Common Errors to Check

Division by 0

Recursion has no terminating condition

Accidentally breaking invariants

Try to read from a Output Iterator

Try to access nullptr after modifying codes

Assume two input source are different (Assume they are the same all the time to avoid tricky corner cases)

Does not add idiot check

Does not consider that the input can be empty range

Does not use `delete` & `delete[]` to free memory, causing memory leak

Off-by-One Error

```
const size_t SIZE = 5;
int x[SIZE];
size_t i;

// Acessing x[SIZE]
for (i = 0; i <= SIZE; ++i)
    x[i] = i;

// Accessing x[SIZE - 1 + 1]
for (i = 0; i <= SIZE - 1; ++i)
    x[i] = x[i + 1];
```

```
// Not accessing x[SIZE - 1]
for (i = 1; i < SIZE; ++i)
    x[i - 1] = i;
```

Use reserve and index at the same time or use resize and push at the same time

```
// would double the size
vector<int> tmp(k);
while (...) {
    tmp.push_back(...);
}

// would cause the size to be empty
vector<int> tmp;
tmp.reserve(k);
for(int i = 0; i < k; ++k) {
    tmp[i] = ...;
}
```

'\0' at the front

```
const  char *s = "hello";
char ss[20];int  length = strlen(s);
for (int i = 0; i < length; ++i)
 ss[i] = s[length  - i];printf("%s", ss);
```

Mistake memory complexity with time complexity

Swap An Array of N Elements (True no matter N even or odd)

`while (i = 0; i < n / 2; ++i) {`
`swap(ar[i], ar[n - i -1]); }`

String

```
sizeof("abcd") = 5;
strlen("abcd") = 4;
"abcd".length() = 4;
"abcd".size() =4;
```

Complexity Analysis

Remember $O(n^2) = O(n^3) = O(n^n)$

Ascending Complexity

- $\Theta(1)$
- $\Theta(n)$
- $\Theta(nlog(n)) = \Theta(log(n!))$
- $\Theta(n^2)$
- $\Theta(2^n)$
- $\Theta(n!)$
- $\Theta(n^n)$

Master Theorem

$$\text{For } a \geq 1, b > 1 \text{ if } f(n) \in \Theta(n^c)$$
$$T(n) = aT(\frac{n}{b}) + f(n) \text{ Then:}$$
$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

Complete binary tree

Every level, except possibly the last, is completely filled, and all nodes are as far left as possible

Heap

kth min-max heap;

$\Theta(n)$ **time complexity to build a heap on a known size array.**

Heapsort has worst-case Θ(nlog n) time complexity.

Fix-up & fix-down has an average-case time complexity of Θ(log n).

insertion is done by inserting the element at the end, and then calling fix-up.

Two ways to build a heap.

> 1. Proceeding from the bottom of the heap to the top, while repeatedly calling fixDown()
> 2. Proceeding from the top of the heap to the bottom, while repeatedly calling fixUp()

Linked List

Sorting algorithm that does not make asymptotic worst-case complexity even worse

> bubble sort
>
> selection sort
>
> insertion sort
>
> quick sort
>
> merge sort

Fast Algorithm on nearly sorted data

> bubble sort
>
> insertion sort

Set Union

Ordered container:

The relative position of other elements will be the same for insertion and deletion.

Stable sort

Important when the you have to sort with multiple keys (e.g. fisrt name then last name.)

> Bubble sort
>
> Insertion sort (Stable: place the to_insert at the back when encountering ties)
>
> merge sort (Stable: place the element with larger index at the back when encountering ties)

Sort

Bubble Sort: Adaptive

> Characteristic
>
> > Front elements are not fully sorted
> >
> > Large elements are bubble to the back
> >
> > A few back elements at the end are sorted
> >
> > Most elements reordered

```
void bubble(Item a[], size_t left, size_t right) {
    for (size_t i = left; i < right - 1; ++i) {
        bool swapped = false;
        for (size_t j = right - 1; j > i; --j) {
            if (a[j] < a[j - 1]) {
                swapped = true;
                swap(a[j - 1], a[j]);
            }  // if
        }  // for j
        if (!swapped)
            break;
    }  // for i
}
```

Selection Sort: Adaptive (Still $O(n^2)$ comparisons but reduce swaps)

> Characteristic
>
> > Front elements sorted
> >
> > Back elements where the original front elements lies are swapped with another element previously lying at the front
> >
> > All other position elements stay their original positions
> >
> > An element can't be placed to the front unless all elements smaller than itself have been picked to the front
> >
> > The swaps can happen a few times, so the element at the corresponding front and back locations may not have a direct mapping relation, be careful when validating & this is also a sign

```
void selection(Item a[], size_t left, size_t right) {
   for (size_t i = left; i < right - 1; ++i) {
       size_t minIndex = i;
       for (size_t j = i + 1; j < right; ++j)
           if (a[j] < a[minIndex])
               minIndex = j;
       if (minIndex != i)
           swap(a[i], a[minIndex]);
   }  // for
}  // selection()
```

Insertion Sort: Adaptive

> Characteristic
>
> > Front elements sorted
> >
> > Back elements untouched
>
> ```
> void insertion(Item a[], size_t left, size_t right) {
>     for (size_t i = left + 1; i < right; ++i) {
>         Item v = a[i]; size_t j = i;
>         while ((j > left) && (v < a[j - 1])) {
>             a[j] = a[j - 1];
>             --j;
>         }  // while
>         a[j] = v;
>     }  // for i
> }  // insertion(
>
>
> void insertion(Item a[], size_t left, size_t right) {
>     for (size_t i = right - 1; i > left; --i)  // find min
> item
>         if (a[i] < a[i - 1])                     // put in
> a[left]
>             swap(a[i - 1], a[i]);                  // as
> sentinel
>     for (size_t i = left + 2; i < right; ++i) {
>         Item v = a[i]; size_t j = i;          // v is new item
>         while ((j > left) && v < a[j - 1]) {  // v in wrong
> spot
>             a[j] = a[j - 1];                    //
> copy/overwrite
>             --j;
>         }  // while
>         a[j] = v;
>     }  // for i
> }  // insertion()
> ```

Quick Sort

> Characteristic
>
> > Front elements are not fully sorted
> >
> > Last element placed somewhere near the middle as the pivot

> > All element before the pivot is smaller than itself
> >
> > All elements after the pivot is bigger than itself
> >
> > If elements are originally in the correct left or correct right, their positions don't change
>
> ```
> size_t partition(Item a[], size_t left, size_t right) {
>     size_t pivot = left + (right - left) / 2; // pivot is middle
>     swap(a[pivot], a[--right]);               // swap with right
>     pivot = right;                            // pivot is right
>
>     while (true) {
>         while (a[left] < a[pivot])
>             ++left;
>         while (left < right && a[right - 1] >= a[pivot])
>             --right;
>         if (left >= right)
>             break;
>         swap(a[left], a[right - 1]);
>     }  // while
>     swap(a[left], a[pivot]);
>     return left;
> }
>
>
> void quicksort(Item a[], size_t left, size_t right) {
>     if (left + 1 >= right)
>         return;
>     size_t pivot = partition(a, left, right);
>     quicksort(a, left, pivot);
>     quicksort(a, pivot + 1, right);
> }
> ```

Merge Sort

> Characteristic
>
> > In Quiz & Exam: a chunk's size equal to the largest odd factor of the total size of the array
> >
> > Every chunk of elements are sorted but all elements are not sorted

> The elements in the original chunk will not be placed in some other chunk

```
void mergeAB(Item c[], Item a[], size_t size_a,
             Item b[], size_t size_b) {
    size_t i = 0, j = 0;
    for (size_t k = 0; k < size_a + size_b; ++k) {
        if (i == size_a)
            c[k] = b[j++];
        else if (j == size_b)
            c[k] = a[i++];
        else
            c[k] = (a[i] <= b[j]) ? a[i++] : b[j++];
    }  // for
}  // mergeAB()


void merge_sort(Item a[], size_t left, size_t right) {
    if (right < left + 2)  // base case: 0 or 1 item(s)
        return;
    size_t mid = left + (right - left) / 2;
    merge_sort(a, left, mid);  // [left, mid)
    merge_sort(a, mid, right);  // [mid, right)
    merge(a, left, mid, right);
}  // merge_sort()


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

    for (size_t i = left, j = mid, k = 0; k < n; ++k) {
        if (i == mid)
            c[k] = a[j++];
        else if (j == right)
            c[k] = a[i++];
        else
            c[k] = (a[i] <= a[j]) ? a[i++] : a[j++];
    }  // for
    copy(begin(c), end(c), &a[left]);
}  // merge()
```

Heap Sort

> Characteristic
>
> > All elements has a heap shape of min / max
> >
> > May pop a few tops to the back and the back have a sorted order of max / min
> >
> > Then, front elements still in heap shape of min / max
> >
> > The front heap does not have more extreme elements than the sorted back
>
> ```
> // pesudo code
> Heapify()
> while(size() > 1) {
>     swap(first, last);
>     pop_back();
>     fix_down();
> }
> ```

C++ STL

Copy Back & Forth

> ```
> #include <vector>
> #include <algorithm>
> using namespace std;
> const size_t N = 100;
>
> int main() {
>     vector<int> v(N, -1);
>     int a[N];
>
>  for (size_t j = 0; j != N; ++j)
>      v[j] = (j * j * j) % N;
>  copy(v.begin(), v.end(), a); // copy over
>  copy(a, a + N, v.begin()); // copy back
>  //copy(begin(a), end(a), begin(v)); // equivalent
>  sort(a, a + N);
> ```

> ```
>  sort(v.begin(), v.end());
>  vector<int> reversed(v.rbegin(), v.rend());
> }  // main()
> ```

iota

```
#include <numeric>
vector<int> v(100);
iota(begin(v), end(v), 0);
// v contains {0, 1, 2, ...,
```

Vector (3 * 8 bytes overhead)

Begin allocated memory

End allocated memory

End used memory

> ```
> vector<vector<int>> twoDimArray(10);
>
> // option 1
> for (size_t i = 0; i < 10; ++i)
>  twoDimArray[i] = vector<int>(20, -1);
>
> // option 2
> for (size_t i = 0; i < 10; ++i)
>  twoDimArray[i].resize(20, -1);
>
> // option 3
> vector<vector<int>> twoDimArray(10, vector<int>(20, -1));
> ```

Deque

> push_back()
>
> back()
>
> pop_back()
>
> push_front()
>
> front()
>
> pop_front()

Priority Queue

> ```
> std::priority_queue<T, std::vector<T>, std::less<T>>
> defaultPQ;
> std::priority_queue<T, std::vector<T>, COMP> customPQ;
> // Range build constructor, copy the vector elements and build
> a heap inplace, tight_bound = O(n).
> std::priority_queue<T, std::vector<T>, COMP>
> rangePQ(sourceVector.begin(), sourceVector.end(),
> COMP_OBJECT);
> ```

Iterators

Index Sorting

```
class SortByCoord {
    const vector<double> &_coords;
public:
    SortByCoord(const vector<double> &z) : _coords(z) {}
    bool operator()(size_t i, size_t j) const {
        return _coords[i] < _coords[j];
    }  // operator()()
};  // SortByCoord{}


vector<size_t> idx(100);
vector<double> xCoord(100);
for (size_t k = 0; k != 100; ++k) {
    idx[k] = k;
    xCoord[k] = rand() % 1000 / 10.0;
}  // for
SortByCoord sbx(xCoord);  // sbx is a function object!
sort(begin(idx), end(idx), sbx);
```

Permutation with a stack and queue

```
template <typename T>
void genPerms(vector<T> &perm, deque<T> &unused) {
    // perm is only a "prefix" until unused is empty
    if (unused.empty()) {
    cout << perm << '\n';
    return;
    }  // if
    for (size_t k = 0; k != unused.size(); ++k) {
        perm.push_back(unused.front());
        unused.pop_front();
        genPerms(perm, unused);
        unused.push_back(perm.back());
        perm.pop_back();
    }  // for
}
```

Tail Recursion Reuse Stack Frame $O(1)$

```
int factorial(int n, int res = 1) {
    if (n == 0)
        return res;
    return factorial(n - 1, res * n);
}

int power(int x, uint32_t n, int result = 1) {
    if (n == 0) {
        return result;
    } else if (n % 2 == 0) { // even
        return power(x * x, n / 2, result);
    } else { // odd
        return power(x * x, n / 2, result * x);
    }
}  // power()
```

Read in

```
int i;
string buffer
while (!(cin >> i)) {
    // Keep reading unless encounter a integer
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

while (getline(cin, buffer)) {
    // Keep reading until reaching cin.eof()
}
```

Write out

```
template <typename T>
ostream &operator<<(ostream &out, const stack<T> &s) {
    // display the contents of a stack on a single line
    // e.g., cout << my_stack << endl;
    stack<T> tmpStack = s;  // deep copy
    while (!tmpStack.empty()) {
        out << tmpStack.top() << ' ';
        tmpStack.pop();
    }  // while
    return out;
}  // operator<<()

template <typename T>
ostream &operator<<(ostream &out, const stack<T> &s) {
    // display the contents of a stack on a single line
    // e.g., cout << my_stack << endl;
    for (auto &el : v) {
        out << el << ' ';
    }
    return out;
}  // operator<<()
```

Lambda

There are several ways to capture variables in a lambda:

[] captures no variables from the surrounding scope

[=] captures all variables in the surrounding scope by value (i.e., a copy of each variable is made)

[&] captures all variables in the surrounding scope by reference

[foo] captures only the variable foo, by value

[&foo] captures only the variable foo, by reference

[=, &foo] captures all variables in the surrounding scope by value except for foo, which is captured by reference

[&, foo] captures all variables in the surrounding scope by reference except for foo, which is captured by value

[this] captures the current object – needed if a lambda defined in an object’s member function needs access to its member variable