

算法第一次上机

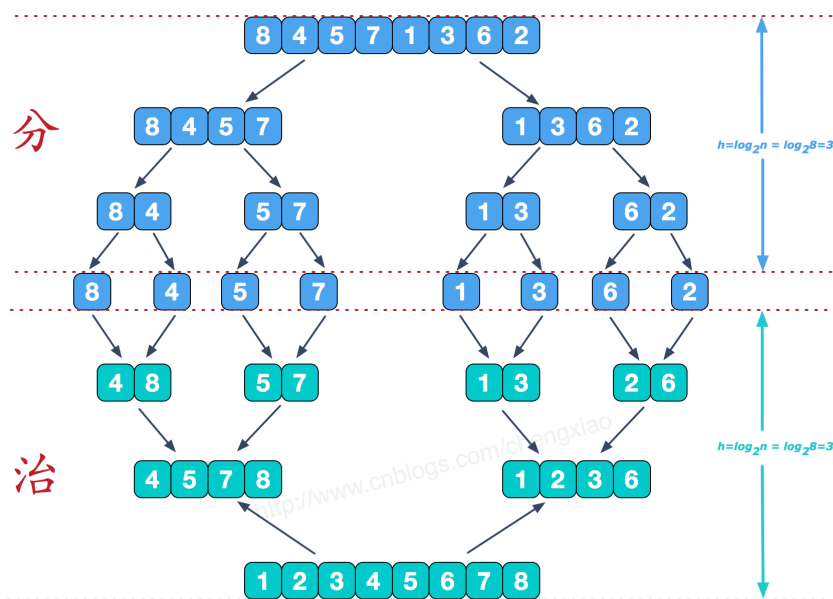
闫一慧20009200331

MERGE-SORT

归并排序是一种经典的分治算法，用于对数组进行排序。它的基本思想是将数组递归地划分为较小的子数组，然后将这些子数组进行合并，最终得到一个有序的数组。

归并排序的代码思路如下：

1. **分解（Divide）**：将待排序的数组递归地分解为较小的子数组，直到子数组的大小为1或为空。
2. **合并（Merge）**：将两个已排序的子数组合并为一个有序的子数组。这个过程中，需要创建一个临时数组来存储合并的结果。
3. **递归（Recursion）**：重复步骤1和步骤2，对左右两个子数组进行递归调用，直到所有的子数组都排序完成。
4. **合并结果**：最后一次合并得到的子数组就是已排序的数组。



具体代码思路如下：

1. 定义一个辅助函数 `merge()`，用于合并两个已排序的子数组。函数参数包括原始数组、临时数组、左侧子数组的起始位置 `left`、中间位置 `mid` 和右侧子数组的结束位置 `right`。

2. 在 `merge()` 函数中，设置三个指针：`l_pos` 指向左侧子数组的起始位置，`r_pos` 指向右侧子数组的起始位置，`pos` 指向临时数组的起始位置。
3. 通过比较 `arr[l_pos]` 和 `arr[r_pos]` 的大小，将较小的元素放入临时数组 `temparr` 中，并将对应指针向后移动。
4. 如果某一侧子数组的元素已经全部放入临时数组，而另一侧子数组还有剩余元素，则直接将剩余元素全部放入临时数组。
5. 最后，将临时数组 `temparr` 中的元素复制回原始数组 `arr` 的对应位置。
6. 定义一个递归函数 `msort()`，用于实现归并排序的分治过程。函数参数包括原始数组、临时数组、左侧子数组的起始位置 `left` 和右侧子数组的结束位置 `right`。
7. 在 `msort()` 函数中，首先判断如果 `left < right`，则继续递归调用 `msort()` 函数进行划分。
8. 将数组划分为两个子数组后，分别对左右两个子数组进行递归调用 `msort()` 函数。
9. 最后，调用 `merge()` 函数将左右两个子数组合并为一个有序的子数组。

```
#include<stdio.h>
#include<stdlib.h>
void print_arr(int arr[],int n)
{
    for(int i=0;i<n;i++)
    {
        printf("%d ",arr[i]);
    }
    printf("\n");
}
void merge(int arr[],int temparr[],int left,int mid,int right)
{
    //临时数组的下标
    int l_pos=left;
    int r_pos=mid+1;
    int pos=left;
    //比较并合并
    while(l_pos<=mid&& r_pos<=right)
    {
        if(arr[l_pos]<arr[r_pos])
            temparr[pos++]=arr[l_pos++];
        else
            temparr[pos++]=arr[r_pos++];
    }
    //防止一边已经放完而另一边没有放入，无法进行比较直接放入即可
    while(l_pos<=mid)
```

```

{
    temparr[pos++] = arr[l_pos++];
}
while(r_pos ≤ right)
{
    temparr[pos++] = arr[r_pos++];
}
// 将临时数组元素传入原始数组
while(left ≤ right)
{
    arr[left] = temparr[left];
    left++;
}
}

void msort(int arr[], int temparr[], int left, int right)
{
    // 如果只有一个元素就不需要划分
    if(left < right)
    {
        int mid = (left + right) / 2;
        msort(arr, temparr, left, mid);
        msort(arr, temparr, mid + 1, right);
        merge(arr, temparr, left, mid, right);
    }
}

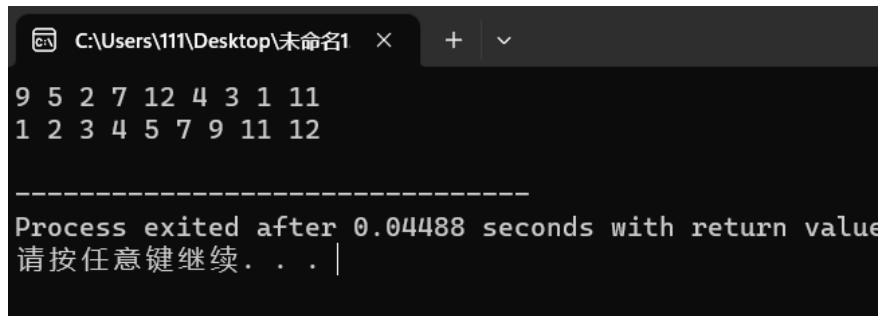
void merge_sort(int arr[], int n)
{
    // 分配辅助数组
    int* temparr = (int*)malloc(n * sizeof(int));
    if(temparr)
    {
        msort(arr, temparr, 0, n - 1);
        free(temparr);
    }
}

int main()
{
    int arr[] = {9, 5, 2, 7, 12, 4, 3, 1, 11};
    int n = 9;
    print_arr(arr, n);
    merge_sort(arr, n);
    print_arr(arr, n);
}

```

```
return 0;
}
```

结果展示：



```
C:\Users\111\Desktop\未命名1 >
9 5 2 7 12 4 3 1 11
1 2 3 4 5 7 9 11 12

-----
Process exited after 0.04488 seconds with return value
请按任意键继续. . . |
```

归并排序具有稳定性和较好的时间复杂度，适用于对大规模数组进行排序。

INSERT-SORT

插入排序（Insertion Sort）是一种简单直观的排序算法。它的工作方式类似于我们打扑克牌时的排序方式，逐个将未排序的元素插入已排序的子数组中，直到所有元素都被插入并完成排序。

算法思路如下：

通过不断地将当前元素与已排序的子数组进行比较和移动，插入排序逐渐构建起一个有序的数组。

插入排序算法的代码思路如下：

1. 创建一个函数 `insertionSort`，它接受一个数组和数组的长度作为参数。
2. 使用一个循环遍历数组，从第二个元素开始（索引为1），将当前元素视为待插入元素。
3. 在内部循环中，将当前元素与已排序的子数组中的元素进行比较，同时将较大的元素向右移动。
4. 当找到已排序的元素小于或等于当前元素时，将当前元素插入到该位置。
5. 重复步骤2至4，直到所有元素都被插入并完成排序。



输入和输出部分与快排一致，这里只展示关键部分插入排序算法的代码：

```
void insertionSort(int arr[], int n) {  
    int i, key, j;  
    for (i = 1; i < n; i++) {  
        key = arr[i];  
        j = i - 1;  
  
        // 将比 key 大的元素向右移动  
        while (j ≥ 0 && arr[j] > key) {  
            arr[j + 1] = arr[j];  
            j = j - 1;  
        }  
        arr[j + 1] = key;  
    }  
}
```

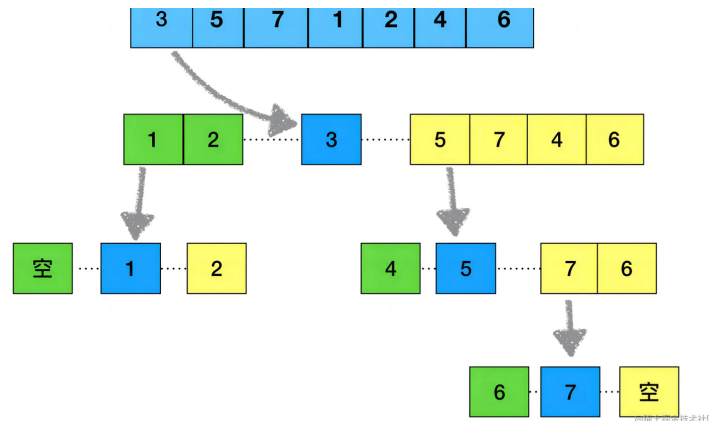
结果展示：

```
12 11 13 5 6  
5 6 11 12 13  
  
-----  
Process exited after 0.03658 seconds with return va  
请按任意键继续. . .
```

插入排序是一个原地排序算法，它的时间复杂度为 $O(n^2)$ ，其中 n 是数组的长度。尽管在最坏情况下的时间复杂度较高，但对于小规模数组或基本有序的数组，插入排序表现良好，并且它是稳定的排序算法，不会改变相等元素的相对顺序。

QUICK-SORT

快速排序是一种高效的排序算法，它基于分治策略。它的核心思想是选择一个主元（pivot），然后将数组划分为两个子数组，一个小于等于主元的子数组，一个大于主元的子数组。然后对这两个子数组分别递归地应用快速排序算法，直到子数组的长度为1或0，即已经有序。



以下是用C语言实现的快速排序算法：

```
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j ≤ high - 1; j++) {
        if (arr[j] ≤ pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void quicksort(int arr[], int low, int high) {
    if (low < high) {
```

```

        int pivotIndex = partition(arr, low, high);
        quicksort(arr, low, pivotIndex - 1);
        quicksort(arr, pivotIndex + 1, high);
    }
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quicksort(arr, 0, n - 1);

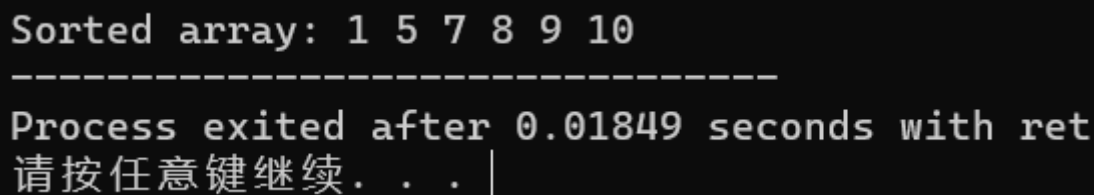
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}

```

这段代码首先定义了一个用于交换两个元素的辅助函数 `swap`，然后实现了 `partition` 函数来进行划分操作。`partition` 函数选择最后一个元素作为主元，并将数组划分为两个子数组。接下来，`quicksort` 函数使用递归来对子数组进行排序。最后，在 `main` 函数中，我们创建一个示例数组并调用 `quicksort` 函数对其进行排序，然后打印排序后的结果。

运行结果如下：



```

Sorted array: 1 5 7 8 9 10
-----
Process exited after 0.01849 seconds with ret
请按任意键继续. . . |

```

1. How many comparisons will Quicksort do on a list of n elements that all have the same value?

列表中的所有元素都相同的情况下，Quicksort 仍然会执行 $n * (n-1) / 2$ 次比较。这是因为选择主元和划分步骤将始终将数组分成大小为 $n-1$ 和 0 的两个子数组。因此，在这种情况下，比较次数的递归关系是 $T(n) = T(n-1) + (n-1)$ ，解得 $T(n) = n * (n-1) / 2$ 。

2. What are the maximum and minimum number of comparisons will Quicksort do on a list of n elements, give an instance for maximum and minimum case respectively.

在最大情况下，主元总是选择子数组中的最大或最小元素。这种情况可能发生在数组已按升序或降序排序的情况下。在这种情况下，Quicksort每个递归层级将执行 $n-1$ 次比较，总共进行 $n * (n-1) / 2$ 次比较。

例如，如果我们有数组[1, 2, 3, 4, 5]，最大比较次数将是10 ($n * (n-1) / 2 = 5 * 4 / 2 = 10$)。

在最小情况下，主元被选择为将数组分成两个大小相等的子数组的位置。这可能发生在数组以某种平衡方式完全平衡的情况下。在这种情况下，Quicksort每个递归层级将执行 $\log_2(n)$ 次比较，总共进行 $n * \log_2(n)$ 次比较。

例如，如果我们有数组[5, 2, 8, 1, 7, 3]，最小比较次数将是9 ($n * \log_2(n) = 6 * \log_2(6) \approx 9$)。

Priority Queue

以下是一个用c语言实现优先队列的示例：

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int priority;
    // 可根据需要添加其他数据成员
} Element;

typedef struct {
    Element* elements;
    int capacity;
    int size;
} PriorityQueue;

PriorityQueue* createPriorityQueue(int capacity) {
    PriorityQueue* queue = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    queue->capacity = capacity;
    queue->size = 0;
    queue->elements = (Element*)malloc(capacity * sizeof(Element));
    return queue;
}

void destroyPriorityQueue(PriorityQueue* queue) {
    free(queue->elements);
    free(queue);
}
```



```

void enqueue(PriorityQueue* queue, Element element) {
    if (queue->size ≥ queue->capacity) {
        printf("Error: Priority queue is full.\n");
        return;
    }

    int i = queue->size - 1;
    while (i ≥ 0 && queue->elements[i].priority > element.priority) {
        queue->elements[i + 1] = queue->elements[i];
        i--;
    }
    queue->elements[i + 1] = element;
    queue->size++;
}

Element dequeue(PriorityQueue* queue) {
    if (queue->size ≤ 0) {
        printf("Error: Priority queue is empty.\n");
        Element emptyElement = {0}; // 返回一个空元素作为错误处理
        return emptyElement;
    }

    Element element = queue->elements[0];
    for (int i = 1; i < queue->size; i++) {
        queue->elements[i - 1] = queue->elements[i];
    }
    queue->size--;
    return element;
}

int main() {
    PriorityQueue* queue = createPriorityQueue(5);

    Element e1 = {3};
    enqueue(queue, e1);

    Element e2 = {1};
    enqueue(queue, e2);

    Element e3 = {5};
    enqueue(queue, e3);
}

```

```

    Element e4 = {2};
    enqueue(queue, e4);

    Element e5 = {4};
    enqueue(queue, e5);

    printf("Dequeuing elements from the priority queue:\n");
    while (queue->size > 0) {
        Element element = dequeue(queue);
        printf("Priority: %d\n", element.priority);
    }

    destroyPriorityQueue(queue);

    return 0;
}

```

程序运行结果如下：

```

Dequeuing elements from the priority queue:
Priority: 1
Priority: 2
Priority: 3
Priority: 4
Priority: 5

-----
Process exited after 0.02227 seconds with return value 0
请按任意键继续. . .

```

在这个示例中，我们定义了 `Element` 结构来表示优先级队列中的元素，它只包含一个 `priority` 成员，但你可以根据需要添加其他数据成员。

`PriorityQueue` 结构表示优先级队列本身，其中包含一个 `elements` 数组用于存储元素，`capacity` 表示队列的最大容量，`size` 表示队列当前的大小。

`createPriorityQueue` 函数用于创建一个具有给定容量的优先级队列，并返回指向该队列的指针。

`destroyPriorityQueue` 函数用于销毁优先级队列，释放内存。

`enqueue` 函数用于将元素插入到优先级队列中，根据优先级进行插入排序。

`dequeue` 函数用于从优先级队列中移除并返回具有最高优先级的元素。

在 `main` 函数中，我们创建了一个优先队列并进行了一系列的 `enqueue` 操作来插入元素。然后，我们使用 `dequeue` 函数逐个移除并打印队列中的元素，直到队列为空。