据结构目的是，在解决问题时可以用分治法提高速度。

三个步骤

- **Divide 分解** - 将原问题分解为若干子问题，这些子问题都是原问题规模较小的实例。
- **Conquer 解决** - 递归地求解各子问题。如果子问题规模足够小，则直接求解该**base case**。
- **Combine 合并** - 将所有子问题的解合并为原问题的解。

Conquer 的顺序

- recursion 递归 / iteration 迭代
- DFS 纵向 / BFS 横向

Rewrite recursive implementation to iterative implementation

- All recursive algorithms can be written as a *iterative* version. Recursive version is a version using the system's call-function stack.
    - The size of call-function is 8M.
    - recursion缺点，可能会stackoverflow
    - 工业界场景中，很多应用是recursion写的。
- if it is a **tail recursion**, it can be easily written as iterative.
    - tail recursion: the recursive call is always the last execution statement
        - preorder/in/post都是dfs都不是tail recursion
- call stack vs. stack
- iteration写法中，如果需要回到目标节点的上一层，需要加入prev节点。

Backtracking Implementation

- Base case
    - can be terminated condition
- Recursive rule

DFS / Backtracking / Recursion problems:

- Subsets Problem
    - Subsets Solution

```
# 1. recursive helper parameter and return type
# for return type, if we needs to record a max or min,
# we can use a global varible or put it both in the parameters and return type.
# Eg. subsets problem
# thisnode = (subsetStart, startIndex)
# background = nums
def recurhelper(background, thisnode, results):
    # 2. recursive helper exit
    if GOAL_TEST(thisnode):
        results.add(thisnode)

    # 3. expand this node and call recurhelper recursively.
    # EXPAND function returns hard copy of nodes
    for node in EXPAND(thisnode):
        recurhelper(background, node, results)

    return results
```

Physical Data Structures

2.1. Array

Five types of problems

- **Sorting** Problems
  - See **Sorting Algorithms**.
- **Binary Search** in Sorted Array & Binary Search Tree
  - See **Binary Search**.
- Selectively Copying Problems
  - See **Two Pointers**
- Partition Problems
  - See **Two Pointers**.
- String Problems
  - String Problems can be seen as char array problems.
  - See **String**

Array vs. Linked List Comparison

- Memory Layout
  - Array: consecutive allocated, no **overhead**
  - Linked List: non-consecutive, **overhead** of multiple objects with the `next` reference
- Random access time
  - Array: $O(1)O(1)_{O(1)}$
  - Linked List: worst case $O(n)O(n)_{O(n)}$
- Search time in unsorted list
  - Array, Linked List: $O(n)O(n)_{O(n)}$
- Search time in sorted list
  - Array: $O(\log n)O(\log n)_{O(\log n)}$
  - Linked List: $O(n)O(n)_{O(n)}$

2.2. Sorting Algorithms

Compare

- **Stablity**
  - remains the **relative order of records with equal keys**
- **Cache Locality**
  - 访问的时候，cache会读取一部分连续的内存，连续访问容易出现cache hit，可以利用Cache读取快的特性。
  - 访问消耗：Memory 100ns, L1 cache 1ns

Sorting Algorithms	Time (worst case)	Time (average)	Space (worst case)	Spa
quick sort	$O(n2)O(n^2)_{O(n2)}$	$O(nlogn)O(nlogn)_{O(nlogn)}$	$O(n)O(n)_{O(n)}$	O(l
merge sort	$O(nlogn)O(nlogn)_{O(nlogn)}$	$O(nlogn)O(nlogn)_{O(nlogn)}$	$O(n)O(n)_{O(n)}$	O(r
heap sort	$O(nlogn)O(nlogn)_{O(nlogn)}$	$O(nlogn)O(nlogn)_{O(nlogn)}$	$O(1)O(1)_{O(1)}$	O(1

Selection Sort

不断地选择剩余元素中的最小者。

- Step 1 - 找到数组中最小元素并将其和数组第一个元素交换位置。
- Step 2 - 在剩下的元素中找到最小元素并将其与数组第二个元素交换，直至整个数组排序。

特点

- 比较次数 = (N-1)+(N-2)+(N-3)+...+2+1~N^2/2
- 交换次数 = N
- **Time** = $O(n2)O(n^2)_{O(n2)}$
  - 运行时间与输入分布无关
- 数据移动最少

Merge Sort

将两个有序对数组归并成一个更大的有序数组。通常做法为递归排序，并将两个不同的有序数组归并到第三个数组中。典型的分治法应用

特点

- Divide Time = $2n-1=O(n)$2n-1 = O(n)
- Merge Time = $n*log2n=O(nlogn)$n * log2n = O(nlogn)
- Time = $O(nlogn)$O(nlogn)
- Space = $2n-1=O(n)$2n-1 = O(n)
- 需要使用额外的空间来存储归并后的数据

k-way merge

- S1. iterative reduction
  - Time O(k^2 * n)
  - Space O(kn)
  - Best when only using memory
- S2. binary reduction
  - Time O(klogk * n)
  - Space O(kn)
- S3. heap k-way altogether
  - Time O(klogk * n)
  - Space O(k)
  - Best when data is to big to fit in memory, Least writing times
  - Online Algortihm (Can input n streams)
- vs. find common elements in k sorted arrays

Quick Sort

快排是一种采用分治思想的排序算法，大致分为三个步骤。

- **定基准** - 首先随机选择一个元素最为基准
- **划分区** — `XXXXXXP` -> `AAABBBP` -> `AAAPBBB`
  - 具体做法，一开始把基准元素换到最左边或者最右边，然后对剩下区域进行分区，最后把基准元素交换到中间。
  - **必须把基准元素放在中间**
    - **到递归下一层的子问题时，子问题不处理基准。保证每次调用必然会减少问题规模。**
    - 保证最终答案基准在排好序的位置。
    - `5, 3, 2, 4` 选择第一个元素作为pivot，如果子问题不去除掉基准会无法减小问题规模。
  - 具体做法，一开始把基准元素换到最左边或者最右边，然后对剩下区域进行分区，最后把基准元素交换到中间。
- **递归调用** - `AAAPBBB` -> `AAA` + `BBB`
  - **循环退出条件** `left >= right`
    - **必须包含等于**

特点

- **Worst Case** - `pivot` 选择不lucky，最差是O(n2)$O(n^2)$O(n2)

Rainbow Sort

```
AAABBBxxxxxxCCC
   |  |    |
   i  j    k
```

把序列分区为三个部分：

- [0,i−1][0, i-1]$[0,i-1]$ are all As.
- [i,j−1][i, j-1]$[i,j-1]$ are all Bs.
- [j,k][j, k]$[j,k]$ are all xes.
- [k+1,size−1][k+1, size-1]$[k+1,size-1]$ are all Cs.

算法步骤

- Step 1 - if `Array[j]` is A, `swap(i, j), i++, j++`
- Step 2 - if `Array[j]` is B, `j++`
- Step 3 - if `Array[j]` is C, `swap(j, k), k--`

特点

- Time = O(n)O(n)$_{O(n)}$

Heap Sort

- Step 1 - heapify
- Step 2 - pop the heap for k times
  - poll出来放在最后，Inplace地利用原数组。

特点

- typical runtime overhead 额外运行时间 - 建立堆需要额外的时间
- poor spatial locality, might swap from first to last (prefer to access data nearby / poor use of cache memory)
- **hard to parallelize/distribute**

2.3. Binary Search

- **Idea - 不断地排除一半错误选项**
  - 使用**循环** `loop` 不断地移动指针 `curr` 。
  - 使用**分类讨论** `if-else` 控制指针 `curr` 的进入其中一个分支，扔掉一半的选项。
- **适用问题范围**
  - **Typical problems**
    - Search problems in **Binary Search Tree** or **Sorted Array**
  - 每次循环会减少解空间，不能死循环；不能在循环中，把正确答案排除掉。
- **循环进入条件**
  - `left + 1 < right`
    - 取决于最后一次循环是否需要检查左右两个pivot确定答案
    - 需要做post processing
  - `left < right`
  - `left <= right`
- **调试循环条件的方法**
  - **使用一个元素的input调试，检测死循环**
  - 使用如下的会触发边界条件的例子测试, Eg. `[0]` , `[0, 1]` , `[0, 1, 2]`
- **二分计算**
  - `mid = start + (end - start) / 2`

- **排除一半的错误选项**
  - 判断条件
    - `nums[mid] < target` - 和mid直接相关
    - `cnt < k` - 和mid间接相关
      - LC 719
  - 经典问题
    - Binary Reduction in two sorted array
    - Quick Select - find k-th smallest in unsorted array
    - Jump out and jump in an array with unknown size

Binary Search Problems

P1. Find the largest element that is smaller than target

```
public int largestSmaller(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    int left = 0, right = nums.length - 1;
    # Here, the entry condition should be left + 1 < right.
    # because when left == 0 and right == 1, then mid = 0
    # and the start may not be updated.
    while (left + 1 < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid;
        } else {
            right = mid - 1; // IMPORTANT
        }
    }
    if (nums[right] < target) {
        return right;
    }
    return left;
}
```

```
public int largestSmaller(int[] nums, int target) {
    if (nums == null || nums.length == 0) {
        return -1;
    }
    int left = 0, right = nums.length - 1;
    int res = -1;
    # Here, the entry condition should be left + 1 < right.
    # because when left == 0 and right == 1, then mid = 0
    # and the start may not be updated.
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            res = mid;
            left = mid + 1;
        } else {
            right = mid - 1; // IMPORTANT
        }
    }
    return res;
}
```

P2. Find smallest in a rotated sorted array

- test cases
  - `2 4 5 6 0 1`
  - `6 0 1 2 4 5`
- cases
  - case 1: `array[mid] > array[right]` => `array[left:mid]` is sorted and continue to search in `array[mid+1:right]`
  - case 2: `array[mid] < array[right]` => `array[mid:right]` is sorted and continue to search in `array[left:mid]`
- if we have duplication in input
  - case 3: `array[mid] == array[right]`
    - worst case - all the elements are the same

Follow-up. Find the K closest number to target

- Find largest smaller or equal number.
- Find the kth cloest number using linear scan or binary elimination

P3. Find mountain peak in array

- test cases
  - `1 3 7 23 57 ... 100 99 86 44 32 21`
- 根据单调性来判断

Jump out and jump in an array with unknown size

Binary Search in a sorted array with unknown size

- Step 1 - jump out
- Step 2 - jump in

	step = 2	**step = 10**
Worst case	$n=2k-1+1$n=2^{k-1}+1$n=2k−1+1	$n=10k-1+1$n=10^{k-1}+1$n=10k−1+1
Jump out - Time	$log2n$log_{2}n$log2n	$log10n$log_{10}n$log10n
Jump in - Time	$log210n$log_{2}10n$log210n	$log22n$log_{2}2n$log22n

Binary Reduction in two sorted array

find the kth smallest / median in two sorted array

Binary Reduction on candidates

- xxxxxxxxxx XXXXXXXXXX
- 0 m
- yyyyyyyyyy YYYYYYYYYY
- 0 n
- move m or n to the right until $m+n==k-1$m + n == k - 1$m+n==k−1
  - choose one from m and n by comparing the A[m2] and B[n2]
  - moving step initially set to `k`

    - moving step decrease by half for each time `k -> k - k/2`

Quick Select in unsorted array

find kth in unsorted array

- `xxxxxxxxxN` -> `xxxxxxxNyyy`
- partition the array into two parts
- remove one part from the two
- keep doing until the pivot is exactly at the kth position

Binary Search in lazy unknown array

LC 719. Find K-th Smallest Pair Distance

- Binary Search + Lazily Count Pair Differences in sorted array
- Binary Search + Bucket sort

2.4. Two Pointers

[jiuzhang Chapter 7]

- 使用 `Pivot` 分隔数组。
- 使用**循环** `loop` 不断地移动指针 `pivot` 。
- 使用**分类讨论** `if-else` 控制指针 `pivot` 的移动。

适用问题

- 要求space complexity为O(1)$O(1)_{O(1)}$的问题

问题类型

- **同向双指针**
    - Array Deduplication
    - Remove leading/tailing spaces
    - Two Sum
- 相向双指针
    - Cannot reserve the initial order of the array
- Partition - Quick Select
    - k-th smallest

- 双数组双指针
  - k-th smallest in two sorted array

同向双指针

Problems:

- P1. Remove Particular Char
- P2. Remove all leading/trailing and duplicate empty spaces
- P3. Remove duplicated and adjacent letters but keep at most one
  - Follow-up. Remove duplicated and adjacent letters but keep at most two
  - Follow-up. Remove duplicated and adjacent letters and keep none
    - use two fast pointers to check the duplicated elements
- P4. Remove duplicated and adjacent letters repeatedly
- Use extra data structure, eg. array/queue/stack
  - compare the end of the Queue and the fast pointed element
  - compare the top of the Stack and the fast pointed element
- **In-place** way of two pointers
  - **a good way of split the array: [0, s), [s, f), [f, len)**
    - imagine the index of the slow pointer is the length of new array
    - if excluding the slow pointer, we do not need to think about too many corner cases
    - usually comes with `s++`
    - if including the slow pointer, use ++s
  - move the fast pointer for each time
  - not necessarily move the slow pointer
- P5. Count pairs with a difference no more than T
  - LC 719. Find K-th Smallest Pair Distance
  - O(n)

相向双指针

2-Sum

Two Pointers in two arrays

Common element problems

Assumptions:

- optimized time or space?
- sorted or unsorted?
- duplicate?
- keep one or all?
- data size
- data type

What about optimized time?

- if we only keep one element: use a hashset
- time O(n) space O(n)

What if the arrays are already sorted?

- use two pointers
- time O(n) space O(1)

What if the arrays are already sorted, but the size are very different(m<

- binary search
- for each element in the small array, do a binary search in the large array
- time O(m log n)

What if unsorted, but we want to optimize space?

- sort them first
- what if the data range is very limited?
    - array[26]

find common elements in 3 sorted arrays of similar size

find common elements in k sorted arrays of similar size

Different from **k-way merge**

Solution 1: iterative

- time O(kn)
- space O(n)

Solution 2: Binary Reduction

- time (1 + 1/2 + 1/4 + ...) * kn = O(kn)
- space O(kn)

// Not an useful method Solution 3: Min Heap

- time O(kn * logk)
- space

2.5. String

Popular Represenatation of Characters

- ASCII
  - A == 65, a == 97
- Unicode

Basic Problems

Idea

- 把string当成char array来思考，最好使用**in-place**写法
- 循环中间改变（删除）string，会影响for循环次数
  - Solution: `i--` / `i++`
- 避免频繁调用 `deleteCharAt(i)` ，因为花费O(n)O(n)O(n)时间。
  - 常使用**Two Pointers**的两个隔板把字符串分成三个部分：已经检查的 | 已经删去的 | 未检查的

P1. Char Removal

- `solution -> soltio`
- 字符串只减小不增大
- Solution: 使用**Two Pointers**的同向的两个 `pivot` 把字符串分成三个部分：已经检查的 | 已经删去的 | 未检查的

P2. De-dupliaction

- `aaaabbbbcc -> abc`
- 字符串只减小不增大
- Solution: 使用**Two Pointers**的同向的两个 `pivot` 把字符串分成三个部分：已经检查的 | 已经删去的 | 未检查的

P3. Sub-string Finding (strstr)

- regular method
- Robin-Carp (hash based string matching)
  - use Hash Function to get the hashcodes of all substrings, which has the same length as the pattern does.
- KMP (Knuth-Morris-Pratt)

P4. String Reversal (swap)

- `I love yahoo -> yahoo love I`
- Step 1: swap the whole sentence
- Step 2: swap every single word (two pointers)

P5. String Replacement

- Eg. replace empty space " " with "20%"
- Case 1: pattern.length >= replacement.length
  - use **Two Pointers**
- Case 2: pattern.length < replacement.length
  - Step 1: count how many times show up in the original string and extend the string to new string with empty space in the right.
  - Step 2: use **Two Pointers**
  - **从右往左**扫描利用输入的array。未扫描区域 | 待复制区域 | 已处理区域

Advanced Problems

P1. String Shuffling

- `ABCD1234 -> A1B2C3D4`
- Merge sort with custom comparator
- Reverse of Merge Sort - Space O(n)$O(n)$
- swap the 2nd and the 3rd part - Space O(1)$O(1)$
- follow-up: sorted array 被push到deque里面（任意方向），然后又被pop出来（任意方向），如何用O(n)$O(n)$的时间恢复sorted array?

P2. String Permutation

- CANNOT have duplicate string in the result
- DFS
- Situation 1: no duplicate letters in the input string
- Situation 2: exists duplicate letters in the input string

P3. String Decoding/Encoding

- `aaaabcc -> a4b1c2`
- 可能越界
  - 这个时候如果只出现一次，那么不使用数字
  - 先拓展string

P4. Sliding Window in a string

- Longest substring that contains only unique chars
- use a sliding window
- use a hash table to record the existence of characters

P4 follow-up. Sliding Window in a string

- find all anagrams （同构异形体） of a substring S2 in a long string S1
- anagrams: reorder of the original string, bcaa -> acab
- Solution 1
  - use a hash table to record the counts of all charcters
  - compare the hash table of each sliding window to that of the original pattern
  - if the two hash tables are the same, return
- Solution 2
  - use a hash table to record the difference of the appearances of each character.
  - use a counter to count the zeros in the hash table
  - if the counter equals 0, return

P4 follow-up. Sliding Window in a string

- given a 0-1 array, you can flip at most k '0's to '1's. Please find the longest subarray that consists of all '1's.
- use a sliding window
- use a counter to record the number of '0's and '1's.

P5. Matching (*, ?)

2.6. Linked List

Common mistakes:

- Never lose the control of the **head** pointer of the Linked List.
- When dereferencing a ListNode, ensure that pointer is not `NULL`.
- When changing the links of the nodes, be very careful and do not lose the **reference**.
    - record the next ListNode in advance

Common techniques:

- **use dummy node**
    - when **constructing new linked list**
    - when the **head** of the returning linked list **could be changed**
- **use a group of pointers when iterating a linked list**
    - eg. use `Prev` and `Cur` pointers
- **use Fast and Slow pointers**
    - to find the **middle** node of the linked list
    - to **find if there exists a circle** in the linked list
    - to **delete the n-th node**
    - Q: Why do we use fast and slow pointers instead of treverse two times?
        - A: Online algorithm vs. Offline algorithm: We can stop in the middle of the program and record the two nodes without losing information.
- Iterative Practice
    - The operations go from head to tail
    - do not do extra link change
- Recursive Practive
    - The operations go from tail to head
    - can set null first and then use the operations ahead to cover it
        - if we do not want to do post processing on the tail

---

Basic Problems

P1. Reverse a linked list

- Space Complexity: $O(1)$
- 改变link方向时，需要使用 `next` 或者 `prev` 去记录因为改变link可能丢失的节点。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseListRecursively(ListNode head) {
        if (head == null || head.next == null){
            return head;
        }

        ListNode newhead = reverseListRecursively(head.next);
        head.next.next = head;
        head.next = null;
        return newhead;
    }

    public ListNode reverseListIteratively(ListNode head){
        if (head == null || head.next == null){
            return head;
        }

        // prev head->next becomes prev<-head next
        // ListNode prev = null, next = head.next;
        // while (head != null){
        //     head.next = prev;
        //     if (next == null){
        //         return head;
        //     }
        //     // update loop control listnodes
        //     prev = head;
        //     head = next;
        //     next = next.next;
        // }
        // return prev;
        ListNode prev = null;
        while (head != null){
```

```
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }

    public ListNode reverseList(ListNode head) {
        // return reverseListRecursively(head);
        return reverseListIteratively(head);
    }
}
```

Follow-up. reverse a linked list by pair

P2. Find middle node of a linked list

- 快慢指针
- 尽量去找middle左边的节点，方便后续调用。

P3. determine if there exists a circle in a linked list

- 快慢指针

P3 Follow-up. 寻找环的开始

- 快慢指针相遇后，在head节点新放一只慢速指针，最终会和slow相遇。

P4. insert a node in a sorted linked list

- corner case - node插入在head之前，tail之后。
- 使用dummy head，因为node可能需要插在head之前。

P5. merge two linked list

- 使用dummy head

P6. Determine if two linked lists intersect

- Step1 - find the length of two linekd lists
- Step2 - get the difference and move the pointer `diff` steps forward in the longer linked list
- Step3 - move two pointers in the two linked lists until find the same node

---

Composed Problems

P5 Follow-up. change order

`N1->N2->...->Nn->null into N1->Nn->N2->Nn-1->...`

- Step 1 - find the middle node
- Step 2 - reverse the second hal
- Step 3 - merge two halves into one solution

P6. Partition List

- 把节点小于target放在左边，大于等于的放在右边。
- Step 1 - 使用两个dummy head
- Step 2 - 分配节点
- Step 3 - 合并两个linked lists
- 易错点 - 合并需要两个步骤，循环+剩余节点链接。

P7. Merge Sort

- Step 1 - Find the middle
- Step 2 - Split the list into two halves

- Step 3 - Recurse: sort each half
- Step 4 - Merge two halves

P8. Add Two Numbers

- Step 1 - Reverse two linked lists
- Step 2 - Add the number and create one new linked list
- Step 3 - Reverse the new linked list

P9. Check if a linked list is palindrome

- Step 1 - findMiddle
- Step 2 - Reverse
- Step 3 - Compare

Graph copy

P1. Graph copy with other pointer__

N1 -> N2 (next) N1 -> N3 (other) N2 -> N3 (next)

N3 is pointed twice, so we need a data structure to avoid duplicated copy.

Logical Data Structures

3.1. Binary Tree

Reference: Laioffer Class 5, Practice Class 7, 8

Definition

- **Binary Tree**
  - *Structure* - For each node, there are at most two children nodes.
- **Balanced Binary Tree**
  - *Structure* - **For every node**, the depth of the left and right subtrees differ by 1 or less. 对于每个节点，左右子树高度差都小于等于 1。
  - Height = O(log2(N))
- **Complete Binary Tree**
  - *Structure* - 除了最后一层的节点可以为null，其他都不为null。并且最后一层的节点都挤在左边。
- **Full / Perfect Binary Tree**
  - *Structure*
- **Binary Search Tree**
  - *Structure* - 没有要求。
  - *Value* - 任何节点，左子节点一定是小于父节点。柚子节点一定大于父节点。不可以等于父节点的值。
  - 中序遍历In-order的遍历结果是从小到大排序的序列。

General ways of solving problems about binary tree:

- **Divide and conquer** is the nature of binary tree. The problems can usually be divided into three parts:
  - solve sub problems for left/right subtree
  - solve sub problem for root
- **Iteration**
  - could be complicated for binary tree but we need to know how.

Recursion & Tree Problems

两种信息传递方式

- 把信息**从上往下**传 - 用**函数参数 parameters**传递
- 把信息**从下往上**传 - 用**返回值 return type**传递

算法的几个要点

- **What do you expect from lchild, rchild?**

- What do you want to tell to your lchild, rchild?
- How do you want to divide the problems into subproblems?
- **What do you want to do in the current layer?**
- **What do you want to report to your parent?**

几种类型

1. 在从上往下的过程中，就可以提前终止程序

取决于是否存在这样的可能，在还没有遍历到底部，就可以终止的情况。如果存在这种可能，说明判断的信息

- **P1. determine if it is BST**
    - @ parameter: the should-be range of root
    - 下层节点为BST = 上层节点为BST
    - 可能提前终止
- **P2. isSymmetric/isIdentical (Node root1, Node root2)**
    - @ return: boolean isSymmetric
    - 对称 = 该层key相同 && 该层结构相同
    - 上层对称 = 该层对称 + 下层对称
    - base case 最下层对称
    - 可能提前终止
- 只有在从下往上的过程里，才能结束程序 - 需要遍历到最底层、需要遍历所有节点的情况
- **P3. getHeight (Node root)**
    - @ return: int Height
    - 上层的层数 = Max(左子树的层数, 右子树的层数) + 1
    - base case 最下层高度为1
    - 上层对高度的计算依赖对下层的遍历
- **P4. isBalanced (Node root)**
    - @ return: boolean isbalanced
    - 上层节点是否平衡 = 下层节点平衡 && 左右子树高度差小于等于1
    - base case 最下层平衡 && 最下层高度为1
    - 上层对高度的计算和平衡的判断依赖对下层的遍历
- **P5. Assign the value of each note to be the total number of nodes that belong its left subtree. 求左子树数字之和**
- **P6. Lowest Common Ancestor 共同子祖先**

// case 1: root is one or two // 1.1: root.left == one or two || root.right == one or two (may not be answer if answer is not gurantee) // => return root // 1.2: root.left != one either two && root.right != one either two (may not be answer if answer is not gurantee) // => return root // ----> root is one or two => return root // case 2: root is not one or two // 2.1: one of root.left and root.right found one or two // => return lres || rres // 2.2: both of root.left and root.right found one or two // => return root // 2.3: return null

- **Follup-up. LCA (with parent node)**
  - Solution 1 - go to the parent recursively and get the number of level
  - Solution 2 - go to the parent from one of the node, and record the path of going up.
- **Follow-up. LCA (one or two is not guranteed in the tree)**
  - check if children return two if root == one
  - check if children return one if root == two
- **P7. Maximum Leaf-to-Leaf Path**
  - What do we expect from left child / right child? / What do we want to report to the parent node?
    - max single path in the left / right subtree
  - What do we want to do in the current layer?
    - update global_max
- **Follow-up. max Node-to-Node path sum**
- **Insert in BST**
  - *Problem* : return the new root after the change.
  - corner case: if the root is null, return the new node.
- **Delete in BST**
  - *Problem* : return the new root of the BST.
  - **Step 1**: find the node to be deleted
    - By transforming the problem into ONE sub problem that delete the key in one of the sub tree.
  - **Step 2**: delete the node
    - **case 1** - the node has no children.
    - **case 2** - the node has no left child.
    - **case 3** - the node has no right child.
    - **case 4** - the node has both left and right children.
      - The problem is transformed into three parts
      - **part 1**: find the largest node in the left subtree or the smallest node in the right subtree and record the replacement
      - **part 2**: delete the replacement in the tree - can be realized using recursive call. It must only hit one of case 1, case 2 and case 3.

        - **part 3**: set the children of the replacement

1. 人字形
2. **P1. Maximum Path Sum Binary Tree II**
    - update maximum的逻辑和recursion传递逻辑有区别
        - update maximum在每个节点，不仅要考虑人字形，也要考虑单臂型
        - recursion传递逻辑只考虑传递其中一条单臂信息
3. Path Problem in Binary Tree
4. 问法可以和一维数组问题结合，比如Maximum Subarray sum，比如2 Sum

Binary Search Tree Problems

- When?
    - 我们想要某种程度上保持数据的顺序，同时还需要(Key, Value)方式的查找。（红黑树）
    - 只需要某种程度上保持数据的顺序。（Heap）
- 为什么BST里面没有重复数据？
    - 工业界中，BST是用来查找的。

Time Complexity in BST

- `search` - worst case O(n)$O(n)_{O(n)}$, average O(logn)$O(logn)_{O(logn)}$
- `insert` - worst case O(n)$O(n)_{O(n)}$, average O(logn)$O(logn)_{O(logn)}$
- `remove` - worst case O(n)$O(n)_{O(n)}$, average O(logn)$O(logn)_{O(logn)}$
- Worst case happen if the binary tree is structured like a linked list.
- In **Balanced Binary Search Tree**, `search`, `insert` and `remove` operations are guaranteed to be O(logn)$O(logn)_{O(logn)}$. For example, **AVL Tree**, **Red-Black Tree** are balanced BST.

优化

- 怎么样让BST效率更高？ - 更加Balanced
- **balanced binary search tree**
    - Eg. AVL tree, Red-black tree, etc 但是面试不会考定义，可能会借这个定义考基本能力。
    - Red-Black tree Implementations
        - Guaranteed **O(log(n))** time cost for `containsKey`, `get`, `put`, `remove`
        - in Java: TreeMap/TreeSet
        - in C++: map/set

    - NavigableMap
        - Methods lowerEntry, floorEntry, ceilingEntry, and higherEntry return Map.Entry objects associated with keys respectively less than, less than or equal, greater than or equal, and greater than a given key, returning null if there is no such key

---

Treverse & Tree (Serialization) Problems

Coding 注意点

- Iteration写法中，如果需要回到目标节点的上一层，需要加入prev节点。
- 使用prev时，记得先更新prev，再更新curr

Problems

- **P1. Pre-order treverse**
    - *Order* : root -> root.left -> root.right
    - Remember to add the right child first, so the left child is popped first.
- **P2. In-order treverse**
    - *Order* : root.left -> root -> root.right
    - 使用Stack记录访问顺序 - 但是并不是遍历顺序
    - 画图，然后观察prev和curr的关系，从而分类讨论处理节点
- **P3. Post-order treverse**
    - *Order* : root.left -> root.right -> root
    - 使用Stack记录访问顺序 - 但是并不是遍历顺序
    - 画图，然后观察prev和curr的关系，从而分类讨论处理节点
- **P4. Level treverse**
    - *Order* : low level -> high level
    - BFS

Deserialization Problems

每一层需要得到左右子树各自的输入（比如是in-order和post-order）， 左右子树分别返回构造好的tree的root。

P1. Reconstruct a BST with post-order traverse

P2. Reconstruct a tree with level-order and in-order traverses

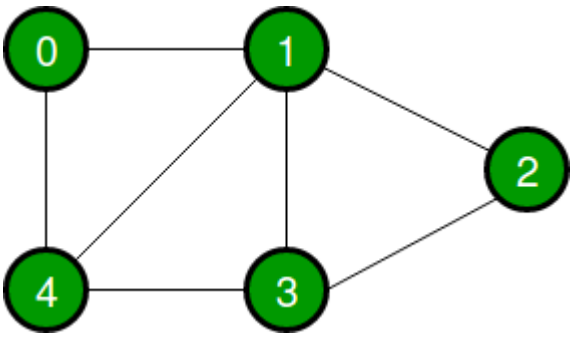
P3. Reconstruct a tree with pre-order and in-order traverses

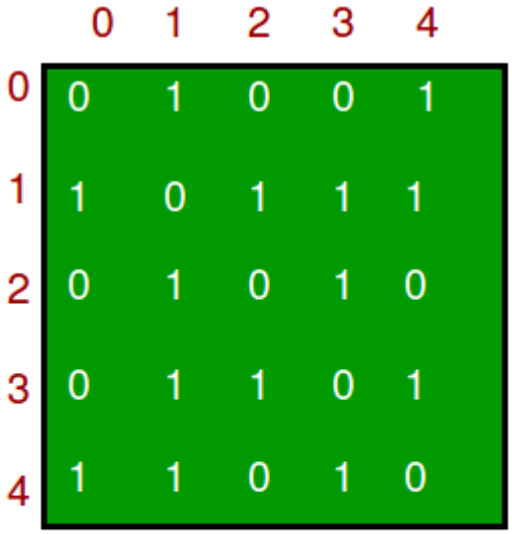
3.2. Graph

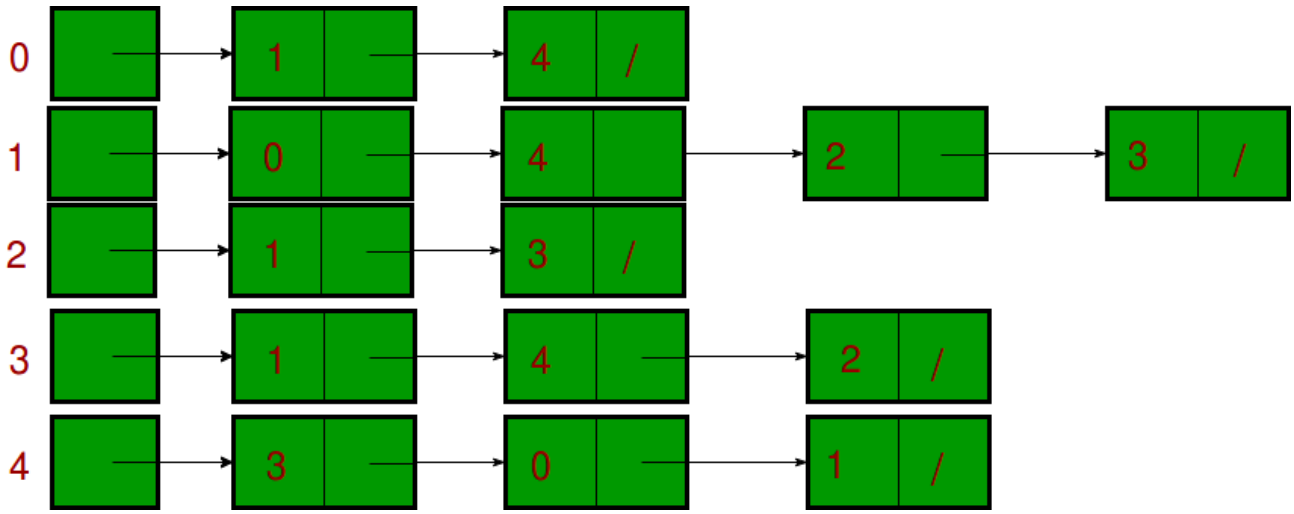
Representation

Graph can be represented using **Adjacency Matrix**, **Adjacency List** and **hash table**.

Following is an example undirected graph with 5 vertices.







Adjacency Matrix vs. Adjacency Lists

- Adjacency Matrix
  - Time: Edge removal $O(1)O(1)_{O(1)}$, Query of Edge $O(1)O(1)_{O(1)}$
  - Space: $O(V2)O(V^2)_{O(V\,2)}$
- Adjacency List
  - Time: Query of Edge $O(V)O(V)_{O(V\,)}$
  - Space: $O(V+E)O(V + E)_{O(V\,+E)}$

Graph Search Algorithms

Draw the Search Graph of Problems

- What does it store on each level?
- How many different states should we try to put on each level?

Tip

- Checking existed and Writing existed nodes should be put at the same place.

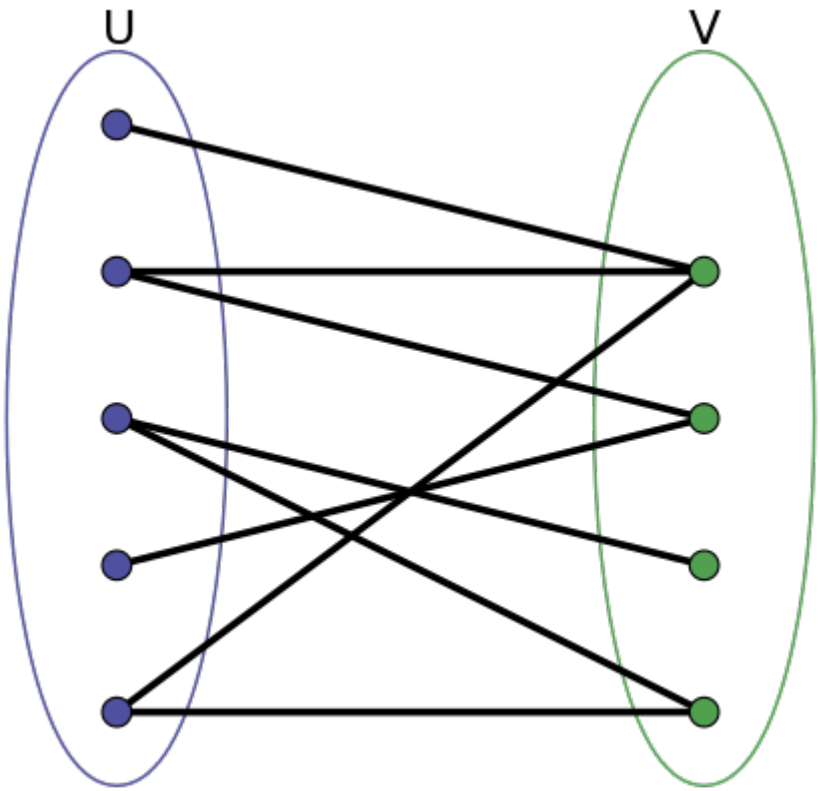
Breadth-First Search (BFS-1)

Problems

P1. Level order treverse

P2. Bipartite

The graph can be divided into two parts, in either of which there is no direct edges between nodes.



- use two explored sets
- make sure nodes of two consequent levels do not have nodes in common.

P3. Determine whether a binary tree is a complete binary tree

- find the first node that misses one or two children node.

- ensure the nodes behind have no child.

Best-First Search (BFS-2)

Dijkstra's algorithm

Problems

P1. find the shortest path cost from source node to any other nodes in the graph

- 点到面
- 使用Min Heap记录candidates
- Any node can only be expanded once but can be generated more than once.

Depth First Search (DFS, Back-tracking)

- Draw recursion tree
  - How many levels are there?
  - How many branches can we make from the current code?
  - **Two kinds of recursion tree**
    - I. in each level, there exists **only two branches**, that is, to add and not to add.
      - **base case should add to result and return**
    - II. in each level, there exists many branches that depends on what choices are left. **(NOT RECOMMENDED)**
      - **base case should add to result and may not return**
- **Issues and tricks**
  - **inplace**
    - **split the input array into two parts** by using `swap` (Problem: permutation)
      - `Arranged characters | Candidate characters`
    - use two resizable container respectively record the arranged charcters and candidate charcters**. In Java, use `StringBuilder` to record arranged characters and use `Set` / `List` / `boolean[]` to record candidates.
  - **remove duplicate**
    - **sort the array first then use a while loop to move index** in each function call when using recursion tree I
    - use extra space like `HashSet` in Java to record used characters in each level

P1. subsets

P2. make up 99 cents with 4 kinds of coins

```
- 4 levels in total
```

P3.

P4. Permutations

P5. List all valid combination of factors that form an integer.

- Recursion I.
  - on each level, we have lots of branches, which represent all possible number of current factor
  - factors go down as along as level increase

```
    12
/           \
6^0          6^1
/   \         /   \
4^0  4^1      4^0  4^1
...
```

- Recursion II.
  - on each level, we have lots of branches, which represent all the possible and smaller or equal factors.
  - only expand factors that are less than or equal to the previous factor

```
          12
  /    |    |    \
  6    4    3     2
 /     |    |     |
2      3    2     2
            |     |
            2(v)  x
```

P6. Print all permutations of ([{

- dfs
- use stack to determine which right parenthesis to add (because we need to do linear scan)

Generic Tree Search Algorithm (BFS and DFS)

Reference - [USC-CSCI561-Lecture-Week2]

Three points

- Initial state
- Expansion/generation rule
- Termination condition

```
function TreeSearch(problem) return a solution or failure

frontier <- makeQueue(makeNodes(problem.INITIAL_STATE))
loop do
    if isEmpty(frontiers) then return failure
    node <- removeFirst(frontiers)
    if goalTest() applied to node.state succeeds
        then return solution(node)
    frontiers <- insertAll(EXPAND(node, problem))
```

Expand Function

```
// @Return: a set of nodes
def expand(node, problem):
    successors = set([])
    for (action, result) in problem.get_successors(node.state):
        s = node(empty)
        s.state = result
        s.parent_node = node
        s.action = action
        s.path_cost = node.path_cost + get_step_cost(node, action, s)
        s.depth = node.depth + 1
        successors.add(s)
    return successors
```

Note:

- Always remove elements from the front
- BFS places new elements at the end of the queue. FIFO.
- DFS places new elements at the front of the queue(stack). LILO.

Generic Graph Search Algorithm

```
function GraphSearch(problem) return a solution or failure

frontier <- makeQueue(makeNodes(problem.INITIAL_STATE))
exploredSet <- empty // change 1
loop do
    if isEmpty(frontiers) then return failure
    node <- removeFirst(frontiers)
    if goalTest() applied to node.state succeeds
        then return solution(node)
    exploredSet <- insert(node, exploredSet) // change 2; after goalTest the node.
    candidateNodes <- EXPAND(node, problem)
    candidateNodes.filter(NOT in frontier && NOT in exploredSet)
    frontiers <- insertAll(candidateNodes)
```

Check if there exists a loop

in a Directed Graph

1. DFS and check if there is duplicate explored node in one path
2. Topological Sorting

Idea

- A circle does not have any nodes with zero in-degree, while other graphs always do.
- So we keep removing nodes with zero in-degrees from the graph and see if there are nodes left.

Steps

- construct edge map (out) and in-degree map

- remove nodes with zero indegree
- keep removing
- if there are nodes left, there exists loops.

in a Undirected Graph

1. Union-find (Disjoint Set)

Princeton Slide

A disjoint-set data structure is a data structure that keeps track of the belongings of nodes.

- Each node has one pointer
- The root has empty pointer
- Each subset only have one root

```
1 -> 2 -> 3
5 -> 2
4 -> 3
```

- Operations
  - ```
    - Eg. ```find(1) = 3```, ```find(4) = 3
    ```
    - 1 and 4 belongs to same subset.
  - `union(A, B)`
    - `rootA = find(A)`
    - `rootB = find(B)`
    - `connect rootA -> rootB`
- Steps to find circle - find the superfluous edge that contributes to the disjoint-set.
  - given nodes and edges
  - use a parent array to record the parent of each node: `parent[node.length]`
  - initialize all nodes to be disjoint sets
    - for each node, `parent[i] = -1`
    - for each edge A -> B,
      - `rootA = find(A)`
      - `rootB = find(B)`
      - if rootA != rootB: union(A, B)
      - else: return Circle Detected

LeetCode 685. Redundant Connection II
    ```

From S^T to S*T

Viterbi Algorithm Coke Range Machine

3.3. Queue & Stack

Reference: Laioffer Class 3

- **Queue** - FIFO (First in first out)
- **Stack** - FILO (First in last out)
- **Deque** - 左右都可以进出的队列

Ascending Stack

Q1: find max rectangle in histogram

Descending Deque

Q1: keep track of the max element of a sliding window of input stream

- use Descending Deque
- use lazy deletion
  - do not take up too much space. worst case O(k)
- Time for each move
  - worst case O(k)
  - amortised O(1)
- Time for all move O(n)

3.4. Heap

1. Also called **Priority Queue**

2. It is implemented by a **Complete binary tree** and can be stored in an array. So the index can be computed easily.
    - *index of left child = index of parent * 2 + 1*
    - *index of right child = index of parent * 2 + 2*
    - *index of parent = (index of child - 1) / 2*
    - *index of root = 0m*
3. 任意节点小于它的所有后裔，最小元素在堆的根上（堆序性）。
4. 根最小的堆叫最小堆，根最大的堆叫最大堆。

Operations

- `insert` - $O(logn)$ - `percolateUp`
- `update` - $O(logn)$
- `get` / `top` - $O(1)$
- `pop` - $O(logn)$ - `percolateDown`
- `heapify` - make an unsorted array into a heap. $O(n)$
    - heap sort

Useless Operations

- *`find`* - $O(n)$ 因为左右儿子没关系
- *`remove(int index)`* - $O(n)$ 需要先find

Problems

Q1. Find smallest k elements from an unsorted array of size n

- **Solution 0** - sort $O(nlogn)$
- **Solution 1** - 一批全吃掉
    - heap sort using a **min heap of size n** $O(klogn+n)$
    - Step 1 - heapify $O(n)$
    - Step 2 - call `pop()` k times to get the k smallest elements $O(klogn)$
- **Solution 2** - 数据量很大，一个批次做不到。
    - use a **max heap of size k** as smallest k candidates $O(k+(n-k)logk)$
    - Step 1 - heapify the first k elements to form a max-heap of size=k $O(k)$
    - Step 2 - iterate over the rest n-k elements one by one to get the real k smallest elements. $O((n-k)logk)$
- Compare **Solution 1** vs. **Solution 2**

Compare	min heap of size n	max heap of size k
Time	O(n + klogn)	O(k + (n-k) log k)
k<	O(n)	O(n log k)
k~~n	O(n log n)	O(n log n)

- **Solution 3** - quick select
  - average time = O(n)$O(n)_{O(n)}$
  - worst time = O(n2)$O(n^2)_{O(n2)}$

Q2. Kth smallest number in sorted matrix

```
12345
34569
489AB
```

- use a N * N-size min heap to store everything and poll k times - O(N∗N+k∗log(N2))$O(N*N + k*\log(N^2))_{O(N∗N+k∗log(N2))}$
- Best First Search
  - use a priority queue to store the candidates and pop the smallest every time
  - pop the smallest for k times
  - O(k∗logk)$O(k * \log k)_{O(k∗logk)}$
- Merge K sorted lists and use a k-size max heap to store the heads of the lists

3.5. Hash Table

- (key, value) pairs
- **NO duplicate keys**
- Hash Table is a general data structure
  - `HashMap` and `HashSet` are its implementationclasses in Java.
- **hash collision**
  - **close addressing**
    - separate **chaining** - use singly linked list
  - **open addressing** -
    - **probe** - put in the next available bucket
      - Linear probing - low cache miss when load factor is low.
      - Quadratic probing

        - Double hashing
- Data Overload
    - **rehashing**

Problems

P1. Top k frequent words

- Step 1 - iterate over the composition for each word and count the words' frequencies using a hashtable.
- Step 2 - maintain a min heap of size k and get top k frequent candidates

P2. find the one missing number in an unsorted array

P3. find the common numbers between two sorted arrays a[N] b[M]

Hash Set

Eg. **P. Kth Smallest Sum From Two Sorted Array** can be used to store visited nodes. time complexity $O(n)O(n)_{O(n)}$ -> $O(k)O(k)_{O(k)}$

Bloom Filter

- A Bloom filter is a data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a **set**.
- The base data structure of a Bloom filter is a **Bit Vector**
    - To add an element to the Bloom filter, we simply **hash it a few times** and **set the bits** in the bit vector at the index of those hashes to 1.
- The price paid for this efficiency is that a Bloom filter is a **probabilistic data structure**
    - the element either is **definitely not** in the set or **may be** in the set.
    - If the bloom filter says yes, it could be not in the set.
    - If the bloom filter says no, it must not be in the set.
- The hash functions used in a Bloom filter should be **independent** and **uniformly distributed**.

3.6. Trie Tree

It is a search tree.

- insert, delete, or search for some kind of "**keys**"
- store **ordered** data

What is the **requirement** of this dictionary?

- n - # of words
- m - average length of the string/word
- k - # of words with the same prefix

- search (word)

- delete
- add
- find all words with given prefix, e.g., auto-completion in google search
    - in HashMap O(nm)

!trie tree

Implementation

```
class TrieNode {
    Map<Character, TrieNode> children;
    // another way: TrieNode[] children; // size 26 array
    Integer value;
    boolean isWord;
}
```

Other problems

4.1. Dynamic Programming

- 把大问题分解为很多个小问题，思考如何用小问题的solution构建大问题的solution (Divide and Conquer)
- 记录Sub-solutions (Memorization)

常见：最大最小值

What is DP? DP vs. Recursion

- recursion：从大到小解决问题，不记录任何sub-solution
  - base case
  - recursive rule
- DP：从小到大解决问题，记录sub-solution（数学归纳法）
  - size(size(n) subsolution
  - base case
  - induction rule
- 满足一定条件的recursion可以用dp
- 空间复杂度可能可以通过只记录M[n-1]从O(n)$O(n)_{O(n)}$优化到O(1)$O(1)_{O(1)}$

When do we use DP?

- **linear scan + look back**
- 大号问题应该可以通过小号问题的解答推导出来
- Eg.
  - 求largest/smallest（变体，在某一段范围求largest/smallest）
  - 分割类问题 (cut rope/wood)
  - 最长连续类问题
  - largest rectangle不行，因为需要找leftest/rightest hi < h，需要用前面所有的解，此时用单调栈会更简单一些。

How?

- 怎样记录子问题解
  - 一维
    - Longest Ascending Subarray
    - Longest Ascending Subsequence
    - cut rope
    - cut palindrome
  - 二维
    - cut wood
    - 沙子归并
    - 两头取pizza
    - 各种matrix问题
    - Two String寻找Minimum Edit distance, Longest common Substring/subsequence
- 需要子问题的数量
  - One
    - Longest Ascending Subarray
  - Many
    - Longest Ascending Subsequence
- 解决大一号问题的方法
  - 大段 + 小段 （子问题解和额外的步骤的组合）
  - 大段 + 大段 （子问题解的组合）

One Dimension DP

1. 左大段 + 右小段
2. 左大段 + 右大段

最小不可分的问题是similar和identical的。 ->是->linear scan回头看 ->否->右大段有可能有不同的pattern （cut postion类型问题）

P1. Longest Ascending Subarray

- Array vs. Sequence
  - sub-Array: phsical contiguous elements in an array
  - sub-Sequence: not necessarily contiguous

- M[i] represents in String[0, i] the max length of the ascending subarrays. The substring must include i-th element. **(以i结尾的最长的substring)**
- M[i]=M[i−1]+1,ifinput[i]>input[i−1]M[i] = M[i - 1] + 1, if input[i] > input[i - 1]$M[i]=M[i-1]$ +1,ifinput[i]>input[i−1]
- M[i]=1,otherwiseM[i] = 1, otherwise$M[i]=1,otherwise$

Follow-up. Longest Ascending Subsequence

- linear scan back
  - M[0] = 1
  - M[i] = max(M[j]) + 1 for all 0 <= j < i and a[j] < a[i]
  - result = max(M[i])
- binary search
  - Refine is an ascending array that records the lowest ending of all the size-M ascending subsequence.
  - Refine[M] = min(A)
  - Refine[M] < Refine[M + 1]
  - update
    - i: 0 -> A.length - 1
    - find index of largest Refine[j] smaller than A[i], that is j
      - binary search
    - Refine[j + 1] = min(Refine[j + 1], A[i])
      - simply update Refine[j + 1] = A[i] is fine. We have that Refine[j] < A[i] <= Refine[j + 1] and Refine[j + 1] > Refine[j]. So Refine[j + 1] = A[i] is always right.
      - M[i] = j + 1
    - result = largest j within initialized Refine[j]
      - if found j == result, result++

P2. Cut ropes

Note: 之所以说DP是从小到大解决问题，是因为我们只解决每个子问题一遍，而且严格地按照从规模小到规模大的问题的顺序解决。使用Recursion比较不容易控制解决问题的顺序。

P3. Maximum subarray sum

- M[i]M[i]$M[i]$ represents the subarray with the largest sum among all subarrays that ends with i.

- $M[i]=M[i-1]+input[i], if M[i-1]>0$M[i] = M[i - 1] + input[i], if M[i - 1] > 0$M[i]=M[i−1]+input[i],ifM[i−1]>0$
- $M[i]=input[i], otherwise$M[i] = input[i], otherwise$M[i]=input[i],otherwise$

P4. Dictionary Concat

Two Dimension DP

P1. Edit Distance

P2. Maximum Square of 1's in a binary matrix

P3. Longest common substring

Follow-up. Longest common subsequence

Big data situations

Situations

- **Source limitation**
  - 单机单核内存足
  - 单机单核内存不足
  - 单机多核内存足
  - 多机内存足
  - 多机内存不足
- **Data input type**
  - data is offline and so big that it **cannot fit into the memory**
  - data can come by **stream**

- **Data input range**
  - small range, like 0-255
  - large range, like any double number

Key points

- **fit data into Memory**
  - When analyzing the problem, **Space Complexity** is key.
  - **partition data into chunks when sorting**
    - deal with each chunk respectively and merge the solutions
    - can use hash or some order, like bucket sort
  - **maintain a heap in memory when finding k-th largest / median**

find median

Source: C18 Q4

P1. Given a stream, keep track of the median of the numbers.

- Data structure
  - Both halves have roughly the same size
  - max(smaller half) <= min(larger half)
  - median = max(smaller half) if odd
  - median = (max(smaller half) + min(larger half)) / 2 if even

Smaller half	Larger half
max heap	min heap

- Maintain the data structure
  - insert the `cur` into the smaller half or the larger half
  - make the quantity of the two halves equal by moving the largest of the smaller half to the larger half or the opposite.

Follow-up. Data cannot be fit into the memory situation

- Data structure
  - Let's say, set the limit of the memory of the two heaps to be 1G.

Smaller sorted array	Smaller max heap	Larger min heap	Larger sorted array
on disk	in memory	in memory	on disk

- Maintain the data structure
  - insert the `cur` into the smaller heap or the larger heap
  - make the quantity of the two halves equal by moving the largest of the smaller half to the larger half or the opposite.
  - reorganize heap and sorted array if one of two situations meet
    - size(heap) is about to exceed 500M
      - merge the max heap and the samller sorted array
      - put the largest 250M elements in the max heap
      - put the remaining data in smaller sorted array
    - max(smaller sorted array) > max(max heap)

Input is not a stream

只有2G内存的pc机，在一个存有10G个整数的文件，从中找到中位数，写一个算法。

1. 外排序 思想：排序-归并 把部分排序结果输出到一个临时文件中。
2. 堆排序 step 1. 先求第1G大：建立一个最小值堆，把所有数都装进去，超出内存的舍弃。 step 2. 利用第1G大，求得第2G大：构造一个最小堆，然后把剩下的数一一丢进去。 step 3. 重复以上直到得到中位数。
3. 位运算 根据内存大小，取整数的前几位，排序全量数据。
4. 桶排序 step 1. 第一次扫描：统计每个区间的数量。 step 2. 统计结果，得到中位数所在区间 step 3. 第二次扫描：拿到区间的所有数字，然后取中位数。

find certain percentile

Source: C18 Q5

find 95th percentile of all urls' length

- Solution 0 - sort O(nlogn)
- Solution 1 - max heap(95%) + min heap(5%)
  - for each step O(logn)
  - total time O(nlogn)
- Solution 2 - bucket sort
  - insight: max length of URLs <= 4096
  - calculate the histogram and its CDF

sort

Source: 加强5 Q3

Given a single computer with a single CPU and a single core, which has 2GB of memory and 1GB available for use, it also has two 100GB hard drives. How to sort 80GB integers of 64bits?

Step 1: **Divide all data into 800 chunks**, each of those having 100MB data. use 1GB memory to sort each 100MB data.

- Quick sort
  - space O(n)
- Merge sort
  - space O(n)
- Why only sort 100MB data?
  - because we need O(n) space. In practice, it can be several n.

Step 2: **k-way merge sort** 800 chunks of 100MB sorted data.

- memory cost
  - each chunk has a reading cache
  - also has a writing cache
  - size-800 heap

Bit Representation

Binary

Decimal -> Binary

One's Complement / **Two's Complement**

- 为什么这么设计？ CPU处理时不用考虑符号

use 32bits for each number

```
 1 == 0000 0000 0000 0000 0000 0000 0000 0001


      1111 1111 1111 1111 1111 1111 1111 1110
-1 == 1111 1111 1111 1111 1111 1111 1111 1111  plus 1
```

Hexadecimal

32-bit signed integer max value = 0x7FFFFFFF

Bit Operation

- & and
- | or
- ~ not
- ^ xor
  - same bits = 0, different bits = 1
- arithmetical shift
  - << left shift
    - add 0 to the right of positive number
  - right shift
    - add 0 to the left of positive number
    - add 1 to the left of negative number
    - `3 >> 1 = 3 / 2 = 1`
    - `-3 >> 1 = -2` 负数除法不建议用位运算做
- logical shift
  - right shift
    - always add 0 to the left of number

bit check

```
x & (1 << k)
```

```
(x >> k) & 1
```

bit set

```
x = x | (1 << k)
```

Probility, Sampling, Randomization, etc

Shuffling algorithm

Random Shuffle - All the permutations have the same probability.

```
Random random = new Random();
```

Data Structure:

Shuffled || To be shuffled

Sampling

How do we sample one element from a stream of unknown/unlimited size?

- use one varible to record the sample and another varible to count the stream
- Replace the sample with the new element with prob. of `1 / counter`

```
Element sample = null;
int counter = 0

Element newElement = stream.getNext();
while (newElement != null){
    counter++;
    if (random(counter) == 0){
        sample = newElement;
    }
    newElement = stream.getNext();
}
```

Follow-up. How do we sample K element from a stream of unknown/unlimited size?

- Reservoir sampling
- Use a counter and Size-k element to store the samples
- Replace one of the samples with the new element with prob. of `k / counter`
- Replace the i-th sample with the new element with prob. of `1 / k`

Follow-up. return a random index of the largest numbers

- Use a counter to record the number of largest and an element to record the largest
- If the largest change, reset the counter to 1
- If the new element is equal to the largest
    - replace the random index with the new index with prob. of `1 / counter`

Design a random generator with other random generator

P1. Design Random5() with Random7()

index	0	1	2	3	4	5	6
-	0	1	2	3	4	5	6

- If the generated number is in [0, 4], output it.

P2. Design Random7() with Random5()

index	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

- Pick a number from [0, 20]

P3. Design Random1000000() with Random2()

- Random125() = Random5() *25* + *Random5()* 5 + Random5()
- Random2() -> Random2^k()
  - 2^k >= 1000000

4.5. Combination of Data Structures

keep track of the max element of a sliding window of input stream

- Time for each move
  - worst case O(k)
  - amortised O(1)
- Time for all move O(n)

LRU Cache

加强6 Q3.1

use cases:

- add()

- get()
    - HashMap
- deleteOldest()
- adjustToNewest()
- for deleteOldest() and adjustToNewest()
    - Array
        - time O(n)
    - X Deque (Deque cannot delete in the middle)
    - LinkedList
        - time O(1)
        - (easy to implement if using Doubly Linked List)
    - X Min Heap

keep track of first non-repeating character in a stream

加强6 Q3.2

use cases:

```
- non-repeating
    - use HashMap
- record the order of characters
    - use Array
    - use LinkedList
    - X use Deque (Deque cannot delete in the middle)
```

A good solution:

- use a HashMap and a doubly linked list
    - if key is not in the hashmap
        - it never appears in the previous input;
    - if key is in the hashmap and the value is null
        - it appears more than once
    - if key is in the hashmap and the value is linked to the node in the linked list
        - it appears once
- not suitable for lazy deletion
    - waste space if input is like: a b b b b b b b b

Exam Room

4.6. Classic Problems with Multiple Solutions

K-Sum

2-Sum

- Sorted input
  - use HashMap
    - HashMap
  - two pointers, time O(n) space O(1)
    - 证明：循环证明。有点绕，大概意思就是所有被排出的数都可以被之前被排除的数的结论推导得到。
    - assume we have Xi0 + Yj0 < t
    - then we got: for all Xi < Xi0, Xi + Yj0 < t
    - all Xi < Xi0 are excluded with Yj >= Yj0 under only one situation that Xi + Yj < t
      - that is, for all Xi < Xi0, exists Yj1 >= Yj0 that Xi + Yj1 < t
    - for Yj > Yj1, it is excluded with Xi1 <= Xi0 under only one situation that Xi1 + Yj1 < t
      - that is, for all Yj > Yj1, exists Xi2 <= Xi1 that Xi2 + Yj > t
    - conclusion: for all Xi < Xi0, we have Yj >= Yj0, Xi + Yj != t
- Unsorted input
  - sort first, then use methods for sorted input
    - quick sort, Space average O(logn) worst O(n)
    - merge sort, Space O(logn)
    - selection sort (recursively select the max/min and swap it with the number in the right place), Space O(1), Time O(n^2)
      - optimized space
- What if the data cannot fit in the memory? (sorted input)
  - use two pointers
  - use cache

3-Sum

```
for (int i = 0; i < input.length; i++) {
    run2Sum(input, i + 1, input.length - 1, target - input[i]);
}
```

Time = O(n^2) sorted it first

4-Sum

Solution 1

```
for i:
    for j:
        run 2Sum
```

Time = O(n^3)

Solution 2

```
sum1 = <i1 + j1> => HashSet
sum2 = <i2 + j2>

for i:
    for j:
        current pair <i, j> with sum = input[i] + input[j]
        loop:
            find pair in the hashSet with value = target - sum
            check duplicatation worst O(n)
```

Time = O(n^3)

Solution 3 - optimized from Solution 2

- We do not want duplication of set of 4 numbers.
  - we only store only one pair in the hashmap.
  - we want to ensure 4 numbers are arranged in order, like i1 < j1 < i2 < j2
  - when we add 2-sum pair into the hashmap, if for each i, we only keep the smallest j
  - for example, we have 1,2,7,9. we only keep pair (1, 2) and pair (7,9). when we have pair (1, 7), it is not add to the hashmap

`java`

DP && DFS && BFS2

largest product of length

Given a dictionary containing many words, find the largest product of two words' lengths, such that the two words do not share any common characters.

laicode

1. no common characters
2. max product of length

Potential ways to solve max/min problem

- DP
- BFS2(best first search)
    - initial state
    - expansion/generation rule
        - poll string pairs in descending order of the product of two string length
    - termination condition
- DFS (brute force)
    - generate all the possible conditions and find the max

We have n*n states:

- pop each state takes O(log(n^2))
- check common elements O(m)

Time = O(n^2 * (log(n^2) + m))

k-th smallest with only 3,5,7 as factors

- BFS2 with min heap
- deduplicate - use set or boolean array

k-th closest point to <0,0,0>0,0,0>

- BFS2 with min heap

Place to put the Chair I

We want the sum of distances from the chair to all the equipments is the shortest.

Assumptions

- 4-connected grid, cost = 1
- 8-connected grid, cost = sqrt(2)
- cannot use Manhattan distance if we have obstacles

Solutions

- Solution 1
    - for for every possible chair location (x,y) { run a Dijkstra(BFS2) to search path from chair to equipments calculate the sum update the global min for each location }
    - time O(n^2 *n*^2 log(n^2)) = O(n^4 * logn)
- Solution 2
    - for each equipment { run a Dijkstra }
    - for each cell { calculate the sum }
    - time O(k *n*^2 log n)

k way merge

find common elements in k sorted array

Other algorithms

Voting algorithms

加强6 Q4

黑帮火拼算法 数第二遍

优势：牺牲时间常数项，减少空间复杂度

证明：

活下来的人一定是出现次数大于1/K的元素（活下来是必要条件） 数第二遍（证明充分条件）

Java

Java

Difference Between Java & C++

Java: Platform compatible, write once, compile once, run everywhere on JVM.

C++: write once, compile everywhere

Coding Style

Google Java Style Guide

1. Naming - use upperCamelCase for class names and all upper cases for const variable, otherwise lowerCamelCase.
2. Separate reversed words and bracelets. reversed words - 保留字
3. Eg. `for ()`
4. Eg. `a && b`
5. Add spaces on both sides of binary or ternary operators.
6. Eg. `a + b`

Primitive types vs. Class types

- `boolean` / `byte` / `char` / `short` / `int` / `long` / `float` / `double`
  - `char` ranges from `0``\u0000` to `\uffff``65535` (16-bit unicode)

- `5` / `5.5` are objects.
- Strings are objects.
- Arrays are objects.

Autoboxing and Unboxing

- Wrapper class
  - allow null
  - immutable internal values
- **Autoboxing & unboxing are done only when it is necessary.**
  - Mannual: `Integer a = Integer.valueOf(4);`
- `==` vs. `equals`
  - `Integer a == Integer b` can be wrong
  - `(Integer)a.equals((Integer)b)` or `Integer.compare(o1, o2)` should be used!
- `int[]` vs. `Integer[]`
  - there is no auto conversion directly between them.
- Unboxing null throws NPE (Null pointer exception).

Compare Integer

- The JVM is caching Integer values. == only works for numbers between -128 and 127

```
Integer i1;
Integer i2;
i1 == i2 // wrong!
i1.equals(i2) // right
```

Converting Number to/from Strings

- factory method

```
int i = 0;
String si = i + "";
si = String.valueOf(i); // factory
si = Integer.toString(i);

Integer w = null;
si = w.toString(); // "null"
si = String.valueOf(w); // factory

Integer.valueOf(si); // factory
Integer.parseInt(si);
```

- from string to integer

```
public int parsePosInt(String str) {
    int res = 0;
    for (char c: str.toCharArray()) {
        res = res * 10 + (c - '0');
    }
    return res;
}
```

Array

```
int[][] array = new int[2][];
array[0] = new int[1];
array[1] = new int[3];
```

取subarray，使用arrays工具箱。

Static & Final

1. static

- class variable vs. instance varible
- class method vs. instance method

- unlike C++, local static varible is not allowed.

2. final

- final class: A class that cannot be dereived. 最终类。不能被继承。
- final method: A method that cannot be overridden.
- final field/var: A varible that once assigned, cannot be assigned again.

define a **constant var** in Java:

```
// Cannot change the value
final int value = 10;
```

define a **constant reference** in Java:

```
// Cannot change the reference value
final Boy b1 = new Boy(10);

// Can change the object field varibles
b1.girlFriend = new Girl("Lady gaga");
```

Parameter Passing

Java parameters are always passed by value.

- **primitive type**: copy of the value itself
- **objects**: copy of the object reference

Inheritance, Interface, Abstract Class

Inheritance（继承） - Base class（父类）, Derived Class（子类）

C++: Abstract class Java: Interface and Abstract Class

class extends class class implements interface interface extends interface

- **Override** vs. **Overload**
  - Override is to redefine a method that has been defined in a parent class

    - Overload is when you define two methods with the same name distinguished by their different signatures.
        - Overload has nothing to do with polymorphism

Java Interfaces & Abstract Classes

Interface helps you (or enforces you) to focus on the API signature definition. **Abstract Class** provides a common base class implementation to derived classes. Use abstract class (or even concrete class) for base class if the derived classes share common implementation

- 为什么要用List作为Interface？

  ```
  List<Integer> myList = new LinkedList<>();
  ```

  把接口和实现分开实现。因为这里我们仅仅需要List的语义，我们要的不是一个特定的List。实现可能会变化。

  Eg.

  ```
  List<Integer> myList = creatListFromConfig(conf);
  ```

- 父类List有的功能，子类LinkedList一定有。

Abstract class vs. interfaces

Java docs

Abstract class	**Interface**
using **extends**	**implements**
methods can be declared without implementation	same
cannot instantiate	same
can give implementation	cannot
doesn't support multiple inheritance	support
...	...

```
interface A {
    public Integer get(int index);
}

interface B {
    public void put(int index, E e);
}

A myDict = new A(); // wrong

class myList implements A, B {
    @override
    public Integer get(int index) {
    }
    @override
    public void put(int index, E e) {
    }
}
```

- When do we use abstract class vs. interface?
  - abstract class provides a common base case implementation to derived classes
  - if we want to declare non-public members
  - if we need to add methods in the future

Polymorphism and Overriding

Polymorphism 多态 : 调用override的function

java会根据实际的object而决定调用的行为。

Design

残疾人是一个有残疾状态的人，不应该是子类。因为残疾可能会变好。

Top Level Class

- One Java file have only one public class.

- One Java file can have more than one classes.
- A helper class is at most package access level but a static nested can be public.

Nested Class

- **Static nested class** - can access class varibles and methods
  - It is a way of logically grouping classes that are only used in one place.
  - It increases encapsulation. Consider we hide B into A. We can set members in A to be private and B can access them and be hidden from outside A.
- **Nonstatic nested class** / **Inner class** - can access varibles and methods
  - **Inner class cannot be created without an instance of outer class while nested class can.**
  - `return Outerclass.this;`
- **Anonymous class**
  - Comparators, lambda

Iterator

Iterator

- `next()`
- `hasNext()`
- `remove()` (optional)

```
for (Iterator<Apple> iter = list.iterator(); iter.hasNext();) {
    Apple apple = iter.next();
    System.out.prinln(apple)
}
```

为什么使用Map的View时候删除会报错Concurrent Modification Exception。

ListIterator

- `previous()`
- `previousIndex()`

- `hasPrevious()`
- `nextIndex()`

Generics

- Types are parameters
- A type or method to operate on objects of various types while providing **compile-time** type safety (vs. runtime safetyl too late, exceptions).
- E T K V U
- 原理：类型擦除保留原始类型

```
List list = new ArrayList();
Apple apple = (Apple) list.get(0);
```

```
// read time, fever codes
List<Apple> list = new ArrayList<>();
Apple apple = list.get(0);

// write time, safety check
list.add(new Orange()); //wrong
list.add(new Fruit()); // wrong

// Itertor gets better (no need to cast)
for (Iterator<Apple> iter = list.itertor(); iter.hasNext();) {
    Apple apple = iter.next();
}

// for each
for (Apple apple : list) {
    apple.printOut();
}
```

- Static methods(needs to declare generic type again)

```
public static <T> T getFirst(List<T> list) {
    return list.get(0);
}
```

- Subclass and Superclass in Generics

```
Apple apple = new Apple();
Fruit fruit = apple; // ok

Fruit fruit = new Fruit();
Apple apple = fruit; // wrong, need cast
Apple apple = (Apple) fruit; // ok at compile time, but may have java.lang.ClassCastException
```

- List of T cast

```
List<Apple> a = new ArrayList<>();
List<Fruit> b = a; // not allowed.
// b can add new Orange(). a cannot add new Orange(). a and b refer to the same object.

List<Fruit> a = new ArrayList<>();
List<Apple> b = a; // not allowed either.
```

- Generics check the type during the compile-time while **Array does not**.

```
Apple[] apples = new Apple[3];
Fruit[] fruits = apples; // ok at compile time
fruits[0] = new Apple(); // ok
fruits[1] = new Fruit(); // ArrayStoreException
fruits[2] = new Orange(); // ArrayStoreException
```

- `? extends T` and `? super T`
  - give read-time flexibility

```
List<? extends Fruit> fruits = apples;
```

```
public static <E extends Comparable<E>> E getMin(E[] arr) {
    ...
}
```

Enum

```
class RainbowColor {
    public static final int RED = 0;
    public static final int BLUE = 1;
}
```

```
enum RainbowColor {RED, ORANGE, YELLOW}

switch(today) {
    case Mon: do(something); break;
    case Tue: do(something); break;
}
```

- `enum.values()`
- `constant.valueOf()`
- `constant.ordinal()` returns the order

```
enum Weekday {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
}

Weekday[] l = Weekday.values();
for (Weekday wd : l) {
    System.out.println(wd + " " + wd.ordinal());
}



Weekday wd = Weekday.valueOf("Mon");
Weekday wd = Weekday.valueOf("Monday");
```

- Extends Enum

```
final class MyDay extends Enum {
    public static MyDay[] values() {return (MyDay[])$VALUES.clone();}
    private MyDay(String s, int i) {
        super(s, i);
    }

    public static final MyDay MONDAY;
    public static final MyDay TUESDAY;
    public static final MyDay $VALUES[];

    static {
        MONDAY = new MyDay("MONDAY", 0);
        TUESDAY = new MyDay("TUESDAY", 1);

        $VALUES = (new MyDay[] {MONDAY, TUESDAY});
    }
}
```

- Enum classes and anonymous class

```
public enum ScoreComparator implements Comparator<Student> {
    MATH_COMP {
        @Override
        public int compare(Student o1, Student o2) {
            return 0;
        }
    },
    ENG_COMP {
        @Override
        public int compare(Student o1, Student o2) {
            return 0;
        }
    }
}
```

Java Collections

Collections Framework Overview

Java 集合系列01之 总体框架

Overview

Interface

- `List`
  - `ArrayList`
  - *`Stack`* (Not Recommended)
  - `LinkedList`
- `Queue`
  - `LinkedList`
  - `ArrayDeque`
  - `PriorityQueue`
- `Deque` (inherited from `Queue`)
  - `LinkedList`
  - `ArrayDeque`

List vs. Queue

- List有index概念，支持随机访问
- Queue没有index概念
- java的Stack被淘汰了，因为性能不够好。

Interface - List

API

- `size()`
- `isEmpty()`
- **Random Access**
  - `set(int index, E e)`
  - `E get(int index)`
  - `add(int index, E e)`
  - `add(E e)`
  - `remove(int index)`
  - `remove(E e)`
- Search

- Iterator
- Range-View

Class - ArrayList

- **inital capacity 10** and expand by **1.5 times** when there is no unused cells available
- **size** = `ArrayList.size()`
- **capacity** = `ArrayList.length` 当前内存容量
- 在删除很多元素后，容量减少到一定程度之后，ArrayList不会主动trimToSize，一般需要人工调用该方法。

Create an ArrayList from an Array

Initialization of an arraylist in one line

create arraylist from array

how to convert int[] into list in java

Interface - Queue & Deque

Java Doc Deque

- Queue is a FIFO queue. (exception PriorityQueue)
- Deque is short for double-ended queue. Deque is FIFO & LIFO. (both queue and stack)
- Java里面实现Stack最好的接口是Deque，用LinkedList实现，不能用List，更不能直接用LinkedList。

API

- Queue
  - `offer(E e)` - offer at the **tail**
  - `E poll()` - poll at the **head**
  - `E peek()` - look at the **head** without polling it out
- Deque
  - `offerFirst(E e)`
  - `offerLast(E e)` = `offer(E e)`
  - `addFirst(E e)` = `push(E e)`

    - `E pollFirst()` = `E poll()`
    - `E pollLast()`
    - `E removeFirst()` = `E pop()`
    - `E peekFirst()` = `E peek()`
    - `E peekLast()`

```
Locations-------First-----------------Last
Operations-----add/push---------------offer
--------------remove/pop------------------
----------------poll----------------------
```

Operations of Queue and Stack

Operations	Queue	Stack
Insert	`offer` / `add`	`push` / `add`
Remove	`poll` / `remove`	`pop` / `remove`
Examine	`peek` / `element`	`peek` / `element`

Note:

- `add`, `remove` and `get` throw exception if error happens
- `offer`, `poll` and `peek` return special value if error happens

Class - LinkedList & ArrayDeque

- Most popular implementation class: `LinkedList`, `ArrayDeque`.
- `ArrayDeque` is a newer implementation by **circular array** for `Deque`. No null values in Deque.
- For a queue or stack, we can just use `LinkedList`, as it implements both `Queue` and `Deque` interfaces.
- Java的LinkedList是Doubly Linked List，存有Head/Tail

Circular Array

- use varible `head` to record the index of the node before the head.
- use varible `tail` to record the index of the node after the tail.
- can store `n-1` elements with n-length array.

Class - ArrayList vs. LinkedList

How do we choose ArrayList or LinkedList?

1. if there are **very many random access**, use ArrayList.
2. **always add/remove at the end**, use ArrayList.
3. If time complexity is similar for both, use ArrayList. Because LinkedList is memory-cost.
4. **Queue/Stack** -> LinkedList
5. Stack & Vector现在不用了，这两个是线程安全的，但是额外消耗了很多开销。 Stack->use LinkedList(ArrayDeque) Vector->use ArrayList

Class - PriorityQueue

- It implements the queue interface but it is not a FIFO queue. It is actually a min heap.
- It arranges the order of the elements by comparing any pair of elements.
- **Time Complexity** of size n `PriorityQueue`
  - `offer(E e)` - $O(logn)O(logn)_{O(logn)}$
  - `peek()` - $O(1)O(1)_{O(1)}$
  - `poll()` - $O(logn)O(logn)_{O(logn)}$
  - `remove()` - $O(logn)O(logn)_{O(logn)}$ - should not be used
  - `size()` - $O(1)O(1)_{O(1)}$
  - `isEmpty()` - $O(1)O(1)_{O(1)}$
  - `private heapify()` - $O(n)O(n)_{O(n)}$
- **heapify** - use one `Collection` as initial argument
- **Order** of the elements
  - It is defaulted by returning -1 if x1 < x2;
  - be CAREFUL not to return an overflowed value.
  - `Comparator.compare(E o1, E o2)` provided when newing a PriorityQueue
    - needs not to change the original class
  - `Comparable.compareTo(E another)` interface in class
  - use `Collections.reverseOrder()`
  - use `Collections.reverseOrder(new myComparator)`
- **Possible ways to provide a comparator class**
  - top-level class (recommended)
  - Static nested class
  - Anonymous class (not recommended)

- Lambda expressions (since Java 8)

```
class Cell implements Comparable<Cell>{
  public int row;
  public int col;
  public int value;
  public Cell(int row, int col, value){
      this.row = row;
      this.cell = cell;
      this.value = value;
  }

  @Override
  public int compareTo(Cell another){
      if (this.value == another.value){
          return 0;
      }
      return this.value < another.value ? -1:1;
  }
}

// will use compareTo() in the class
PriorityQueue<Cell> minHeap = new PriorityQueue<Cell>(11);

//interface Comparator{
//    int compare(E o1, E o2);
//}

class MyComparator implements Comparator<Cell>{
  @Override
  public int compare(Cell c1, Cell c2){
      if (c1.value == c2.value){
          return 0;
      }
      // !IMPORTANT needs to return -1 or 1, or it might over MAX_INT.
      return c1.value - c2.value ? -1:1;
  }
}

// will use compare() in the provided Comparator
PriorityQueue<Cell> minHeap = new PriorityQueue<Cell>(11, Collections.reverseOrder())
// self-defined comparator class
```

```
PriorityQueue<Cell> minHeap = new PriorityQueue<Cell>(11, new MyComparator());
// anonymous comparator class
PriorityQueue<Cell> minHeap = new PriorityQueue<Cell>(11, new Comparator<Cell>(){
  public int compare(Integer one, Integer two){
      return 0;
  }
});
// lambda
PriorityQueue<Cell> minHeap = new PriorityQueue<Cell>(11, (lhs, rhs) ->{
  return 0;
}
);
```

Heap Implementation

- `percolateUp(int index)`
  - when need to move up one element
  - eg. offering a new element into the heap.
- `offer()` - use `percolateUp`
- `percolateDown(int index)`
  - when need to poll the root element from the heap.
- `poll()` - use `percolateDown`
- `heapify()`
  - convert an array into a heap in O(n)$O(n)_{O(n)}$ time.
  - perform `percolateDown` action in the order of from the nodes on the deepest level to the root. Only needs to do for the first half of the array. range: [0,n/2−1][0, n/2-1]$_{[0,n/2-1]}$
    - No need to know how to prove
  - if we perform `offer()` and `percolateUp`, the time complexity becomes O(nlogn)$O(nlogn)_{O(nlogn)}$
- `update()`
  - find the element in O(n)$O(n)_{O(n)}$
  - use either `percolateUp` or `percolateDown`

Java Map

Java Collection Interfaces Overview

Interface - Map / Set

API

- `Map`
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(K key)`
  - `boolean containsKey(Object key)`
  - get view
    - `Set<K, V>> entrySet()`
    - `Set<K> keySet()`
    - `Collection<V> values()`
  - `boolean containsValue(V value)` - O(n)O(n)$_{O(n)}$
  - `void clear()`
  - `int size()`
  - `boolean isEmpty()`
- `Set`
  - `boolean add(E e)`
  - `boolean remvoe(Object o)`
  - `boolean contains(Object o)`
  - `void clear()`
  - `int size()`
  - `boolean isEmpty()`

Implementation Classes

1. HashSet
   - stores its elements in a hashtable
   - doesn't not gurantee the order of iteration
2. TreeSet
   - stores its elements in a red-black tree(balanced binary search tree)
   - gurantee the order of iteration (inorder traversal)
3. LinkedHashSet
   - It is a HashSet and also a LinkedList.

Class - HashMap / HashSet

- `HashMap` and `Hashtable` are like `ArrayList` and `Vector`.
  - `HashMap` allows one null key.
  - `Vector` and `Hashtable` operations are synchronized.
- `HashSet` is backed up by a `HashMap` instance.
- Data Storage - `array` + `linkedlist`
  - maintain array of entries
    - `Node<K, V>[] array`
    - `ArrayList<K, V>> array` Java supports better.
    - each entry is actually a single linked list(handle collision) and contains the (key, value) pairs
- Distribution - hashcode()
  - Java1.8的hashmap的hashcode：(h = k.hashCode()) ^ (h >>> 16)
- **Capacity Expansion**
  - check load factor if exceeds 0.75 and double the size
  - rehash
- **Access key (get / put)** process
  - hash - `int hash = hashCode(key)`
  - index - `int index = hash % table.length()`
  - iterate the bucket
    - check if the key has already existed
    - NO -> add a ney Entry node
    - YES -> update the value of the Entry

Time Complexity

```
Operation | Average | **Worst**
----|----|----
search: containsKey/get | O(1)O(1)O(1) | **O(n)O(n)O(n)**
insert/update: put | O(1)O(1)O(1) | **O(n)O(n)O(n)**
delete: remove| O(1)O(1)O(1) | **O(n)O(n)O(n)**
```

Check key

- `==`
  - determine if two primitive type has the same type
  - determine if two reference are pointed to the same object
- `.hashCode()`

- `.equals()`

Contract between equals() and hashCode()

- true `equals` => same `hashCode`
- reduce collesion as much as you can.

Override equals(), hashCode()

```
public class Element {
    int x;
    int y;
    int sum;
    ...
    @Override
    public boolean equals(Object obj) {
        if (this == obj){
            return ture;
        }
        if (!(obj instanceof Element)) {
            return false;
        }
        Element other = (Element) obj;
        return this.x == other.x && this.y == other.y && this.sum == other.sum;
    }

    @Override
    public int hashCode(){
        return this.x * 31 * 31 + this.y * 31 + this.sum;
    }

}
```

hashCode Design

```
return x * 31 + y;
```

```
return hashNumber & 0x7fffffff;
```

- 需要取绝对值，位运算计算效率最快。
    - 10 % 3 = -1，会影响后续取index

- pair设计hashcode，当交换Pair(A, B)得到Pair(B, A)，应该得到同样的hashcode

Efficient way of accessing the HashMap

> Laioffer Practice Class 13

- count the occurence
  - `containsKey -> get // two times`
  - `Integer sc = hm.get("snap") // one time`
  - `getOrDefault // one time`
- remove when traverse
  - `for (Map.Entry<String, Integer> entry : map.entrySet())` NOT RECOMMENDED
  - `Iterator<String, Integer>> iter = map.entrySet().iterator();` and `while (iter.hasNext())`

Class - TreeMap / TreeSet

- Red-Black tree Implementations
- Guaranteed **O(log(n))** time cost for `containsKey`, `get`, `put`, `remove`
- in Java: TreeMap/TreeSet
- implements `NavigableMap`
  - Methods `lowerEntry`, `floorEntry`, `ceilingEntry`, and `higherEntry` return `Map.Entry` objects associated with keys respectively **less than**, **less than or equal**, **greater than or equal**, and **greater than** a given key, returning `null` if there is no such key

Java String

- `String`
  - `int capacity()`
  - **`char charAt(int index)`**
  - `int indexOf(String str)`
  - `int indexOf(String str, int fromIndex)`
  - `int lastIndexOf(String str)`
  - `int lastIndexOf(String str, int fromIndex)`
  - **`int length()`**
  - `String substring(int start)`

    - **`String substring(int start, int end)`** output = input[start, end)
    - `String toString()`
    - `boolean startsWith(String prefix)`
    - `boolean endsWith(String suffix)`
    - `int compareTo(String anotherString)`
        - compares two strings lexicographically.
        - `"abc"` < `"abcd"`
        - `"abc"` < `"abcd"`
        - `"abc"` < `"b"`
    - `String[] split(String regex)`
    - `String[] split(String regex, int limit)`
        - `"a/b/c/d"` -> `.split("/", 2)`
    - `String trim()`
        - remove leading and trailing white spaces.
- `StringBuilder`
- `StringBuffer` - thread-safe / *synchronized*
    - same as `String`
    - **`StringBuffer append(String s)`**
    - `StringBuffer reverse()`
    - `delete(int start, int end)`
    - **`deleteCharAt(int index)`**
    - `insert(int offset, int i)`
    - `replace(int start, int end, String str)`

```
public class Test{
  public static void main(String args[]){
    StringBuffer sBuffer = new StringBuffer("菜鸟教程官网：");
    sBuffer.append("www");
    sBuffer.append(".runoob");
    sBuffer.append(".com");
    System.out.println(sBuffer);
  }
}
```

String to Char Array & Char Array to String

- `substring` - [startIndex, endIndex)
- `new String` - [offset, offset + count)

```
public char[] convert(String input){
    return input.toCharArray();
}

public String substring(String input, int offset, int count){
    return new String(input, offset, count);
}



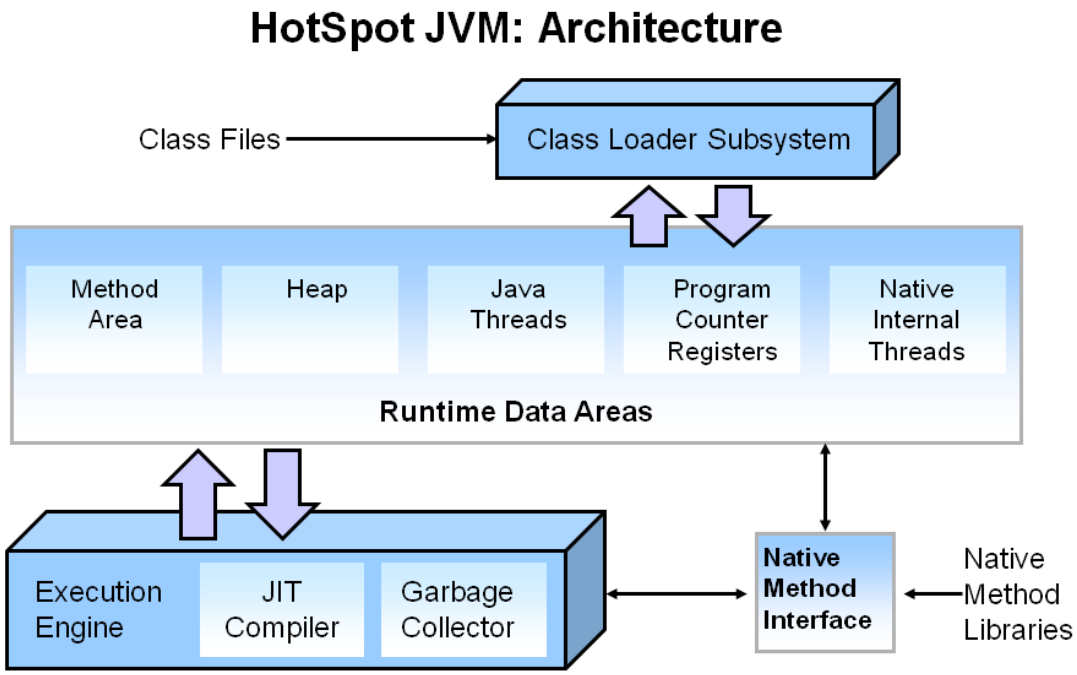
public String convert(char[] input, int startIndex, int endIndex){
    return input.substring(startIndex, endIndex);
}

// String definition
```

Java Virtual Machine JVM

- **JDK**
  - Java Development Kit
  - JDK contains the tools for developing Java programs running on JRE.
  - for example, it provides the compiler 'javac'
- **JRE**
  - Java Runtime Environment
  - JRE is the JVM program

- **JVM**



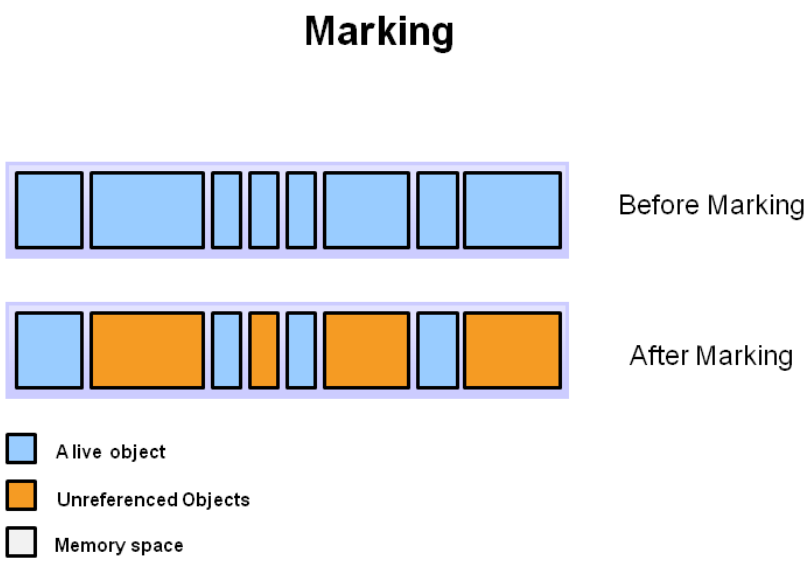
  -
  - **Class Loader**
  - **Java Threads**
  - **Garbage Collector**

Garbage Collection

Java tutorial

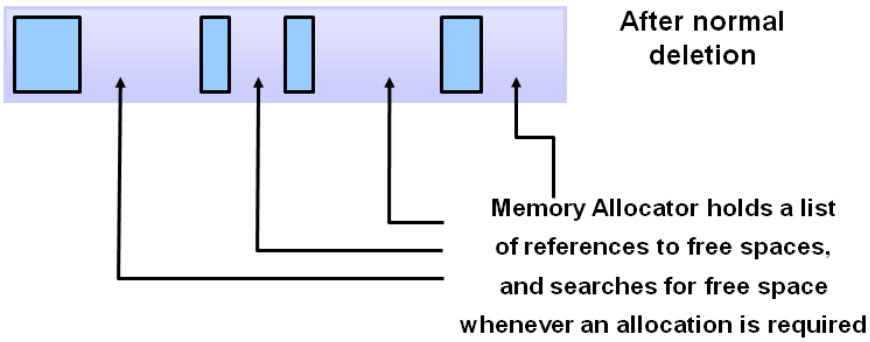
- What is GC?
  - GC is the mechanism to automatically **detect/delete** the unused objects.
  - Automatic garbage collection is the process of looking at heap memory, **identifying** which objects are in use and which are not, and **deleting** the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the **memory** used by an unreferenced object **can be reclaimed**.
- Why do we need it?
  - More efficiently manage dynamic memory allocation.
- Where is it implemented?
  - Stack
    - local varible, tracking method call flow, return address etc.

        - one per thread
    - **Heap**
        - dynamic memory allocation
        - The heap is where your object data is stored.
        - only one per JVM
- How does it work?
    - Step 1. **Marking**
        - This is where the garbage collector **identifies** which pieces of memory are in use and which are not.

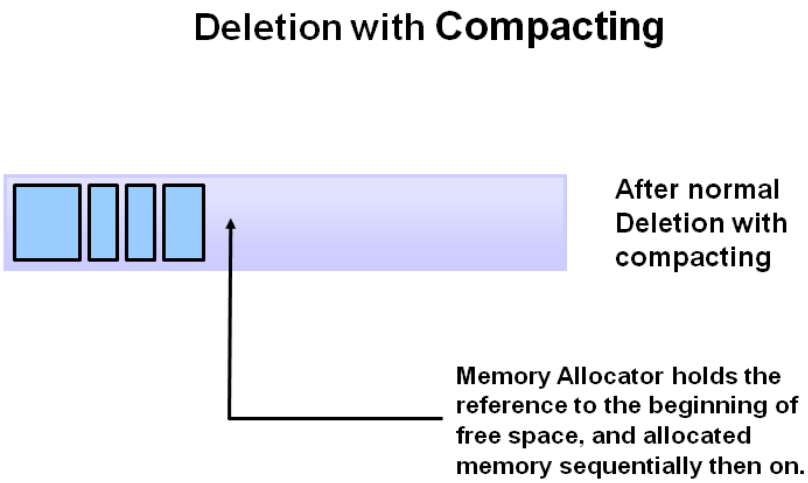


        -
    - Step 2. **Normal Deletion**
        - Normal deletion **removes** unreferenced objects leaving referenced objects and pointers to free space.
        - Inconvenient to allocate space again due to too **many fragments**, so we need compacting.

Normal Deletion

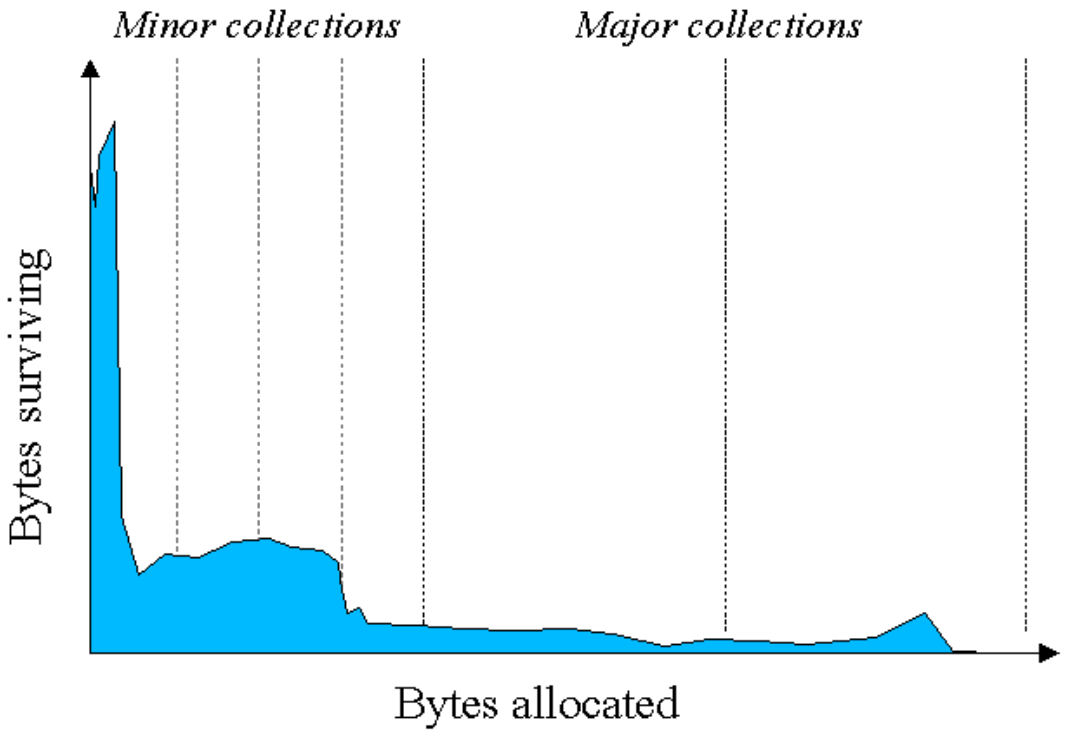


-
- Step 2a **Deletion with Compacting**
  - To further improve performance, in addition to deleting unreferenced objects, you can also **compact** the remaining referenced objects.
  - By **moving referenced object together**, this makes new memory allocation much easier and faster.



Why Generational Garbage Collection?

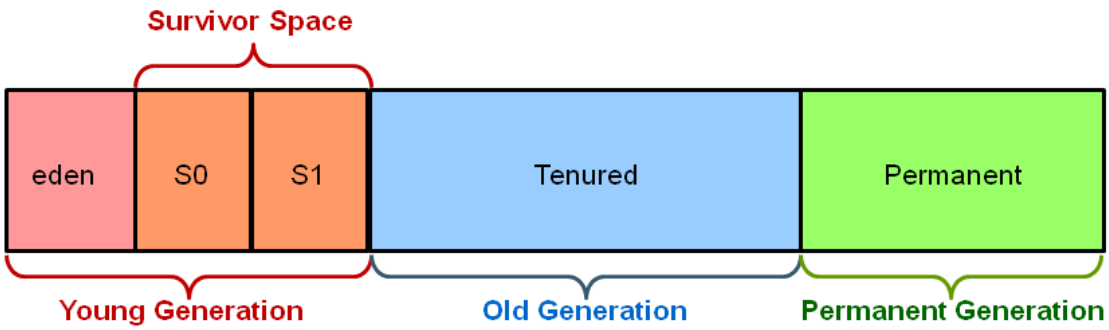
As stated earlier, having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short lived.



JVM Generations

The information learned from the object allocation behavior can be used to enhance the performance of the JVM. Therefore, the heap is broken up into smaller parts or generations. The heap parts are: **Young Generation**, **Old or Tenured Generation**, and **Permanent Generation**

Hotspot Heap Structure



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a **minor garbage collection**. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

Stop the World Event - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are always Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

How to know if an object can be GCed or not?

from a set of the root references/objects, find all the objects reachable through the paths on the dependency graph, other ones are not needed and can be GCed.

4 kinds of GC roots in Java

1. **Local varibles** are kept alive by the stack of a thread.
2. **Active Java threads**
3. **Static varibles**
4. **JNI (Java Native Interface) References**

finalize method

- **finalize method**
    - It is a method that the Garbage Collector always **calls just before the deletion/destroying** the object which is eligible for Garbage Collection, so as **to perform clean-up activity**.
    - **Clean-up activity** means *closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation*. Remember it is not a reserved keyword.
    - Once finalize method completes immediately Garbage Collector destroy that object. finalize method is present in Object class and its syntax is:

      ```
      protected void finalize throws Throwable{}
      ```

- **Exception in finalize method**
    - If Garbage Collector calls finalize method, while executing finalize method some unchecked exception rises then **JVM ignores that exception** and rest of program will be continued normally. So in this case the program termination is Normal and not abnormal.

Class Loader

1 2

5.6. Exceptions

Error indicates serious problems that a reasonable application should not try to catch.

- `StackOverflowError`

Exception indicates conditions that a reasonable application might want to catch.

`Error` and `Exception` extend from `Throwable`

`Error` and all exceptions in `Runtime Exception` can be not caught in the compiling.

`Exception` 是 `Throwable` 的子类，包含一些有用的函数。

Catch an exception

```
try {
    return;
} catch (IOException e) {
    return;
} catch (Exception e) {
    return;
} finally {
    return;
}
```

Define an exception

`Exception` 是 `Throwable` 的子类，包含一些有用的函数。

```
public class MyException extends Exception {
    public MyException(){
        super();
    }
    public MyException(String msg){
        super(msg);
    }
}
```

New feature in Java 7

```
try (BufferedReader br = new new BufferedReader(new FileReader(path))) {
    return br.readline();
}
```

old style:

```
BufferedReader br = new BufferedReader(new FileReader(path));
try {
    return br.readline();
} finally {
    if (br != null) br.close();
}
```

Throw vs. Throws

1. throw instance (throwable)
2. throws

throws is a keyword in Java, which is used in the signature of method to indicate that this method might throw one of the listed type exceptions.

- re-throw
- multiple catches

Java File

Stream

- Byte Stream
  - 8-bit byte
  - `FileInputStream`
  - `FileOutputStream`
- Character Stream
  - 16-bit unicode
  - `FileReader`
  - `FileWriter`
- Standard Stream
  - standard input
  - standard output
  - standard error

Common Java Read/Write Operations

- stream handle input/output byte by byte
- we want to do it line by line
  - `BufferedReader`
    - `.readLine()`

5.8. JUnit Test

- what kinds of tests do you know?
  - unit test (dev, component or function level, white box) *
  - integration test (see if your code can be integrated into other codes)
  - regression test (if new code breaks existing logic) *
  - smoke test
  - end-2-end test, feature test (external behavior test, black test) *
  - black/white box (white box is when you know how the functions are implemented inside)
  - performance test *
- agile 快速迭代 ci (continuous integration) + regression test

JUnit Test

- Annotations
- Assertions
  - `assertSame(Object a, Object b)` same references

```
public class CalTest {
    @BeforeClass
    public static void setupAll() {
    }

    @AfterClass
    public static void teardownAll() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    @Test
    public void test() {
        fail("Not yet implemented");
    }

    @Test(expected=NullPointerException.class)
    public void test2() {
        Cal c = new Cal();
        assertEquals(c.getMedian(new int{1,2,3}), 2);
    }
}
```

OOD

Objected-Oriented Design

Books Recommendation

for beginners: *head first java*

for intermediates:

- *effective java*
- Java source code (e.g. ArrayList, LinkedList, HashMap, PriorityQueue...)

Book on Computer Systems: *A programmer's perspective CSAPP*

Design Pattern Java tutorial The Pragmatic Programmer: from Journeyman to Master

Introduction

3 advices to answering design questions

1. 文（design）无第一，武（coding）无第二
    - 很有可能最优design不存在的，换一个场景可能就换了个解法
    - coding则会有好坏之分
2. 尽信书则不如无书，过程比结果更重要
3. 冰冻三尺非一日之寒

no silver bullet - 真正的银弹并不存在；所谓的没有银弹是指没有任何一项技术或方法可使软件工程的生产力在十年内提高十倍。

What is good code/design?

- Complete functionality
- **Easy to use (by others)**
    - clear, elegant, easy to understand, no ambiguity
    - prevent users from making mistakes
- **Easy to evolve**

Motivation of OOD

- Structured programming : code + data 代码和数据的合理组织
  - Each object is defined by two kinds of information: state/behavior.
    - state: field - what things it maintains;
    - behavior: method - what it can do.
- Limitedly exposed interfaces/API

Object-Oriented Design vs. Procedure-Oriented Design

-	**面向对象**	**面向过程**
解决问题的模块	函数（怎么做）	对象（谁来做）
角度	先具体逻辑细节，后抽象问题整体	先抽象问题整体，后具体逻辑细节
封装特性支持	继承、多态	重载、覆盖

Note:

重载：使用同名函数，但是形参不同，可以是个数，也可以是类型，顺序。

Concepts

- **Class** / **Object**
  - **class** - scheme
    - a blueprint for a data **type**
  - **object** - instance
    - a specific realization of any class
- **Encapsulation 封装**
  - encapsulate data and methods in class
  - **Data Abstraction** and **Access Lebels**
- **Inheritance 继承**
  - base class and derived class
  - In Java, **Interface**, **Abstract class** and Concrete class
- **Polymorphism 多态**
  - **Overriding**
  - In Java, `List.get()` has different implementations. It is different for each call of different class implementation.

Encapsulation

Data Encapsulation: Data Abstrction and Access Levels

- Providing only essential information to the outside world and hiding their background details
- Separate interface and implementation
- Access Labels: public, private, protected, default

Access	public	protected	private
Same class	yes	yes	yes
Derived class	yes	yes	no
Outside class	yes	no	no

In Java, add one access level - **no modifier**(**default**) and one kind of class - **package**

- no modifier is accessible to classes in the same package but not to a subclass

Note:

- 存在嵌套关系的package并不属于同一个package，import的时候也只会import父亲，而不会import子孙。

How to select appropriate access labels?

As strict as possible.

- API: public
- Internal Implementation: private
- Class inheritance: do we need to use protected for methods/attributes
  - Protected methods: sometimes useful when we want to override an implementation in subclasses
  - Protected attributes: be careful, try to use private first
- default in java(package-private)

General steps for OOD

1. Understand/Analyze the **use case**
   - 换位思考
   - use cases -> API, what is input/output?
   - from top level to bottom level

2. Classes and their relationships
    - Single-responsibility Pricinple - A class should have only one job
    - **Class relationships**
        - **Delegation**
            - **Dependency** (A use B)
                - temporate connection
            - **Association** (A has B)
                - permernent connection
                - eg. Vehicles ***are associated with*** Parking Spot
            - **Aggregation**/**Composition**(A owns B)
                - **Composition**
                    - House ***contains*** one or more rooms. Room's ***lifetime*** is controlled by House as Room will not exist without House.
                    - Parking Lot is composed of levels. Levels are composed of Parking Spots.
                - **Aggregation**
                    - Toy house built from blocks. You can ***disassemble*** it but blocks will ***remai n***.
        - **Inheritance/is-a**
            - Truck ***is a*** Car.
3. For complicated designs, first focus on public methods (APIs). Discuss the implementation details later!
    - Basic functionality: use case
    - Possible extensions (Product Road Map): provide available spot locations, ...

1. Complete implementation Details

```
public class ParkingLot{
    private Level[] levels;

    public boolean hasSpot(Vehicle v){
    // TODO
    }

    class Level{
        // boolean hasSpot(Vehicle)
    }

    class ParkingSpot{
        // boolean fit(Vehicle): check size and availability
    }

    class abstract class Vehicle{
        // getSize()
    }

    class car, bus, ... extends Vehicle
}
```

- 用enum常对象给VehicleSize的项目赋予size的属性。

```
public enum VehicleSize{
    Compact, Large
}
```

- use unmodifiable class

```
Colltions.unmodifiableList(list);
```

- **Do not include the mapping from Vehicle to ParkingSpot in the field of the Vehicle class.**
    - Reasons：逻辑不清楚，从属不清楚。实现容易出错，最好不要让对象相互环形持有对方。

```
HashMap<Vehicle, ParkingSpot>
```

Steps

1. Understand / Analyze the **functionality** and its **use case**

   What is the problem and its use case?

   Details: Describe the parking lot building? Vehicle monitoring? What kind of parking lot?

   use cases -> **functionalities** -> **APIs**

   functionalities

   - the main functionality
   - input/output of the main functionality
     - **Request** -> **Schedule & move elevator** -> Load onto elevator -> schedule and move elevator -> unload

```
**Modeling**


- Visble
    - Elevators
    - Users
    - Floors
- Invisble
    - Requests
    - Controller
    - Simulation program itself
```

1. Classes and their relationships

   Data - Classes and their member fields Action - Methods

   Single-responsibility Principle: A class should have only one job.

   Abstraction & **Decouple**: to sperate logics.

Example 1. Parking Lot

Example 2. In card games like **BlackJack**, the data are to describe the states of cards and game.

- Data - 公共道具（牌堆）

- Action - 流程规则

The actions are the rules and the game process.

1. What **data** are used.
2. What manipulation are done on the data.

Example 3. Elevators

Elevator

- State
  - id
  - maximum capacity
  - current laod size
  - current location
  - loaded requests
  - next destination
  - moving direction
- Behavior
  - load(weight)
  - unload(weight)
  - move design
    - X--moveTo(floor)--
    - moveUp()/moveDown() (stateless design)
    - move()/stop()/Change moving direction (stateful design)

Note:这里的目的是为了把电梯的行为拆分，分析的关键在于把调度的功能和移动的功能解耦。调度功能放在simulator里面。

Users

- State
  - current location
  - request
- Behavior
  - send request
  - enter elevator
  - leave elevator

Requests

Simulator

6.2. Design Pattern

Builder Pattern

Motivation

- **reduce the number of constructors**
- **control the exposure of limited data while having multiple options to initialize an instance**
    - Data can be accessed when initializing but not after.
- ensure when an instance is completedly initialized semantically
- Use: `User user = new User.UserBuilder("San", "Zhang")`
  `.age(25) .phone("1234567890") .address("Fake Address") .build()`

```
public class User {
    private final String firstName; // required
    private final String lastName; // required
    private int age;
    private String phone;
    private String address;
}

private User(UserBuilder builder){
    this.firstName = builder.firstName;
    this.lastName = builder.lastName;
    this.age = builder.age;
    this.phone = builder.phone;
    this.address = builder.address;
}

public static class UserBuilder {
    private final String firstName; // required
    private final String lastName; // required
    private int age = 0;
    private String phone = "";
    private String address = null;
    public UserBuilder(String firstName, String lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public UserBuilder age(int age){
        this.age = age;
        return this;
    }
    public UserBuilder phone(String phone){
        this.phone = phone;
        return this;
    }
    public UserBuilder address(String address){
        this.address = address;
        return this;
    }
    public User build(){
```

```
        return new User(this);
    }
}

public static void main(String[] args){
    User user = new User.UserBuilder("San", "Zhang")
    .age(25)
    .phone("1234567890")
    .address("Fake Address")
    .build()
}
```

Factory Pattern

Motivation

- Create objects **without specifying the exact class** of object that will be created. Separate instance/object creation logic from its usage
  - eg. We want lots of `Shape`s. Some of them can be `Rectangle`s. Some of them can be `Circle`s. We want to create those from its `Name`s.

Note

- Create lots of Shapes, including `Rectangle`, `Triangle` and `Circle` using the names or other properties.
- Do not need to know about the constructor detail.

Abstract Factory

Motivation

- Provides a way to encapsulate a group of individual factories that have a common theme without specifying the concrete classes.

Note

- abstract class `ControllerFactory` defines the way to create classes like `Button`, `InputBox` and other controllers.
- `IOSControllerFactory`, `AndroidControllerFactory` extends the abstract factory `ControllerFactory`.

Singleton

Motivation

- Ensure a class has o**nly one instance** and **provide a global access point** to that instance.

```
public class Singleton {
    private _static_ final Singleton INSTANCE = new Singleton();
    _private_ Singleton() {}

    _public static_ Singleton getInstance(){
        return INSTANCE;
    }
}
```

Operating Systems

7.1. Concurrency

Concurrency vs. Parallel

Concurrency - multiple tasks run simultaneoutly (Semantic Concept)

Parallel - multiple tasks **physically** run simultaneouly (Implementation level)

- Eg. multilane highway, multicore machines, Hadoop clusters

Explanations:

- Two definitions have nothing to do with the order of different tasks
- On a single core machine, we still have concurrency and parallel. (pretend to do so)
- 冯诺依曼体系：（每个核）一个时刻只能执行同一条**指令**。但是和上面概念不矛盾。

Ways to perform parallel programming (to physically launch multiple executors)

- Multiprocess

- multi-thread
- I/O multiplexing

Process vs. Thread

Ways	memory space	call stack	heap	os resource
Process	independent	independent	independent	independent
Thread	shared	independent	shared	shared

- Independent memory space is realized using **virtual memory** techniques in OS.
- Java concurrency is focusing on multithread cases in real life.
- 创建线程开销相对更小

Problems in multi-thread

- Communication overhead
- Resource isolation (fault tolerance)
- Creation/destroy overhead

Java Thread

Steps:

- create the thread objects
  - each Java thread is mapped to one System thread (managed and scheduled by OS kernel rather than in user space)
- tell the threads what you want them to do
- start the thread

Ways of creating threads and make them running:

1) extends Thread

```
Thread t = new Thread() {
    @Override
    public void run(){
        System.out.println("Hello");
    }
};
t.start(); // schedule the created thread and make ready to go
//
// concurrency here
System.out.println("Main Thread"); // main thread
//
t.join(); // make sure the thread finished after this line
```

2) implements Runnable (for the purpose of being not conflict with single inheritence rule)

```
interface Runnable {
    void run();
}

class HelloRunnable extends Client implements Runnable {
    @Override
    public void run(){
        //..
    }
}

Thread newThread = new Thread(new HelloRunnable());
newThread.start();
newThread.interrupt();
```

- **When the JVM will exit?**
  - **no alive non-daemon threads**
    - default thread is non-daemon
    - create daemon thread using `Thread.setDaemon()` method

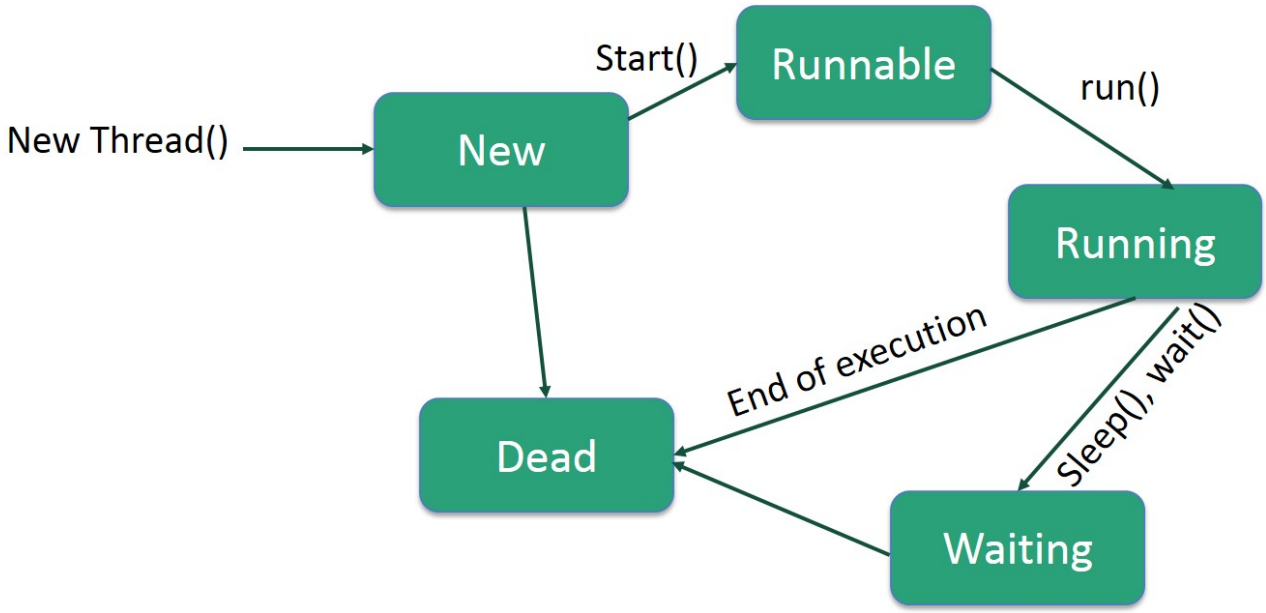
Thread Operations

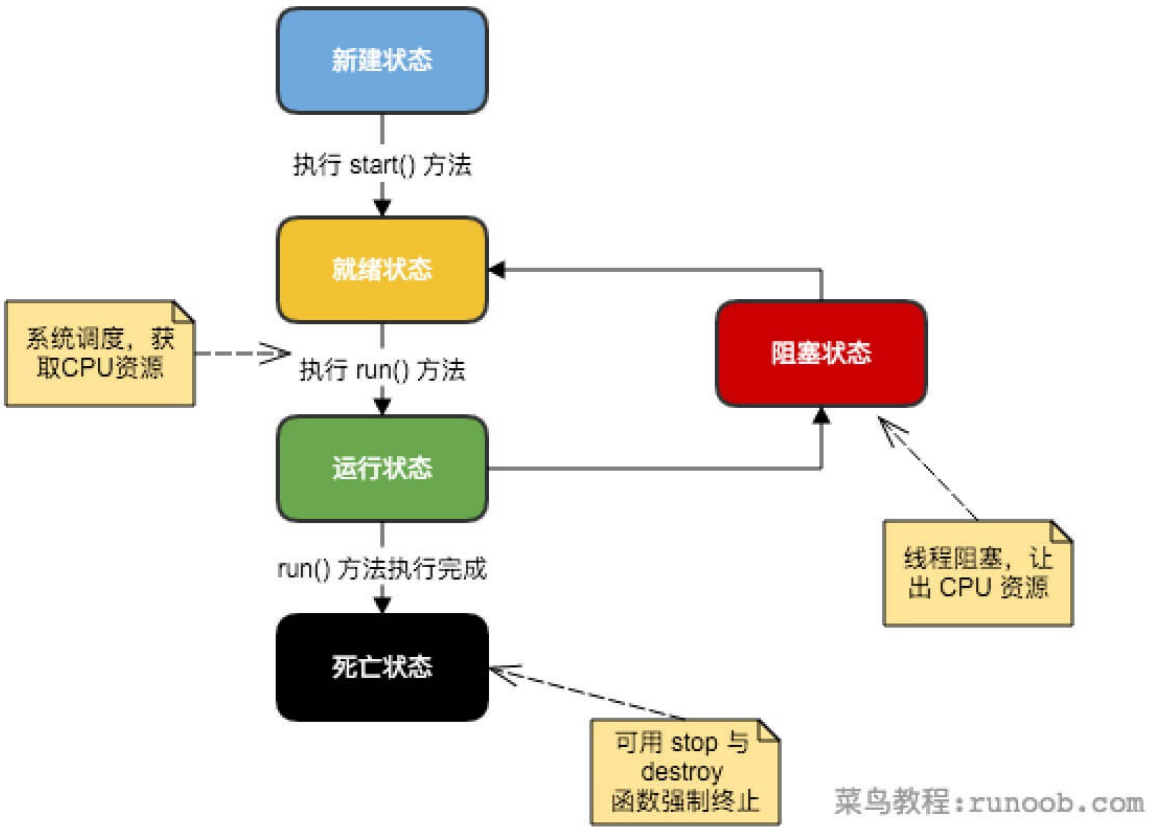
- `.start()`
- `.run()`
- `.join()`

- `.sleep()`
- `.yield()` - 让出CPU资源。通常不需要手动使用。
- `.interrupt()`

How do we determine which line is run at each time?

- Scheduler + Queues: threads/processes are waited in queues, and there is a scheduler controls which thread/process is going to be run at the current time
- Each thread accepts a sequence of executed instructions





- Ways to block a thread
  - `.sleep()` / `.suspend()`
  - `.wait()`
  - wait for synchronized lock
  - `.join()`
  - other blocks

Synchronous procedure call Vs. Asynchronous procedure call

Thread or process synchronization#Thread_or_process_synchronization)

> Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as critical section.

Asynchrony)

异步调用的三种方式

1. 单向调用：客户端发出请求之后不再关心服务端的情况
2. 延迟响应：在第一次发出调用请求的时候，服务端需要返回一个称为票据（Ticket）的对象。客户端通过后续的调用查询请求结果。票据对象会作为第二次发出检索结果请求时的一个参数。（有些地方称为同步非阻塞，因为通过轮询方式不断地查询本身就是同步的）
3. 请求回调：客户端发出调用请求，服务端立刻返回。服务端得到请求的结果之后，通过回调方式通知客户端触发响应。

关于Tornado

> Tornado is a Python web framework and **asynchronous networking library**, originally developed at FriendFeed. **By using non-blocking network I/O**, Tornado **can scale** to tens of thousands of open connections, making it ideal for long polling, WebSockets, and other applications that require a long-lived connection to each user.

- Tornado provides asynchronous functions (in its library) to call
- Tornado ioloop is condition synchronization by epolling for non-blocking
- By using non-blocking I/O, we can increase our programme's performance

Synchronization & race

Synchronization - the **coordination** of events to operate a system in unison.

Impose orders on (previously) concurrent events. 我们人为提供时间执行的顺序。

How? Locks (synchronized), concurrent data structures, wait/notify (condition synchronization), volatile, ...

Data races - if two "conflicting operations" are in different threads and are not properly synchronized (concurrent), they will introduce data races.

冲突可能发生在CPU指令层次，最后会导致读操作对写操作产生影响。所以一定要同时对读操作和写操作采用合适的同步操作。

3 factors can form a data race

- More than one operations work on the same memory location
- At least one oepration is a `write`
- At least two of those opreations are concurrent

Mutual exclusion, critical section, and lock

The most typical way to get rid of data races: **locks**

Or, on the semantic side: **mutual exculsion**

lock锁的是代码不是资源

No two concurrent processes are in their critical section at the same time. And a **critical section** can be defined as:

A part of a multi-process program that may not be concureently executed by more than one of the program's processes/threads.

How to create a critical section?

- A general lock has two operations: lock and unlock.
- Lock: wait no one is there and go into the critical section
- Unlock: Leave the critical section and mark no one is here

Our goals in parallel programming

1. Data race freedom
2. determinism

Lock **Granularity** 粒度:

- coarse grained lock (CGL) 粗粒度锁
- fine grained lock (FGL) 细粒度锁
- critical section
    - avoid super slow opeartions in a **critical section**, such as IO

Mutual exclusion in Java

1. synchronized keyword

```
private int value;
public void increase() {
    synchronized(this) {
        value++;
    }
}

public synchronized void decrease(){
    value--;
}
```

- synchronized on non-static method == synchronized(this)
- synchronized on static method == synchronized(Counter.class) != synchronized(all instances)
- `synchronizedMap` is a wrapper of `Map`
- `concurrentHashMap` separates HashMap into segements and perform sync operations on each one of them separately (sharding, federation).

Locks in Java

```
Lock L = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

ReentrantLock

Dead Lock

- **Defintion**
  - P1: holding Lock 1, trying to acquire Lock 2
  - P2: holding Lock 2, trying to acquired Lock 1
- **Conditions to form a deadlock**
  - **Mutual Exclusion**
    - Cannot avoid. We want mutual exclusion for synchronization
  - **Hold and Wait** or **Resource Holding**
    - Can avoid by using one lock at a time
  - **No Preemption** (抢占)
    - Cannot avoid. We do not have this kind of mechanism sometimes
  - **Circular Wait**
    - Can avoid by arranging the order of locks

Live Lock

live lock will happen if each side is actively resolving the problem (backoff and retry)

Solutions

- priority
- randomize

Atomicity

volatile

- guarantees **visibility** (semantically, happens-before), much like acquiring and releasing a lock.
  - **为单个变量提供独占性，避免data race**
- does not guarantee **atomicity**
  - only guarantees single read/write operation is atomic. But in a read-then-update case (or more complicated), we want to gurantee `value++` is atomic.
  - use `synchronized` or `AtomicInteger` instead
  - use double checked locking in Singleton case
- **Common use case**
  - flag to stop (read/write only for each thread)

```
public volatile boolean flag = false;
```

Condition synchronization

Think this in real life: wait for some important mail. I can:

- keep waiting there (very low efficiency)
- periodic check and go (better efficiency, but may delay, how long shall I wait?)
- use tracking service and notification it will send you messages when it comes! (high efficiency, almost no delay)

Here we can do similar things: instead of busy waiting or periodic checking, we need a mechanism to:

- allow threads to wait on a condition
- (Hopefully) Allow a thread satisfies the condition to notify all waiting threads

Producer consumer problem

Build a **blocking queue**, such that:

- when the queue is empty, consumer will wait until produce provide one element
- when the queue is full, producer will wait until consumer release one element

Both operations happens in a mutual exclusive fashion so that all operations are ordered.

- Mutual exclusive -> use locks
- Wait -> condition synchronization

What is **Monitor**?

- There are two queues controlling the access to enter the room.
- lock queue, there is only one thread at a time from the queue can enter
- wait queue, there is a queue for the condition provided, all the thread wait(), is in the queue, and wait for the notify() call to remove from the condition queue.

BlockingQueue Realization Using Wait NotifyAll

```
class Q {
    private Queue<Integer> q;
    private final int limit;
    public Q(int limit) {
        this.q = new LinkedList<>();
        this.limit = limit;
    }

    public synchronized void put(Integer ele) {
        while (q.size() == limit) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        if (q.size() == 0) { // Comsumer wait under only on circumstance that the queue is emp
            notifyAll();
        }
        q.offer(ele);
    }

    public synchronized Integer take() {
        while (q.size() == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrack();
            }
        }
        if (q.size() == limit) {
            notifyAll();
        }
        return q.poll();
    }
}
```

Other Languages

8.1. Python

Basics

```
def main():
    x = [1, 2, 3, 4]
    for i in range(len(x)):
        print("The %dth number of List x is %d".format(i, x[i]))

if __name__ == "__main__":
    main()
```

print

python3: {} python2: %d

delivering function parameters

for loop

range is removed in python3 xrange is replaced with range for i in range(end, start-1, -1): print(i)

devide

1/2=0.5 in python3 1//2 = 0 in python3

File Input and Output

```
file = open("input.txt")
x = file.readline()
print(x)
```

String Format

Python2 vs Python3 https://pyformat.info/

Basic Operation

List

python2: zip() python3: list(zip())

Set

a = set([2,3,4]).union([1,2,3])

Dict

b = {}.get(1, None) elegantly initialize dict a={}.fromkeys([1,2,3],0)

enumerate

1 in {1:2} is faster than 1 in {1:2}.keys()

Map key to multiple values

http://python3-cookbook.readthedocs.io/zh_CN/latest/c01/p06_map_keys_to_multiple_values_in_dict.html

Deque

```
q = collections.deque()
course = q.popleft()
q.append(i)
```

Generator

Generators

- Generator Pattern is to make iterator
- Python make Generator more simplified by using `yield`

Python异步编程

参考：

- 深入理解Python异步编程（上）
- 代码参考
- Tornado关于异步非阻塞IO的文档
- Tornado关于协程的文档
- 解释Future那一块的文字

假设存在业务逻辑 A --> B --> C

A，B，C内部都是同步的，并且把阻塞等待IO的操作用注册回调的方式取代了。

A，B，C可以是别人写好的异步库，也可以是自己写的异步的逻辑。

异步A，B负责：

- 调用注册回调函数，以便让程序进入下一个状态。
  1. 回调函数负责**提取IO返回的结果**，并存储在Future对象里
  2. 回调函数也负责执行**返回上文**操作。
- A，B都会返回Future对象，对象内部的内容记录了回调函数和A，B操作执行后IO返回的结果。

Future的设计目标是作为协程(coroutine)和IOLoop的媒介，从而将协程和IOLoop关联起来。

```
class Future:
    def __init__(self):
        self.result = None
        self._callbacks = []

    def add_done_callback(self, fn):
        self._callbacks.append(fn)

    def set_result(self, result):
        self.result = result
        for fn in self._callbacks:
            fn(self)
```

协程写法可以如下：

```
def app():
    yield A()
    yield B()
    yield C()
```

事件驱动的主Loop负责检查Event，然后执行注册Event的回调函数如下：

```
def loop():
    events = selector.select()
    for event_key, event_mask in events:
        callback = event_key.data
        callback()
```

使用Task对象的Step函数

1. **调用上文，并把下文的入口放在上文里**
2. **把IO输出的结果输出给下文**

实现的方式是在上文返回的Future对象中，加入下文入口的回调函数采用。类似于Tornado的Runner：

```
class Task:
    def __init__(self, coro):
        self.coro = coro
        f = Future()
        f.set_result(None)
        self.step(f)

    def step(self, future):
        # 1. 执行上文，得到返回的Future对象
        try:
            next_future = self.coro.send(future.result)
        except StopIteration:
            return

        # 2. 把下文入口加入到上文的Future的回调中
        next_future.add_done_callback(self.step)
```

为什么要大费周章发明异步框架？

- 性能好
- 代码逻辑易于理解（类同步写法）

问题收集

1. 内存泄漏的产生方式 【内存泄漏】没有释放被分配的内存，最后可用的内存会越来越少。 【野指针】访问被释放的内存。
2. 全局变量和局部变量的区别 作用域不同。全局变量在任何函数都可以访问，但是局部变量只能在当前函数内访问。
3. 全局变量添加static关键字 加static的关键字只会被初始化一次。加static的全局变量不能被其他文件使用。