目 录

1 主定理

分治法的运用：将一个规模为 n 的实例分为若干个规模为 n/b 的实例，其中 a 个实例需要求解。对于算法的运行时间，有如下公式：

$$T(n) = aT(n/b) + f(n) \tag{1}$$

其中，f(n) 是一个函数，用于表示将问题分解为小问题和将结果合并起来所消耗的时间。

可以使用主定理计算算法复杂度：

$$T(n) \in \begin{cases} \Theta(n^d), & \text{当}a < b^d\text{时} \\ \Theta(n^d \log n), & \text{当}a = b^d\text{时} \\ \Theta(n^{\log_d^a}), & \text{当}a > b^d\text{时} \end{cases} \tag{2}$$

2 合并排序

算法思路：对于一个需要排序的数组 A[0...n-1]，将其分为 $A[0, \lfloor n/2 \rfloor - 1]$ 和 $A[\lfloor n/2 \rfloor ...n-1]$，并对每个字数组递归排序，然后把这两个排好序的子数组合并为一个有序数组。

算法实现如下：

```
void merge(int *B, int *C, int *A, int len_b, int len_c, int left)
{
    int index_b = 0;
    int index_c = 0;
    while(index_b < len_b && index_c < len_c)
    {
        if(B[index_b] < C[index_c])
            A[left++] = B[index_b++];
        else
            A[left++] = C[index_c++];
    }
    while(index_b < len_b)
        A[left++] = B[index_b++];
    while(index_c < len_c)
        A[left++] = C[index_c++];
}

// A的元素范围为[left, right)
void merge_sort(int *A, int left, int right)
{
    int i;
    int j = 0;
    int len = right - left;
```

```
    if(len > 1)
    {
        int mid = (right + left) / 2;
        int *B = (int*)malloc((len/2)*sizeof(int));
        int *C = (int*)malloc((len - len/2)*sizeof(int));

        for(i = 0; i < len/2; ++i)
            B[i] = A[i];
        for(; i < len; ++i)
            C[j++] = A[i];

        merge_sort(B, left, mid);
        merge_sort(C, mid, right);
        merge(B, C, A, len/2, len-len/2, left);
        free(B);
        free(C);
    }
}
```

算法分析:

- 算法基本操作是键值比较操作，键值比较次数 $C(n)$ 的递推关系式如下:

$$\text{当}n > 1\text{时}, C(n) = 2C(n/2) + C_{merge}(n), C(1) = 0 \tag{3}$$

- $C_{merge}(n)$ 是合并阶段进行键值比较的次数。在最坏情况下，$C_{merge}(n) = n-1$。
- 在最坏情况下，如果 $n = 2^k$，最差效率递推式的精确解为 $C_{worst} = n\log_2^n - n + 1$。
- 在最坏情况下，算法时间复杂度为 $C_{worst}(n) \in \Theta(n\log^n)$。
- 合并排序的主要缺点就是该算法需要线性的额外空间。

3 快速排序

算法思路：对给定数组中的元素进行重新排列，得到一个分区。在这个分区中，所有在 s 下标之前的元素都小等于 A[s]，所有在 s 下标之后的元素都大等于 A[s]。随后对 A[s] 前和 A[s] 后的子数组进行相同的操作。

算法实现如下:

```
int partition(int *A, int left, int right)
{
    int index = left;
    int pivot = A[right];
    int i;
    int temp;
    for(i = left; i < right -1; ++i)
```

```
        {
            if(A[i] < pivot)
            {
                temp = A[index];
                A[index] = A[i];
                A[i] = temp;
                ++index;
            }
        }
        temp = A[index];
        A[index] = A[right];
        A[right] = temp;
        return index;
    }
    int hoare_partition(int *A, int left, int right)
    {
        int index = left;
        int pivot = A[left++];
        int temp;
        while(left < right)
        {
            while(A[left] < pivot)
                ++left;
            while(A[right] > pivot)
                --right;
            if(left >= right)
                break;
            temp = A[right];
            A[right] = A[left];
            A[left] = temp;
        }
        temp = A[right];
        A[right] = A[index];
        A[index] = temp;
        return right;
    }
    // A的元素范围为[left, right]
    void quick_sort(int *A, int left, int right)
    {
        int mid;
        if(left < right)
        {
            mid = partition(A, left, right);
            quick_sort(A, left, mid-1);
            quick_sort(A, mid+1, right);
        }
    }
```

算法分析：

- 如果扫描指针交叉，那么建立分区之前所执行的键值比较次数是 n+1；如果它们相等，那么键值比较次数是 n。
- 最优情况是所有分裂点位于相应子数组的中点。在最优情况下，键值比较次数

$C_{best}(n)$ 满足下面的递推式:

$$当\ n>1\ 时, C_{best}(n) = 2C_{best}(n/2) + n, C_{best}(1) = 0 \tag{4}$$

根据主定理，$C_{best}(n) \in \Theta(n \log_2^n)$。当 $n = 2^k$ 时，$C_{best}(n) = n \log_2^n$。

- 在最差的情况下，所有的分裂点抖趋于极端：两个子数组有一个为空，而另一个子数组仅仅比被分区的数组少一个元素。当输入的数组已经被排过序时，最差情况就会发生。最差情况下键值比较次数为 $C_{worst}(n) = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$。
- 平均键值比较次数 $C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2^n$。

4 折半查找

算法思路：对于一列有序数组，比较查找键 K 和数组中间元素 A[m]。如果它们相等，则算法结束。如果 K<A[m]，则对数组的前半部分执行该操作。如果 K>A[m]，则对数组的后半部分执行该操作。

算法实现如下:

```
int binary_search(int *A, int K, int len)
{
    int left = 0;
    int right = len-1;
    int mid;
    while(left <= right)
    {
        mid = (left + right) / 2;
        if(K == A[mid])
            return mid;
        else if(K < A[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1;
}
```

算法分析:

- 键值比较次数不仅取决于 n，还取决于输入的特征。
- 在最坏的情况下，算法在进行了一次比较后，除了数组规模变为原来的二分之一，算法仍然面临同样的情况。最坏情况下的键值比较次数有如下递推式:

$$当\ n>1\ 时, C_w(n) = C_w(\lfloor n/2 \rfloor) + 1, C_w(1) = 1 \tag{5}$$

对于任意的正整数 n，$C_w(n) = \lfloor \log_2^n \rfloor + 1 = \lceil \log_2(n+1) \rceil$。

- 在最差情况下，折半查找的时间复杂度为 $\log_2^n$。
- 折半查找的平均键值比较次数为 $C_{avg}(n) \approx \log_2^n$。在查找成功的情况下，$C_{avg}(n) \approx \log_2(n-1)$。在查找失败的情况下，$C_{avg}(n) \approx \log_2(n+1)$。

5 二叉树的遍历

二叉树高度的定义：二叉树高度是根的左、右子树的最大高度加一。空树的高度为-1。计算二叉树高度的代码如下：

```
typedef struct tree_node tree_node;
struct tree_node
{
    int key;
    tree_node* left;
    tree_node* right;
}
int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
int height(tree_node* root)
{
    if(root == NULL)
        return -1;
    return max(height(root->left), height(root->right)) + 1;
}
```

算法分析：

- 算法的基本操作是检查树是否为空。
- 外部顶点的数量总是比内部顶点的数量大一。
- 检查树是否为空的比较操作次数为 $C(n) = 2n + 1$，加法操作的次数为 $A(n) = n$。

二叉树的三种经典遍历算法如下：

- 在前序遍历中，根在访问左右子树之前被访问。
- 在中序遍历中，根在访问左子树之后，但在访问右子树之前被访问。
- 在后序遍历中，根在访问左右子树之后被访问。

6 大整数乘法

7 Strassen 矩阵乘法

8 用分治法解最近对问题

9 用分治法解凸包问题