算法设计与分析 期末

~~算法的抽象分析~~

证明正确性

肯定用**数学归纳法**

性能指标

最坏情况时间复杂度

平均情况时间复杂度：$A(n)=\sum_{I\in D_n} Pr(I)\cdot f(I)$，$I$是输入，$f(I)$是该输入的具体时间复杂度

算法中的数学

渐进增长率

$$f(n)=O(g(n)) \text{ iff } \lim_{n\to\infty}\frac{f(n)}{g(n)}=c<\infty, c>0$$

$$f(n)=o(g(n)) \text{ iff } \lim_{n\to\infty}\frac{f(n)}{g(n)}=0$$

两个都说明了f(n)增长率不如g(n)，但$o(g(n))$强调f与g间存在实质性的差距（更不如g了）

$$f(n)=\Theta(g(n)) \text{ iff } \lim_{n\to\infty}\frac{f(n)}{g(n)}=c, 0<c<\infty$$

$$f(n)=\Theta(g(n)) \text{ iff } f(n)=O(g(n)) \wedge f(n)=\Omega(g(n))$$

说明f和g同级

$$f(n)=\Omega(g(n)) \text{ iff } \lim_{n\to\infty}\frac{f(n)}{g(n)}=c>0, c\text{可以为}\infty$$

$$f(n)=\omega(g(n)) \text{ iff } \lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$$

两个都说明了f(n)增长率优于g(n)，但$\omega(g(n))$强调f与g间存在实质性的差距（更胜于g了）

求解分治与递归

替换法

（期中考过）

利用数学归纳法，归纳证明$T(n)$的复杂度也小于等于$cO(n)$，c是某常数。

例如，求$T(n) = 2T(n/2) + n$，猜测为$O(n \log n)$，步骤如下：

1. 假设对于小于n的参数都成立
2. 证明n也成立，如：

$$\begin{aligned} T(n) =& 2T(n/2) + n \\ \le& 2c\frac{n}{2}\log\frac{n}{2} + n \\ =& cn\log\frac{n}{2} - c'n + n \\ \le& cn\log n (c \ge 1) \end{aligned}$$

递归树

用不上

如果一定要用就点了吧

主定理

有式子$T(n) = aT(\frac{n}{b}) + f(n)$，其中$a >= 1, b > 1$

根据$n^{\log_b a}$与$f(n)$的大小关系分以下3种情况：

1. $f(n) = \Omega(n^{\log_b a - \epsilon})$，其中$\epsilon > 0$

表明$f(n)$的影响力不如递归

$T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a} \log^\epsilon n)$，其中$\epsilon \ge 0$

注意n的次方没有减去ϵ，表明同级（但不完全同）。ϵ可以为0表明log项可以没有

$T(n) = \Theta(n^{\log_b a} \log^{\epsilon+1} n)$

3. $f(n) = \Omega(n^{\log_b a + \epsilon})$，其中$\epsilon > 0$，并且存在$c < 1, n \to \infty, af(\frac{n}{b}) \le cf(n)$

后面那个条件不知道什么意思，反正考试也不会考没法用主定理的，所以直接忽视

$f(n)$占支配地位

$T(n) = \Theta(f(n))$

蛮力算法

憨憨查找、选择排序、插入排序

~~狗都会~~

分治 in 排序

快速排序

```
Func Partition(A[], low, high)
    pivot = A[low]
    while low < high:
        while low < high && A[high] >= pivot:
            high--
        A[low] = A[high]
        while low < high && A[low] <= pivot:
            low++
        A[high] = A[low]
    A[low] = pivot
    return low

Func QuickSort(A[], low, high)
    if low < high:
        pivot = Partition(A, low, high)
        QuickSort(A, low, pivot - 1)
        QuickSort(A, pivot + 1, high)
```

归并排序

$$W(n) = 2W(\frac{n}{2}) + O(n)$$

决策树

引入决策树，说明算法结果需要一步一步走到叶节点，从而证明，比较排序的最坏情况时间复杂度的下界：

$$\Omega(n \log n)$$

逆序对及计数

计算逆序对数，可以在归并排序中顺便完成

```
long long merge_count(long long array[], long long start, long long end)
{
    if (start == end)
        return 0;
    long long mid = (start + end) / 2;
    long long lcount = merge_count(array, start, mid);
    long long rcount = merge_count(array, mid + 1, end);
    long long p1 = mid, p2 = end;
    long long* copyarray = new long long[end - start + 1];
    long long copy_index = end - start;
    long long count = 0;
    while (p1 >= start and p2 >= mid + 1)
        if (array[p1] > array[p2])
            count += p2 - mid, copyarray[copy_index--] = array[p1--];
        else
            copyarray[copy_index--] = array[p2--];
    while (p1 >= start)
        copyarray[copy_index--] = array[p1--];
    while(p2 >= mid+1)
        copyarray[copy_index--] = array[p2--];
    for (long long i = 0; i < end - start + 1; i++)
        array[start + i] = copyarray[i];
    return lcount + rcount + count;
```

```
}
```

分治例子

整数乘法

长度都为n的xy相乘，直接计算复杂度为$O(n^2)$

$$x = x_1 \cdot 2^{n/2} + x_0, y = y_1 \cdot 2^{n/2} + y_0$$

那么计算变为：

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \\ &= x_1 y_1 \cdot 2^n + [(x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0] + x_0 y_0 \end{aligned}$$

将问题分解为$x_1 y_1$ $(x_1 + x_0)(y_1 + y_0)$ $x_0 y_0$三个子问题

$$W(n) \le 3W(\frac{n}{2}) + O(n)$$

不知道为什么是小于等于

根据主定理

$$W(n) = O(n^{\log_2 3})$$

检测异类

期中考过

列出所有情况组合，每次都只做降低规模至一半的操作

$$W(n) \le \frac{n}{2} + \frac{n}{2^2} + \cdots$$
$$W(n) = O(n)$$

$O(\log)$时间算法

二分查找

有序下二分查找、峰值查找、$\sqrt{N}$计算

~~狗都会~~

红黑树

红黑树性质如下：

- 节点颜色只有红色或黑色
- 根节点必黑，叶节点必黑
- 节点若有子节点，必有2子；或者节点完全无子节点
- 红色节点不能连续出现
- 所有外部节点的黑色深度（到根路径上除根的黑色节点数）相等，称为黑色高度

准红黑树即根节点是红色，但其他性质都满足

不存在ARB_0

对于$h \ge 1$，红黑树RB_h左右子树分别为RB_{h-1}或ARB_h

对于$h \ge 1$，准红黑树ARB_h左右子树都为RB_{h-1}

假设T为一个有n个内部节点的红黑树，则红黑树的普通高度不超过$2\log(n+1)$，基于红黑树的查找代价为$O(\log n)$

$O(n)$线性选择k阶元素

> 想要找到阶为k的元素，最笨的方法是每次$O(n)$找最小（或最大）并取出，找k次即可，用时$O(kn)$

期望情况

期望下，使用快速排序的Partition每次规模减半，可在$O(n)$内找到

```
Func Partition(A[], low, high)
    pivot = A[low]
    while low < high:
        while low < high && A[high] >= pivot:
            high--
        A[low] = A[high]
        while low < high && A[low] <= pivot:
            low++
        A[high] = A[low]
    A[low] = pivot
    return low
```

一顿期望的数学计算后，反正是$O(n)$

最坏情况

考虑最坏情况，每次都精准地选出最小或最大数作为基准，那么每次规模只-1，那么$T(n)=T(n-1)+O(n)$，退化成$O(n^2)$

通过将数据分组，用选出基准**组**的方式来避免过于不平衡。已知，5个一组最好（期中考过，必不可能再考）

算法SELECT-WLINEAR：

1. 所有元素分5组，凑不齐一组的元素暂放（也可按课本的分为一组）
2. 找出每组中位数，共$\lfloor\frac{n}{5}\rfloor$个
3. 对这$\lfloor\frac{n}{5}\rfloor$个中位数递归地使用SELECT-WLINEAR找出其中的中位数$m*$（同时调整好了组的位置）
4. 基于$m*$的大小对所有元素（含第1步凑不齐一组的元素）进行划分，假设有$x-1$个元素小于$m*$
5. 若$k=x$，返回$m*$；若$k<x$，对小于$m*$的元素调用SELECT-WLINEAR找阶为k的元素；若$k>x$，对大于$m*$的元素调用SELECT-WLINEAR找阶为$k-x$的元素

$$W(n)\leq W(\lceil\frac{n}{5}\rceil)+W(\frac{7}{10}n+6)+O(n)$$

第1项是找组中的中位数的中位数

第2项是划分结果。在所有$\lceil\frac{n}{5}\rceil$组中，至少有一半的小组要贡献3个比$m*$大的元素，其中不包括$m*$所在组以及最后凑不满的组，那么至少也淘汰掉$3(\frac{1}{2}\lceil\frac{n}{5}\rceil-2)\geq\frac{3n}{10}-6$个元素，子问题最大也不过$n-(\frac{3n}{10}-6)=\frac{7}{10}n+6$

第3项是杂七杂八的用时

DFS

~~DFS树~~

- 树边Tree edge
- 回边Back edge
- 子嗣边Descendant edge
- 跨边Cross edge

有向图4种边都有

无向图没有Cross edge，证明如下：

> 反设遍历点u时，发现一条边指向v，uv是CE（即u与v无祖先后继关系）
>
> 首先，v不是白色，否则uv是TE；v不是灰色，否则u与v有祖先后继关系，uv是BE；那么v应该是黑色。
>
> 那么，v在u访问前已完成访问，即访问v时u尚未访问，那么vu应为TE（无向图），u与v有祖先后继关系，与假设矛盾

~~活动区间~~

在遍历过程中，一个节点的活动区间定义为从该节点被发现到遍历结束的时间区间：

$$active(v) = [discoverTime, finishTime]$$

与DFS树相关有以下定理：

$$\begin{aligned}
&\text{u是v在DFS树中的后继节点 iff } active(u) \subseteq active(v) \\
&\quad \text{u和v没有祖先后继关系 iff } active(u) \text{ 和 } active(v) \text{ 没有重叠} \\
&\qquad\qquad \text{uv是TE iff } active(v) \subset active(u) \wedge \text{不存在第3个点x，使得 } active(v) \subset active(x) \subset active(u) \\
&\qquad\qquad \text{uv是DE iff } active(v) \subset active(u) \wedge \text{存在第3个点x，使得 } active(v) \subset active(x) \subset active(u) \\
&\qquad\qquad \text{uv是BE iff } active(u) \subset active(v) \\
&\qquad\qquad \text{uv是CE iff } active(v) \text{ 在 } active(u) \text{ 前面}
\end{aligned}$$

拓扑排序

- 无向图无拓扑序
- 有向图如有环，则不存在拓扑序
- 有向无环图必存在拓扑序

```
func Topo(Graph G)
    while G 非空:
        找到当前G中任一入度为0的点x
        x加入拓扑序列
        从G中删除x
```

关键路径

任务之间的依赖关系可以用有向图G表示，任务对应点，边$i \to j$表示a_i依赖于a_j

定义每个任务的最早开始时间earlist start time，记为est_i

每个任务的est_i和时长l_i唯一地确定了任务的最早结束时间earlist finish time，记为$eft_i = est_i + l_i$

- 不依赖任何其他任务的任务$est_i = 0$
- 依赖于若干其他任务的任务，其est_i为依赖任务中eft_j的最大值，$est_i = max\{eft_i | a_i \to a_j\}$
- 任何任务的est_i确定后，$eft_i = est_i + l_i$

关键路径是一组任务$v_0, v_1, \cdots, v_k$，满足：

- v_0没有任何依赖
- 对任意$1 \le i \le k$：$v_i \to v_{i-1}, est_i = eft_{i-1}$
- 任务v_k的eft_k是所有任务的eft的最大值

```
func CRITICAL-PATH(v)
    v染成灰色
    v.est = 极小值
    v的关键依赖设为空
    for v的每一个邻居w :
        if w是白色：
            CRITICAL-PATH(w)
            if w.eft >= v.est:
                v.est = w.eft
                v的关键依赖设为w
        else if w.eft >= v.est:
            v.est = w.eft
            v的关键依赖设为w
    v.eft = v.est + v.l
    v染成黑色
```

有点点类似tarjan

强连通分量 in 有向图

给每个节点一个DFS标号$v.index$，以时间戳表示访问顺序；给每个节点一个追溯值$v.lowlink$，表示从v出发可达的节点的最小的$index$

$$v\text{是强连通分量的根 iff } v.index = v.lowlink$$

```
输入图G=(V,E)
index = 0
S初始化为空栈

for each i in V:
    if i未访问过:
        strongconnect(i)

func strongconnect(v)
    v.index = v.lowlink = index
    index++
    v入栈S

    for each(v,w) in E:
        if w未访问过:
            strongconnect(w)
            v.lowlink = min(v.lowlink, w.lowlink)
        else if w在栈S中:
            v.lowlink = min(v.lowlink, w.index)

    if v.index == v.lowlink:
        栈S栈顶元素不断出栈直到v出栈，本次出栈的元素为一强连通分量
```

需要一个栈来留存访问状态以生成强连通分量

每个节点调用一次`strongconnect`，每条边最多一次访问，判断元素在栈中用标记保存每次用时$O(1)$，总计时间复杂度$O(n+m)$

割点 in 无向图

割去这个点，使得新图不再连通

tarjan即可

若一个节点v有至少一个子节点w使得$w.lowlink \geq v.index$说明w无法回到v的祖先，即v是割点

对于根节点（DFS入口），根节点是割点当且仅当根节点的子树数量大于1

```
输入图G=(V,E)
index = 0

for each i in V:
    if i未访问过:
        标记i为根节点
        strongconnect(i)

func strongconnect(v)
    v.index = v.lowlink = index
    index++

    for each(v,w) in E:
        if w未访问过:
            strongconnect(w)
            if v是根节点:
                v的子树数量++
            else if w.lowlink >= v.index:
                v是割点
        v.lowlink = min(v.lowlink, w.index)

    if v是根节点且v的子树数量超过1:
        v是割点
```

割边 in 无向图

割去此边，新图不再连通

一条边是割边，即要求（DFS树中）孩子节点无法回溯到父亲节点（更无法向上回溯）

$$\text{边uv，u是父亲，v是孩子}$$
$$\text{uv是割边 iff } v.lowlink > u.index$$

此处要求严格大于，即连父亲都回溯不到

```
输入图G=(V,E)
index = 0

for each i in V:
    if i未访问过:
        strongconnect(i)

func strongconnect(v)
    v.index = v.lowlink = index
    index++

    for each(v,w) in E:
        if w为v父亲:
            continue
        if w未访问过:
```

```
            标记v是w的父亲
            strongconnect(w)
            v.lowlink = min(v.lowlink, w.index)
            if w.lowlink > v.index:
                (v,w)是割边
        else:
            v.lowlink = min(v.lowlink, w.index)
```

只有w未访问过（w是v孩子节点）才判断$lowlink > index$

BFS

BFS树

- TE：有
- BE：有向图可以有，无向图没有

证明无向图没有BE：

> 假设(u,v)是BE，v是u的祖先
>
> 若v是白色，与v是u的祖先矛盾；
>
> 若v是灰色，与v是u的祖先矛盾；
>
> 若v是黑色，v应在变黑前将u入队，(u,v)应为TE，与(u,v)是BE矛盾
>
> 综上，无向图没有BE

- DE：不存在

用有向图证明，无向图同理：

> 反设uv是DE，u是v的祖先且不是v的父亲（若u是v的父亲，uv应为TE）
>
> 考虑u刚出队、即将处理邻居的时刻：
>
> 若v是白色，uv是TE，与uv是DE矛盾
>
> 若v是灰色或黑色，v比u先出队，与u是v的祖先矛盾
>
> 综上，不存在DE

- CE：有

有向图中，若有CE记为$x \to y$，则$y.dis \le x.dis + 1$。y最多在x的下一层（要求在同一棵BFS树上）

无向图中，若有CE记为(x, y)，则$y.dis = x.dis \lor y.dis = x.dis + 1$。y和x在同一层或下一层（要求在同一颗BFS树上）

二分图

太简单

寻找k度子图

以为是多厉害算法，就是用个队列把不够k的节点放进去排队tck罢了

```
func k-degree-subgraph(G, k)
    初始化空队列Q
    for each v in V:
        if v.degree < k:
            v入队列Q
    while 队列Q非空:
        v = 队列Q出队首
        for each (v,w) in E:
            w.degree--
            if w.degress < k && w不在队列Q中:
                w入队列Q
        从G中删除v
    G即为k度子图
```

最小生成树

Prim

贪心地构建最小生成树

每次都从不在当前最小生成树的节点中选出一个节点，它有一条边使得它和最小生成树内一点相邻且此边权值最小

```
func Prim
    初始化空图MST
    初始化空优先队列Q
    初始化candidateEdge为空
    s入Q
    while Q非空:
        v = Q取队头
        if v在MST中:
            continue
        candidateEdge[v]加入MST
        for each v->w in E:
            if candidateEdge[w]为空 || candidateEdge[w] > v->w权值:
                candidateEdge[w] = v->w
                w入Q
```

对于每个点，都要进出队列Q；对于每条边，都要被确认一次代价

$$T(n,m) = O(n \cdot C_{EXTRACT-MIN} + n \cdot C_{INSERT} + m \cdot C_{DECREASE-KEY})$$

操作	数组	堆
INSERT	$O(1)$	$O(\log n)$
EXTRACT-MIN	$O(n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$

- 数组实现优先队列：$O(n \cdot n + n \cdot 1 + m \cdot 1) = O(n^2 + m)$
- 堆实现优先队列：$O(n \cdot \log n + n \cdot \log n + m \cdot \log n) = O((n+m)\log n)$

Kruskal

Kruskal先将所有边排序，再逐条尝试是否加入MST（中间过程很可能不连通）

```
Func Kruskal
    对M中的边按权值从小到大排序
    for each (u,v) in M:
        if MST中点数 == n:
            break
        if find(u) != find(v):
            (u,v)加入MST
            union(u,v)
```

将边排序，用时$O(m\log m)$

对每条边，可能都要调用1次find和union，用时$O(n+m\log n)=O(m\log m)$

只要并查集实现得较为高效，总代价由边排序支配：$O(m\log m)=O(m\log n)$

Minimum-weight Cut-crossing Edge

跨越切的最小权值边

> 切：图$G=(V,E)$有两点集V_1V_2满足$V_1\cup V_2=V, V_1\cap V_2=\emptyset$，$V_1V_2$构成一个切

- 如果存在一个切使得某边e成为该切的MCE，则e一定属于某最小生成树

证明：

> 反设，e不属于任何最小生成树
>
> 考虑某个最小生成树T，它必然存在一条跨越切的边，记为e'
>
> 将e加入T成环。因为T是最小生成树，所以e是环上权值最大的边之一，即$e.weight\geq e'.weight$
>
> 又因为e是MCE，所以$e.weight\leq e'.weight$
>
> 所以，$e.weight=e'.weight$
>
> 将e'从T中去除并加入e得到T'，则T'是最小生成树，与e不属于任何最小生成树矛盾

最短路径

Dijkstra 单源非负

类似Prim，每次都从候选点中选择最近的加入最短路径树

无法处理非负边权

```
func Dijkstra(G, s)
    初始化空优先队列Q
    for each i from 1 to n:
        dis[i] = MAX
    dis[s] = 0
    for each (s,i) in E:
        i进Q
        i前驱记为s
    while Q非空:
        x = Q取队首
        for each (x,i) in E:
            if dis[x] + xi.weight < dis[i]:
                dis[i] = dis[x] + xi.weight
                i前驱记为x
                if i不在Q中:
                    i进Q
```

对于每个点，都要进出队列Q；对于每条边，都要被确认一次代价

$$T(n,m)=O(n\cdot C_{EXTRACT-MIN}+n\cdot C_{INSERT}+m\cdot C_{DECREASE-KEY})$$

操作	数组	堆
INSERT	$O(1)$	$O(\log n)$
EXTRACT-MIN	$O(n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$

- 数组实现优先队列：$O(n\cdot n+n\cdot 1+m\cdot 1)=O(n^2+m)$
- 堆实现优先队列：$O(n\cdot\log n+n\cdot\log n+m\cdot\log n)=O((n+m)\log n)$

Floyd-Warshall 多源无负环

算法利用不断地计算传递闭包

$D_{ij}^{(k)}$表示i到j的只利用x进行中转的最短路径$i\to x+x\to j,1\le x\le k$，其中x不超过k

```
Func Floyd-Warshall(G)
    for each i from 1 to n:
        for each j from 1 to n:
            D[i][j] = MAX
    for each i from 1 to n:
        D[i][i] = 0
    for each (i,j) in E:
        D[i][j] = D[j][i] = ij.weight
    for k from 1 to n:
        for i from 1 to n:
            for j from 1 to n:
                if D[i][k] + D[k][j] < D[i][j]:
                    D[i][j] = D[i][k] + D[k][j]
                    Go[i][j] = Go[i][k]//记录路径
    return D
```

`Go[i][j] = x` 表示i到j最短路上i后的下一个节点，递归地访问 `Go[x][j]` 即得下一跳

Bellman-Ford 单源 负环检查

算法无脑进行$n-1$次所有边尝试松弛。每次不断增加已算出最短路径的点的数量。算法在第n次判断是否仍然能松弛，若能则有负环

```
Func Bellman-Ford(G, s)
    for each i from 1 to n:
        dis[x] = MAX
    dis[s] = 0
    for i from 1 to n - 1:
        relax = False
        for each (u,v) in E:
            if dis[u] + uv.weight < dis[v]:
                dis[v] = dis[u] + uv.weight
                v的前驱记为u
                relax = True
        if relax == False:
            break
    for each (u,v) in E:
        if dis[u] + uv.weight < dis[v]:
```

```
            存在负环，退出
    return dis
```

算法最多需要遍历n轮所有m条边，复杂度$O(nm)$

贪心

相容任务调度

记输入的任务集合为$A = a_1, a_2, \cdots, a_n$，每个任务定义为区间$a_i = [s_i, f_i)$，si和fi分别为任务的开始和结束时间。找出A中最大（包含任务个数最多）的相容任务集。

任务之间没有价值区别，只追求数量最大

正解：“最早结束任务”。将所有任务按照结束时间的先后进行排序，然后从前往后依次扫描所有任务。如果一个任务不和已选择的任务冲突，则选择它；否则忽略。

```
func COMPATIBLE-TASKS(A)
    sort A accordint to Fi
    Compatible = empty()
    while A != empty:
        选出a0
        将与a0冲突的任务从A中删除
        a0加入Compatible
    return Compatible
```

正确性证明（贪心代表）

假设COMPATIBLE-TASKS选出的任务列表为$C = i_1, i_2, \cdots, i_k$，任务顺序为被算法选出的顺序即时间顺序

假设存在一个问题的最优算法得到任务集合$O = j_1, j_2, \cdots, j_m$，任务顺序为时间顺序

- 证明：$k = m$，即COMPATIBLE-TASKS总选出和最优解一样大的任务集合（不一定完全一致）

下面归纳证明“令$f(a)$表示任务a的结束时间，有$r \leq k, f(i_r) \leq f(j_r)$”：

当$r = 1$时，贪心算法选择全局结束最早的任务，显然$f(i_1) \leq f(j_1)$，命题成立

假设$r - 1$时，命题成立，$f(i_{r-1}) \leq f(j_{r-1})$，则

显然，i_{r-1}和i_r是相容的，j_{r-1}和j_r是相容的

则i_{r-1}和j_r一定相容，i_{r-1}后j_r可选

又因为贪心算法总是选择最早结束的任务，所以$f(i_r) \leq f(j_r)$

当r时，命题成立。综上，得证

- 证明：COMPATIBLE-TASKS总能选出最大相容任务集合

反设$m > k$，那么O中至少有一个任务在j_k后面，记为j_{k+1}

由上得，$f(i_k) \leq f(j_k)$，所以i_k与j_{k+1}相容

贪心算法未选择j_{k+1}与算法内容矛盾，假设不成立

综上，得证

Huffman Code

```
func HUFFMAN-ENCODING(A)
    初始化优先队列Q空
    for each a_i in A:
        <a_i,f(a_i)>入Q
    while Q内元素个数大于1:
        i = Q取队首
        j = Q取队首
        新建节点a_k,左子树为i，右子树为j
        f(a_k) = f(i) + f(j)
        <a_k,f(a_k)>入Q
```

用时瓶颈在于优先队列Q取最小时：

$O(n^2)$：数组实现

$O(n\log n)$：堆实现

简单数据结构

堆

- 堆结构特性：比完美二叉树在底层少若干节点，且底层左侧连续排列
- 堆偏序特性：根节点的值大于所有子节点的值（大根堆）

堆的修复

取出堆顶后，左右子树仍满足两性质。先恢复堆结构特性，再恢复堆偏序特性：

1. 底层最右侧的元素移至堆顶（堆结构fixed）
2. 从堆节点开始递归地，将父亲节点与两子节点中大者交换，直到叶子节点（堆偏序fixed）

修复次数不超过堆高度为$O(\log n)$，每次代价为$O(1)$，堆修复代价为$O(\log n)$

堆的构建

1. 将元素摆放在堆中
2. 从叶子节点开始，子节点中的最大值若大于父亲节点则与父亲节点交换，并从该子节点位置开始向下修复堆

$$W(n) = 2W(\frac{n}{2}) + 2\log n$$
$$W(n) = O(n^{\log_2 2}) = O(n)$$

堆排序

基于大根堆进行升序排序

堆顶与底层最右侧的叶子交换后，堆大小-1使原堆顶不在堆的处理范围；反复进行直到堆大小为0

```
func HEAP-SORT(A)
    A建堆
    for i from 1 to n:
        swap(A[1], A[n+1-i])
        堆A大小--
        FIX-HEAP(1)
```

优先队列

增加了INSERT插入和EDIT-KEY改权操作

- INSERT：新增叶子节点（从左到右），堆增大，新节点向上上浮同时向下修复
- EDIT-KEY：直接修改权值，向上上浮同时向下修复

并查集

变化的、扩增的等价关系，即**动态等价关系**

- FIND(a_i)：返回a_i所在的**等价类的代表元**
- UNION(a_i,a_j)：将a_i和a_j所在的等价类合并成一个等价类

普通并+普通查

慢

$O(nl)$

加权并+普通查

feature: **WEIGHTED-UNION**

把节点数更少的挂到更多的上，需要在根节点记录根树的大小信息

- 基于WEIGHTED-UNION的并查集，k个节点的树高不超过$\lfloor \log k \rfloor$，证明：

$k = 1$时显然成立

假设对任意$m < k$，m个节点的树高不超过$\lfloor \log m \rfloor$

假设有k个节点的树T是由k1个节点的子树T1和k2个节点的子树T2合并而成的，不妨设$k_1 \geq k_2$

此时T树高$h = \max\{h_1, h_2 + 1\}$

而$h_1 \leq \lfloor \log k_1 \rfloor \leq \lfloor \log k \rfloor$且$k_2 \leq \frac{k}{2}, h_2 + 1 \leq \lfloor \log k_2 \rfloor + 1 \leq \lfloor \log \frac{k}{2} \rfloor + 1 = \lfloor \log k \rfloor$

综上，$h \leq \lfloor \log k \rfloor$

- 采用WEIGHTED-UNION和FIND的并查集，最坏代价为$O(n + l \log n)$

树高不超过$\log n$，那么FIND代价不超过$O(l \log n)$

初始化，需要$O(n)$

WEIGHTED-UNION代价不超$O(n)$

所以n个节点长度为l的并查集程序代价为$O(n + l \log n)$

加权并+路径压缩

在C-FIND找到祖先节点后，立即更新沿途的节点的父亲为该祖先

- 最坏情况时间复杂度为$O((n + l) \log^* n) \approx O(n + l)$

哈希表

哈希表实现了接近下界$O(1)$的准常数时间性能$O(1 + \alpha)$

定义**负载因子**$\alpha = \frac{n}{m}$，它反映了哈希表的拥挤程度

直接寻址表

键值空间U为每个元素分配了空间，查找每个元素用时$O(1)$但空间开销惊人

简单均匀哈希

多用$h(k) = k \mod n$等函数

封闭寻址

又叫链式寻址

在哈希表的每个位置放的是指向一个链表头部的指针

冲突元素会直接插入到链表头部而不是尾部，以实现$O(1)$的插入

一次成功查找，代价为$O(1)$

一次不成功查找，平均代价为$\Theta(1+\alpha)$。不成功查找的比较次数为找到链表尾所用次数。

开放寻址

i都是从0开始

- 线性探测：直接往后一个个挪看有没有位置

$$h(k,i) = (h'(k)+i) \mod m$$

- 二次探测：第i次从$+1^2, -1^2, +2^2, -2^2, +3^2, -3^2, \cdots, +n^2, -n^2$里选第i项加上（不是累加）

$$h(k,i) = (h'(k)+c_1 i + c_2 i^2) \mod m$$

- 双重哈希：如果冲突，两个函数一起算

$$h(k,i) = (h_1(k) + i h_2(k)) \mod m$$

假设$\alpha = \frac{n}{m} < 1$:

- 不成功查找的平均代价不超过$\frac{1}{1-\alpha}$
- 成功查找的平均代价不超过$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

动态规划

矩阵链相乘

枚举low（从大到小）、high（从low+1递增），在每一组(low,high)中枚举中间的k（此过程能用到前面计算好的），在$O(n^3)$内解

```
func MATRIX-MULTI-DP(Dimelist[1...n])
    for low from n-1 to 1:
        for high from low+1 to n:
            if high-low == 1:
                bestcost = 1
                bestlast = -1//记录分割位置
            else
                bestcost = MAX
            for k from low+1 to high-1:
                a = cost[low][k]
                b = cost[k][high]
                c = multCost(Dimelist[low], Dimelist[k], Dimelist[high])
                if a+b+c < bestcost:
                    bestcost = a+b+c
                    bestlast = k
            cost[low][high] = bestcost
```

```
            last[low][high] = bestlast

func EXTRACT(low, high)
    if high - low > 1:
        k = last[low][high]
        EXTRACT(low, k)
        EXTRACE(k, high)
        k进输出队列
```

编辑距离

将一个单词变成另一个单词所需最少“编辑操作”次数：插入、删除、替换。

考虑`dp[i][j]`：

- 插入，直接加上`B[j]`，从`dp[i][j-1] + 1`递推而来
- 删除，直接删掉`A[i]`，从`dp[i-1][j] + 1`递推而来
- 替换，判断是否一致，从`dp[i-1][j-1]`递推而来

```
dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1)
if A[i] == B[j]:
    dp[i][j] = min(dp[i][j], dp[i-1][j-1])
else:
    dp[i][j] = min(dp[i][j], dp[i-1][j-1] + 1)
```

背包问题

01背包

每件物品只能装0件或1件

```
func 01packet(w[1...n], v[1...n], C)
    dp[0...C] = 0
    for i from 1 to n:
        for j from C to w[i]:
            dp[j] = max(dp[j], dp[j-w[i]] + v[i])
    return dp[C]
```

如果使用`dp[i][j]`表示状态，则表示把$1\cdots i$这些物品都考虑后，大小j下所能取得的最大价值

但是实际上我们最后只关心`dp[n][C]`，所以将二维dp压缩至一维，那么**j的循环必须从大到小，以免重复拿某物品（完全背包）**

i的循环顺序无所谓，保证遍历即可

完全背包

每件物品能装0件或**任意数量**件

```
func allpacket(w[1...n], v[1...n], C)
    dp[0...C] = 0
    for i from 1 to n:
        for j from w[i] to C:
            dp[j] = max(dp[j], dp[j-w[i]] + v[i])
    return dp[C]
```

i的循环顺序仍然无所谓

j的循环必须从小到大，这样大容量才可以从（同一物品的）小容量转移而来，即实现重复拿取

多重背包

每件物品能装0件或**指定上限**件

```
func mulpacket(w[1...n], v[1...n], s[1...n], C)
    dp[0...C] = 0
    for i from 1 to n:
        for j from C to w[i]:
            for k from 1 to s[i] && w[i] * k <= j:
                dp[j] = max(dp[j], dp[j - w[i] * k] + v[i] * k)
```

大致和01背包相同，只是01背包中考虑i这单个物品的过程被扩展为判断1个i、2个i、…、s[i]个i这堆物品。所以也有方法直接把物品复制s[i]个直接转变为01背包

仍然要注意j的循环从大到小，本次更新的值不会在同一个i循环里被用上

硬币兑换问题

类似完全背包。最外层循环是最新参与考虑的硬币，次层循环是从小到大的金额枚举

最大和连续子序列

`dp[i]`表示`i`在末尾的连续子序列的和的最大值

全局答案应该是$ans = \max\{dp[1], \cdots, dp[n]\}$

考虑每一个数，它只有2种情况：

- 接上前一位。`dp[i] = dp[i-1] + a[i]`
- 自己新起一串。`dp[i] = a[i]`

得状态转移方程$dp[i] = \max(dp[i-1] + a[i], a[i])$

相容任务调度

整个不懂课本在说什么

反正就是递归吧

平摊分析

C_{actual}实际代价，直接、精准地反映了每次的代价

$C_{accounting}$记账代价，让一些操作产生的额外的代价（正）以抵消另外一些操作的代价（负的代价）

$C_{amortized} = C_{act} + Caccounting$平摊代价

运行花费较低的operations时先存credit未雨绸缪, 供未来花费较高的operations使用

设置每个操作的平摊成本(amortized cost)后，要做valid check确保任何时刻credit不可以是0

具体看书

对手论证

将算法设计者与算法分析者看作对手，同时扮演两个角色进行算法分析。

1. 算法设计者：尽量多的创造更多信息
2. 算法分析者：尽量少的给予信息，拥有着**随时合理改变取值**的能力

具体看书

N NP NP-hard NP-complete

> 优化问题：优化问题关注某种特殊的结构，希望优化该结构的某种指标
>
> 判定问题：不再关注指标的最大/最小值，而是引入参数k，并回答一个“是或否”的问题：本结构的指标与参数k是否满足某种关系

- P问题：如果存在关于n的多项式$\mathrm{poly}(n)$使得存在一个**解决问题**的算法的代价$f(n)=O(\mathrm{poly}(n))$

证明P问题：找到多项式时间的解决问题的算法即可

- NP问题：如果存在关于n的多项式$\mathrm{poly}(n)$使得存在一个**验证问题**的算法的代价$f(n)=O(\mathrm{poly}(n))$，**不要求解决**

证明NP问题：找到多项式时间的验证问题的算法即可

> 归约 reduction
>
> 问题P可以归约到问题Q（P is reducible to Q）的含义是解决问题P可以间接地通过解决问题Q来实现
>
> 判定P到Q的归约为一个函数$T(x)$满足：
>
> - 将P的任一合法输入x转换成Q的任一合法$T(x)$
> - PQ的输出保持一致（一荣俱荣，一损俱损）
>
> 需要证明Q的输出一定就是P的输出
>
> 如果T是多项式的，那么P多项式时间归约到Q，记为$P\leq_P Q$（符号旁边的P表示poly多项式）
>
> 符号也表明，Q难度高于P

- NP-hard问题：比所有的NP问题都难（它自己不需要是NP问题），$\forall Q\in\mathrm{NP},Q\leq_P P$

证明NP-hard问题：课本没教

- NP-complete问题：是NP问题的NP-hard问题

证明NP-complete问题：

0. 欲证问题Q是NP完全问题
1. 找来一个已知的NP完全问题 P， 构造$P\leq_P Q$，由传递性，Q比所有NP问题都难了，Q是NP难问题
2. 证明Q是NP问题
3. 综上，得证