合并排序-分治

时间复杂度：最好 = 最坏 = *O(n log n)*

```
void FZ(int *arr, int l, int r) {
    m = (l + r) / 2;
    FZ(arr, l, m);
    FZ(arr, m + 1, r);
    HB(arr, l, r, m);
}
```

```
void HB(int *arr, int l, int r, int m) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1];
    int R[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }

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

    int i, j;
    i = j = 0;

    while (i < n1 && j < n2) {
        if (L[i] < R[j])
            arr[l++] = L[i++];
        else
            arr[l++] = R[j++];
    }

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

    while (j < n2)
        arr[l++] = R[j++];
}
```

快速排序-分治

时间复杂度

1. **最好情况**
   当每次分区操作都能将数组分为两个大致相等的部分时，快速排序的性能最好。
   时间复杂度：*O(n log n)*
2. **最坏情况**
   当数组已经有序或几乎有序，且分区点总是选择到数组的最小值或最大值时，
   时间复杂度：$O(n^2)$
3. **平均情况**
   时间复杂度：*O(n log n)*

时间上限通常指的是算法在最坏情况下的时间复杂度，因此快速排序的时间上限是：n^2

```
void QuickSort(int *arr, int l, int r) {
    if (l < r) {
        int p = Sort(arr, l, r);
        QuickSort(arr, l, p - 1);
        QuickSort(arr, p + 1, r);
    }
}
```

```
int Sort(int *arr, int l, int r) {
    int p = random(l, r);
    swap(arr[l], arr[p]);
    int i = l;
    int j = r + 1;

    while (true) {
        while (arr[++i] < arr[l] && i < j) {}
        while (arr[--j] > arr[l]) {}
        if (i >= j)
            break;
        swap(arr[i], arr[j]);
    }

    swap(arr[l], arr[j]);
    return j;
}
```

矩阵连乘-动态规划

时间复杂度：由于矩阵连乘问题使用动态规划求解时，其时间复杂度不随输入数据的不同而变化，
最好 = 最坏 = $O(n^3)$

```
void Max(int n, int *p, int **m, int **s) {
    for (int i = 1; i <= n; i++)
        m[i][i] = 0;  // 初始化对角线
```

```
    for (int r = 2; r <= n; r++) {  // 连乘矩阵的数量
        for (int i = 1; i <= n - r + 1; i++) {
            int j = i + r - 1;       // 最后一个矩阵的下标
            m[i][j] = m[i][i + 1] + m[i + 1][j] + p[i - 1] * p[i] * p[j];
            s[i][j] = i;

            for (int k = i + 1; k < j; k++) {
                int t = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (t < m[i][j]) {
                    m[i][j] = t;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

最长公共子序列-动态规划

时间复杂度:
最好 = 最坏 = *O(mn)*，其中 *m* 和 *n* 分别是两个序列的长度。

```
void Max(int m, int n, int *X, int *Y, int **s, int **b) {
    s[0][0] = 0;
    b[0][0] = 1;  // 1, 2, 3 都行，无实际意义，用不到

    // 初始化，考虑只有一个序列有元素的情况
    for (int i = 1; i <= m; i++)
        s[i][0] = 0;
    for (int i = 1; i <= n; i++)
        s[0][i] = 0;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i] == Y[j]) {
                s[i][j] = s[i - 1][j - 1] + 1;
                b[i][j] = 1;
            } else if (s[i - 1][j] > s[i][j - 1]) {
                s[i][j] = s[i - 1][j];
                b[i][j] = 2;
            } else {
                s[i][j] = s[i][j - 1];
                b[i][j] = 3;
            }
        }
    }
```

```
}
```

最大字段和-动态规划

时间复杂度：
最好 = 最坏 = *O(n)*，其中 *n* 是数组的长度。
这是因为算法只需要一次遍历数组，通过维护一个动态规划变量来记录到当前位置为止的最大子数组和。

```
void Max(int n, int *arr) {
    int max = 0;
    int curmax = 0;

    for (int i = 0; i < n; i++) {
        curmax += arr[i];
        if (curmax > max)
            max = curmax;
        else if (curmax < 0)
            curmax = 0;
    }
}
```

图像压缩-动态规划

时间复杂度：
最好 = 最坏 = *O(n)*，其中 *n* 是像素序列的长度。
算法中有一个循环，其次数不会超过 256，对于每个确定的 *i*，可以在 *O(1)* 时间内完成，因此整个算法的时间复杂度为 *O(n)*。

```
void YS(int n, int *p, int *s, int *l, int *b) {
    int Lmax = 256;
    int header = 11;
    s[0] = 0;

    for (int i = 1; i <= n; i++) {
        int bmax = length(p[i]);
        b[i] = bmax;
        s[i] = s[i - 1] + bmax;
        l[i] = 1;

        for (int j = 2; j <= i && j <= Lmax; j++) {
            if (bmax < b[i - j + 1])
                bmax = b[i - j + 1];
            if (s[i] > s[i - j] + j * bmax) {
                s[i] = s[i - j] + j * bmax;
                l[i] = j;
            }
            s[i] += header;
```

```
        }
    }
}
```

0-1 背包-动态规划

时间复杂度：
最好 = 最坏 = *O(nW)*，其中 *n* 是物品的数量，*W* 是背包的容量。
算法需要填充一个二维的 `dp` 表格，其中有 *n* 行（物品数量）和 *W* 列（背包容量）。每个单元格的计算复杂度为 *O(1)*。

```
void Backpack(int n, int C, int *W, int *V, int **m) {
    // 初始化，先考虑最后一个物品
    for (int i = 0; i <= C; i++) {
        if (i < W[n])
            m[n][i] = 0;
        else
            m[n][i] = V[n];
    }

    for (int i = n - 1; i > 1; i--) {
        for (int j = 0; j <= C; j++) {
            if (j < W[i])
                m[i][j] = m[i + 1][j];
            else {
                m[i][j] = max(m[i + 1][j], m[i + 1][j - W[i]] + V[i]);
            }
        }
    }

    m[1][C] = max(m[2][C], m[2][C - W[1]] + V[1]);
}
```

活动安排-贪心算法

```
void GreedySelector(int n, int s[], int f[], bool A[]) {
    A[0] = true;
    int j = 0;

    for (int i = 1; i < n; i++) {
        if (s[i] >= f[j]) {
            A[i] = true;
            j = i;
        } else {
            A[i] = false;
        }
    }
}
```

最优装载-回溯法

时间复杂度

1. **时间复杂度**
   回溯法的时间复杂度通常取决于解空间的大小和每个节点的计算时间。在最优装载问题中，解空间是一棵二叉树，每个物品有两种选择：装入或不装入。因此，解空间的大小为 2n2^n，其中 nn 是物品的数量。
   时间复杂度：$O(2^n)$
2. **最好情况**
   在最好情况下，回溯法可能通过剪枝提前找到最优解，而不需要遍历整个解空间。例如，如果第一个物品的重量超过背包容量，那么解空间的一半可以被剪枝。然而，时间复杂度仍然是指数级的。
   时间复杂度：$O(2^n)$
3. **最坏情况**
   在最坏情况下，回溯法需要遍历整个解空间，即 2n2^n 个节点。
   时间复杂度：$O(2^n)$
4. **时间上限**
   最坏情况下的时间复杂度为 $O(2^n)$。

```
int Maxloading(int n, int *W, int C) {
    Loading X;
    X.n = n;
    X.W = W;
    X.C = C;
    X.bestw = 0;
    X.cw = 0;
    X.Backtrack(1);
    return X.bestw;
}
```

```
void Backtrack(int k) {
    if (k > n) {
        if (cw > bestw)
```

```
            bestw = cw;
        return;
    }

    if (cw + W[k] <= C) {
        cw += W[k];
        Backtrack(k + 1);
        cw -= W[k];
    }

    Backtrack(k + 1);
}
```

最优装载-右子树剪枝-回溯法

时间复杂度：与普通回溯法相同，最坏情况下为 $O(2^n)$。

```
int Maxloading(int n, int *W, int C) {
    Loading X;
    X.n = n;
    X.W = W;
    X.C = C;
    X.bestw = 0;
    X.cw = 0;
    X.rw = 0;  // 初始化剩余重量
    for (int i = 1; i <= n; i++) {
        X.rw += W[i];
    }
    X.Backtrack(1);
    return X.bestw;
}
```

```
void Backtrack(int k) {
    if (k > n) {
        bestw = cw;
        return;
    }

    if (cw + W[k] <= C) {
        cw += W[k];
        Backtrack(k + 1);
        cw -= W[k];
    }

    rw -= W[k];
    if (rw + cw > bestw) {
        Backtrack(k + 1);
```

```
    }
    rw += W[k];
}
```

m 着色问题-回溯法

时间复杂度：

时间复杂度：$O(nm^n)$

在 m 着色问题中，有 n 个顶点，每个顶点有 m 种颜色可以选择。回溯法在最坏情况下会尝试所有可能的颜色组合，因此时间复杂度是 $O(nm^n)$。

```
int mColor(int n, int m, int **a) {
    Coloring X;
    X.n = n;
    X.m = m;
    X.a = a;
    X.sum = 0;
    X.x = new int[n + 1];
    X.Backtrack(1);
    return X.sum;
}
```

```
void Backtrack(int k) {
    if (k > n) {
        sum++;
    } else {
        for (int i = 1; i <= m; i++) {
            x[k] = i;
            if (OK(k))
                Backtrack(k + 1);
        }
    }
}
```

```
bool OK(int k) {
    for (int i = 1; i <= n; i++) {
        if (a[k][i] == 1 && x[k] == x[i])
            return false;
    }
    return true;
}
```

n 皇后问题-回溯法

时间复杂度：

时间复杂度：*O(n!)*

在 n 皇后问题中，回溯法尝试将 n 个皇后放置在 n×n 的棋盘上，每个皇后必须放在不同的行、列和对角线。回溯法在最坏情况下需要遍历所有可能的放置方案，因此时间复杂度为 *O(n!)*。

```
int NQueen(int n) {
    Queen X;
    X.n = n;
    X.x = new int[n + 1];
    X.sum = 0;
    X.Backtrack(1);
    return X.sum;
}
```

```
void Backtrack(int k) {
    if (k > n)
        sum++;
    else {
        for (int i = 1; i <= n; i++) {
            x[k] = i;
            if (Place(k))
                Backtrack(k + 1);
        }
    }
}
```

```
bool Place(int k) {
    for (int i = 1; i < k; i++) {
        if (x[k] == x[i] || abs(x[k] - x[i]) == abs(k - i))
            return false;
    }
    return true;
}
```