Study notes for some algos

http://wdxtub.com/interview/14520609088903.html

数组和字符串

解题策略

* 一般来说，一旦出现“unique”，就落入使用哈希表或者bitset来判断元素出现与否的范畴。
* 一旦需要统计一个元素集中元素出现的次数，我们就应该想到哈希表。

Hash Table:

* Time complexity Read O(1) Write O(1)
* C++标准库中提供map容器，底层以平衡二叉搜索树的方式实现, 严格来说，map并不是一个哈希表，原因是查找时间从O(1)变为了O(log n).
* 在C++11中，标准库添加了unordered\_map，更符合哈希表的传统定义，平均查找时间O(1)。

[String searching algorithm](https://en.wikipedia.org/wiki/String_searching_algorithm)

* Brute-Force算法：

顺序遍历母串，将每个字符作为匹配的起始字符，判断是否匹配子串。时间复杂度 O(mn)。

* [Rabin-Karp算法](https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm)：

将每一个匹配子串映射为一个哈希值。例如，将子串看做一个多进制数，比较它的值与母串中相同长度子串的哈希值，如果相同，再细致地按字符确认字符串是否确实相同。顺序计算母串哈希值的过程中，使用增量计算的方法：扣除最高位的哈希值，增加最低位的哈希值。因此能在平均情况下做到O(m+n)。

* [KMP算法](https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm)

[易懂说明](https://liam0205.me/2016/12/20/KMP-Algorithm/)：建立匹配表让搜索跳跃式前进，对每一个 j，在模式串中寻找最大的 k，使得 P[0:k] == P[j - k:j]。遇不同，查找表中位置回退跳跃比较。

j: 0 1 2 3 4 5 6

P: a b a b a c b

T: -1 0 0 1 2 3 0

* [Boyer-Moore字符串搜索算法（常用最优）](https://zh.wikipedia.org/wiki/Boyer-Moore%E5%AD%97%E7%AC%A6%E4%B8%B2%E6%90%9C%E7%B4%A2%E7%AE%97%E6%B3%95)

[易懂说明](http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html) or [Example](http://www.cs.utexas.edu/~moore/best-ideas/string-searching/fstrpos-example.html)：从搜索词尾端开始比较，对齐跳跃

Stack: LIFO: For depth first search

Queue: FIFO: For breadth first search

链表

## 快慢指针应用：

* 快慢指的是指针向前移动的步长
* 快速找出未知长度单链表的中间节点：＊f and ＊s指向头节点，＊f移动速度是＊s两倍，＊f指向结尾时，＊s正好在中间
* 判断单链表是否有环：＊f and ＊s指向头节点，＊f移动速度是＊s两倍，＊f＝＝null则无环，＊f==＊s则有环
* 找倒数第 N 个节点：＊f and ＊s指向头节点，＊f先移动N步，然后一起前进，＊f到达结尾，＊s即为倒数第N个节点

递归和动态规划

动态编程的核心在于，如果在一个问题的解决方案中，子问题被重复计算，那么就可以利用记录中间结果，达到用空间换取时间的目的

## 算法策略

* 分而治之(Divide and Conquer)

这里只谈狭义的D&C，即将问题分成几个部分，每一部分相互独立，互不重叠，假定每个部分都可以得到解决来进行递归调用，合并每一部分的结果。例如Merge Sort， Quick Sort (Merge Sort的divide容易，但Conquer/Merge复杂，Quick Sort的divide复杂，但Conquer/Merge容易)

* 动态编程(Dynamic Programming)

尽可能不重复计算每个子问题，而是将计算结果存储下来，以确定后驱问题的解。与贪心算法的区别是，会记录下所有可能通向全局最优解的局部解，以便在计算后驱问题时综合考虑多个前驱问题的解。

* 贪婪算法(Greedy Algorithm)

只做出当下最优的判断，并且以此为基础进行下一步计算。当前判断最优时，不考虑对全局/未来的影响，所以所以从全局来说并不能保证总是最优。贪心算法每次更新当前的最优解。如Dijkstra算法就是贪心算法的实例之一。

* 回溯 (Backtracking)

一种暴力(穷举)的深度优先搜索法：搜索，直到节点空间的尽头，然后再返回到上次的节点，再往其他方向深度搜索。树或图的DFS是回溯的实例之一。

广度优先

先访问当前节点，一旦发现未被访问的邻近节点，推入队列，以待访问. 白色表示未被发现，灰色表示被发现，黑色表示已访问

BFS(G, s)

For each vertex u exept s

Do Color[u] = WHITE

Distance[u] = MAX

Parent[u] = NIL

Color[s] = GRAY

Distance[s] = 0

Parent[s] = NIL

Enqueue(Q, s)

While Q not empty

Do u = Dequeue(Q)

For each v is the neighbor of u

Do if Color[v] == WHITE

Color[v] = GRAY

Distance[v] = Distance[u] + 1

Parent[v] = u

Enqueue(Q, v)

Color[u] = BLACK

深度优先

public void DFS(GraphNode start){

Stack<GraphNode> s = new Stack<GraphNode>();

q.push(start);

visited.add(start);

while(!s.empty()){

GraphNode cur = s.pop();

for(GraphNode next: cur.children){

if(!visited.contains(next)){

s.push(next);

visited.add(next); // mark node as visited when adding to stack!

}

}

}

}

cycle detection using DFS: 当前节点的邻居有灰色。因为该节点状态为灰, 即还没有把它以后的节点全部遍历, 所以当前节点v肯定可以从u到达, 而现在又可以从v到达u, 所以构成一个回路

### 单源最短路径问题

[Dijkstra算法](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

* 指从某个节点出发，到其他节点的最短距离
* 仅适用于均为非负权值边的情况，O(V^2)
* 利用贪心的思想，在剩下的节点中选取离源点最近的那个加入集合，并且更新其邻近节点到源点的距离，直至所有节点都被加入集合

**function** Dijkstra(*Graph*, *source*):

create vertex set Q // priority queue

**for each** vertex *v* in *Graph*: *// Initialization*

dist[*v*] ← INFINITY *// Unknown distance from source to v*

prev[*v*] ← UNDEFINED *// Previous node in optimal path from source*

add *v* to *Q* *// All nodes initially in Q (unvisited nodes)*

dist[*source*] ← 0 *// Distance from source to source*

**while** *Q* is not empty:

*u* ← vertex in *Q* with min dist[u] *// Node with the least distance will be selected first. Terminate if only a known target is wanted*

remove *u* from *Q*

**for each** neighbor *v* of *u*: *// where v is still in Q.*

*alt* ← dist[*u*] + length(*u*, *v*)

**if** *alt* < dist[*v*]: *// A shorter path to v has been found*

dist[*v*] ← *alt*

prev[*v*] ← *u*

**return** dist[], prev[]

* If only interested in a shortest path between *source* and *target*, terminate the search if *u* = *target*. Now read the shortest path from *source* to *target* by reverse iteration:

*S* ← empty sequence

*u* ← *target*

**while** prev[*u*] is defined: *// Construct the shortest path with a stack S*

insert *u* at the beginning of *S* *// Push the vertex onto the stack*

*u* ← prev[*u*] *// Traverse from target to source*

insert *u* at the beginning of *S* *// Push the source onto the stack*

Bellman-Ford算法

* 适用于一般情况(包括存在负权值的情况，但不存在从源点可达的负权值回路
* O(V\*E)
* Dijkstra算法的基本操作“拓展”是在深度上寻路，而relax操作则是在广度上寻路，这就确定了贝尔曼-福特算法可以对负边进行操作而不会影响结果。

**function** BellmanFord(*list* vertices, *list* edges, *vertex* source)

::distance[],predecessor[]

*// This implementation takes in a graph, represented as*

*// lists of vertices and edges, and fills two arrays*

*// (distance and predecessor) with shortest-path*

*// (less cost/distance/metric) information*

*// Step 1: initialize graph*

**for each** vertex v **in** vertices:

distance[v] := **inf** // At start, all vertices weighted infinity

predecessor[v] := **null** // And a null predecessor

distance[source] := 0 // Except for the Source, where the Weight is zero

*// Step 2: relax edges repeatedly*

**for** i **from** 1 **to** size(vertices)-1:

**for each** edge (u, v) **with** weight w **in** edges:

**if** distance[u] + w < distance[v]:

distance[v] := distance[u] + w

predecessor[v] := u

*// Step 3: check for negative-weight cycles*

**for each** edge (u, v) **with** weight w **in** edges:

**if** distance[u] + w < distance[v]:

**error** "Graph contains a negative-weight cycle"

**return** distance[], predecessor[]

Floyd-Warshall算法

* 解决任意两点间的最短路径的一种算法
* 原理：设D(i,j,k)为从i到j的只以（1…k）集合中的节点为中间节点的最短路径的长度
  + 若最短路径经过k，则D(i,j,k)=D(i,k,k-1)+D(k,j,k-1)
  + 若不过k，则D(i,j,k)=D(i,j,k-1)

**let** dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)

**for each** vertex *v*

dist[*v*][*v*] ← 0

**for each** edge (*u*,*v*)

dist[*u*][*v*] ← w(*u*,*v*) *// the weight of the edge (*u*,*v*)*

**for** *k* **from** 1 **to** |V|

**for** *i* **from** 1 **to** |V|

**for** *j* **from** 1 **to** |V|

**if** dist[*i*][*j*] > dist[*i*][*k*] + dist[*k*][*j*]

dist[*i*][*j*] ← dist[*i*][*k*] + dist[*k*][*j*]

**end if**

[Sort](https://zh.wikipedia.org/wiki/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95)

QuickSort

**function** partition(a, left, right, pivotIndex)

pivotValue := a[pivotIndex]

swap(a[pivotIndex], a[right]) *// 把pivot移到結尾*

storeIndex := left

**for** i **from** left **to** right-1

**if** a[i] <＝ pivotValue

swap(a[storeIndex], a[i])

storeIndex := storeIndex + 1

swap(a[right], a[storeIndex]) *// 把pivot移到它最後的地方*

**return** storeIndex

**procedure** quicksort(a, left, right)

**if** right > left

select a pivot value a[pivotIndex]

pivotNewIndex := partition(a, left, right, pivotIndex)

quicksort(a, left, pivotNewIndex-1)

quicksort(a, pivotNewIndex+1, right)