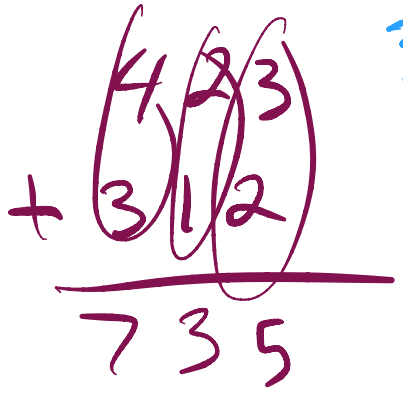
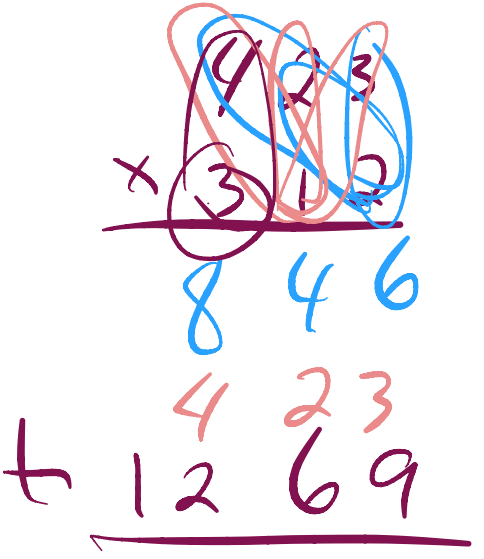
Warm-Up

If you had to add or multiply the following two numbers in your head as fast as possible, which would you rather do? Why?



3 steps



9 steps

5 steps

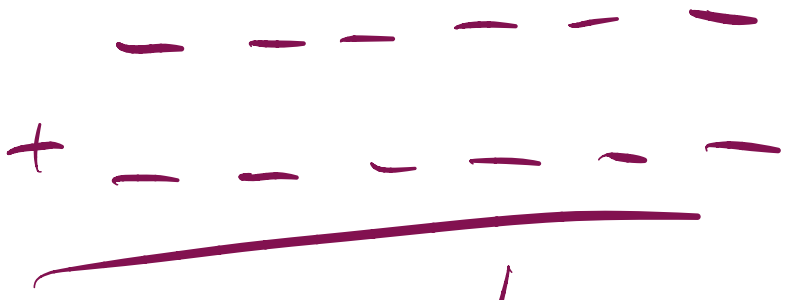
25 steps

5 6 4 2 3

× 7 2 3 1 2

+

n digits



n operations

n^2 operations

Efficiency

List Insertion

```
def double(s, v):
    """Insert a v after each v in list s.

    >>> s = [2, 7, 1, 8, 2, 8]
    >>> double(s, 2)
    >>> s
    [2, 2, 7, 1, 8, 2, 2, 8]
    """
    i = 0
    while i < len(s):
        if s[i] == v:
            s.insert(i+1, v)
            i += 2
        else:
            i += 1
```

Linked List Insertion

```
def double_link(s, v):
    """Insert a v after each v in linked list s.

    >>> t = Link(2, Link(7, Link(1,
                 Link(8, Link(2, Link(8))))))
    >>> double_link(t, 2)
    >>> print(t)
    (2 2 7 1 8 2 2 8)
    """
    while s is not Link.empty:
        for _ in range(10000): 1+1
        if s.first == v:
            s.rest = Link(v, s.rest)
            s = s.rest.rest
        else:
            s = s.rest
```

time → steps

n elements

loops n times

n time

n^2 steps

10000 n steps

loop n times

few steps

4 + 3 + 2 + 1

insert time

$n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \frac{n(n-1)}{2}$

n loops

$= \frac{n^2}{2} - \frac{n}{2}$

If you have a function that takes in a list of length n and takes f(n) steps to finish running, which f(n) is the fastest runtime and which is the slowest?

Rank the following f(n) from fastest to slowest runtime.

$f(n) = \frac{n(n-1)}{2}$ 5

$f(n) = (1.3)^n / 998799$ 6

$f(n) = 10000n$ 4

$f(n) = 1.2n$ 3

$f(n) = 98 * \log(n)$ 2

$f(n) = 34 * 10^{14}$ 1

Quantitative → Qualitative

Constant $O(1)$

logarithmic $O(\log(n))$

exponential gap

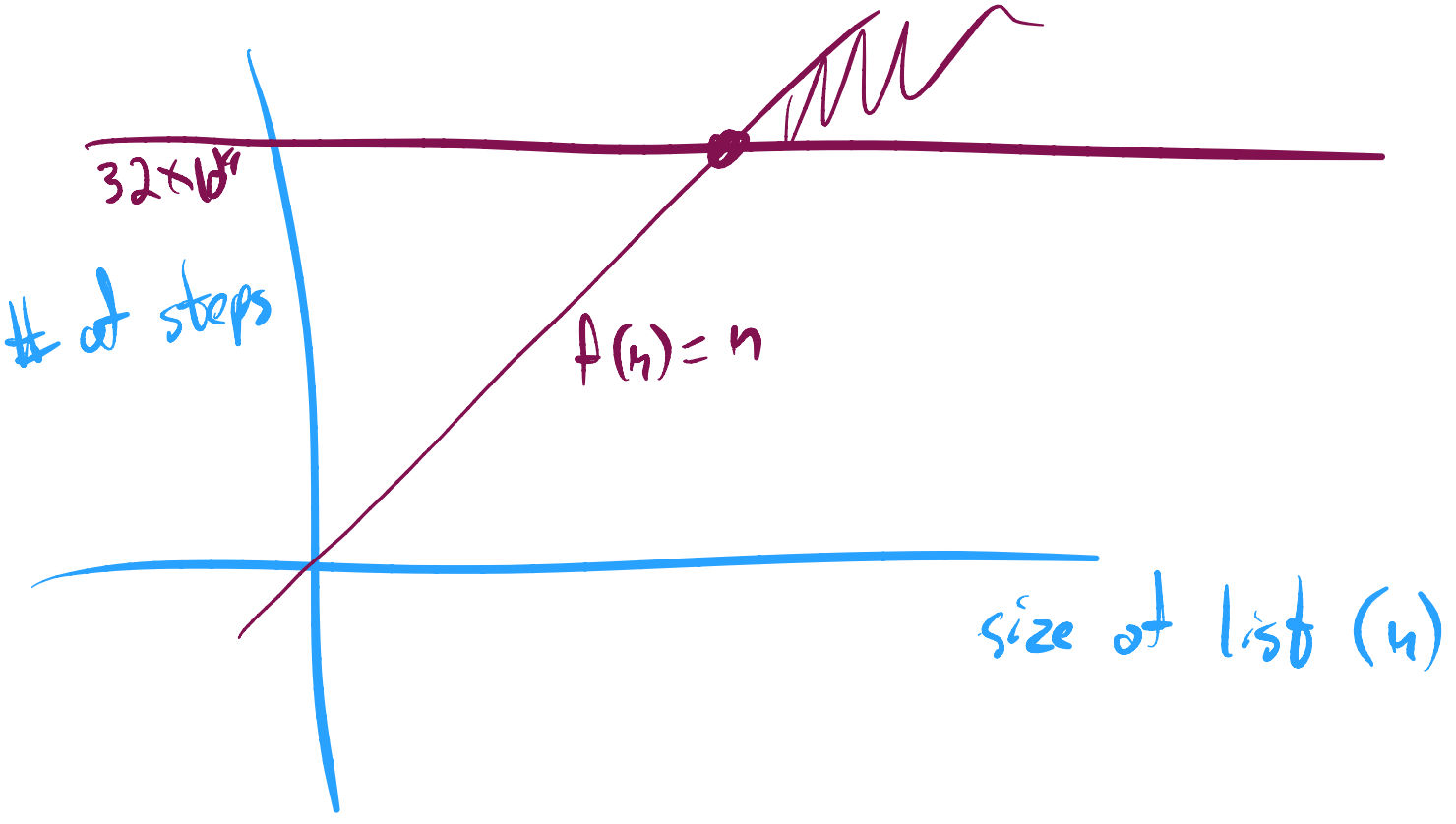
linear $O(n)$

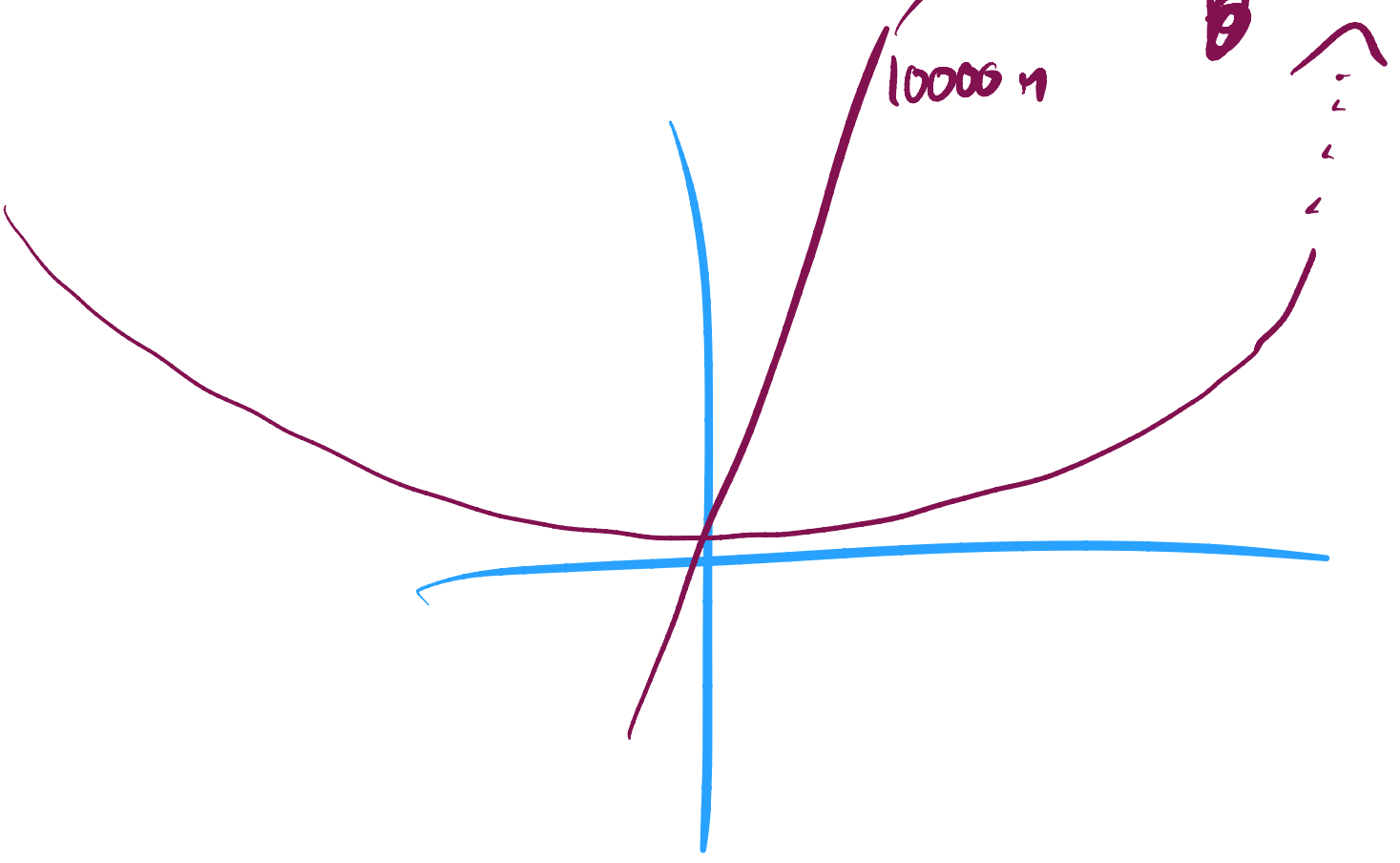
exponential gap

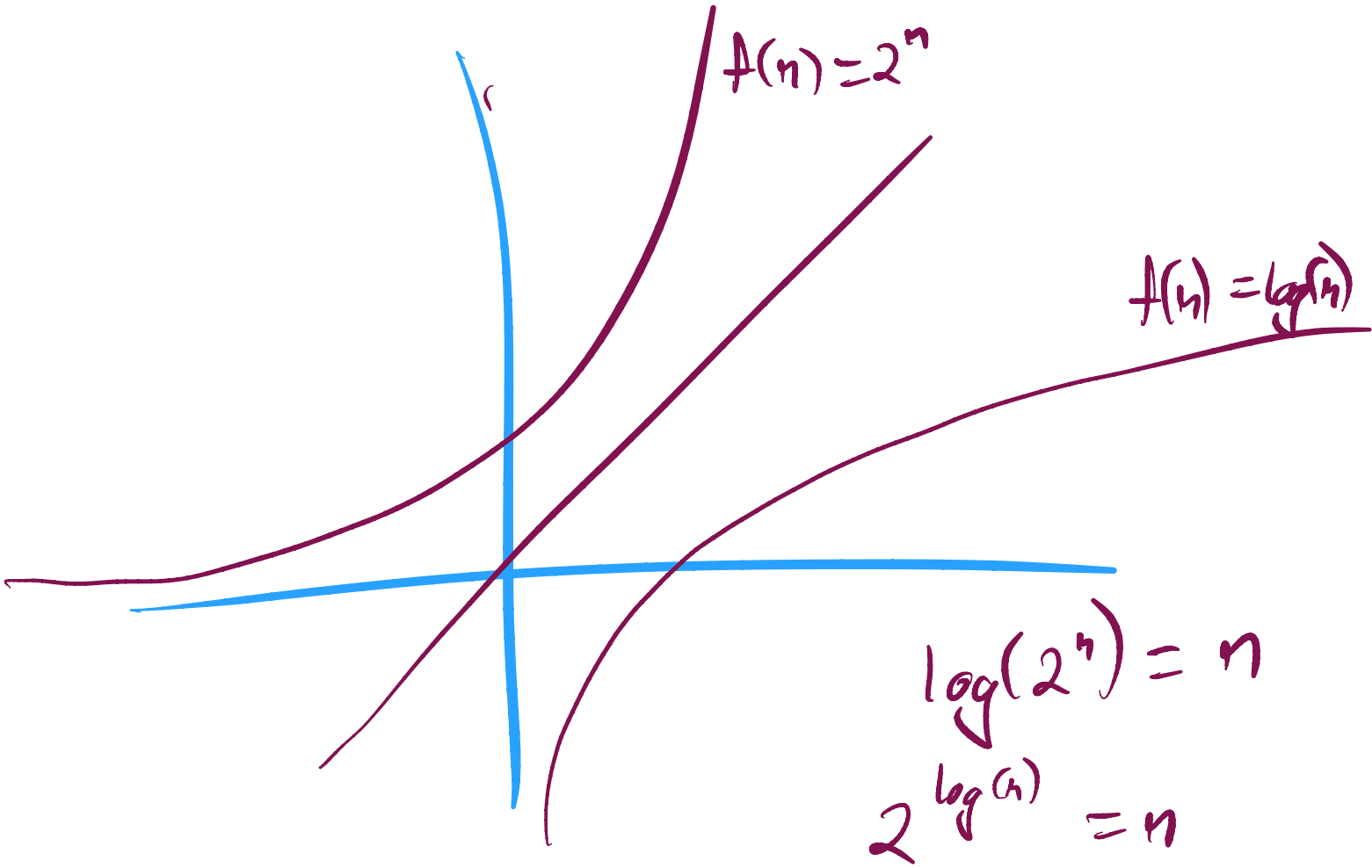
exponential $O(2^n)$

quadratic $O(n^2)$

cubic $O(n^3)$


32×16⁶
# of steps
f(n) = n
size of list (n)


10000 n



$$\log(2^n) = n$$

$$2^{\log(n)} = n$$

Common Orders of Growth

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

Common examples:

Tree recursion
(without memoization)

Quadratic growth.

Incrementing *n* increases *time* by *n* times a constant

Two nested loops:
For each element in a list,
process a whole list

Linear growth.

Incrementing *n* increases *time* by a constant

For each element in a list,
do a fixed amount of work

Logarithmic growth.

Doubling *n* only increments *time* by a constant

Each step shrinks the problem
by half (or some fraction)

Constant growth. Increasing *n* doesn't affect time

Just process the first few
elements of a list

Linear or Quadratic

```
def g(lst):
    for num1 in lst:
        for num2 in lst:
            print(num1 + num2)
```

n loops

n^2

```
def g(lst):
    sum = 0
    for num1 in lst:
        sum += num1
    for num2 in lst:
        sum -= num1
    return sum
```

n time + n time

$2n = O(n)$

```
def g(lst):
    for bonus in range(10):
        for i in range(len(lst)):
            list[i] += bonus
```

10 times

n time

$10n = O(n)$

```
def g(lst):
    for num1 in lst:
        insert(0, num1 - 1)
```

n loops

n time

n^2

```
def g(lst):
    max_num = lst[0]
    for num in lst:
        max_num = max(max_num, num)
    return max_num
```

n times

$O(n)$

$O(1)$

```
def g(lst):
    if len(lst) == 1:
        return lst[0]
    return max(lst[0], g(lst[1:]))
```

$O(n)$ time & space