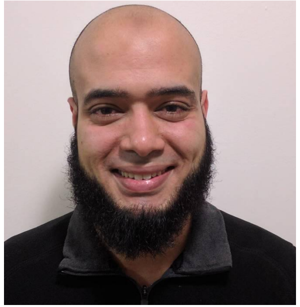
Data Structures
Space Complexity

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Time vs Space

- We learned how to compute the time order O() of code
- But our code also consumes **memory**. So we also compute its **space** order
- Very similar thoughts to the time
  - The order is about worst case, an upper bound!
  - What is the **largest needed memory** at any point of time during the program?
  - We mainly focus when N goes so large
    - Ignore constants and factors
  - It is all about estimates, nothing exact, but this is enough in practice!
  - 2 algorithms of the same time/space order may have different constants

O(1) memory

- We knew this is O(nm) **time**
- But how much memory?
- We have a few integers defined
- This means, regardless N, the same memory is used
- This is O(1) **memory**
- How many bytes in the data types?
  - We don’t care. Matter of small factors

```
void f3(int n, int m) {      // O(nm)
    int cnt = 0;
    for (int i = 0; i < 2 * n; ++i)
        for (int j = 0; j < 3 * m; ++j)
            cnt++;
}
```

Tips

- Ignore all fixed variables
    - n, sum, i, j
- The memory has dynamic array created of size n
    - So O(n) memory here
- No other memory creation
- The nested loops is the largest for time = O(n^2) time

```
int* f(int n) { // Total O(n) memory, O(n^2) time

    // This line: O(n) time and O(n) memory
    int *p = new int[n] {};

    for (int i = 0; i < n; ++i) // O(n) time
        p[i] = i;

    int sum = 0;     // O(n^2) time
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            sum += p[i];
        }
    }
    return p;
}
```

Tips

- This function creates constant memory
    - Basic variables: n, i, j
    - 10k integers (fixed)
- All is fixed: time and memory
- O(1)
- Tip: Fixed? Ignore

```
int* f2() { // Total O(1) memory, O(n^2) time
    int n = 10000;
    int *p = new int[n] {};

    for (int i = 0; i < n; ++i)
        p[i] = i;

    int sum = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            sum += p[i];
    return p;
}
```

Tips

- What is the largest memory **at any time**?
- Only O(n)
    - Whenever we create, we delete

```
int* f3B(int n) {   // O(n) time/memory
    return new int[n] {};
}

void f3A(int n) {
    // O(n^2) time but still O(n) memory
    for (int i = 0; i < n; ++i) {
        int *p = f3B(n);
        delete [] p;
    }
}
```

Tips

- Now, we delete nothing
- We create 1 + 2 + 3 + 4 …. N memory
    - Which is O(n^2)
- Note: This is memory leak

```
int* f4B(int n) {    // O(n) time/memory
    return new int[n] {};
}

void f4A(int n) {
    // O(n^2) time & memory
    for (int i = 0; i < n; ++i) {
        int *p = f3B(n);
        // we accumulate memory!
    }
}
```

Tips

- Like time, we focus on the largest term
    - 10n, 20n = 30n ⇒ O(n)

```
int* f5(int n) {    // O(n) time/memory
    int p1 = new int[10 * n] {};
    int p2 = new int[20 * n] {};
}
```

Tips

- O(n)
- O(m)
- So we sum O(n+m)

```
void f6(int n, int m) { // O(n+m) time/memory
    int p1 = new int[10 * n] {};
    int p2 = new int[20 * m] {};
}
```

Tips

- I prefer to exclude parameters with reference or pointers from order
  - Some courses don't
- f7B doesn't create new memory other than fixed
- Total O(n) memory

```
int f7B(int *arr, int n) {
    // O(1) excluding parameters with reference
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += arr[i];
    return sum;
}

void f7A(int n) {
    int *x = new int[n];    // O(n) memory
    f7B(x, n);  // O(1) memory
}
```

Tips

- f8B receives a vector by reference, so O(1) for this param
    - Created by someone else
- Function f8A max point is O(n) memory

```cpp
int f8B(vector<int> & v, int f) {
    // O(n) time and O(1) memory
    int sum = 0;
    for (int i = 0; i < v.size(); ++i)
        sum += v[i] * f;
    return sum;
}

void f8A(int n) {
    //vector of n numbers: On) memory
    vector<int> v(n, 1);

    // O(n^2) time and O(1) memory
    for (int i = 0; i < n; ++i)
        f8B(v, i);
}
```

Tips

- The change here v is **not by reference**
- In every call, a temporary vector of n items is created
- So it is O(n) memory

```
int f9B(vector<int> v, int f) {
    // O(n) time and O(n) memory
    // The vector n items will copied each time!
    int sum = 0;
    for (int i = 0; i < v.size(); ++i)
        sum += v[i] * f;
    return sum;
}

void f9A(int n) {
    //vector of n numbers: On) memory
    vector<int> v(n, 1);

    // O(n^2) time and O(n) memory
    for (int i = 0; i < n; ++i)
        f9B(v, i);
}
```

Tips

- Clearly this is O(1) memory
- What if we wrote it recursively?
- Same memory? Think

```
int factorial1(int n) {
    // O(n) time and O(1) memory
    int res = 1;
    for (int i = 1; i <= n; ++i)
        res *= i;
    return res;
}
```

Tips

- Recursion is a bit tricky.
- If we have N recursive calls, then the variables in each call remains in memory
- E.g. we will have N copies of subres variables
- So O(n) memory
- We call it **auxiliary space** (*extra temporary space used by an algorithm*)

```
int factorial2(int n) {
    // O(n) time and O(n) memory
    if(n <= 1)
        return 1;

    int subres = factorial1(n-1);
    return n * subres;
}
```

Tips

- Again, we have N active recursive calls, each call keeps in memory N values
- Total O(n^2) memory!

```
void f10(int n) {   // O(n^2) memory
    if(n <= 0)
        return;
    int *p = new int[n];    // O(n)
    f10(n-1);
    delete[] p;
}
```

Tips

- Before the call, p is created and removed
- Creation itself is O(n) any time
- But the N recursive active calls, each has **only O(1) memory**
- In the last recursive calls
    - N calls each with O(1) ⇒ O(n)
    - P creation ⇒ O(n)
    - 2n ⇒ O(n)

```
void f11(int n) {   // O(n) memory
    if(n <= 0)
        return;
    int *p = new int[n];    // O(n) memory
    delete[] p;
    f10(n-1);

}
```

So...

- As we have a few specific areas with memory creation, we only look to them
- Be careful from loops with function calls
- Recursive functions
    - What is the actual O() memory before the call
        - If constant, then N recursive calls need O(n)
        - If no, assume m, then N recursive calls need O(nm)

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."