Time Complexity (Easy)

1.
```
int f(int n) {
    int i = 0;
    while (i<(n/2))
        i += 2;
    return 0;
}
```
O(n)

```
int f(int n) {
    for (int i=1;i<(n/2);i*=2)
        O1function()
    return 0;
}
```
O(log n)

```
int f(int n) {
    for(int i=0;n>i;i++)
        for(int j=0;n/10>j;j++)
            O1function();
    return 0;
}
int g(int n) {
    for (int i=0;i<(n);i++)
        Q1function();
}
```
$O(n^2)$

```
int f(int n) {
    for (int i=0;n*n>i;i+=n)
        O1function();
    return 0;
}
```
O(n)

```
void f(int n) {
    while (n>1000)
        n = n / 2;
}
```
O(log n)

```
void f(double n) {
    if (n < 0) return;
    f(n - 0.005);
}
```
O(n)

Time Complexity (Medium)

```
int f(int n) {
    for (int i=2040;i<(n*1000);i+=i)
        O1function();
    return 0;
    }
```

O(log n)

```
void f(int n) {
    for (int i=0;i<(n);i++)
        g(n);
}
void g(int n) {
    if (n < 10) return;
    g(n/2);
}
```

O(n log n)

```
void g(int n);
void f(int n) {
    for(int i=0;i<(n);i++) g(n/2);
}
void g(int n) {
    for(int i=0;i<(n);i++) O1function();
}
```

$O(n^2)$

```
int f(int n) {
    for (int i=1;i<(n);i+=20)
        if (i%2==0) i*=2;
    return 0;
    }
```

O(n)

```
int g(int n);
int f(int n) {
    for(int i=0;i<(n/2);i++) g(i);
}
int g(int n) {
    for(int i = 0;i<10000;i++) O1function();
}
```

O(n)

Time Complexity (Hard)

```
void f(int n) {
    if (n < 10) return;
    for(int i=0;i<(n);i++)
        O1function();
    f(n/2);
}
```

O(n)

```
void f(int n) {
    if (n<10) return;
    for (int i=0;i<10;i++)
        f(n/10);
    for (int i=0;i<(n);i++)
        O1function();
}
```

O(n log n)

```
void f(int n) {
    if (n<10) return;
    for (int i=0;i<(n/10);i++)
        g(i*10);
}
void g(int n) {
    if (n<10) return;
    g(n-10);
}
```

$O(n^2)$

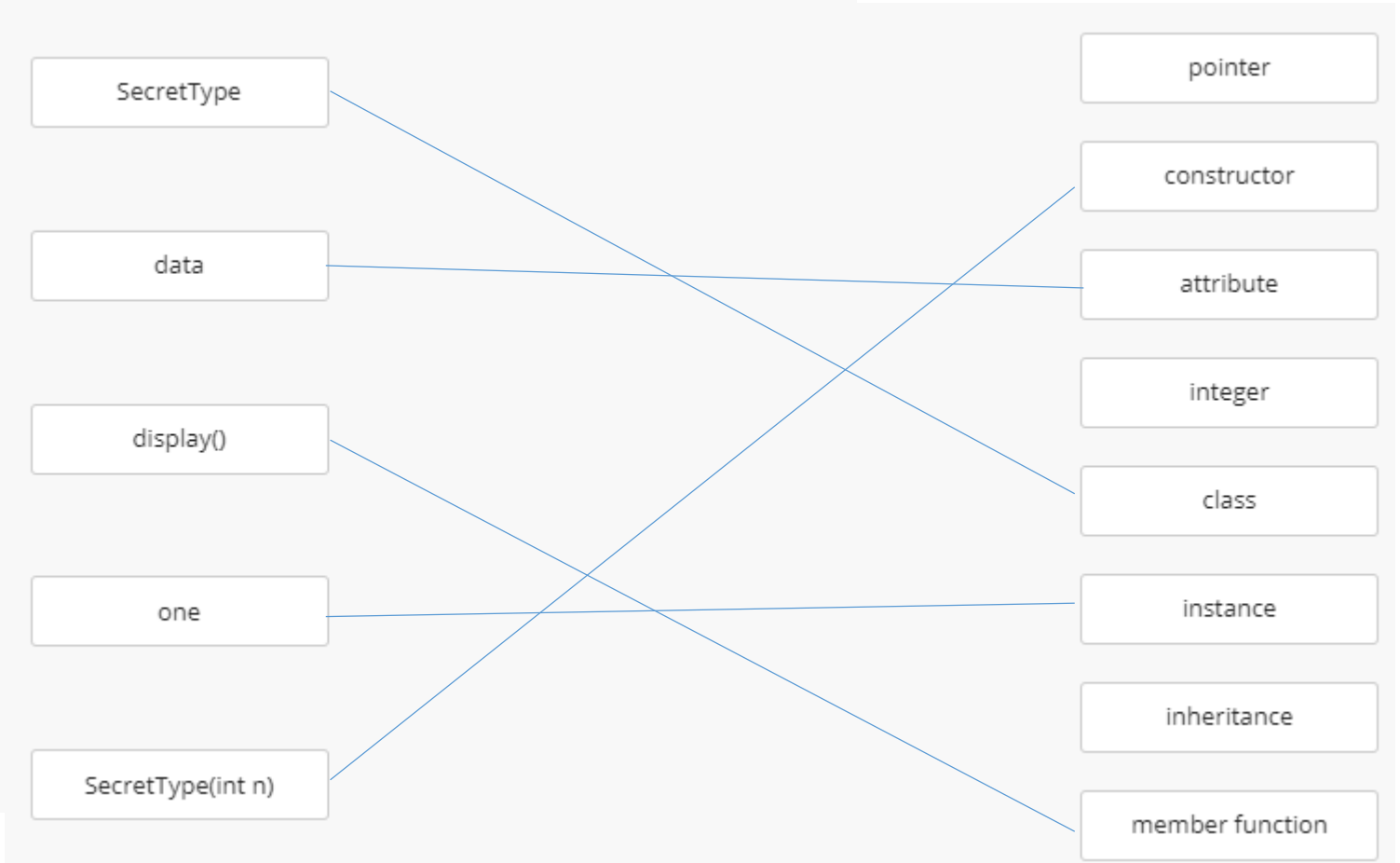
```
void f(int n) {
    int counter = 1;
    while (counter < (n)) {
        g(n);
        counter *= 2;
    }
}
void g(int n) {
    if (i<10) return ;
    for (int i=0;i<(n);i++)
        return g(9);
}
```

O(log n)

C++ OOP Basic

```
#include
class SecretType{
    private:
        int data;
    public:
        SecretType(int n) {data = n;};
        ~SecretType(){};
        void display() {printf("%d\n",data);};
};

int main() {
    SecretType one(99);
    one.display();
}
```



True or False

T • The time complexity is the same when we perform Insertion Sort on an array or a linked list.

T • A C++ class can have no constructor with one destructor

F • A compiler will report an error and cannot compile if a C++ class has no member function, including no constructor nor destructor.

F • If we are lucky, Quicksort can finish sorting in O(n) for some particular special cases.

O(log n)

- Given a sorted array with indices from 1 to n. We use binary search but we change the mid = (begin+end)/10 instead of mid = (begin+end)/2 What will be the new time complexity?

```
Search(A, key, n)
    begin = 0
    end = n
    while begin < end - 1 do:
        if key < A[(begin+end)/10] then
        end = (begin+end)/10
        else begin = (begin+end)/10
    return A[begin]
```

True or False

T • There are sorting algorithms with their worst case complexity equals to O(n) with n is the number of items to be sorted.

T • *This statement is just a recap of the lecture and it is true.*

• *We introduced the paranoidQuickSort that repeat partitioning until the pivot is good, i.e. the pivot divides the array into two such that each of them is at least n/9 items. And the expected number of times to repeat is O(1).*

• Please decide if the **following statement** is true or false:

• If we want to repeat the partitioning such that the pivot divides the array into two parts such that each of them is at least n/3 items, The expected number of repeating until we found such a pivot randomly is still O(1).

Linked List Coding

```
bool List::checkUnique() {
    ListNode* current = __1__;
    while (current) {
        ListNode* temp = current;
        while (temp->__2__) {
            temp = temp->__3__;
            if (temp->__4__ == current->__5__)
                return false;
        }
        current = current->_next;
    }
    return true;
    }
```

```
bool List::checkUnique() {
    ListNode* current = _head;
    while (current) {
        ListNode* temp = current;
        while (temp->_next) {
            temp = temp->_next;
            if (temp->_item == current->_item)
                return false;
        }
        current = current->_next;

    }
    return true;
}
```

Linked List Coding

- For the previous question (about checking uniqueness in a linked list), what is the time complexity? And do you have a better suggestion. Please describe briefly and give the time complexity for your new idea.
- Old: $O(n^2)$
- Sample answer:
  - Sort and compare O(n log n)
  - There could be solution of O(n) if you can justify it

```
bool List::checkUnique() {
    ListNode* current = __1__;
    while (current) {
        ListNode* temp = current;
        while (temp->__2__) {
            temp = temp->__3__;
            if (temp->__4__ == current->__5__)
                return false;
        }
        current = current->_next;
    }
    return true;
}
```