在计算机科学中，有许多常用的排序算法，下面我将介绍十大常用的排序算法的思路，并提供使用C++实现的示例代码。

我们先从最简单的冒泡排序开始，它的思路是比较相邻的元素并交换，将较大（或较小）的元素逐渐“冒泡”到列表的一端。

1. 冒泡排序（Bubble Sort）：

思路：比较相邻元素，若顺序错误则交换，每次循环将最大（或最小）的元素冒泡到末尾。

C++代码实现：

```
#include <iostream>

void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // 交换元素
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, size);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出结果为：11 12 22 25 34 64 90

这是冒泡排序的一个简单示例，接下来我们将继续介绍其他排序算法的思路和代码实现。

2.选择排序（Selection Sort）：

思路：在未排序序列中找到最小（或最大）的元素，放到序列的起始位置，然后再从剩余未排序元素中找到最小（或最大）元素，放到已排序序列的末尾。

C++代码实现：

```
#include <iostream>

void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // 交换元素
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, size);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出结果为：11 12 22 25 34 64 90

选择排序通过不断选择最小的元素并交换位置来进行排序，它的时间复杂度为O(n^2)。接下来我们将介绍其他的排序算法。

3.插入排序（Insertion Sort）：

思路：将数组分为已排序和未排序两部分，初始时将第一个元素视为已排序部分，然后将未排序部分的元素逐个插入到已排序部分的正确位置，直到全部元素有序。

C++代码实现：

```
#include <iostream>

void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
```

```
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, size);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出结果为：11 12 22 25 34 64 90

插入排序的时间复杂度也是O(n^2)，但它在对部分有序的数组进行排序时有较好的性能表现。下面我们将继续介绍其他排序算法。

4.希尔排序（Shell Sort）：

思路：希尔排序是插入排序的一种改进版本，它将数组按照一定的间隔分组，对每组使用插入排序。然后逐渐缩小间隔，重复进行分组和排序操作，直到间隔为1，完成最后一次插入排序。

C++代码实现：

```
#include <iostream>

void shellSort(int arr[], int size) {
    for (int gap = size / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < size; i++) {
            int temp = arr[i];
            int j = i;

            while (j >= gap && arr[j - gap] > temp) {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = temp;
        }
```

```
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    shellSort(arr, size);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出结果为：11 12 22 25 34 64 90

希尔排序通过逐步缩小间隔并对分组进行插入排序，以减少插入排序的次数，从而提高效率。它的时间复杂度取决于间隔序列的选择，一般为O(nlogn)。接下来我们将继续介绍其他的排序算法。

5.归并排序（Merge Sort）：

思路：归并排序采用分治的思想，将待排序数组分成两个子数组，分别对子数组进行排序，然后将两个排序好的子数组合并成一个有序数组。不断重复这个过程，直到最终数组有序。

C++代码实现：

```
#include <iostream>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // 创建临时数组
    int* L = new int[n1];
    int* R = new int[n2];

    // 将数据复制到临时数组
    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    // 合并临时数组到原数组
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
```

```
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // 将剩余元素复制到原数组
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    // 释放临时数组的内存
    delete[] L;
    delete[] R;
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // 递归地对左右子数组进行排序
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // 合并两个子数组
        merge(arr, left, mid, right);
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, size - 1);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出结果为：11 12 22 25 34 64 90

归并排序具有稳定性和较高的效率，它的时间复杂度为O(nlogn)。接下来我们将继续介绍其他的排序算法。

6.快速排序（Quick Sort）：

思路：快速排序也是一种分治算法，选择一个基准元素，将数组分成两部分，使得左边的元素都小于等于基准元素，右边的元素都大于等于基准元素。然后递归地对左右子数组进行快速排序。

C++代码实现：

```
#include <iostream>

int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // 选择最后一个元素作为基准
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            // 交换元素
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // 将基准元素放到正确位置
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        // 递归地对两个子数组进行快速排序
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, size - 1);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
```

```
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出结果为：11 12 22 25 34 64 90

快速排序是一种高效的排序算法，它的时间复杂度为O(nlogn)，但在最坏情况下可能达到O(n^2)。下面我们将继续介绍其他的排序算法。

7.堆排序（Heap Sort）：

思路：堆排序利用堆这种数据结构进行排序，将待排序的数组构建成一个最大堆（或最小堆），然后依次将堆顶元素与最后一个元素交换，并调整堆，重复这个过程直到整个数组有序。

C++代码实现：

```
#include <iostream>

void heapify(int arr[], int size, int root) {
    int largest = root;    // 最大值的索引
    int left = 2 * root + 1;
    int right = 2 * root + 2;

    // 找出左子节点和右子节点中的最大值
    if (left < size && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < size && arr[right] > arr[largest]) {
        largest = right;
    }

    // 若最大值不是根节点，则交换根节点和最大值，并递归调整堆
    if (largest != root) {
        int temp = arr[root];
        arr[root] = arr[largest];
        arr[largest] = temp;

        heapify(arr, size, largest);
    }
}

void heapSort(int arr[], int size) {
    // 构建最大堆
    for (int i = size / 2 - 1; i >= 0; i--) {
        heapify(arr, size, i);
    }

    // 逐个将堆顶元素与最后一个元素交换，并调整堆
    for (int i = size - 1; i > 0; i--) {
```

```
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            heapify(arr, i, 0);
        }
    }

    int main() {
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        int size = sizeof(arr) / sizeof(arr[0]);

        heapSort(arr, size);

        std::cout << "排序结果: ";
        for (int i = 0; i < size; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;

        return 0;
    }
```

输出结果为：11 12 22 25 34 64 90

堆排序利用了堆这种数据结构的特性，它的时间复杂度为O(nlogn)，且具有稳定性。接下来我们将继续介绍其他的排序算法。

8.计数排序（Counting Sort）：

思路：计数排序是一种非比较排序算法，它适用于排序范围相对较小的整数序列。该算法通过统计每个元素出现的次数，然后根据统计结果重构有序序列。

C++代码实现：

```
#include <iostream>

void countingSort(int arr[], int size) {
    int maxVal = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > maxVal) {
            maxVal = arr[i];
        }
    }

    // 创建计数数组
    int* count = new int[maxVal + 1]();

    // 统计元素出现次数
    for (int i = 0; i < size; i++) {
        count[arr[i]]++;
    }
```

```
    // 根据统计结果重构有序序列
    int k = 0;
    for (int i = 0; i <= maxVal; i++) {
        while (count[i] > 0) {
            arr[k] = i;
            count[i]--;
            k++;
        }
    }

    // 释放计数数组的内存
    delete[] count;
}

int main() {
    int arr[] = {4, 2, 2, 8, 3, 3, 1};
    int size = sizeof(arr) / sizeof(arr[0]);

    countingSort(arr, size);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出结果为：1 2 2 3 3 4 8

计数排序的时间复杂度为O(n + k)，其中n是待排序数组的大小，k是计数范围。计数排序在排序范围较小且元素分布均匀时具有较好的性能。下面我们将继续介绍其他的排序算法。

9.桶排序（Bucket Sort）：

思路：桶排序是一种分布排序算法，它将待排序元素划分为多个桶，每个桶内的元素进行排序，然后将桶中的元素按顺序合并得到最终有序序列。

C++代码实现：

```
#include <iostream>
#include <vector>
#include <algorithm>

void bucketSort(float arr[], int size) {
    // 创建桶
    std::vector<float> buckets[size];

    // 将元素放入对应的桶中
    for (int i = 0; i < size; i++) {
```

```
            int bucketIndex = size * arr[i];
            buckets[bucketIndex].push_back(arr[i]);
        }

        // 对每个桶内的元素进行排序
        for (int i = 0; i < size; i++) {
            std::sort(buckets[i].begin(), buckets[i].end());
        }

        // 合并桶中的元素得到有序序列
        int index = 0;
        for (int i = 0; i < size; i++) {
            for (float num : buckets[i]) {
                arr[index] = num;
                index++;
            }
        }
    }

    int main() {
        float arr[] = {0.42, 0.32, 0.64, 0.12, 0.22, 0.11, 0.90};
        int size = sizeof(arr) / sizeof(arr[0]);

        bucketSort(arr, size);

        std::cout << "排序结果: ";
        for (int i = 0; i < size; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;

        return 0;
    }
```

输出结果为：0.11 0.12 0.22 0.32 0.42 0.64 0.9

桶排序的时间复杂度取决于桶的个数和每个桶内的排序算法，一般情况下是O(n + k)，其中n是待排序元素的数量，k是桶的数量。桶排序在元素分布均匀时有较好的性能表现。接下来我们将介绍最后一个排序算法。

10.基数排序（Radix Sort）：

思路：基数排序是一种按照元素的位数进行排序的算法。它从最低位开始，依次按位比较元素，将元素分配到对应的桶中，然后按照桶的顺序将元素重新组合得到有序序列。这个过程重复进行，直到最高位，完成排序。

C++代码实现：

```
#include <iostream>

int getMax(int arr[], int size) {
    int maxVal = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > maxVal) {
```

```
            maxVal = arr[i];
        }
    }
    return maxVal;
}

void countingSort(int arr[], int size, int exp) {
    int output[size];    // 存储排序结果的临时数组
    int count[10] = {0}; // 存储计数的数组

    // 统计元素出现次数
    for (int i = 0; i < size; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // 计算累加次数
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // 构建排序结果
    for (int i = size - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // 将排序结果复制到原数组
    for (int i = 0; i < size; i++) {
        arr[i] = output[i];
    }
}

void radixSort(int arr[], int size) {
    int maxVal = getMax(arr, size);

    // 对每位进行计数排序
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        countingSort(arr, size, exp);
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    radixSort(arr, size);

    std::cout << "排序结果: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

输出结果为：11 12 22 25 34 64 90

基数排序的时间复杂度为O(d * (n + k))，其中d是最大元素的位数，n是待排序元素的数量，k是基数（一般为10）。基数排序适用于整数排序，并且对于位数较少的整数具有较好的性能。以上就是十大常用排序算法的思路和C++代码实现。