**1. 算法概述**

* **时间复杂度**：  
  时间复杂度描述算法的运行时间与输入规模之间的关系，用大O记法表示（如 O(n)、O(log⁡n)O(\log n)O(logn) 等），是衡量算法效率的主要指标之一。
* **怎样考量时间复杂度的好坏**：
  + 时间复杂度越低，算法运行越快，通常越好。
  + 考虑不同输入情况下的表现（平均、最坏、最好）。
* **平均、最坏、最好时间复杂度**：
  + **最坏情况**：算法在最不利输入时的运行时间，常用 OOO 表示。
  + **最好情况**：算法在最优输入时的运行时间。
  + **平均情况**：所有输入下的运行时间的期望值。
* **O, Ω, Θ 表示法**：
  + O(f(n))：上界，表示算法运行时间不超过<= f(n)。
  + Ω(f(n))：下界，表示算法运行时间至少为>= f(n)。
  + Θ(f(n))：精准界，表示运行时间正好为 f(n)。

**算法的定义及满足的4条性质**

**定义**：算法是为解决特定问题而定义的有限指令序列，它能够在有限时间内获得问题的解。

**算法的4条性质**：

1. **有穷性**：算法必须在有限的步骤后终止。
2. **确定性**：算法的每一步必须有明确的定义，不会产生歧义。
3. **输入**：算法具有零个或多个输入，这些输入决定了问题的初始状态。
4. **输出**：算法有一个或多个输出，表示问题的解。

**2. 分治法**

* **思想与步骤**：
  + **分**：将问题分成多个规模更小的子问题。
  + **治**：递归解决子问题。
  + **合**：将子问题的解组合成原问题的解。
* **基本性质**：
  + 子问题互相独立
  + 原问题可以划分为规模更小的子问题
* **最优子结构**：问题的最优解由其子问题的最优解组成。
* **经典问题**：
  + **二分查找**：在有序数组中查找目标值，复杂度 O(logn)。

int binarySearch(int arr[], int left, int right, int target) {

if (left > right) return -1; // 未找到

int mid = left + (right - left) / 2; // 防止溢出

if (arr[mid] == target) return mid; //找到了

else if (arr[mid] < target) //在右边界

return binarySearch(arr, mid + 1, right, target);

else //左边界

return binarySearch(arr, left, mid - 1, target);

}

* + **归并排序**：分解数组，递归合并，复杂度 O(nlogn)。

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++) L[i] = arr[left + i];

for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) arr[k++] = L[i++];

else arr[k++] = R[j++];

}

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

* + **快速排序**：选定基准值，分区递归，复杂度O(nlogn)。

int partition(int arr[], int low, int high) {

int pivot = arr[low];

int i = low + 1, j = high;

while (i <= j) {

while (i <= j && arr[i] <= pivot) i++;

while (i <= j && arr[j] > pivot) j--;

if (i < j) swap(arr[i], arr[j]);

}

swap(arr[low], arr[j]);

return j;

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

**3. 动态规划（DP）**

* **基本思想**：通过将问题分解为子问题，记录已经解决的子问题的解（避免重复计算），逐步构建出最终问题的解。
* **基本要素**：
  1. **最优子结构**：问题的最优解可以由子问题的最优解构成。
  2. **状态转移方程**：递推关系，用于从子问题构造解。
  3. **重叠子问题**：子问题重复计算，可通过记忆化或表格优化。
* **解题步骤**：
  1. 定义状态。
  2. 建立状态转移方程。
  3. 初始化边界条件。
  4. 通过递推或迭代求解。
* **经典问题**：
  1. **矩阵连乘问题**：找出矩阵连乘的最优顺序。

int matrixChainOrder(int p[], int n) {

int dp[n][n];

for (int i = 1; i < n; i++) dp[i][i] = 0;

for (int l = 2; l < n; l++) { // l 是链长

for (int i = 1; i < n - l + 1; i++) {

int j = i + l - 1;

dp[i][j] = INT\_MAX;

for (int k = i; k < j; k++) {

int cost = dp[i][k] + dp[k+1][j] + p[i-1]\*p[k]\*p[j];

dp[i][j] = min(dp[i][j], cost);

}

}

}

return dp[1][n-1];

}

* 1. **最长公共子序列（LCS）**：找出两个序列的最长公共子序列。

int LCS(string X, string Y) {

int m = X.length(), n = Y.length();

int dp[m+1][n+1];

for (int i = 0; i <= m; i++) {

for (int j = 0; j <= n; j++) {

if (i == 0 || j == 0) dp[i][j] = 0;

else if (X[i-1] == Y[j-1])

dp[i][j] = dp[i-1][j-1] + 1;

else

dp[i][j] = max(dp[i-1][j], dp[i][j-1]);

}

}

return dp[m][n];

}

**4. 贪心算法**

* **基本概念**：通过局部最优选择逐步构造全局最优解。
* **思想**：在每一步都选择当前看来最优的解。
* **贪心与动态规划的区别**：
  + 贪心：每步选择是局部最优，不能回溯。
  + 动态规划：考虑所有可能性，求全局最优，允许回溯。
* **经典问题**：
  + **活动安排问题**：选择最大不冲突活动集。

void activitySelection(vector<pair<int, int>> &activities) {

sort(activities.begin(), activities.end(), [ ](auto &a, auto &b) {

return a.second < b.second; // 按结束时间排序

});

int last\_end = 0;

for (auto &[start, end] : activities) {

if (start >= last\_end) {

cout << "选这个活动: " << start << "-" << end << endl;

last\_end = end;

}

}

}

* + **混乱编码问题**：重新排列字符串的最优方法。

void dfs(vector<int> &path, vector<int> &choices, vector<vector<int>> &result) {

if (path满足条件) {

result.push\_back(path);

return;

}

for (int i = 0; i < choices.size(); i++) {

if (choice合法) {

path.push\_back(choices[i]);

dfs(path, 更新后的choices, result);

path.pop\_back(); // 回溯

}

}

}

**5. 回溯算法（DFS）**

* **基本概念**：通过递归搜索所有可能解，遇到无解的路径时回退。
* **算法框架**：
  1. 从初始节点开始尝试，逐步向前扩展解空间。
  2. 如果当前解非法，则回溯。
  3. 如果找到解则记录或停止搜索。
* **节点分类**：
  1. **扩展节点**：当前需要进一步搜索的节点。
  2. **活节点**：等待扩展的节点。
  3. **死节点**：不可能产生解的节点。

**6. 分支限界法**

**基本概念**：通过分支生成解空间树，通过计算**下界**来剪掉那些不可能产生最优解的分支，减少搜索空间。

* **算法框架**：
  1. 用优先队列或递归生成解空间树。
  2. 每次选择最优节点扩展。
  3. 使用限界函数估计节点是否需要进一步扩展。
* **剪枝策略**：
  1. 剪去无法生成更优解的分支
  2. 减少无效计算，提高效率。

**19. 渐进复杂性的意义**

**定义**

渐进复杂性描述算法在输入规模趋于无穷大时的增长率，通常用**大O符号**来表示。

**意义**

1. **预测性能**：渐进复杂性帮助我们评估算法在大规模数据上的性能。
2. **对比算法**：通过比较复杂性，可以选择最优算法。
3. **优化方向**：通过分析复杂性，可以找到算法优化的潜在点。

**动态规划 vs. 贪心算法**

* **问题特点  
  动态规划：**最优子结构 + 重叠子问题。  
  **贪心算法：**最优子结构 + 贪心选择性质。
* **解决方式  
  动态规划：**考虑所有可能性，求全局最优，允许回溯。 **贪心算法：**每步选择是局部最优，不能回溯。
* **优缺点  
  动态规划：**能解决复杂问题，但可能占用较多存储空间。 **贪心算法：**实现简单、效率高，但只适用于满足贪心性质的问题。

**回溯算法 vs. 分支限界法**

* **搜索方式  
  回溯算法：**深度优先搜索（DFS） **分支限界法：**广度优先搜索（BFS）
* **剪枝策略  
  回溯算法：**根据约束条件剪枝，排除不满足要求的解。 **分支限界法：**根据下界估计剪枝，排除无可能成为最优解的分支。
* **优缺点  
  回溯算法：**简单灵活，但依赖剪枝优化效率。 **分支限界法：**剪枝更强，更适合大规模问题。