CSP 复赛复习 - 基础算法(1)

3. 基础算法

贪心法

核心思想：每一步都选择当前最优解，希望最终得到全局最优解

适用条件

- 最优子结构性质
- 贪心选择性质

时间复杂度：通常为 $O(n)$ 或 $O(n \log n)$

贪心法示例：部分背包问题

问题：n 个物品，第 i 个物品价值 v_i，重量 w_i，背包容量 W，可以取物品的一部分，求最大价值

贪心策略：按单位重量价值 $\frac{v_i}{w_i}$ 从大到小排序

```cpp
struct Item {
    double v, w; // 价值和重量
    bool operator<(const Item& other) const {
        return v / w > other.v / other.w; // 按单位价值降序
    }
} items[N];

int main() {
    int n;
    double W;
    cin >> n >> W;
    for (int i = 0; i < n; i++) {
        cin >> items[i].v >> items[i].w;
    }

    sort(items, items + n);

    double ans = 0;
    for (int i = 0; i < n && W > 0; i++) {
        if (items[i].w <= W) {
            ans += items[i].v;
            W -= items[i].w;
        } else {
            ans += items[i].v * (W / items[i].w);
            break;
        }
    }

    printf("%.2f\n", ans);
    return 0;
}
```

贪心法示例：排队打水问题

问题：n 个人打水，第 i 个人需要 t_i 时间，求最小总等待时间

贪心策略：按打水时间从小到大排序

总等待时间：$\sum_{i=1}^{n}\left(\sum_{j=1}^{i-1} t_j\right)$

```cpp
int t[N]; // 打水时间数组

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> t[i];
    }

    sort(t, t + n); // 按打水时间排序

    long long total_wait = 0;
    long long current_time = 0;

    for (int i = 0; i < n; i++) {
        total_wait += current_time; // 当前人的等待时间
        current_time += t[i];       // 更新当前时间
    }

    cout << total_wait << endl;
    return 0;
}
```

贪心法示例：线段覆盖问题

问题：在数轴上有 n 条线段 $[l_i, r_i]$，选择最多的不相交线段

贪心策略：按右端点从小到大排序

```cpp
struct Segment {
    int l, r;
    bool operator<(const Segment& other) const {
        return r < other.r; // 按右端点排序
    }
} seg[N];

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> seg[i].l >> seg[i].r;
    }

    sort(seg, seg + n);

    int cnt = 0, last_r = -1e9;
    for (int i = 0; i < n; i++) {
        if (seg[i].l >= last_r) { // 不相交
            cnt++;
            last_r = seg[i].r;
        }
    }

    cout << cnt << endl;
    return 0;
}
```

贪心法示例：区间覆盖问题

问题：选择最少的点，使得每个区间至少包含一个点

贪心策略：按右端点从小到大排序，点覆盖区间时尽量靠后，向后可能覆盖更多区间。

```cpp
struct Interval {
    int l, r;
    bool operator<(const Interval& other) const {
        return r < other.r;
    }
} intervals[N];

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> intervals[i].l >> intervals[i].r;

    sort(intervals, intervals + n);

    int cnt = 0, last = -1e9;
    for (int i = 0; i < n; i++)
        if (intervals[i].l > last) {
            cnt++;
            last = intervals[i].r;
        }

    cout << cnt << endl;
    return 0;
}
```

递推法

核心思想：从已知的初始条件出发，逐步推导出后续结果

特点

- 自底向上的计算方式
- 避免重复计算
- 常用于动态规划的预处理

递推法示例：斐波那契数列

递推公式：$f(n) = f(n-1) + f(n-2)$，**时间复杂度**：预处理 $O(n)$，查询 $O(1)$

边界条件：$f(0) = 0, f(1) = 1$

```
int fib[N]; // 斐波那契数组

void precompute(int n) {
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i <= n; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }
}

// 获取第 n 个斐波那契数
int getFibonacci(int n) {
    return fib[n];
}
```

递推法示例：组合数计算

递推公式：$C(n,k) = C(n-1,k-1) + C(n-1,k)$

边界条件：$C(n,0) = C(n,n) = 1$

```
int C[N][N]; // 组合数数组

void precomputeCombination(int n) {
    for (int i = 0; i <= n; i++) {
        C[i][0] = C[i][i] = 1;
        for (int j = 1; j < i; j++) {
            C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }
}

// 获取组合数 C(n, k)
int getCombination(int n, int k) {
    return C[n][k];
}
```

递归法

核心思想：函数调用自身来解决问题

三要素

1. 递归终止条件
2. 递归调用
3. 返回值处理

时间复杂度分析：通常使用主定理

递归法示例：选数问题

问题：从 n 个数中选出 k 个数，求所有可能的组合

```cpp
int arr[N];     // 原始数组
int chosen[N];  // 当前选择的数
int n, k;

// depth: 当前深度, start: 从哪个位置开始选择
void selectNumbers(int depth, int start) {
    // 递归终止条件：已选够 k 个数
    if (depth == k) {
        for (int i = 0; i < k; i++) {
            cout << chosen[i] << " ";
        }
        cout << endl;
        return;
    }

    // 递归调用：从 start 开始选择
    for (int i = start; i < n; i++) {
        chosen[depth] = arr[i];           // 选择当前数
        selectNumbers(depth + 1, i + 1);  // 递归选择下一个数
    }
}

int main() {
    cin >> n >> k;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    selectNumbers(0, 0);
    return 0;
```

递归法示例：全排列

```
int path[N];      // 当前路径
bool used[N];     // 标记数组

void permutation(int arr[], int n, int depth) {
    // 递归终止条件
    if (depth == n) {
        for (int i = 0; i < n; i++) {
            cout << path[i] << " ";
        }
        cout << endl;
        return;
    }

    // 递归调用
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            used[i] = true;
            path[depth] = arr[i];
            permutation(arr, n, depth + 1);
            used[i] = false; // 回溯
        }
    }
}
```

递归法示例：汉诺塔问题

```
void hanoi(int n, char from, char to, char aux) {
    if (n == 1) {
        cout << "Move disk 1 from " << from << " to " << to << endl;
        return;
    }

    hanoi(n - 1, from, aux, to);
    cout << "Move disk " << n << " from " << from << " to " << to << endl;
    hanoi(n - 1, aux, to, from);
}
```

移动次数：$T(n) = 2T(n-1) + 1 = 2^n - 1$

二分查找

核心思想：在有序序列中通过比较中间元素快速定位目标

前提条件：序列必须有序

时间复杂度：$O(\log n)$

二分查找标准实现

```
// 在有序数组 arr 中查找 target
int binarySearch(int arr[], int n, int target) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid; // 找到目标
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // 未找到
}
```

二分查找变体：查找第一个大于等于目标的值

```
/* 找第一个 ≥target 的位置 */
int binarySearch(int x)
{
    /* 开区间 ( l,r ) */
    int l = 0, r = n + 1, mid;
    while (l + 1 < r)
    {
        mid = l + r >> 1;
        if (a[mid] >= x)
            r = mid;
        else
            l = mid;
    }
    return r;
}
```

二分查找变体：查找第一个大于目标的值

```
// 在有序数组 a 中查找第一个 > target 的元素位置
int binarySearch(int x)
{
    /* 开区间 ( l,r ) */
    int l = 0, r = n + 1, mid;
    while (l + 1 < r)
    {
        mid = l + r >> 1;
        if (a[mid] <= x)
            l = mid;
        else
            r = mid;
    }
    // 注意，此时的 l 为最后一个 <= target 的位置，+1 后即为 > target 的首个位置
    return l + 1;
}
```

lower_bound & upper_bound()

`lower_bound(begin,end,target)`：查找首个大于或等于目标元素的位置。

`upper_bound(begin,end,target)`：查找首个大于目标元素的位置。

样例输入
5
1 2 2 2 3
2
样例输出
2
4

```
int a[N];
cin >> n;
for(int i=1;i<=n;i++)
    cin >> a[i];
cin >> target;
cout << lower_bound(a+1,a+1+n,target)-a << endl;
cout << upper_bound(a+1,a+1+n,target)-a << endl;
```

二分答案

核心思想：对答案进行二分查找，通过验证函数判断可行性

适用场景

- 求最大值的最小值
- 求最小值的最大值
- 答案具有单调性

二分答案模板

```
bool check(int mid) {
    // 验证 mid 是否可行的函数
    // 返回 true 表示可行, false 表示不可行
}

int binarySearchAnswer(int left, int right) {
    int ans = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (check(mid)) {
            ans = mid;       // 记录可行解
            right = mid - 1; // 尝试更小的值
        } else {
            left = mid + 1; // 需要更大的值
        }
    }
    return ans;
}
```

二分答案示例：木材切割问题

问题：将 n 根木材切成至少 k 段，求每段的最大长度

```cpp
int woods[N]; // 木材长度数组
int n, k;

bool check(int length) {
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        cnt += woods[i] / length;
    }
    return cnt >= k;
}

int main() {
    cin >> n >> k;
    int max_len = 0;
    for (int i = 0; i < n; i++) {
        cin >> woods[i];
        if (woods[i] > max_len) max_len = woods[i];
    }

    int left = 1, right = max_len;
    int ans = 0;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (check(mid)) {
            ans = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    cout << ans << endl;
    return 0;
}
```

算法复杂度总结

算法	平均时间复杂度	最坏时间复杂度	空间复杂度
贪心法	$O(n)$	$O(n \log n)$	$O(1)$
递推法	$O(n)$	$O(n)$	$O(n)$
递归法	取决于问题	取决于问题	$O(\text{递归深度})$
二分查找	$O(\log n)$	$O(\log n)$	$O(1)$
二分答案	$O(\log R \times C)$	$O(\log R \times C)$	$O(1)$

其中 R 为答案范围，C 为检查函数复杂度

祝大家 CSP 复赛取得好成绩！