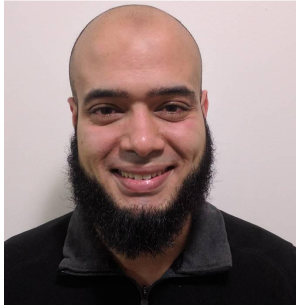
Data Structures

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 process** to time complexity
    - The order is about the worst case, an upper bound!
    - What is the **largest needed memory** at any point in 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 know this is O(nm) **time**
- But how much is it in memory?
- We have a few integers defined
- This means, regardless of N, the same amount of memory is used
- This is O(1) **memory**
- How many bytes in the data types?
    - We don’t care. Matter of small factors

```python
def f3(n, m):    # O(nm)
    cnt = 0
    for i in range(2 * n):
        for j in range(3 * m):
            cnt += 1
```

Tips

- Ignore variables creation that takes fixed memory
  - i, j, sum
- Ignore all operations that doesn't create memory
  - Sum +=, p[i]
- **We created a list of size n**
  - So O(n) memory here
- The nested loops are the largest for time = O(n^2) time

```python
def f_n2_n_a(n):                # Total O(n) memory, O(n^2) time
    p = [0] * n                 # O(n) time and memory

    for i in range(n):          # O(n) time
        p[i] = i

    sum = 0                     # O(n^2) time
    for i in range(n):
        for j in range(n):
            sum += p[i]

    return p
```

Tips

- Again ignore O(1) variables
- Now, even the list size is based on predefined constant
  - So ignore
- In total O(1) time/memory

```python
def f_const():                 # Total O(1) memory, O(1) time
    n = 10000                  # Constant
    p = [0] * n                # O(1) time and memory

    for i in range(n):         # O(1) time
        p[i] = i

    sum = 0                    # O(1) time
    for i in range(n):
        for j in range(n):
            sum += p[i]

    return p
```

Tips

- What is the largest memory **at any time**?
- Only O(n)
    - Whenever we create, it is deleted soon, not **accumulated**

```python
def f_n_n_a(n):
    # O(n) time and memory
    return list(range(n))


def f_n_n_b(n):
    # O(n) time and memory
    p1 = f_n_n_a(n)
    p2 = f_n_n_a(n)
    return p1, p2


def f_n2_n_b(n):
    # O(n^2) time but still O(n) memory
    for i in range(1, n):
        p = f_n_n_a(n)
        # memory released in the background
```

Tips

- Memory accumulation
- We create 1 + 2 + 3 …. N memory, each is O(N)
    - Which is O(n^2)

```python
def f_n_n_a(n):
    # O(n) time and memory
    return list(range(n))


def f_n2_n2(n):
    # O(n^2) time and O(n^2) memory
    ret = []
    for i in range(1, n):
        p = f_n_n_a(n)
        ret.append(p)    # accumulate memory
        # append itself is (1) amortized time
```

Tips

- Focus on the **largest** term

```
def f_n_n_c(n):    # # O(n) time and memory
    p1 = f_n_n_a(10 * n)
    p2 = f_n_n_a(100 * n)
    return p1, p2


def f_n_n2_c(n):    # # O(n) time and O(n^2) memory
    p1 = f_n_n_a(10000 * n)
    p2 = f_n_n_a(n * n)
    return p1, p2
```

Tips

- Time complexity can’t be lower than memory complexity
- To create K numbers, you need K steps
- Extend of K elements takes O(K) on average

```python
def f_nm_nm_a(n, m):  # O(nm) time and memory
    return [None] * (n+m)


def f_nm_nm_n(n, m):  # O(nm) time and memory
    lst = []
    for i in range(n):
        lst.extend(list(range(m)))
```

Tips

- I prefer to exclude parameters space
  - Some courses don't
  - Computed at caller

```python
def f_n_const(lst):         # O(n) time and O(1) memory
                            # don't compute memory of parameters
    sum = 0
    for i in lst:
        sum += i
    return sum


def f2(n):                  # O(n^2) time and O(n) memory
    p = f_n2_n_a(n)         # O(n^2) time and O(n) memory
    f_n_const(p)            # O(n) time and O(1) memory
```

Tips

- Clearly this is O(1) memory
- What if we wrote it recursively?
- Is it the same in memory? Think!

```python
def factorial1(n):
    # O(n) time and O(1) memory
    res = 1
    for i in range(1, n + 1):
        res *= i

    return res
```

Tips

- Recursion is a bit tricky.
- If we have N recursive calls, then the variables in each call remain 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*)

```python
def factorial2(n):
    # O(n) time and O(n) memory
    if n <= 1:
        return 1
    return n * factorial1(n - 1)
```

Tips

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

```
def frec1(n):              # O(n^2) memory
    if n <= 0:
        return
    p = [0] * n            # in the stack
    frec1(n-1)
```

Tips

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

```python
def frec2(n):                # O(n) memory
    if n <= 0:
        return
    p = [0] * n
    del p                    # release memory
    frec2(n-1)
```

```python
def frec3(n):                # O(n ^ 3) memory
    if n <= 0:
        return
    p = [0] * n * n * n
    del p                    # release memory
    frec3(n-1)
```

So...

- As we have a few specific areas with memory creation, we only look to them
    - Space complexity is easiest to calculate than time complexity as 99% of lines of the code will be just O(1).
- Be careful about 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 not, 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."