

Complexity of set operations

Initialize: $O(1)$ 如果 unsorted; $O(n \log n)$ if sorted (需要排序);
 clear: $O(n)$
 isMember: $O(\log n)$, 需要 binary search
 copy: $O(n)$, 即和空集 union
 union, intersect: $O(n)$

Ternary Operator

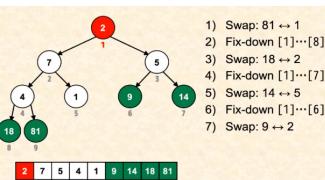
```
c[k] = (a[i] <= b[j]) ? a[i++]: b[j++];
```

等价于

```
if (a[i] <= b[j]) {
    c[k] = a[i];
    ++i;
} else {
    c[k] = b[j];
    ++j;
}
```

Heapsort 一个 vector $a[n]$

- 对 $a[1 \text{ to } n]$ 进行 heapify
- 把 top element 和 tail element 互换
- 把换来的 tail element 在 $a[1 \text{ to } n-1]$ 范围内进行 fixdown
- $a[1 \text{ to } n-1]$ 现在有 heap structure. 我们在 $a[1 \text{ to } n-1]$ 上重复这个过程.



假如 node 的值被修改得更低了, 那么 fix down: 和 descendants 比较, 交换(left child)到两个 children 都比他小为止.

top to bottom 进行 fix up: 要遍历完所有元素, 每个元素 swap 的次数最多是所在层的数目, 越往下越多, 因而是 $O(n \log n)$

bottom to top 进行 fix down: 最后一层不用 fix, 从导数第二层开始, 每一层 k 对于每个上层节点, 左右两个 n_{1k}, n_{2k} 只需要 fix 一个就可以; 并且每个元素 swap 的次数从导数第二层的 1 开始, 往上一层就+1 (同时元素也更少), 复杂度的 sorted array:

binary heap:

- $O(n)$ insertion (排序)
- $O(1)$ inspect top
- $O(1)$ pop (移走end)
- $O(n\log n)$ create
- $O(n)$ create
- $O(n)$ push
- $O(n)$ pop
- $O(1)$ top

sort	time	memory	stable
bubble	n^2	1	yes
selection	$(n^2-n)/2$	1	no
insertion	n^2	1	yes
count	n	$n+k$	yes
merge	$n \log n$	n	yes
heap	$n \log n$	1	no
quick	$n \log n$	$\log n$	no

```
int lower_bound(double a[], double val, int left, int right) {
    while (right > left) { //comp1
        int mid = left + (right - left) / 2;

        if (a[mid] < val) //comp2
            left = mid + 1;
        else
            right = mid;
    }

    return left
}
```

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), f(n) \in \Theta(n^c)$$

直观上 a 越大, b 越小, f(n) 越大则 T 越大. 可以得到:

如果 $a > b^c$, 那么 a, b 的组合 dominate,
 $T(n) \in \Theta(n^{\log_b a})$

如果 $a = b^c$, 那么 a, b 的组合和 f 同时 take dominance,
 $T(n) \in \Theta(n^c \log n)$

如果 $a < b^c$, 那么 f take dominance, $T(n) \in \Theta(n^c)$

MST

Cut Property: 任意切分一个图, 在 cross 两个顶点集的所有边上, 如果其中一条严格小于其他所有边, 那么这条边一定在 MST 中;

corollary: shortest edge for one vertex must be in all MST; in some if unstrict: 如果其中的一条边比 cycle 中的其他边都严格长, 那么它一定不在任何 MST 中

Prim:

记录三个 vector, 每个都 of $|V|$ size

- visited vector: 每个 node v 是否被 visited
- minimal edge weight vector: 每个 node v 的 minimal edge weight
- parent vector: 每个 node v 的 parent

loop $|V|$ 次, 每次选择 intie set 和 outie set 边缘上最短的一个 edge, 把它连接的 outie node 加入 intie set 中, 这样就无 cycle 地添加了 $|V-1|$ 条 edges, 根据连通图的性质, 最后一定会得到一个 spanning tree!

每一次我们把一个 node 加入 intie set, 我们就更新它所有 neighbors 的边的长度, 放入 minimal weight edges vector

既然每次都要找到 outies 的 minimal edges 里面的最小值, 不如起一个 PQ, 每次更新 minimal edges, 我们都把更新好的 <node, min_edge_val> 放进 PQ. 每次从 PQ 中弹出 top 元素, 检查它是不是 outie 元素, 是的话就正常操作, 不是就忽略. |

- 不使用 heap 的朴素实现

loop: $|V|$; 每个 loop 一层 outie loop 来选择 minimal edge: $|V|$; 更新 neighbors 的 min edges: $O(1 + |E|/|V|)$
 因而是 $O(|V|^2 + |E|)$

- 使用 heap 的实现:

While pq nonempty: loop 是 $O(|E|)$ 的

PQ.Getmin: $O(\log |E|)$

在 loop 内遍历更新 neighbors 的 min edges: $O(1 + |E|/|V|)$
 因而是 $O(|E|\log|E|)$

在 graph 比较 sparse 的情况下, 使用 heap 更快

Minimum-cost edge property: 如果 graph 中 minimum cost 的 edge 是 Unique 的, 那么它一定在任何 MST 中; Corollary: **sorting property:** 对一个 connected graph 的所有边长进行排序, 前 k 个不形成 cycle 的 edges 一定是某个 MST 的 subgraph.

Kruskal's algorithm: 我们 sort edges, 然后 loop through all edges, 跳过形成 cycle 的.

当我们想添加一条边的时候, 它会形成一个 cycle 当且仅当它的两个顶点已经 connected, 所以用一个 union-find set 来 keep track of connectivity, 每当放一个新的 edge 等待判断的时候, 我们首先查看它们是否在同一个集合, 进入 MST 的时候, 我们就把它们的顶点 union. (设置其中一个的 parent 为另一个)

排序: $O(E\log E)$; 查找和合并: $O(E \cdot \alpha(V))$ 约等于 E, 合计 $O(E\log E)$

Backtracking:

```
Algorithm checknode(node v)
    if (promising(v))
        if (isSol(v))
            done
        else
            for each node u adjacent to v
                checknode(u)
```

BnB:

```
Algorithm checknode(Node v, Best currBest)
    Node u
    if (promising(v, currBest))
        if (solution(v))
            update(currBest)
        else
            for each child u of v
                checknode(u, currBest)
    return currBest
```

通常的 recursive algorithm 是把整个问题 recursively 划分为 independent 的子问题

而 DP 则用来处理可以分成 subproblems, 但它们之间却不是 independent 的问题

DP

Knight Move:

问题: 从某个格子 ($startX, startY$) 出发, 走 exactly K 步, 有多少种走法可以到另一个格子 ($destX, destY$)? 3D table, 其中第一个维度表示第几步, 第二第三维度是整个棋盘

N 步的棋盘: 遍历第 N-1 步的棋盘, 所有 > 0 (说明第 N-1 步可能到达这个地方) 的格子的 possible moves.

$$dp[k][x][y] = \sum_{(nx, ny) \text{ is有效位置}} dp[k-1][nx][ny] \quad (2)$$

第 N-1 步的棋盘上的一格上面的数字多大, say it is M, 就表示 前 N-1 步有 M 种方法在第 N-1 步时到达这个格子, 于是第 N-1 步到第 N 步, 这个格子上 8 个方向上的 moves 都有 M 重. (直观) 也就是 for all 8 directions, $dp[k][nx][ny] += dp[k-1][x][y]$; 超过棋盘边界不算算。

```
// 0-1 Knapsack Function
int knapsack(const vector<int>& weights, const
vector<int>& values, int W) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1,
0));

    for (int i = 1; i <= n; ++i) {
        for (int w = 0; w <= W; ++w) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(dp[i - 1][w], dp[i - 1]
[w - weights[i - 1]] + values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][W];
}
```

Hash

dict: 1. sorted vector: insert O(n); 2. unsorted vectors: search O(n); 3. linked list: Search O(n); 4. BST (比如 std::map 使用 red-black tree): search 平均 O(logn), worst O(n); insert 平均 O(logn), worst O(n); 5. hash table (比如 std::unordered_map<>使用 hash table): search 平均 O(1), worst O(n); insert 平均 O(1), worst O(n)

Hash table: 把 key 以某个函数转化成一个 Int, 使用 arithmetic operations to calculate a table index from a given key

要做: 1. translation: 把一个 key 翻译为一个 int; 2. compression: limit an int to a valid index; 3. collision resolution: 解决 hash to same table index 的 search keys 的冲突; 前两步合称一个 hash function

Tree Def

1. 无 cycle 的 connected graph
2. 任意两个 node 之间的 paths 都存在唯一的 shortest one.

Properties:

1. 如果一个 tree 有 n 个顶点, 那么它一定有 n-1 个 edges.

$$|E_{tree}| = |V_{tree}| - 1$$

这个性质是源于: tree 是一个连通图。

一个有 $|V|-1$ 条边的连通图一定是 tree. 其实这三条性质: 边数n-1, 连通, 无环中只要两条就可以推导出另一条并表明这是一个树

2. 只要往 tree 中加入任意一条新的边, 就会形成 cycle

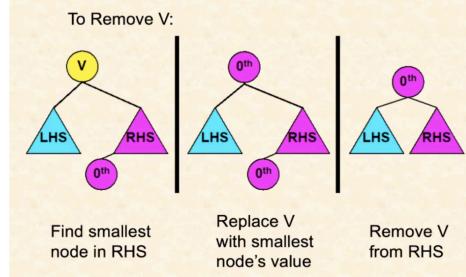
3. 只要在 tree 中去掉任意一条边, 就会 disconnect

4. 每个 tree 都是二分图. (图是二分图当且仅当不包含奇数长度的环

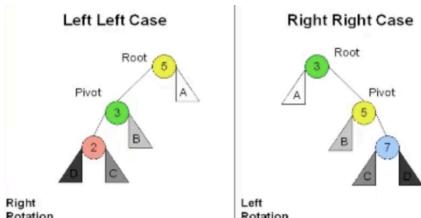
BST: node 比 left subtree 中的所有 nodes 要大, right subtree 中的所有 nodes 要 \leq (可 duplicate)

Search, insert, remove: O(logn) average. worst case: 退化成链表, O(n)

Remove:



average O(logn), worst O(n).



List: 假设 edges 是随机分布的, 每个 vertex 的 vertex list 长度是 $O(E/V)$, space 是: $O(1+|E|/|V|)$ for each vertex, $O(|V|+|E|)$ 总共

找到一个 edge 的 time: $O(|E|/|V|)$

查看两点间是否有边: Matrix: O(1); List: worst O(|V|), best O(1), average O(1 + |E|/|V|)

查找离一个点最近的一个点: Matrix: O(|V|) 一行; List: worst O(|V|), best O(1), average O(1 + |E|/|V|), 遍历一个 list

translate floating pts

key in [0,1)

$$h(key) = \lfloor key * M \rfloor$$

key in [s,t)

$$h(key) = \lfloor \frac{key - s}{t - s} * M \rfloor$$

example: range = [1.38, 6.75], M=13

$$h(3.65) = \lfloor \frac{3.65 - 1.38}{6.75 - 1.38} * 13 \rfloor = 5$$

T	84
TO	$10 * 84 + 79$
TOM	$10 * (10 * 84 + 79) + 77$
TOMI	$10 * (10 * (10 * 84 + 79) + 77) + 32$,

Binary Tree Array: Root 在 index 1, 0dummy, left child of node i 在 index $2i$, right child of node i 在 index $2i + 1$

1. Insert: 平均 O(1), grow O(n)
2. Remove: 稍微有些麻烦, 不能直接 remove 一个节点不管它的子树. 根据需求, 要么 fix down 要么删除整个子树, 但是都要 O(n)
3. Parent, child: O(1), 直接 $n/2, 2n, 2n + 1$

- (1) Space: best case, complete, $O(n)$; worst case: sparse 每层只有一个, $O(2^n)$

Transform a tree into Binary tree

对于 v 的所有 children $\{v_1, v_2, \dots, v_k\}$, 我们把 v_1 变为 new tree v 的 left child, 而 $\{v_2, \dots, v_k\}$ 变为 a chain of right children of v_1 . Recursively do this to v_2, \dots, v_k 的 children .

preorder DFS: visit node, visit left subtree, visit right subtree

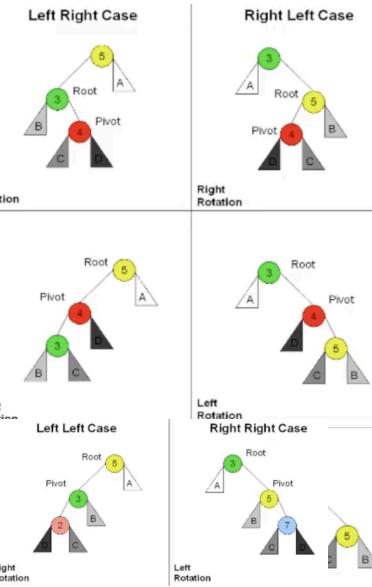
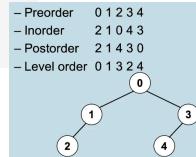
```
void preorder(Node *p) {
    if (!p) return;
    visit(p->key);
    preorder(p->left);
    preorder(p->right);
}
```

inorder DFS: visit left subtree, visit node, visit right subtree

postorder DFS: visit left subtree, visit right subtree, visit node

level order BFS:

```
void levelorder(Node *p) {
    queue<Node *> q;
    while (!q.empty()) {
        Node *n = q.front(); q.pop();
        if (n->left) q.push(n->left);
        if (n->right) q.push(n->right);
    }
}
```



一旦检测到 balance 绝对值为 2, 我们找到第一个 unbalance 的节点 V (一定在 Path from new node to root 上): 它的 left, right subtree 不平衡. 由于之前所有的 node 的 balance factor 绝对值都 ≤ 1 , 我们知道这两个 subtree 中长的一边的左右 subtree (V 的 subtree) 的左右一定从等高变成了不等高, 否则之前就不会平衡.

所以, 这里一定出现了三个高度: 1. V 的短子树, 最短; 2. V 的长子树的一边子树, 其中间; 3. V 的长子树的另一边子树, 最长

```
Algorithm GraphDFS
    Mark source as visited
    Push source to Stack
    While Stack is not empty
        Get/Pop candidate from top of Stack
        For each child of candidate
            If child is unvisited
                Mark child visited
                Push child to top of Stack
            If child is goal
                Return success
        Return failure
```

```
Algorithm GraphBFS
    Mark source as visited
    Push source to back of Queue
    While Queue is not empty
        Get/Pop candidate from front of Queue
        For each child of candidate
            If child is unvisited
                Mark child visited
                Push child to back of Queue
            If child is goal
                Return success
        Return failure
```

```
Dijkstra(G, s)
for all u ∈ V \ {s}, d(u) = ∞
d(s)=0, R = {}
while R != V
    pick u not in R with smallest d(u)
    R = R ∪ {u}
    for all vertices v adjacent to u
        if d(v) > d(u) + l(u, v) {d(v) = d(u) + l(u, v)}
```

$O(|V|^2 + |E|)$, $|E|$ 是更新节点距离总耗, $|V|^2$ 遍历所有 lists