

Lecture 5 Worksheet: Complexity Analysis

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 $f(N) \leq C * g(N)$ for large N values and some fixed constant C , then $f(N) \in O(g(N))$

1

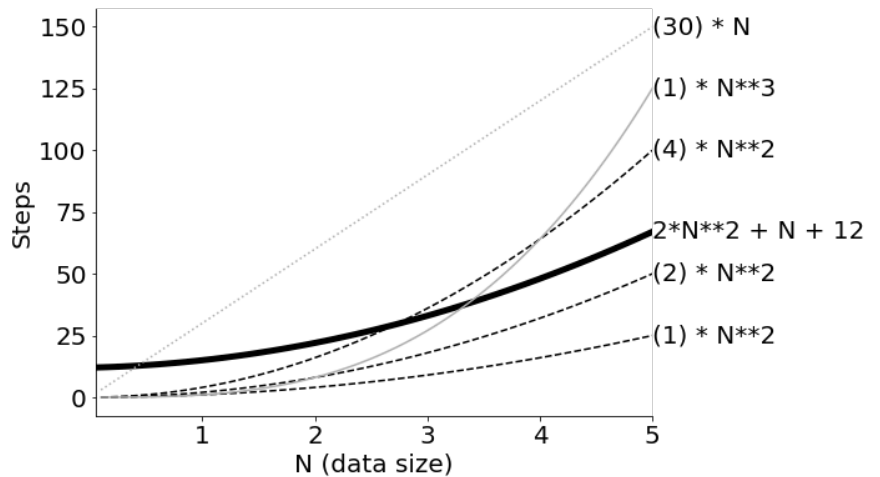
Let $f(N) = 2N^2 + N + 12$

If we want to show $f(N) \in O(N^3)$, what is a good lower bound on N ? Let's have $C=1$.

If we want to show $f(N) \in O(N^2)$, do we pick 1, 2, or 4 for the C ? After picking C , should we choose for N 's lower bound?

What is more informative to show?
 $f(N) \in O(N^3)$ or $f(N) \in O(N^2)$?

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



2

Each of the following list operations are either $O(1)$ or $O(N)$, where N is $\text{len}(L)$. 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`

3

```
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 $N=\text{len}(L)$. What is $f(N)$ in each case?

Worst Case (target is at end of list): $f(N) = \underline{\hspace{2cm}}$

Best Case (target is at beginning of list): $f(N) = \underline{\hspace{2cm}}$

Average Case (target in middle of list): $f(N) = \underline{\hspace{2cm}}$

4

```
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
        L[idx_min], L[i] = L[i], L[idx_min] # swap values at i and idx_min
```

if this runs $f(N)$ times, where $N=\text{len}(L)$, then $f(N) = \underline{\hspace{2cm}}$

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

The complexity of selection sort is
 $O(\underline{\hspace{2cm}})$

5

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 $f(N) = \underline{\hspace{2cm}}$

The complexity of binary search is
 $O(\underline{\hspace{2cm}})$

6

```
def merge(L1, L2):
    rv = []
    idx1 = 0
    idx2 = 0

    while True:
        done1 = idx1 == len(L1)
        done2 = idx2 == len(L2)

        if done1 and done2:
            return rv

        choose1 = False
        if done2:
            choose1 = True
        elif not done1 and L1[idx1] < L2[idx2]:
            choose1 = True

        if choose1:
            rv.append(L1[idx1])
            idx1 += 1
        else:
            rv.append(L2[idx2])
            idx2 += 1

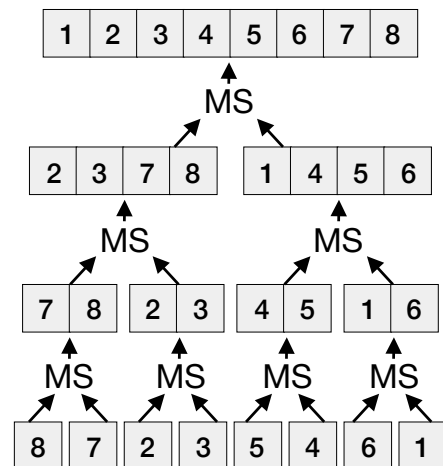
    return rv
```

`merge([1, 3], [2, 4])` will return .

`merge(L1, L2)` implements an $O(N)$ algorithm. But how
can we measure the size of the input? $N = \underline{\hspace{2cm}}$.

```
def merge_sort(L):
    if len(L) < 2:
        return L
    mid = len(L) // 2
    left = L[:mid]
    right = L[mid:]
    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)
```

`merge_sort([4, 1, 2, 3])`



If we double the list size, there will be
more level(s). Level count grows
 $O(\underline{\hspace{2cm}})$. Work per level is $O(\underline{\hspace{2cm}})$.
merge_sort complexity: $O(\underline{\hspace{2cm}})$

7

```
nums = [...]
```

```
first100sum = 0
```

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

If we increase the size of `nums` from 20 items to 100 items, the code
will probably take 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 $O(\underline{\hspace{2cm}})$, with $N=\text{len}(\text{nums})$.