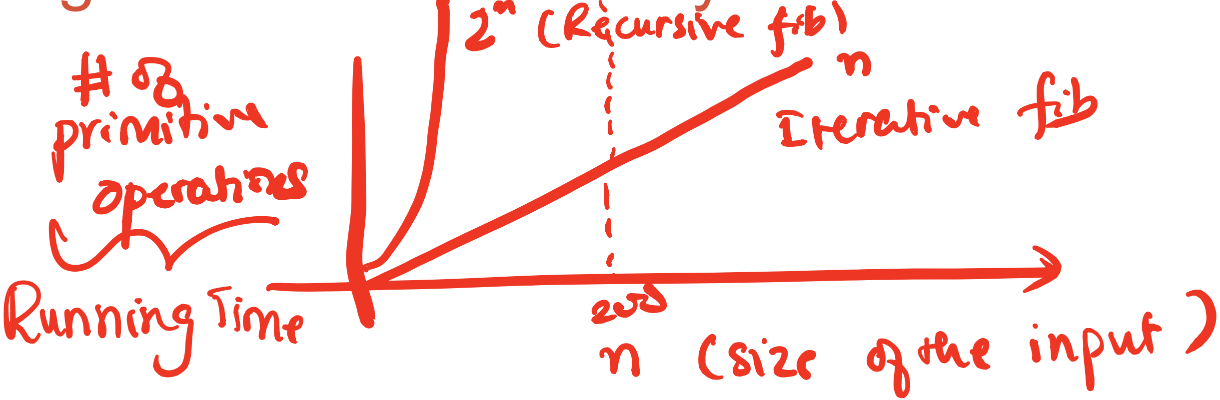
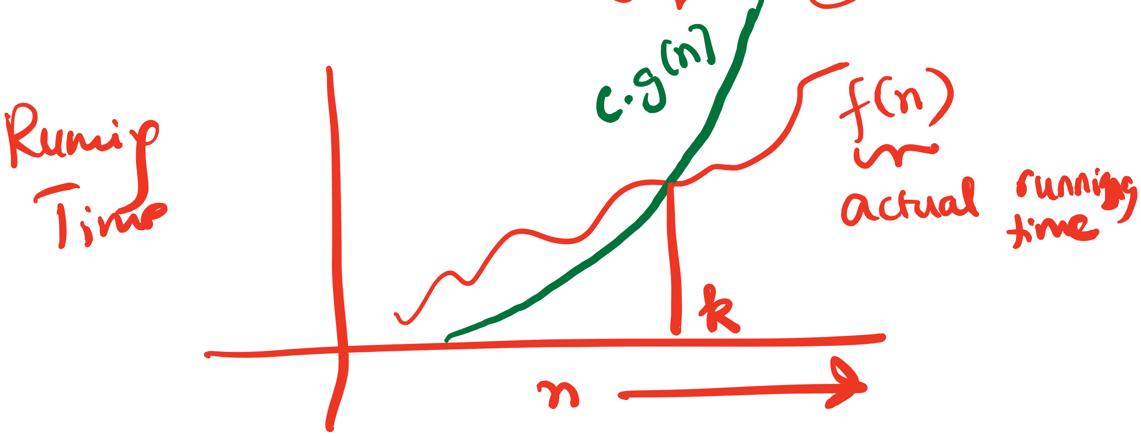
Big O: What does it really mean?



Big O is an upper limit on the running time of an algorithm as n gets large

$$f(n) = \underset{\uparrow}{5}\,\underline{N^2 \log N} + \cancel{N} + \cancel{1000}$$

$$= O\big(\underbrace{N^2 \log N}\big) \rightarrow g(n)$$



$$\underline{f(n)} = 5N^2 \log N + \underline{N} + \underline{1000}$$

$$< 5N^2 \log N + N^2 \log N + N^2 \log N$$

$$= \underset{\nwarrow c}{7}\,\underbrace{N^2 \log N}_{g(n)}$$

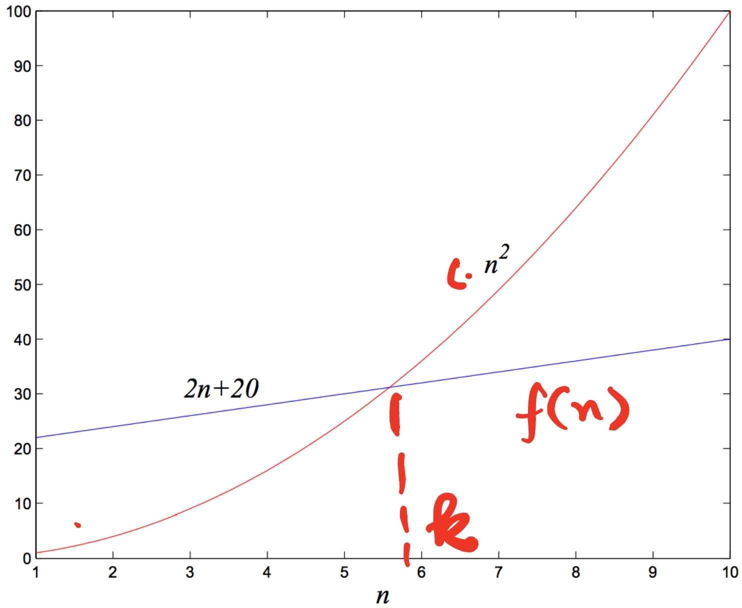
$$f(n) = O(N^2 \log N)$$

A more precise definition of Big-O

- f(n) and g(n): running times of two algorithms on inputs of size n.
- f(n) and g(n) map positive integer inputs to positive reals.

We say f = O(g) if there is a constant c > 0 and k>0 such that f(n) ≤ c · g(n) for all n >= k.

f = O(g)
means that "f grows no faster than g"



According to the definition on the previous slide if $f = O(g)$, $f = O(h)$ for any $h > g$

So if $f = O(N^2 \log N)$, then technically

$f = O(N^3)$

$f = O(N^4)$ .... and so on

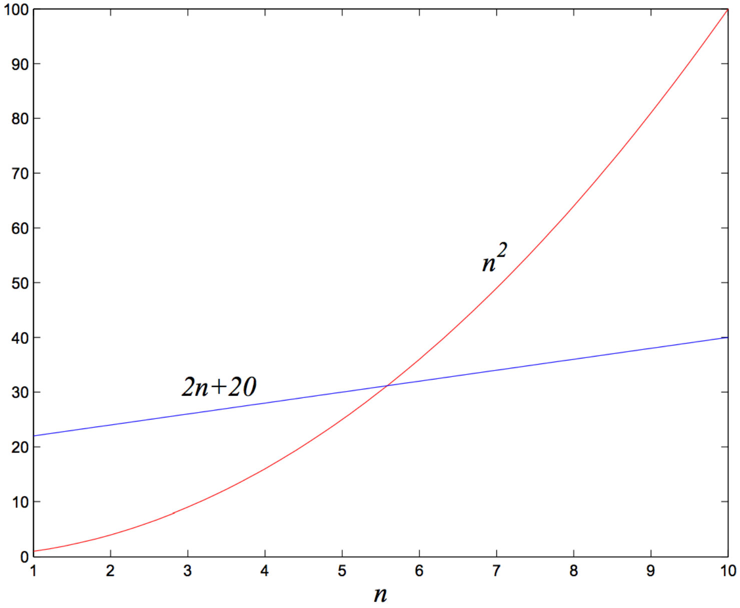
But in practice when doing Big-O analysis we look for the lowest order function that satisfies the definition of Big-Oh (the tightest upper bound to $f(n)$)

Big-Omega

- f(n) and g(n): running times of two algorithms on inputs of size n.
- f(n) and g(n) map positive integer inputs to positive reals.

We say f = Ω(g) if there are constants $c > 0$, $k>0$ such that $c \cdot g(n) \le f(n)$ for $n >= k$

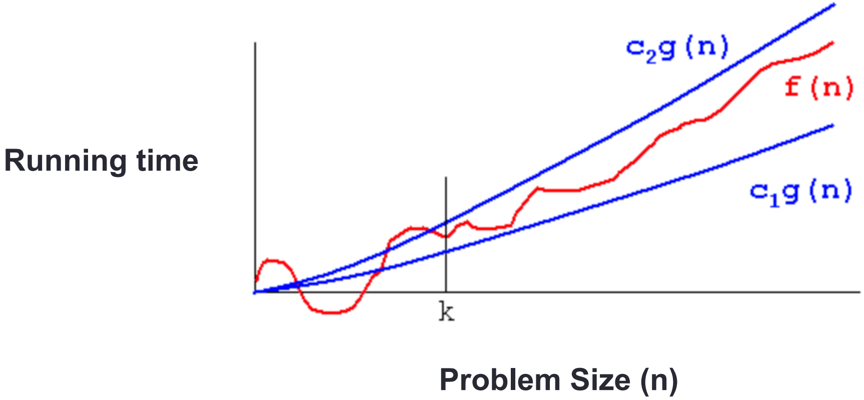
f = Ω(g)
means that "f grows at least as fast as g"



Big-Theta

- $f(n)$ and $g(n)$: running times of two algorithms on inputs of size n.
- $f(n)$ and $g(n)$ map positive integer inputs to positive reals.

We say $f = \Theta(g)$ if there are constants c_1, c_2, k such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$, for $n >= k$



What is the Big-O running time of algoX?

- Assume **dataA** is some data structure that supports the following operations with the given running times, where n is the number of keys stored in the data structure:
  - insert: O(log n)
  - min: O(1)
  - delete: O(log n)

n is the no. of keys in ds

input size

A. $O(N^2)$

[✓ B.] O(N logN)

C. O(N)

D. O(log N)

E. Not enough information to compute

```
void algoX(int arr[], int N)
{
        dataA ds;//ds contains no keys
        for(int i=0; i < N;   i++)
                ds.insert(arr[i]);
        for(int i=0; i < N;   i++){
                arr[i] = ds.min();
                ds.delete(arr[i]);
        }
}
```

```
[for(int i=0; i < N;    i++)
          ds.insert(arr[i]);
```

Running time of this loop is <u>less than</u>

$c_1 N \log N$

```
for(int i=0; i < N; i++){
         arr[i] = ds.min();
         ds.delete(arr[i]);
}
```

Running time of this loop is <u>less than</u>

$N(c_2 + c_3 \log N)$

Overall running time is

$c_1 N \log N + c_2 N + c_3 N \log N$

$= O(N \log N)$

<u>Reason</u>:

Each insert takes a different amount of time because the running time depends on the number of keys already in ds.

The first insert takes the least time, the last one takes the most.

Although we don't know the exact number of operations for each insert, we can find an upper limit.

Specifically, the running time of each insert is less than

c_1 ~~N~~ $\log N$