Worksheet: Complexity Analysis

①

```
def search(L, target):
    for x in L:
        if x == target: #line A
            return True
    return False
```

assume this is asked unless otherwise stated

Let **f(N)** be the number of times line A executes, with `N=len(L)`. What is **f(N)** in each case?

Worst Case (target is at end of list): f(N) = $N \in O(N)$

Best Case (target is at beginning of list): f(N) = $1 \in O(1)$

Average Case (target in middle of list): f(N) = $N/2 \in O(N)$

A **step** is any unit of work with bounded execution time (it doesn't keep getting slower with growing input size).

We classify algorithm complexity by classifying the **order of growth** of a function f(N), where f gives the number of steps the algorithm must perform for a given input size.

Big O definition: if **f(N) ≤ C * g(N)** for large **N** values and some fixed constant **C**, then **f(N) ∈ O(g(N))**

---

②

Let f(N) = $2N^2 + N + 12$

If we want to show **f(N) ∈ O(N³)**, what is a good lower bound on N? Let's have C=1.

$N \geq 4$

To show **f(N) ∈ O(N²)**, do we pick 1, 2, or [✓ 4] for the C? After picking C, what should we choose for N's lower bound?

$N \geq 3$

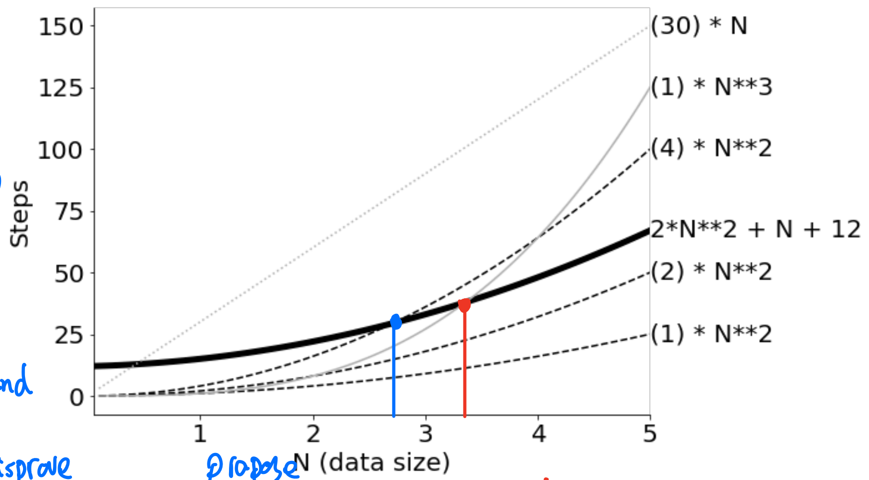
What is more informative to show?
f(N) ∈ O(N³) or $\boxed{f(N) \in O(N^2)}$? tighter upper bound

Somebody claims **f(N) ∈ O(N)**, offering C=30 and N>0. Suggest an N value to counter their claim.

disprove $2N^2 + N + 12 \leq \underset{\overset{\shortparallel}{30}}{C}N$

propose $N = 20$

$800 + 20 + 12 \not\leq 600$



---

③

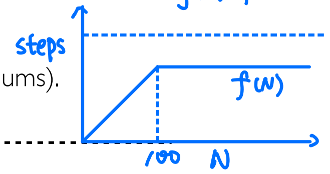
```
nums = [...]

first100sum = 0

for x in nums[:100]:
    first100sum += x
print(first100sum)
```

If we increase the size of nums from 20 items to 100 items, the code will probably take 5 times longer to run.

If we increase the size of nums from 100 to 1000, will the code take longer? Yes / [✓ No]

The complexity of the code is O(1), with N=len(nums).

C=110
g(N)=1



---

④

Each of the following list operations are either **O(1)** or **O(N)**, where N is len(L). Circle those you think are **O(N)**.

[✓ `L.insert(0, x)`] [✓ `L.pop(0)`] `x = L[0]` [✓ `x = max(L)`] `x = len(L)`

`L.append(x)` `L.pop(-1)` [✓ `L2.extend(L)`] [✓ `x = sum(L)`] [✓ `found = X in L`]

---

⑤

```
L = [...]
for x in L:
    avg = sum(L) / len(L)
    if x > 2*avg:
        print("outlier", x)
```

N+1

N

$(N+1) \cdot N$

What is the big O complexity?

$\rightarrow O(N^2)$

Is there a way to optimize the code?

⑥

```
A = [...]
B = [...]

for x in A:
    for y in B:
        print(x*y)
```

len(A) = M

len(B) = N

M+1

N+1

how would you define the variable(s) to describe the size of the input data?

The complexity of code is

$(M+1)(N+1) \Rightarrow$ O(MN)

---

⑦

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when **N** = 1? **N** = 2? **N** = 4? **N** = 8?

If **f(N)** is the number of times this step runs, then f(N) = ______________

The complexity of binary search is O(______________)

---

⑧

```
s1 = tuple("...") # could be any string
s2 = tuple("...")

# version A
import itertools

matches = False
for p in itertools.permutations(s1):
    if p == s2:
        matches = True
```

$len(s_1) = N$

$len(s_2) = N$

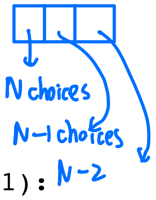
Ex. $S_1 = ("A", "B", "C")$

permutations of S_1

ABC BCA

ACB CAB

BAC CBA



N!

N

$N * (N-1) * (N-2) * \cdots * 2 * 1$

choices in total

```
# version B
s1 = sorted(s1)
s2 = sorted(s2)
matches = (s1 == s2)
```

N log N

N log N

N

assumed sorted is O(N log N) merge sort

quick sort

what is the complexity of version A? O($N * N!$)

what is the complexity of version B? O($N \log N$)

---

⑨

```
def selection_sort(L):
    for i in range(len(L)):
        idx_min = i
        for j in range(i, len(L)):
            if L[j] < L[idx_min]:
                idx_min = j
        # swap values at i and idx_min
        L[idx_min], L[i] = L[i], L[idx_min]

nums = [2, 4, 3, 1]
selection_sort(nums)
print(nums)
```

if this runs **f(N)** times, where **N=len(L)**,

then **f(N)** = $N + (N-1) + (N-2) + \cdots + 2 + 1 + 0$

The complexity of selection sort is

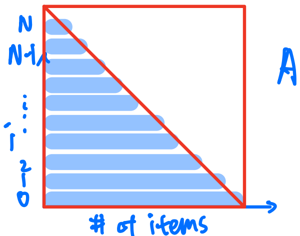
O(N^2) ← simplify $O(N^2/2)$

i	# of items for the inner for loop
0	N
1	N-1
2	N-2
⋮	⋮
N-1	1
N	0

Adding together

$N + (N-1) + (N-2) + \cdots + 1 + 0$

visualize it



Area = $\frac{N^2}{2}$

⑦

```
# assume L is already sorted, N=len(L)
def binary_search(L, target):
    left_idx = 0 # inclusive
    right_idx = len(L) # exclusive
    while right_idx - left_idx > 1:
        mid_idx = (right_idx + left_idx) // 2
        mid = L[mid_idx]
        if target >= mid:
            left_idx = mid_idx
        else:
            right_idx = mid_idx

    return right_idx > left_idx and L[left_idx] == target
```

how many times does this step run when N = 1? N = 2? N = 4? N = 8?

$\log_2 1 = 0$ $\log_2 2 = 1$ 2   3

If f(N) is the number of times this step runs, then f(N) = $\log_2 N$

The complexity of binary search is O($\log N$)