For sorted:

（add 的复杂度是 n, 因为先要找到应该加入的位置）

	array-based	link-list-based
add	n	n
remove(val)	n	1
remove(iterator)	**n**	**n (1 for doubly linked)**
find(val)	$\log n$	**n**
op*	1	1
op[]	**1**	**n**

从 ordered 到 sorted container，除了 add 变了外唯一的区别是 array_based 的 `find(val)` 复杂度降了，因为可以在 contiguous 的内存上使用 Binary search.

所以对于 sorted 的 container，使用 array instead of list 作为 underlying d.s. 在 find 上有额外的好处.

For ordered container:

(add 的复杂度是 $O(1)$, 因为直接加在最后)

	array-based	link-list-based
add	1	1
remove(val)	n	n
remove(iterator)	**n**	**n (1 for doubly linked)**
find(val)	n	n
op*	1	1
op[]	**1**	**n**
insert	**n**	**n (1 for doubly linked)**

```
int lower_bound(double a[], double val, int
left, int right) {
    while (right > left) {  //comp1
        int mid = left + (right - left) / 2;

        if (a[mid] < val)   //comp2
            left = mid + 1
        else
            right = mid;
    }

    return left
}
```

$$T(n) = aT(\frac{n}{b}) + f(n) \ \ , f(n) \in \Theta(n^c)$$

直观上 a 越大，b 越小，f(n) 越大则 T 越大. 可以得到:

如果 $a > b^c$，那么 a, b 的组合 dominate，$T(n) \in \Theta(n^{\log_b a})$

如果 $a = b^c$，那么 a, b 的组合和 f 同时 take dominance，$T(n) \in \Theta(n^c \log n)$

如果 $a < b^c$，那么 f take dominance，$T(n) \in \Theta(n^c)$

但是如果 $f(n) \in \Theta(n^{\log_b a} \log^k n)$ 时是一个特殊情况，称为 fourth condition，此时可以直接得到 $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$.

比如 $T(n) = 2T(\frac{n}{2}) + n \log n$，那么 $T(n) = \Theta(n \log^2 n)$

Complexity of set operations

Initialize: $O(1)$ 如果 unsorted；$O(n \log n)$ if sc
排序）；

clear: $O(n)$

isMember: $O(\log n)$ ，需要binary search

copy: $O(n)$，即和空集 union

union, intersect：$O(n)$

```
    while (first1 != last1 && first2 != last2)
{
        if (compare(*first1, *first2)) {
//compare 返回的是 (x<=y)
            *result++ = *first1++;  // set1
element less than set2 element
        } else if (compare(*first2, *first1)) {
            *result++ = *first2++;  // set2
element less than set1 element
        } else {    //else: 两个元素相等, both
index++
            *result++ = *first1++;
            ++first2;
        }
    }
    // get remainings elements
    while (first1 != last1)
        *result++ = *first1++;
    while (first2 != last2)
        *result++ = *first2++;
    return result;  // sorted union of set1 and
set2
```

Bubble Sort

bubble sort 的做法是：对于把每个元素 `a[i]` 都替换成 `a[i:end]` 中最小的元素

具体是从 end 往左遍历所有 `a[i::]` 的元素，右边比左边小
就交换，这样 greedily 可以得到最后 left 的元素一定是范围
内最小的。

```
void bubble(Item a[], size_t left, size_t
right) {
    for (size_t i = left; i < right - 1; i++) {
// last element 不用排序
        for (size_t j = right-1; j > i; j--) {
            if (a[j] < a[j-1]) {
                std::swap(a[j], a[j-1]);
            }
        }
    }
}
```

差别是本来在任何 case 都是 n-1 swaps；现在 worst case n-1 swaps, best case 0 swap

但是 comparisons 的数量多了（多出了 n-1 个 if

本来是 $\frac{n^2-n}{2}$ comps，现在是 $\frac{n^2-n}{2} + n - 1$ comps

```
void adaptive_selection_sort(Item a[], size_t
left, size_t right) {
    for (size_t i = left; i < right - 1; i++) {
        size_t min_index = i;
        for (size_t j = i + 1; j < right; j++)
{
            if (a[j] < a[min_index]) {
                min_index = j;
            }
        }
        if (min_index != i) {
            std::swap(a[i], a[min_index]);
        }
    }
```

```
void improved_insertion_sort(Item a[], size_t
left, size_t right) {
    for (size_t i = right - 1; i > left; --i) {
/ first find the min item, move to left
        if (a[i] < a[i-1]) {
            std::swap(a[i], a[i-1]);
        }
    }

    for (size_t i = left + 2; i < right; i++) {
        // a[left:i-1] is sorted
        Item v = a[i];
        size_t j = i;
        // while v in the wrong position j, we
et a[j]=a[j-1],
        // ready to move v to a[j-1] (when j-1
s the right pos)
        while (v < a[j-1]) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = v;
    }
```

```
struct IsOddPred {
    bool operator()(int n) {
        return n%2 == 1;
    }
```

Property 1: 当 data 有一个 **const upper limit to the number of inversions for each element** (即：partially sorted，没有一个元素存在O(n) 个相关的 inversion) 的时候，**insertion/bubble sort 的 comparisons 和 swaps 数量几乎是 linear 的.**

Property 2: 当 **{a ∈ data : a 相关的 inversions 数量 constant} 这个集合大小是 const** 时，**insertion sort 的 comparisons 和 swaps 数量几乎是 linear 的.**

Property 3: 当 key 很小时，即便 item 大，selection sort 几乎是 linear 的（comparison 主导，swap cheap，comparison 消耗小的时候很好）

```
Array &=operator=(const Array &other) {
  Array temp(other);    //temporary object

  // assign by swap, 原 object 数据和 temp 交换
  std::swap(length, temp.length);
  std::swap(data, temp.data);

  return *this; //temp out of scope 自动被 dtor
掉
}
```

这个方法去除了 explicit deallocation in the assignment operator，同时很好地处理了 Self-Assignment Handling.

Lambda

一个 lambda expression 是一个这样的形式:

form 1:

```
[captures list] (parameter list) {function
body}
```

（captures list 意思是 variables from the surrounding scope 中所有对这一条 Lambda 表达式 available 的.）

ex:

```
[] (int n1, int n2) { return abs(n1 - n2) > 5;
}
```

sort	time	memory	stable
bubble	n^2	1	yes
selection	(n^2-n)/2	1	no
insertion	n^2	1	yes
count	n	n+k	yes
merge	n log n	n	yes
heap	n log n	1	no
quick	n log n	log n	bo

bubble, insertion: efficient when data are closed to final pos

insertion: also efficient when add new item to sorted data

selection: sorted when key is small, especially when items large

std::sort take 两个 iterator 和一个 operator<

```
size_t partition_middle(int a[], size_t left, size_t right) {
    size_t pivot = left + (right - left) / 2; // middle point as partition
    swap(a[pivot], a[--right]); // move the content of pivot to the end
    // then everything is the same as partition_right

    pivot = right;
    while(true) {
        while (a[left] < a[pivot]) {
            left++;
        }
        while (left < right && a[right] >= a[pivot]) {
            right--;
        }
        if (left >= right) {
            break;
        }
        swap(a[left], a[right - 1]);
    }
    swap(a[left], a[pivot]);
    return left;
```

```
void quicksort2(int a[], size_t left, size_t right) {
    if (left+1 >= right) {
        return;
    }
    size_t pivot_index = partition_middle(a, left, right);

    // sort the smaller partition first
    if (pivot_index - left < right - pivot_index) {
        quicksort2(a, left, pivot_index);
        quicksort2(a, pivot_index + 1, right);
    } else {
        quicksort2(a, pivot_index + 1, right);
        quicksort2(a, left, pivot_index);
    }
```

```
void quicksortWithHeapsortFallback(int a[], int left, int right, int depth) {
    if (right - left <= CUTOFF) {
        insertionSort(a, left, right);  // 小数组使用插入排序
        return;
    }

    if (depth > MAX_DEPTH) {
        heapsort(a, left, right);  // 超过最大深度，使用堆排序
        return;
    }

    size_t pivot_index = partition_middle(a, left, right);
    quicksortWithHeapsortFallback(a, left, pivot_index, depth + 1);
    quicksortWithHeapsortFallback(a, pivot_index + 1, right, depth + 1);
}
```

Ternary Operator

```
c[k] = (a[i] <= b[j]) ? a[i++] : b[j++]
```

等价于

```
if (a[i] <= b[j]) {
    c[k] = a[i];
    ++i;
}
else {
    c[k] = b[j];
    ++j;
}
```

```
void merge(Item a[], size_t left, size_t middle, size_t right) {
    size_t n = right - left;
    vector<Item> c(n);

    for (size_t i = left, j = middle, k = 0; k < n; ++k) {
        if (i == middle) {
            c[k] = a[j++];
        } else if (j == right) {
            c[k] = a[i++];
        } else {
            c[k] = (a[i] < a[j]) ? a[i++] : a[j++];
        }
    }
}
```

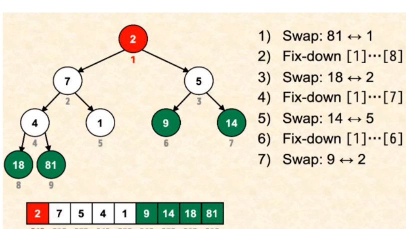
Top-down merge sort

```
void topDown_Mergesort(Item a[], size_t left, size_t right) {
    if (right < left + 2) return;
    size_t middle = left + (right - left) / 2;
    topDown_Mergesort(a, left, middle);
    topDown_Mergesort(a, middle, right);
    merge(a, left, middle, right);
}
```

Heapsort 一个 vector a[n]

1. 对 a[1 to n] 进行 heapify
2. 把 top element 和 tail element 互换
3. 把换来的 tail element **在 a[1 to n-1] 范围内进行 fixdown**
4. a[1 to n-1] 现在有 heap structure. 我们在 a[1 to n-1] 上重复这个过程.

```
void heapsort(Item heap[], int n) {
    heapify(heap, n);
    for (int i = n; i >= 2; --i) {
        swap(heap[i], heap[1]);
        fixDown(heap, i-1, 1);
    }
}
```



top to bottom 进行 fix up: 要遍历完所有元素，每个元素 swap 的次数最多是所在层的数目，越往下越多，因而是 $O(n\log n)$

bottom to top 进行 fix down: 最后一层不用 fix，从导数第二层开始，每一层 k 对于每个上层节点，左右两个 $n1_k, n2_k$ 只需要 fix 一个就可以；并且每个元素 swap 的次数从导数第二层的 1 开始，往上一层就+1（同时元素也更少），复杂度的结果是 $O(n)$

bottom to top fix down: $O(n)$

top to bottom fix up: $O(n\log n)$

比如 linear growth 的 vector 的 "push" 这一操作的 amortized complexity 是

$$\frac{O(n)+n\cdot O(1)}{n}=O(1)$$

的 amortized cost.

但是 constant growth 的 vector 则是:

假设 vector 每次 grow const c 个单位

那么 constant growth 的 vector 的 "push" 这一操作的 amortized complexity 是

$$\frac{O(n)+c\cdot O(1)}{c}=O(n)$$

```
void push(Item newItem) {
    heap[++heapsize] = newItem;
    fixUp(heap, heapsize);
}
```

```
void pop() {
    heap[1] = heap[heapsize--];
    fixDown(heap, heapsize, 1);
}
```

sorted array:

1. $O(n)$ insertion （排序）
2. $O(1)$ inspect top
3. $O(1)$ pop (移走end)
4. O(nlogn) create

PQ 也可以通过普通的 unsorted array 和 sorted array 而不是 heapified array 实现，复杂度是:

unsorted array:

1. $O(1)$ insertion （insert end）
2. $O(n)$ inspect top （排序后最大的里面的最前面
3. $O(1)/O(n)$ pop （排序后最大的里面的最后面，可以 keep track
4. O(n) create

假如我们有一个 node 的值被修改得更高了，那么我们要把它 fix up：和 ancestor nodes 更换值直到 parent 比他大为止.

```
void fixUp(Item heap[], int k) {
    while (k > 1 && heap[k/2]<heap[k]) {
        swap(heap[k], heap[k/2]);
        k /= 2;
    }
}
```

假如 node 的值被修改得更低了，那么 fix down：和 descendents 比较，交换(左child) 到两个 children 都比他小为止.

```
void fixDown(Iten heap[], int heapsize, int k)
{
    while (2 * k <= heapsize) {
        int j = 2*k;    // left child index
        if (j < heapsize && heap[j] < heap[j+1])
            ++j;    // right child greater?
        if (heap[k] >= heap[j]) // node > max(left, right), heap alreday restored
            break;
        swap(heap[k], heap[j]);
        k = j;  // move down
    }
}
```

fixup, fixdown 是 $O(\log n)$ 的

nary heap:

1. O(n) create
2. O(log n) push
3. O(log n) pop
4. O(1) top