# 二分查找

template <typename T>

bool IsSorted(const T\* arr, int arrSize)

{ return true; }

// 二分查找：求query第一次出现的位置

template <typename T>

int BinarySearch(const T\* arr, int arrSize, const T& query)

{

if (!arr || arrSize == 0 || !IsSorted<T>(arr, arrSize)) return -1;

// 1 or 2 elements

if (arrSize < 3) {

for (int i=0; i<2; ++i)

if (arr[i] == query) return i;

return -1;

}

int lower = 0, upper = arrSize - 1, mid = 0;

// invariant: (lower < upper) && (arr[lower] < query <= arr[upper])

while (lower + 1 < upper) // at least 3 elements to check

{

mid = lower + (upper - lower) / 2;

if (arr[mid] <= query) // 如果要找query最后一次出现的位置，应改为<=

lower = mid; // new lowerbound

else

upper = mid; // new upperbound

}

// invariant: (lower+1 == upper) && (arr[lower] < query <= arr[upper])

// assert: there must be 2 elements left

// assert: first occurrence of query may be in arr[upper] only

if (arr[upper] == query) return upper;

else if (arr[lower] == query) return lower;

else return -1;

}

选择排序方法：

n较小(<50): 插入、选择

基本有序：插入、冒泡

n较大：快速、归并、堆

可在归并排序中对小数组采用插入排序

时间 空间 稳定(不改变等值元素顺序)

quick 快速 nlogn ~ n^2 logn N

merge 归并 nlogn n Y

heap 堆 nlogn 1 N

shell 希尔 n^1.x 1 N

线性复杂度排序：计数排序、桶排序

//// 插入排序

template <typename T>

int IntertionSort(T\* arr, int arrSize)

{

if (!arr || arrSize == 0) return -1;

for (int i=1; i<arrSize; ++i) // for each element to insert

{

T newElem = arr[i];

int hole = i;

while (hole > 0 && arr[hole-1] > newElem) {

arr[hole] = arr[hole-1]; // move previous element to hole

--hole;

}

arr[hole] = newElem;

}

return 0;

}

// 冒泡排序

template <typename T>

int BubbleSort(T\* arr, int arrSize)

{

if (!arr || arrSize == 0) return -1;

for (int i=0; i<arrSize; ++i)

{

for (int j=arrSize-1; j>i; --j)

{

if (arr[j-1] > arr[j])

swap(arr[j-1], arr[j]);

}

}

return 0;

}

# 归并排序

template<class T>

void MergeSort(T\* A, int l, int r) // 归并 A[l]~A[r]

{

printf("%d - %d\n", l, r);

if (A && l < r) {

int m = l+(r-l)/2;

MergeSort(A,l,m);

MergeSort(A,m+1,r);

MergeSortedArrays(A,l,m,r); // merge A[l:m] and A[m+1:r]

}

}

template<class T>

void MergeSortedArrays(T\* A, int l, int m, int r)

{

if (l > m || m+1 > r) return;

int n=r-l+1; // total number of elements in the two subarray

T\* B = new T[n];

memset(B, 0, sizeof(T) \* n);

int i=l, j=m+1, k=0;

while (i<=m && j<=r) {

if (A[i] < A[j])

B[k++] = A[i++];

else

B[k++] = A[j++];

}

// append remaining elements in either subarray

while (i<=m)

B[k++] = A[i++];

while (j<=r)

B[k++] = A[j++];

// copy result back to array A

k=0;

while (k<n)

A[l++] = B[k++];

delete [] B; B=NULL;

}

# 快速排序

// A[lower:mid] < threshold; A[mid+1:upper] >= threshold

int Partition(int\* A, int lower, int upper)

{

assert(A);

int randIndex = lower + rand() % (upper-lower+1);

swap(A[upper], A[randIndex]);

const int& threshold = A[upper];

int lastSmallIndex = lower-1; // last element that is smaller than threshold

for (int i=lower; i<upper; ++i)

{

if (A[i] < threshold)

swap(A[++lastSmallIndex], A[i]);

}

++lastSmallIndex; // first element >= bound

swap(A[lastSmallIndex], A[upper]);

return lastSmallIndex;

}

void QuickSort(int\* A, int lower, int upper) // 快速 A[lower]~A[upper]

{

if (!A || lower >= upper) return;

int mid = Partition(A, lower, upper); // A[m] is at the correct position

QuickSort(A, lower, mid-1);

QuickSort(A, mid+1, upper);

}