FEIT 飞天教育 EDU

— 选择飞天，重仓未来 —

期末总复习课程 – COMP2123

by

梨老师

FEIT EDUCATION

飞天教育

FEIT小助手【USYD】

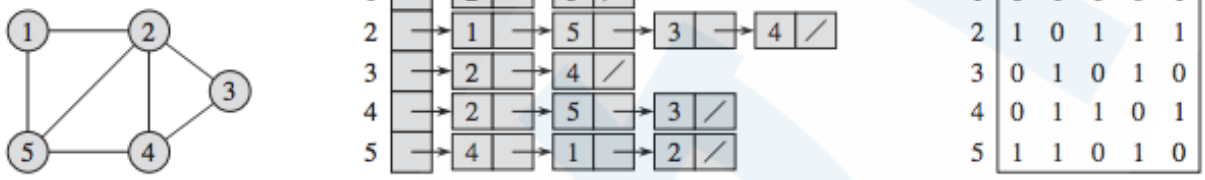
澳大利亚 悉尼

Table of Contents

Graph

Graph 和 Tree 的知识点联系:
两者都可以用 Node 结构来表示.
Tree 只能表达无环的, 一对多的关系.
Graph 可以表达有环的, 多对多的关系. 一般认为 Tree 是 Graph 的特殊情况, 即可以用 graph 的形式表达 Tree.

Graph 的表达形式与术语说明



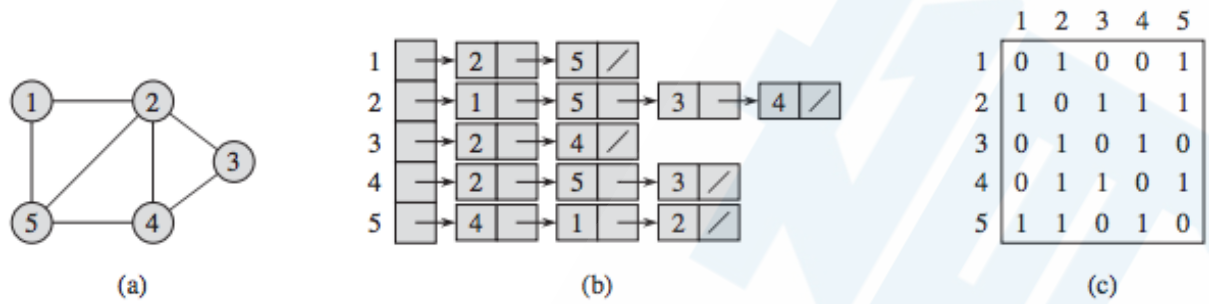
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

V: Node/Vertices 点
E: Edges 边
neighbour: 邻居, 一个 vertex 只需要走一条边就能到达的其他 vertex.
Degree of a vertex: 一个 vertex 的邻居个数, 简称 deg(V)
Directed Graph: 有向图.
意思是比如有一条边从 A -> B, 并不意味着也同时有一条边从 B -> A.
这样的图里, 如果给了条件 A->B, A 的邻居可以有 B, 但 B 的邻居并没有 A.
所以, 有向图里如果 Edge(u, v)并不等同于 E(v, u).
Undirected Graph: 无向图.
意思是比如有一条边从 A -> B, 那同时这条边也从 B -> A, 就可以写成 A-B .
这样的图里, 如果给了条件 A->B, 如果 A 的邻居有 B, 那 B 邻居里就有 A.

Path: 路径, 比如 [1-2, 2-3, 3-4] 就可以构成由 1 到 4 的一条路径
Simple Path: 一条路径里所有的 vertex 都不出现第二次. 换言之, simple path 没有 cycle. 有 cycle 的 path 例如: [1-2, 2-3, 3-1], 从 1 回到了 1, 就构成了环.
Cycle: 正式定义是由 [v_1-v_2, v_2-v_3, ... v_{k-1}-v_k] 组成的一条 path 中, $v_1 = v_k$ (k>2), 而且前 k-1 个点都不相同.
简单的解释就是一条 path 开始和结束是重复的点, 从某点开始又回到了自己.

Connected graph: 一个 graph 里任意两个的 vertex 都可以通过某条 path 连在一起.
Tree: 树. 如果(undirected)图既是 connected, 也没有 cycle. 一个
Rooted tree: 有 hierarchy 层次结构的树, 你可以分清楚 parent child 这样的结构. 比如大家熟悉的二叉树 binary tree 就是这种.
Spanning Tree: 在一个 graph 里, 如果某一个 tree 可以连接所有图中的点,

图的实现方式



Adjacency matrix

简称 adj matrix

它的表达方法很直观. 每一行是 node, 列也是 node, 两个 node 之间是否有 edge 记为成 0 或者 1.

比如我们有三个 node 的一个 graph:

V = {1,2,3,4,5}

E = {1-2, 2-4, 2-3, 3-4, 5-4, 1-5, 2-5}

那么写成 adj matrix 一般可以这样:

N = 5

adj_matrix = [[0 × N] × N]

#我们并不在意一个 Vertex 自己到自己的情况, 比如[1][1]的情况. 可以保留 0.

adj_matrix[1][2] = 1, adj_matrix[2][1] = 1

adj_matrix[2][4] = 1, adj_matrix[4][2] = 1

adj_matrix[2][3] = 1, adj_matrix[3][2] = 1,

adj_matrix[3][4] = 1, adj_matrix[4][3] = 1,

adj_matrix[5][4] = 1, adj_matrix[4][5] = 1,

…

因为可以直接访问 index, Adjacency List 可以很方便的存储"两条边之间是否存在 Edge 的信息.

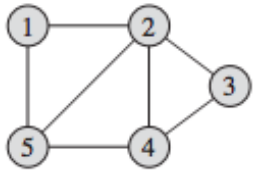
- 空间: $O(n^2)$.
- 检查某两个 vertex 之间是否有 edge 需要 O(1)的时间
- 已知这样的一个 matrix, 获得所有 edge 信息所需要的时间也是 O(n^2). 因为要把这个二维数组扫一遍.
- 如果是无向图(Undirected graph)的话只需要填写一半的空间, 或者可以这么说, 每当写入 [A][B]=1 的时候也写入 [B][A]=1. 这样出来的 matrix 应该是沿着对角线对称的.

Adjacency List

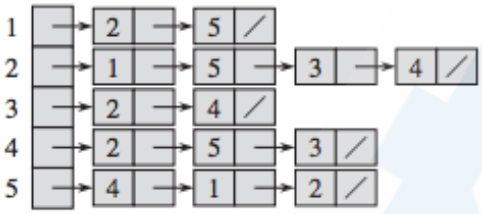
比较经典的例子就是用 dictionary/map 来存所有的 vertex 作为 key, 存储的值为每个 vertex 的 neighbour.

有两种选择: 每个值对应的 neighbour list 里可以是 edge 也可以是 node.

因为我们只需要一个 node + 一个 edge 就可以推出另一个端点 node. 反之亦然.



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

```
class Graph:
  def __init__(self):
    self.vertices = []

class Edge:
  def __init__(self, u, v, val):
    self.u = u
    self.v = v
    self.val = val

class Vertice:
  def __init__(self, v):
    self.val = v
    self.edges = [] #this also records neightbour

  def add_edge(self, e):
    self.edges.append(e)
```

以下是在 neighbour 里直接存 node 的表现形式:

```
class Graph:
  def __init__(self):
    self.graph = defaultdict(list)

  def addEdge(self,u,v):
    self.graph[u].append(v)
```

- 空间: m + n (edge 个数 + vertex 个数)
- 检查某两个 vertex 之间是否有 edge, 需要 O(degree of vertex)的时间

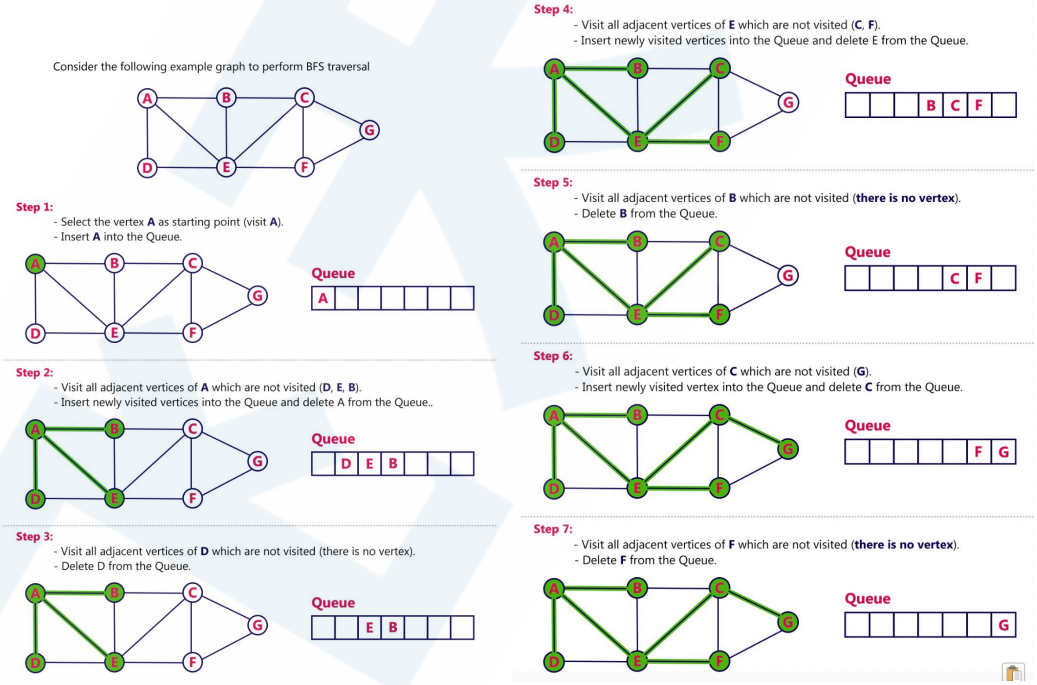
比如我们想求 vertex 1, 0 之间有没有一条边. origin 是 1, destination 是 0.
那么我们需要通过这个字典 O(1)时间找到 1 这个 key 存储邻居用的 list.
因为这里用了 list 来村邻居, 所以找到 destination 需要的时间是这个 key **所有的邻居个数** 也就是 **degree of vertex**.
这个例子里, 1 有两个邻居, 2 和 3. 找到了底也没有发现 0, 所以可以返回有或者没有这条边的结论.

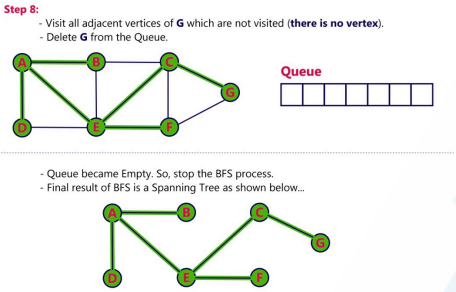
BFS

BFS 适合拿来解决的问题包括:

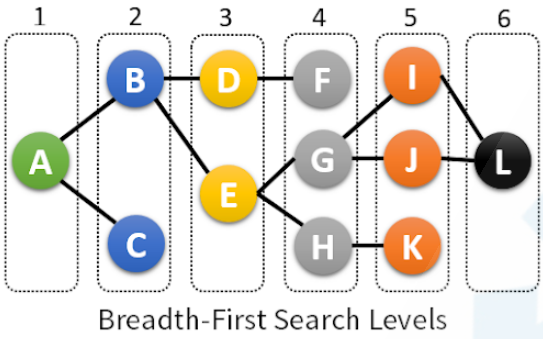
1. 最短路径问题
2. 需要由近及远的一层层的遍历一个图的问题.
   比如获取从当前 vertex 到其他 vertex 需要走几步.

```
class Graph:
    . . .
    def BFS(s):
        # Mark all the vertices as not visited
        visited = [False] * (len(vertices))
        queue = []
        # Mark the source node as visited and enqueue
        queue.append(s) # queue.enqueue()
        visited[s] = True
        while len(queue) > 0:
            # Dequeue a vertex from queue and 'visit it'
            s = queue.pop(0) # queue.dequeue()
            print (s)
            # Get all adjacent vertices of the dequeued vertexs.
            # If a adjacent has not been visited, then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```





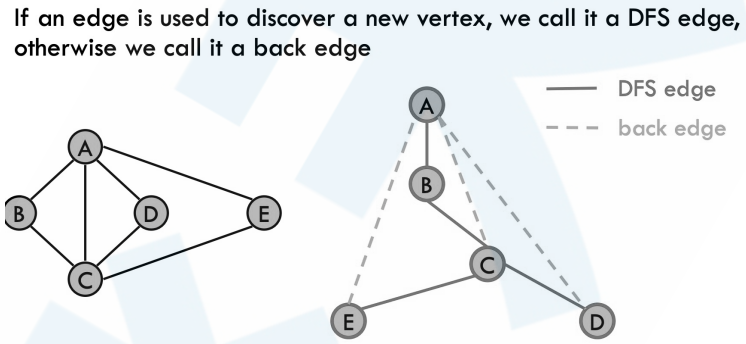
大家可以看到 BFS 的 visit 顺序是向洋葱一样一层一层由内而外的.



所以, 这个方法可以帮我们找到最短路径.
如果是没有 weighting 的最短路径问题, BFS 由于会给你层数, 直接可以用.
如果是有 weighting 的最短路径问题, 则需要对 BFS 进行一些修改.

DFS

DFS 概念里有 **DFS edge** 和 **back edge** 这两种 edge.
DFS edge 是你在某个 node 上第一次 visit 某个邻居的路径时经过的那条路.
其他 edge 都是 back edge.



DFS 适合拿来解决的问题包括:
1. 从一个 vertex 可以到达的所有其他点
2. 搜索, Backtracking 类型的问题, 比如 print all path between S to T.
3. 图是否有 cycle

```
class Graph:
  ...
  def DFS(s):
    # Mark all the vertices as not visited
```

```
    visited = [False] * (len(vertices))
    stack = []
    # Mark the source node as visited and enqueue
    stack.push(s)
    visited[s] = True
    while len(stack) > 0:
      # Dequeue a vertex from queue and 'visit it'
      s = stack.pop()
      print (s)
      # Get all adjacent vertices of the dequeued vertexs.
      # If a adjacent has not been visited, then mark it visited and enqueue it
      for i in self.graph[s]:
        if visited[i] == False:
          stack.push(i)
          visited[i] = True
```

有一种代码更短更简单, 但是实际写起来比较难 debug 的递归式写法.
其作用与上面的非递归式样方法相同.

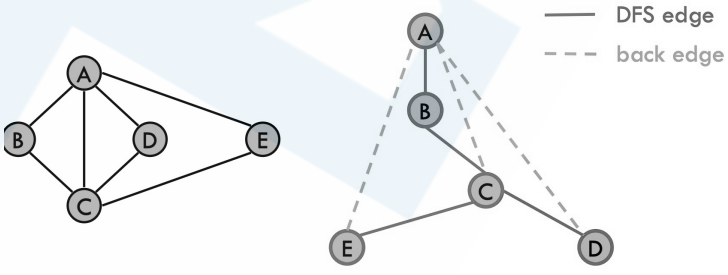
```
def DFS(v, visited):
  # Mark the current node as visited and print it
  visited[v] = True
  print(v)
  # Recurion for all the vertices adjacent to this vertex
  for neighbour_node in self.graph[v]:
    if visited[neighbour_node] == False: # if it is not visited -> backedge.
      DFS(neighbour_node, visited)
DFS(vertex, [False for every vertex])
```

- **Run time analysis**

对于 adjacency list 的实现方法, BFS 和 DFS 都需要$O(m+n)$的时间.
DFS 和 BFS 的优势和劣势
BFS:对于解决最短或最少问题特别有效，而且寻找深度小.
DFS: 可以找环和 connectiviey 问题, 但不能找最短路

例题

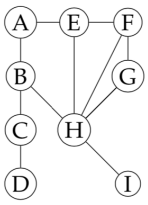
如何在 O(N)时间内确定一个 Graph 是否有环?



用 DFS, 如果我们在访问自身 neighbour 的时候发现了一个 back edge (是一个从未 explore 的 edge, 但所连的 neighbour 已经加入过 stack) 这个 graph 里就有 cycle.

“如果某个 node 在检查邻居时发现除了自己的 parent, 还有其他的已访问 node, 则说明有 back edge 也就有环”

BFS/DFS 综合



a) Starting from A, give the layers the breadth-first search algorithm finds.
b) Starting from A, give the order in which the depth-first search algorithm visits the vertices.

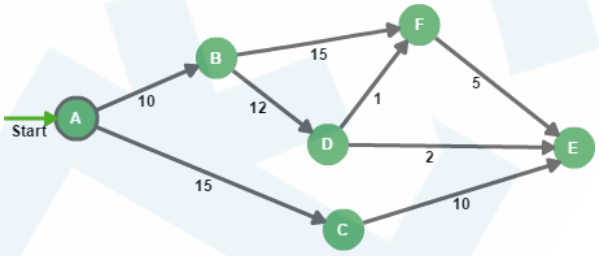
增加问题: 在 DFS 的过程中, 哪些是 back edge, 哪些是 DFS edge

最短路

Dijkstra 求的是单源最短路径, 并且不能适用于 negative edge.
对单源最短路径, 事实上我们只需要知道起点, 终点不重要. 这个方法会计算从源点到所有其他的点的最短距离. 如果某个点在 run 之后还是显示距离为 infinite, 说明没有可能从源点到达.

过程:

1. 把所有除了起点以外的 vertex 标记成 infinite distance
2. 从起点开始, 每一次探索当前最小 distance 的 vertex 向外的边, 并将累积的和记下来. 如果遇到已探索过的部分, 需要比较新的路径是否比已知路径更小并进行更新.



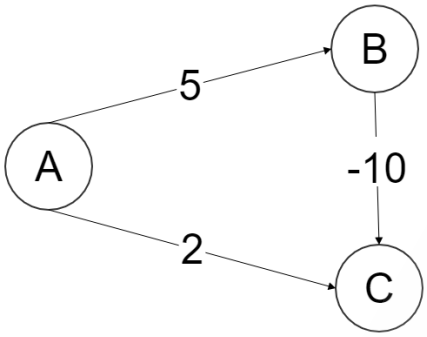
过程:

Priority Queue

PQ	Visited	Removed from PQ
a	-	a

b, c	a	b
c, d, f	a, b	c
d, f, e	a, b, c	d
f, e	a, b, c, d	f
e	a, b, c, d, f	e
Completed	a, b, c, d, f, e	-

为什么 Dijkstra 不能用于 Negative Edge?

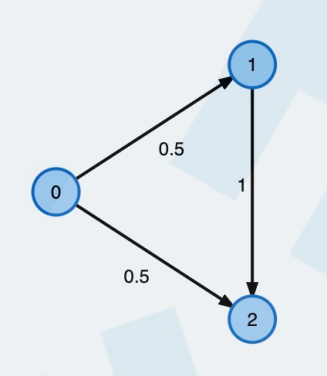


会导致最小值无限被更新下去.

例题

如果已知 graph G 里从 s 到 t 的最短路, 如果我把所有的 edge cost 都加上一个正数形成 G', 原先的最短路还是最短路吗?

不是, 看以下例子 (ignore 箭头 此处假设是无向图)



从 2 到 1 原本有两条最短路 (0.5 + 0.5 vs 1).
如果给每个 edge 都加上 1, 从 2 到 1 就只剩下了确定的一条最短路.

这个技巧也被用在调整最短路算法: 如果想优先步数, 就给 edge 的数量单独加权重. 如果想优先 cost, 就不给 edge 加.

Minimum Spanning Tree (MST)

重要定义 :

Tree: 没有 cycle 的 graph

Spanning Tree: 没有 cycle 而且连接 graph 里所有 vertices 的 tree.
Minimum Spanning Tree: 所有可能的 spanning tree 里 edge weight 之和最小的 spanning tree.
Cut: a cut is a subset of vertex V. V 里随意选的 vertex 集合.
Cut set: the subset of edges with exactly one endpoint in S. 把这些随意选出来的 vertex 视作一个整体之后, "向外的" edge 之和.
Cycle: Set of edges of the form a-b, b-c, c-d, ⋯, y-z, z-a. (需要至少有一个 back edge, 从 z 指向 a)
Cut property: 任意一个 cutset 里, weight 最小的一个 edge, 一定在 MST 里. (一句话概括 Prim 的方法)
Cycle property: 任意一个 cycle 里, weight 最大的一个 edge, 一定不在 MST 里.

一定要知道这些概念名词, 可能 final 考卷里不会解释. 这些概念不仅仅只出现在第一题, 而是后面的题一旦和 graph 有关就会使用这些概念. 所以非常重要

找 MST?

- Kruskal 方法(推荐):
  把所有 edge 按照 weight 从小到大来排列. 开一个空集 A 来存 MST 里所需要的 edge. 从小到大来往 A 里加 edge, 在每次加 edge 的时候检查加入之后会不会产生 cycle, 会就不加.

- Prim 方法:
  选定一个随意的 vertex (考试题目里会给), 从这个 vertex 的 edge 里找一个 weight 最小的 edge 扩展出去, 把这个 edge 另一端的 vertex 加进 V 里来. 当前 V 就变成了两个.从这两个 vertex 所有**向外的(内部的不算!)** edge 里找一个 weight 最小的 edge 扩展出去⋯ 重复这个过程, 直到所有 vertex 都加到集合里为止.

如何证明 Kruskal 和 Prim 的方法是对的?
Invariant: Prior to each iteration, A is a subset of some minimum spanning tree.

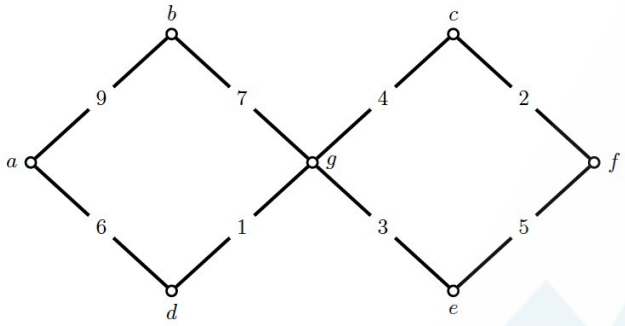
1. Start from empty set. Empty set is certainly a subset of solution.
2. During each iteration, we make sure the edge (u,v) added does not hurting the invariant that A is still a tree.
3. And since we sorted the edge by weight, the edge (u,v) adds smallest weight to the set A, so A's total weight is minimized.
4. Therefore invariant still holds after we add edge (u,v)

例题
如果已知 graph G 里有一个 MST, 如果我把所有的 edge cost 都加上一个正数形成 G', 原先的 MST 还是改过图里的 MST 吗?

是的, 因为以 Kruskal 为例, MST 的选择纯粹看 edge cost 的排列顺序.
给所有 edge cost 加上同一个值以后, edge cost 的排列不变, 所以选出来的 MST 也不变.

Graph 算法综合例题

Final 第一题一定是一道找 MST 的题目, 指定一种算法并要求你描述每一步添加的过程.难度和真实考试一致, 很可信.



如果是 Prim: 会告诉你从哪一个点开始.

如果是 Kruskal: 你自己应该知道从哪个 edge 开始.

Sample Exam:

1. 用 Prim 算法, 从 vertex a 开始找 MST.

过程: 保持一个由 a 开始慢慢延伸出去的 explored set. 一开始只有{a}, 慢慢加入新的 vertex 进来.

每次选择这个 explored set 的 cut edge 里的最小值加入.

{a} 的 cut 有 9 和 6. 选择 6. 把 d 加入 explored set.

{a, d} 的 cut 有 9 和 1, 选择 1. 把 g 加入 explored set.

{a, d, g} 的 cut 有 9, 7, 3, 4. 选择 3. 把 e 加入.

{a, d, g, e} 的 cut 有 9, 7, 4, 5. 选择 4. 把 c 加入.

{a, d, g, e, c} 的 cut 有 9, 7, 5, 2, 选 2, 加入 f.

{a, d, g, e, c, f} 的 cut 有 9, 7. 选 7. 加入 b.

结束. 最终被选定的是{1,2,3,4,6,7}

2. 用 Kruskal 算法:

从最小的 edge 开始加入, 如产生环则不加入.

Edge: {}, Vertex: {}

Edge set	Vertex Set	Note
1	{d, g}	-
1, 2	{d, g}, {c, f}	
1, 2, 3	{d, g, e}, c, f	
1, 2, 3, 4	{D, g, c, f, e}	
1, 2, 3, 4, 6	{D, g, c, f, e, a}	不选择 5 是因为加入 5 会产生 cycle
1, 2, 3, 4, 6, 7	{d, g, c, f, e, a b}	结束

Run Dijkstra's Algorithm on G starting from node a, to calculate the length of the shortest path between a and each other node. Write down the order in which each vertex was extracted from the priority queue.
题目的意思有些绕: Write down the order in which each vertex was extracted from the priority queue. 但他的意思是 “每一次循环里被探索的新的 vertex 是哪个”?

过程:
Priority Queue

PQ	Visited	Removed from PQ
a	-	a
d, b	a	d
g, b	a, d	g
B, e, c	A, d, g	b
E, c	A, d, g, b	e
C, f	A, d, g, b, e	c
f	A, d, g, b, e, c	f
Completed	A, d, g, b, e, c, f	-

Greedy

Greedy 的证明算法:

1. exchange argument.

Transform any solution to the one found by the greedy algorithm without hurting its quality.

2. Structural

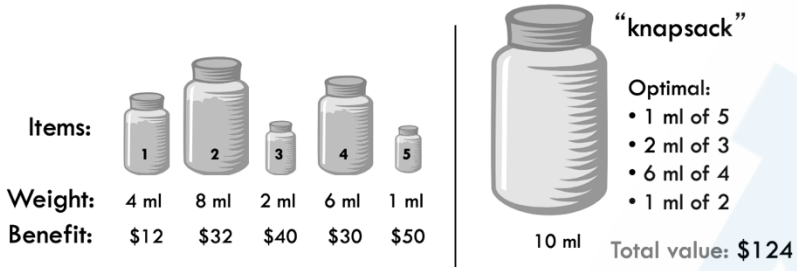
Discover a bound asserting that every possible solution must have a certain value and your algorithm always achieves this bound.

Exchange Argument

常用于 greedy 解需要排序的算法.
它所证明的是排序解的合理性.

给 N 个 item, 每一个都有 benefit b_i和 weight w_i, 最多能选 weight 为 W 的物品. 请问选哪一些能最大化总的 benefit?

Fractional 的意思是: 某样东西可以取一半, 不需要整个.



最优解: 按性价比/单位价值(b_i/w_i)排序, 每次找单位价值最高的物品, 尽量多取.

为了简单化问题, 假设没有两个 task 有相同的 (b_i/w_i) 值
假设有$(b_i/w_i) > (b_j/w_j), J < I$.
假如有一种 optimal 方法, 在某个位置, 优先选择了J, (b_j/w_j) , 然后才选择了 I (b_i/w_i)
如果我们把这个顺序换过来, 我们可以增加总价值.
可以增加多少?

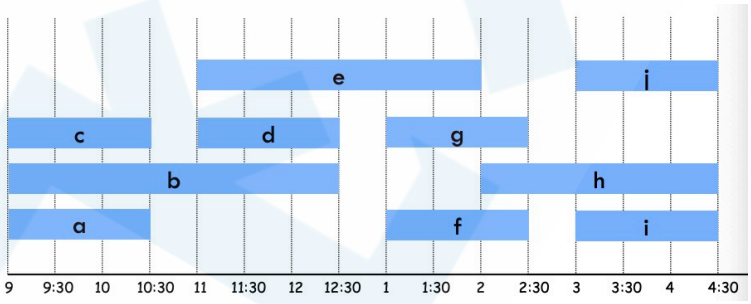
1. 假如 i, j 都可以全选, 调转顺序不变.
2. 如果 i, j 不能全选, 假如我们必须拿走一块 重量为 E 的部分:
   原价值为 bi + bj
   我们可以从 Optimal 中拿走 E*(b_j/w_j), 替换 E*(b_i/w_i)进来.
3. 增加的部分价值为: $E(\frac{b_i}{w_i} - \frac{b_j}{w_j})$, 根据我们的定义, 这部分的值大于 0.

所以, 我们可以调转 J 和 I 的顺序, 来增加总价值.
不断重复这个调转过程, 直到所有顺序严格地遵守b_i/w_i排 s 序, 总价值会越来越高.

Structural

首先, 通过分析问题, 得到一个问题至少/最多需要某些步骤的 upper/lower bound.
然后证明你的算法一定会满足这个 upper/lower bound.
常用于 greedy 解是直接出一个数字的类型.

有 N 个 lecture. 每个 lecture 有开始时间和结束时间. 请问至少需要几个教室可以排下所有的 lecture.



这个问题里的隐含定理是: 一个解至少需要'depth'个教室.
Depth 的定义是某个时间点 overlap 的 task 个数.

Greedy 算法: 先把所有 lecture 按开始时间排序. 如有空余 classroom, 就把这个 lecture 安排上. 如果已经没有空余了, 就开一个新的 room.

这个算法的时间复杂度是$O(NlogN)$, 由 sorting 这步决定.

这个算法的正确性如下:
首先, 显然我们不可能把任何两个时间冲突的课程放在一起.
假如在某个时间, 我们需要 d 个 classroom. 说明我们有一个课程 i 它的时间与至少 d-1 个其他的 room 都冲突了 (其他都满了)
因为我们把所有课程都排过序, 所以其他课程里的开始时间都不晚于S_i.由此可见, 有 d 个课程在S_i的这个时间点有重叠.
所以, 至少需要使用 d 个教室. Greedy 给出的方法精确的给出了 d 个教室, 所以 greedy 的正确性可以保证.

例题

Assignment 5 的最后一题就是典型例子.

maximum degree d 的无向图. 你有 d + 1 种颜色, 编号 0~d.
如何分配这些颜色使任意一条边两个端点的颜色不同?

首先要用 structural 的证明方法, 必须先 prove 一个 lower/upper bound.
这里的 upper bound 是: 一个 maximum degree d 的无向图最多只需要 d+1 种颜色就可以 color 了.
我们 prove 的方式是 induction.
假如一个 graph 里有 N 个 vertex.
第一步:
当 N = 1, degree 最多为 0, d+1= 1
我们确实只需要一种颜色.

第二步: 假设 N=x 的情况
我们假设以下结论成立:
在 N=X, 最大 degree 为 d 的图里, 只需要 d+1 种颜色即可.

第三步: 基于第二步证明
我们给这个图加多一个 vertex, 同样需要保证最大 degree 为 d.
我们可以给这个 vertex 选一个与其邻居都不一样的颜色
因为这个 vertex 最多只有 d 个邻居, 而我们有 d+1 种颜色可以选, 所以一定不会有重复的颜色.
所以 N=X+1 时也成立.

基于前三步, 可以证明对于一个 max degree 为 d 的图, 只要有 d+1 种颜色, 就一定可以涂色.

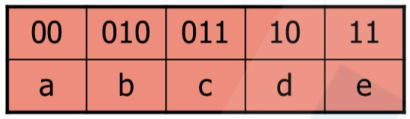
到这一步, upper bound 证明完毕了.
接下来需要描述一种可以满足这个 upper bound 的贪心算法:

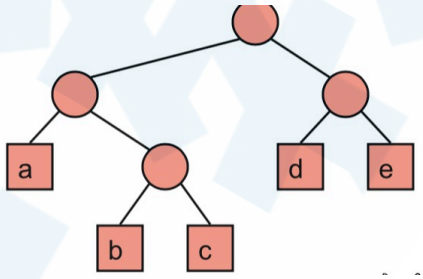
1. 从某个 vertex v 开始, 放置 color 1. O (1)
2. 对于其他所有的 vertex:
    a. 对于目前正在考虑的 vertex v, 检查它的所有 neighbour 的颜色. 排除了这些颜色之后, 选择第一个可以使用的颜色.
    b. 如果发现所有已使用的 color 都不能用, 就用一个新的 color.
3. 当所有 vertex 都被放置完毕后, 这个放置方案就是我们想要的结果.

因为整个 graph 里最多 degree 是 d, 所以一共 vertex 最多有 d 个邻居.
当我们来涂色的时候, 这个 vertex 也最多就排除 d 种颜色
因此, 我们的算法不会产生超过 d+1 种颜色, 满足 upper bound, 所以我们的贪心算法是正确的.

Huffman Tree

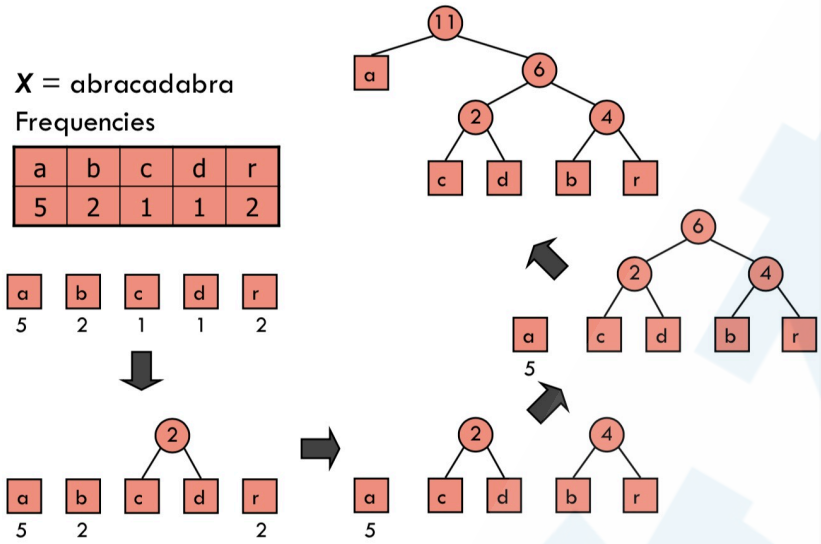
Huffman 树应用于字母到 binary encoding 的编码. (只能是 binary encoding)





从 root 开始, 往左走代表 0, 往右走代表 1.
例如 a, 从 root 到达它需要走 a, a 两步. 代表 0, 0.
我们希望越频繁出现的字母所需要的编码越短, 也就是希望所有字母的 {频率 * 编码长度} 最小化.

方法:
先按照所有字母的频率排序, 存入一个 min heap.
频率越低的两两为一组, 将其发生频率相加成为新的 root node, 并重新加入 min heap.



这样, 出现越频繁的字母会越晚从 min heap 里拿出来进行组合.
熟悉这个过程即可.

Randomised Algorithm

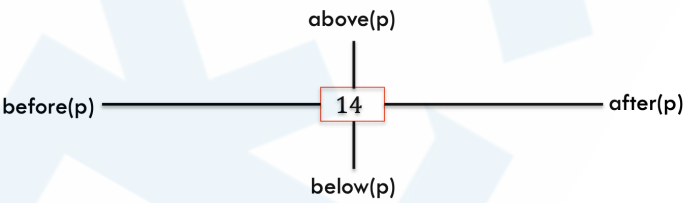
被考的可能性偏低
这类算法是指含有一定随机性, 其时间复杂度不完全取决于 input size 的问题.

Generate Random Permutation

给你一个数字 n, 如何产生一组随机的 {1,2,...n}的组合数?
它需要保证: 每一组组合数产生的概率都是随机的.

```
def FisherYates(A):
    N = len(A)
    for i in range(N):
        j = random.randint(i, N)
        A[i], A[j] = A[j], A[i]
    return A
```

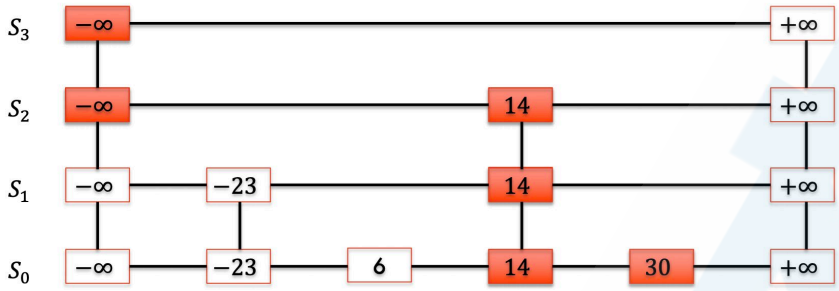
SkipList



每一个 node 有四个 reference: 上下左右

Search 的过程:
当你还不是最后一层且右侧元素 key > k 的时候, 先往右侧搜索. 搜完这一层再往下方搜索.

假如有重复的情况, Search 最终会找到最底下的那个 key.



Insert
从最底下一层开始, 每完成一次加入就抛硬币决定是不是在上一层里也要加入这个值
比如这里 14 就被加入了 3 次, 说明连抛了 2 次硬币都是正面.

Deletion
通过 search 找到最底下的 key, 删除, 然后向上重复删除. (由于复制的 key 一定是在自己上方)

为什么一个内含 N 个 key 的 skiplist 高度为 logN?

- 第 x 层存在的原因是因为某个 key 实现了 x 连正面
- 实现 x 连正面的概率是$\frac{1}{2^x}$.
- 在第 x 层存在至少一个 element 的概率是$\frac{N}{2^x}$
- 要求 x 的高度为 logN 的话, 概率是$\frac{N}{2^{\log_2 N}}$ -> 就是 1. (这里被我简化了, 没有考虑 coefficient)

时间复杂度:
Search/Delete/Insert: Olog(N) 与高度一致.

Divide & Conquer

什么是 divide & conquer

1. Divide : 把一个大问题分成小部分
2. Conquer: 假设小问题都已经解决完了
3. Combine: 把两个已解决的小问题的答案合并并返回, 这个过程中需要做些额外的 computation.

正确性证明

其证明过程结合了 Induction(的风格)和 Invariant
模板:

1. 首先证明 base case 的情况是合理的.
2. Combine 步骤: Assume 在某一个 recursive case, 子 case 返回了两个符合算法要求的正确 input (符合 Invariant)
3. 证明 merge 步骤结束后, 大 case 的 return value 也是合理的. 符合 Invariant.

4. 由此整个算法正确性证明完毕: 无论是 base case 还是 recursive case 都可以给出正确的答案.

举例:
证明 Merge Sort 的正确性

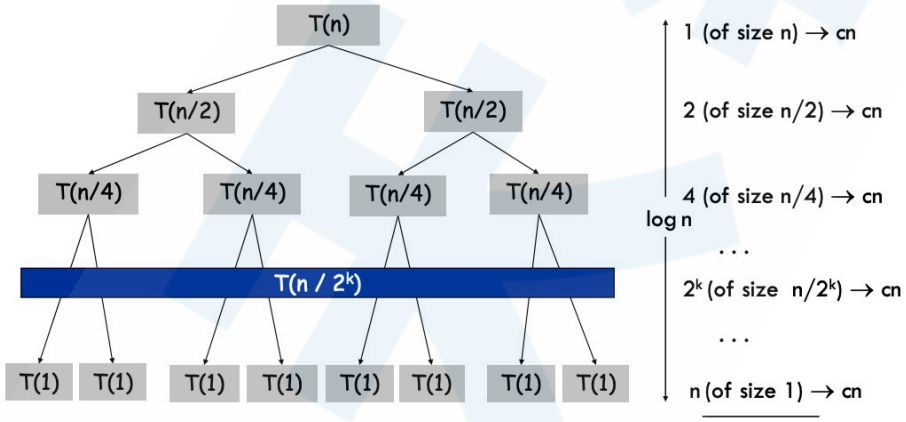
1. Base case 是 array size = 1 或 0. 此时不需要 sort. 它本身即 sorted.
2. Recursive case: 假设 merge 的是两个 sorted sub list .
3. Merge 的每一步都在新 list 里安插两个 sorted sub list 里最小值中(头部位置) 更小的那个值, 产生的新 list 必然 sorted
4. 所以整个算法正确性可以保证.

时间复杂度计算

Unrolling

首先要确定我们的公式: T(N) = aT(N/b)+O(?)
一个大 case 再次 recursion 的时候要被分成几个小 case? 要进行几次 recursion?
被分的份数: b
进行 recursion call 的次数: a.

还是以 Merge sort 举例:



大 case 是一整个 list. 每次一分二.
最底下的 case 是 base case (array 长度为 1).
问最多分几次能分成最底下这样? LogN. 就是这棵树的高度.
每次 Merge 的复杂度是 O(N), 所以总的时间复杂度是 O(NlogN)

Master Theorem

$$T(N) = aT\left(\frac{N}{b}\right) + O(N^d \log^k N)$$

$$\mathrm{T(N)} = \begin{cases} O(N^d)\ if\ d > \log_b a & ----case\ 1 \\ O(N^d \log^{k+1} N)\ if\ d = \log_b a & ----case2 \\ O(N^{\log_b a})\ if\ d < \log_b a & ----case3 \end{cases}$$

将 T(N)表达式转化成 O(N)复杂度的例子:

$\mathrm{T(N)} = 2T\left(\frac{N}{2}\right) + O(N^1)$
$here\ d = 1, a = 2, b = 2.\ 2 = \log_2 1$, 符合 case2 $. O(N^1 logN) = \mathrm{O}\ (NlogN)$

$\mathrm{T(N)} = T\left(\frac{N}{2}\right) + O(NlogN)$
$here\ d = 1,\ k = 1, a = 2, b = 2.\ 2 = \log_2 1$, 符合 case2 $. O(N^1 \log^{1+1} N) = \mathrm{O}\ (N \log^2 N)$

$\mathrm{T(N)} = 4T\left(\frac{N}{2}\right) + O(N)$
$here\ d = 1,\ k = 1, a = 4, b = 2.\ 1 < \log_2 4$, 符合 case1 $. O\left(N^{\log_2 4}\right) = \mathrm{O}\ (N^2)$

$\mathrm{T(N)} = 4T\left(\frac{N}{2}\right) + O(N^3)$
$here\ d = 3,\ k = 1, a = 4, b = 2.\ 3 > \log_2 4$, 符合 case1 $. O\left(N^d\right) = \mathrm{O}\ (N^3)$

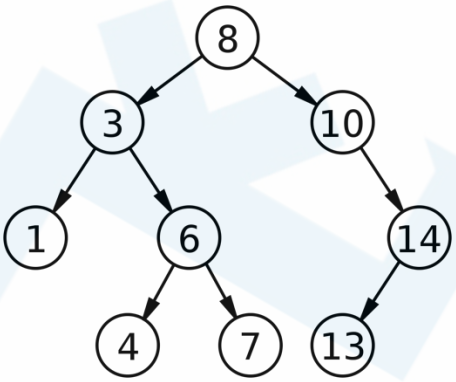
必须要知道的算法

经典算法

Binary Search

算法描述: 对于一个已经排序的数组, 寻找某个数 N. 通过不断缩小包含 N 的搜索范围，最终就可以找到。
一开始范围覆盖整个数组
将数组的中间项与 N 进行比较，可以排除一半元素，范围缩小一半
反复比较，反复缩小范围，最终就会在数组中找到 N, 或 N 不存在.
对于包含 N 个元素的表，整个查找过程大约要经过$log_2 N$次比较.

其过程与 Binary Search Tree 过程一样:



[1, 3, 4, 6, 7, 8, 10, 13, 14]

标准模板

```
def StandardBinarySearch(arr, target):
    L , R = 0, len(arr) - 1
    while L <= R:
        mid = (L + R)//2
        if arr[mid] == target:
            return mid
        if target > arr[mid]:
            L = mid + 1
        else:
            R = mid - 1
    return -1
```

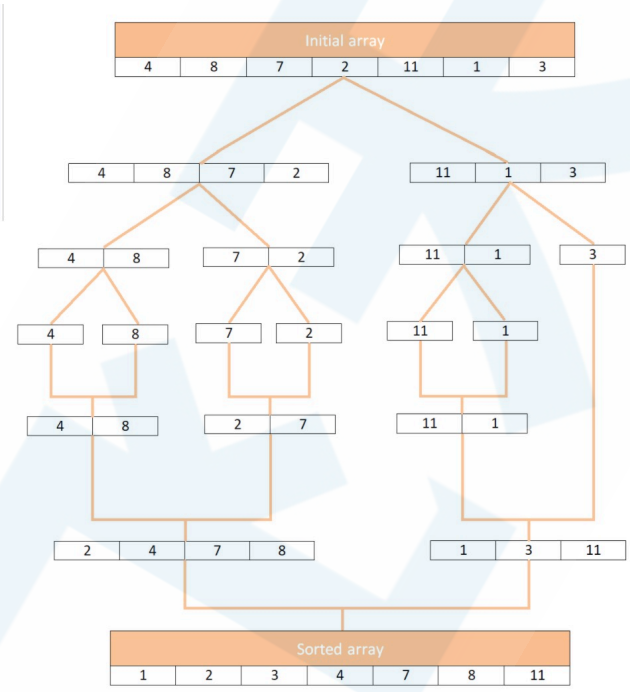
时间复杂度

$T(N) = T\left(\frac{N}{2}\right) + O(1) \rightarrow O(\log N)$

一般而言，当一个题目出现以下特性时，你就应该立即联想到它可能需要使用 Binary Search:
待查找的数组有序
要求时间复杂度低于 O(n)，或者直接要求时间复杂度为 O(log n)

Merge Sort

这个算法的思路非常非常常用, 是很多 Divide and Conquer 问题的基本模板.
里的 merge step 是非常常见的双指针思路.
Merge Sort 里数据的分割过程:



伪代码

A 是一个 array input.

```python
def mergesort(A):
    if len(A) <= 1:
        return A
    mid = len(A)//2
    lefthalf = A[:mid]
    righthalf = A[mid:]
    left = mergesort(lefthalf)
    right = mergesort(righthalf)
    return merge(left, right)

def merge(L, R):
    res = []
    while len(L) > 0 and len(R) > 0:
        if L[0] < R[0]:
            res.append(L.pop(0))
        else:
            res.append(R.pop(0))
    if len(L) > 0:
        res += L
    elif len(R) > 0:
        res += R
    return res
```

时间复杂度:

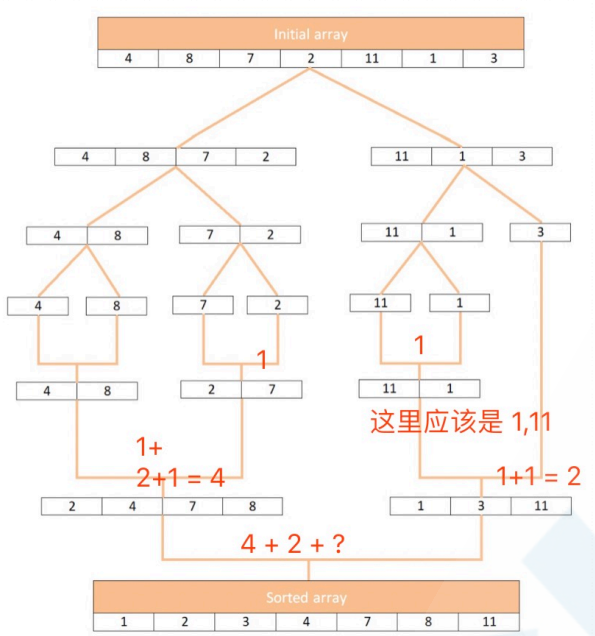
$T(N) = 2T\left(\frac{N}{2}\right) + O(N)$ → O(N log N)

Count Inversion

在一个 array 里有多少对 inversion – 一对 inversion 指 A > B 但是 index A < index B.

这里有

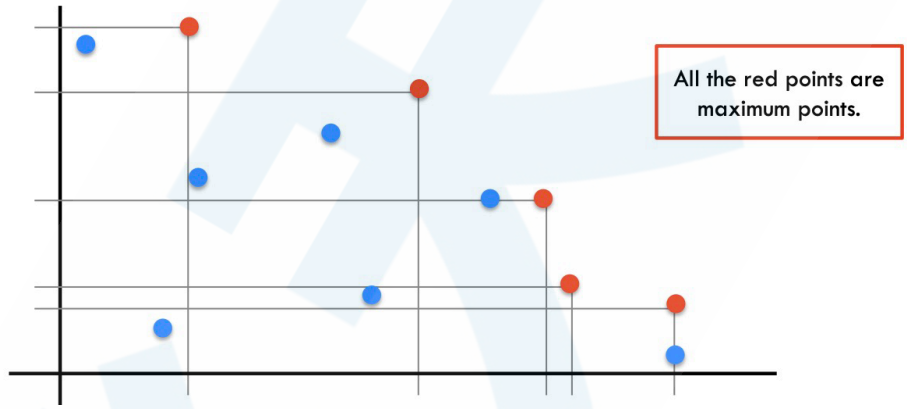
[4,2][4,1][4,3]
[8,7][8,2][8,1],[8,3]
[7,2][7,1][7,3]
[11,1][11,3]
[2,1]



原理同 merge sort.
每次 merge 的时候, 如果右侧 array 先 pop, 则左侧 array 里剩余的部分都是 inversion count.
每次 merge 除了计算自己的 inversion 之外还需要加上来自 subproblem 的 inversion.

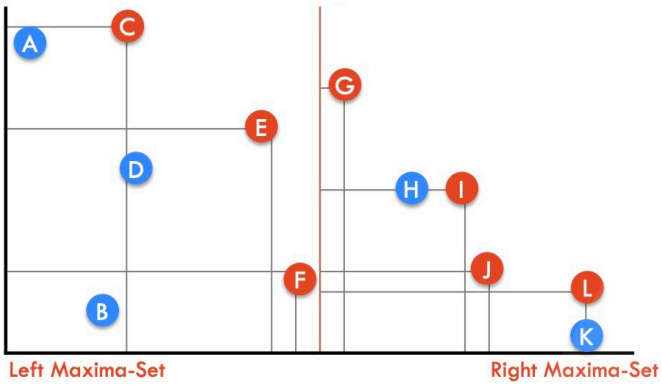
Maxima Set

Maxima point: 某个点的 y 值比所有它左侧的点都大.
求这些点的集合.



观察结论: maxima set 可以 “遮盖” 它左边的值.
又因为按 x 值对半分, 所以可以保证右边的 maxima set 点可能遮盖左边的 maxima set 点, 而反过来则无可能.
如果已知两个 maxima set, 可以通过对比 y 值来消除其中的一部分, 而不需要关注其他不在 maxima set 中的点.

Preprocessing: 先按 y 值排序.
Divide: 按 x 值对半分割进行左右两边的 recursive call



Base case: 如果只有一个点, 它自己就是 Maxima set

Merge: 如果 input 是两个 maxima set, 按 x 轴的分布称作 L 与 R. 合并后的 MS 集为 MergedMS.

1. 所有的 R 都可以加入 MergedMS.
2. 找到 R 中最高点的 y 值. 如果 L 中有任何点比这个 y 值小, 就不能加入 MergedMS, 否则可以加.
3. 返回 MergedMS.

时间复杂度:

$T(N) = aT\left(\frac{N}{b}\right) + O(?)$

每按 x 值对半分进行左右两边的 recursive call: a = 2, b = 2

Merge 步骤: 显然最多进行 O(N)次比较. (R 部分只取 y 最大的点来与所有 L 里的点比较)

$T(N) = 2T\left(\frac{N}{2}\right) + O(N)$ → O(N log N)

其他例题

Majority Element

有一个长度为 N 的 array, 已知某个数字占了过半数. 求在 O(NlogN) 时间里找到这个数字.

例子: [2, 2, 1, 1, 1, 2, 2] , 返回 2.

a) Describe your algorithm

将 array 一分为二, 对左边和右边的 subarray 分别求 majority element. 如果返回的数字一样, 就返回这个数. 否则分别数一下待定的 majority element candidate 出现的次数, 哪个大就返回哪个.

b) Justify the correctness of your algorithm

原理: 一个 majority element 一定会在一分为二的 subarray 里被返回一次, 因为它的数量在 1/2 以上.

举个例子: [1,1,1,2,2,2,2] 算是一个比较极端的例子了. 中间切一刀之后, 左边的 majority element 是 1, 右边的 majority element 是 2. 由于它占比超过 1/2 的缘故, 它必然至少是左右 subarray 里的一个的 majority element. 有的时候,

当数字分布比较均匀的时候, 它还会是左右 subarray 共同的 majority element, 比如 [2,1,2,1,2,1,2].

中间切一刀之后, 左右两边都有理由返回 2 作为自己的 majority element.

当左右返回同样的数字的时候, 可以直接返回, 因为这证明了这个数字在左右 subarray 的出现次数都大于 N/2*(1/2), 也就是这个数字的总数大于了 N/2.

如果返回不同样数字的时候, 就用数数的方式找这个大于 N/2 的数字. 如果发现这个数字并不成立, 则返回没有 majority element.

c) Analyse time complexity

显然是 O(N log N), 来自$2T\left(\frac{N}{2}\right) + O(N)$.

Search in a 2D matrix

例子: 在这个样的一个 matrix 里用 logX 的时间找到某个数是否存在. N 是方形边长. 已知条件是这个长方形里所有 row 和 column 都是分别 sort 好的. 比如观察第一行从上到下, 是严格递增的; 第二列从上到下, 也是严格递增的.

[1, 4, 7, 11, 15],
[2, 5, 8, 12, 19],
[3, 6, 9, 16, 22],
[10, 13, 14, 17, 24],
[18, 21, 23, 26, 30]

思路:

我们可以把这里的一个 2D matrix 分成四个 sort 好的 2D matrix 并排除两个.

比如我们要找 10, 第一次观察正中间那列: [7,8,9, 14,23]. 如果我们发现了 10, 就直接返回 YES. 我们发现 10 在 9 和 14 中间. 以此我们可以确定, 以 9 为右下角和以 14 为左上角的那两个方形, 都可以不用看了, 因为不可能在内部. 所以每次子问题的大小

复杂度: O(N log N). (N 是 matrix 边长, 总元素数量是N^2)

来自 T(x) = $2T\left(\frac{x}{4}\right) + O(x^{0.5})$, where x = N^2

每次子问题的大小为 2 次*1/4 个方块. 额外的搜索次数是 N. 也就是$x^{0.5}$. 为什么这里用了两个符号 x 和 N 是为了方便应用 master theorem.