CS 344

Design and Analysis of Computer Algorithms

Outline for Today

- Algorithmic Analysis
  - Analyzing runtime of algorithms
  - Proving runtime with asymptotic analysis
  - Big-O notation
- Divide and Conquer I
  - Integer Multiplication

Algorithmic Analysis
and
big-O notation

Example 1: Find the Number!

Find the Number

Input: Given an array of numbers **`A[0:n-1]`**, sorted in ascending order:

n numbers in total, i.e., length(A)==n

1	3	4	5	7	8	9	11	14	16	23	28	31	34	37	41	50

Problem: Given a number **x**, locate the number in the array

Our algorithm: Sequential Search

We call this “Pseudo-code”

```
algorithm sequential_search(A, x):
  for i = 0 to length(A)-1:
    if A[i] == x:
          return i;
  return -1;
```

Output: if output is `i>=0`, we know number **x** exists in the array, and its position is **`A[i]`**
if output is `i=-1`, we know number **x** does not exist in the array.

Find the Number

Question: How many **basic operations** the algorithm needs to do in the **worst case**?

n numbers in total, i.e., length(A)==n

1	3	4	5	7	8	9	11	14	16	23	28	31	34	37	41	50

```
algorithm sequential_search(A, x):
  for i = 0 to length(A)-1:
    if A[i] == x:  //one basic operation
          return i;
  return -1;
```

What is **Basic Operation?** : In this case: compare **x** with a number in **A** to see if **`A[i]==x`**

What is **Worst Case?** : In this case: when **x==A[n-1]** or when **x** does not exist in **A**

How many basic operations in the worst case? The answer is **n**

Later, we are going to say the **computational complexity** of the algorithm is **O(n)**

Find the Number

More about Basic Operations:

In Assembly Language, Basic operations are those that can be finished within a constant steps of CPU register operations (usually one step)

Some examples of basic operations:
Basic Math Operation: +, -, ×, /
Comparator: ==, >, <, >=, <=
Value Assignment: =
Digital Shift: >>, <<

Some examples that are not basic operations:
Logarithm: log(x)
Modulo: x mod y
Multiplying long integers

Can we do better?

Question: Can we do **fewer** **basic operations** in the **worst case**?

n numbers in total, i.e., length(A)==n

1	3	4	5	7	8	9	11	14	16	23	28	31	34	37	41	50

```
algorithm binary_search(A, x):
  set L = 0, R = n-1;
  while L <= R:
      set i = L + ⌊(R-L)/2⌋;
      if A[i] == x:  //one basic operation
          return i;
      else if A[i] < x: //one basic operation
          set L = i + 1; //one basic operation
      else if A[i] > x: //one basic operation
          set R = i - 1; //one basic operation
  return -1;
```

Can we do better?

Question: Can we do **fewer basic operations** in the **worst case**?

n numbers in total, i.e., length(A)==n

1	3	4	5	7	8	9	11	14	16	23	28	31	34	37	41	50

```
algorithm binary_search(A, x):
  set L = 0, R = n-1;
  while L <= R:
        set i = L + ⌊(R-L)/2⌋;
      if A[i] == x:    //one basic operation
              return i;
        else if A[i] < x:
              set L = i + 1;
        else if A[i] > x:
              set R = i - 1;
  return -1;
```

What is **Basic Operation?** : In this case: compare **x** with a number in **A** to see if **A[i]==x**

What is **Worst Case?** : In this case: when **x==A[n-1]** or when **x** does not exist in **A**

How many basic operations in the worst case? The answer is **log(n)**

Later, we are going to say the **computational complexity** of the algorithm is **O(log(n))**

What do we learn?

Problem: Given a sorted array **A**, located the given number **x** in the array.

n numbers in total, i.e., length(A)==n

$$\overbrace{\boxed{1}\boxed{3}\boxed{4}\boxed{5}\boxed{7}\boxed{8}\boxed{9}\boxed{11}\boxed{14}\boxed{16}\boxed{23}\boxed{28}\boxed{31}\boxed{34}\boxed{37}\boxed{41}\boxed{50}}^{n}$$

```
algorithm sequential_search(A, x):
  for i = 0 to length(A)-1:
    if A[i] == x:
        return i;
  return -1;
```

Time complexity is **O(n)**

```
algorithm binary_search(A, x):
  set L = 0, R = n-1;
  while L <= R:
        set i = L + ⌊(R-L)/2⌋;
      if A[i] == x:
            return i;
        else if A[i] < x:
            set L = i + 1;
        else if A[i] > x:
            set R = i - 1;
  return -1;
```

Time complexity is **O(log(n))**

1. Even for the same problem, there could exist different algorithms to solve the problem.
2. Some algorithms are faster, some are slower.
3. What do we mean by faster (or slower)?: Fewer (or more) basic operations in the worst case.
4. We say faster algorithms are more efficient, slower algorithms are less efficient.
5. We use time complexity to measure the efficiency of an algorithm.

Example 2: Integer Multiplication!

What is the best way to multiply two numbers?

Multiplication: The Problem

Input: 2 non-negative numbers, x and y (n digits each)

Output: the product $x \cdot y$

$$
\begin{array}{r}
5678 \\
\times\ 1234 \\
\hline
\mathbf{7006652}
\end{array}
$$

Grade-School Multiplication

Algorithm description (informal):
compute partial products (using multiplication & "carries" for digit overflows), and add all (properly shifted) partial products together

```
   45
x  63
-----
  135
 2700
-----
 2835
```

Grade-School Multiplication

```
  45123456678093420581217332421
x 63782384198347750652091236423
-------------------------------
                             ):
```

The long number cannot be stored using one integer, it is stored as an array.

$2^{32} - 1 = 4294967295$, $2^{64} - 1 = 18446744073709551615$

Grade-School Multiplication

$$\overbrace{\begin{array}{r} 451234566780934205812173324 21 \\ \times\ 63782384198347750652091236423 \end{array}}^{\textbf{\textit{n}}\ \textit{digits}}$$

):

How efficient is this algorithm?

(How many single-digit operations are required?)

Grade-School Multiplication

$$\overbrace{\texttt{45123456678093420581217332421}}^{\textbf{\textit{n}}\ \textit{digits}}$$
$$\underline{\texttt{x 63782384198347750652091236423}}$$
$$\texttt{):}$$

How efficient is this algorithm?

(How many single-digit operations *in the worst case*?)

n partial products: ~2n² ops (at most n multiplications & n additions per partial product)

adding n partial products: ~2n² ops (a bunch of additions & "carries")

Grade-School Multiplication

$$\overbrace{\begin{array}{r} 4512345667809342058121733242 1 \\ \times\ 6378238419834775065209123642 3 \\ \hline): \end{array}}^{n \textit{ digits}}$$

How efficient is this algorithm?
(How many single-digit operations *in the worst case*?)

n partial products: ~$2n^2$ ops (at most n multiplications & n additions per partial product)

adding n partial products: ~$2n^2$ ops (a bunch of additions & "carries")

~ $4n^2$ operations in the worst case

Can we do better?

What does "Better" mean?

Is **1000000n** operations better than $4n^2$?

Is **0.000001n^3** operations better than $4n^2$?

Is **2n^2** operations better than $4n^2$?

- **The answers for the first two depend on what value n is…**
  - $1000000n < 4n^2$ only when n exceeds a certain value (in this case, 250000)
- **These constant multipliers are too environment-dependent...**
  - An operation could be faster/slower depending on the machine, so $2n^2$ ops on a slow machine might not be "better" than $4n^2$ ops on a faster machine

What does "Better" mean?

INTRODUCING...

ASYMPTOTIC ANALYSIS

If you still remember our **Find the Number** example:

```
algorithm sequential_search(A, x):
  for i = 0 to length(A)-1:
    if A[i] == x:
        return i;
  return -1;
```

Time complexity is **O(n)**

Slower

```
algorithm binary_search(A, x):
  set L = 0, R = n-1;
  while L <= R:
       set i = L + ⌊(R-L)/2⌋;
       if A[i] == x:
           return i;
       else if A[i] < x:
           set L = i + 1;
       else if A[i] > x:
           set R = i - 1;
  return -1;
```

Time complexity is **O(log(n))**

Faster

What does “Better” mean?

ASYMPTOTIC ANALYSIS

- **The Key Idea:** we care about how the number of operations *scales* with the size of the input (i.e. the algorithm’s *rate of growth*).

- We want some measure of algorithm efficiency that describes the nature of the algorithm, regardless of the environment that runs the algorithm, such as hardware, programming language, memory layout, etc.

Asymptotic Analysis

We'll express the asymptotic runtime of an algorithm using

BIG-O NOTATION

- We would say
  - The Sequential Search algorithm **"runs in time O(n)"**
  - The Binary Search algorithm **"runs in time O(log(n))"**
  - The Grade-school Multiplication algorithm **"runs in time $O(n^2)$"**
    - Informally, this means that the runtime of the algorithm "scales like" n^2
- We'll introduce more formal definitions of Big-O at the end of the lecture

The key point of Asymptotic Big-O notation is:

Ignore **constant factors** and **lower-order terms**

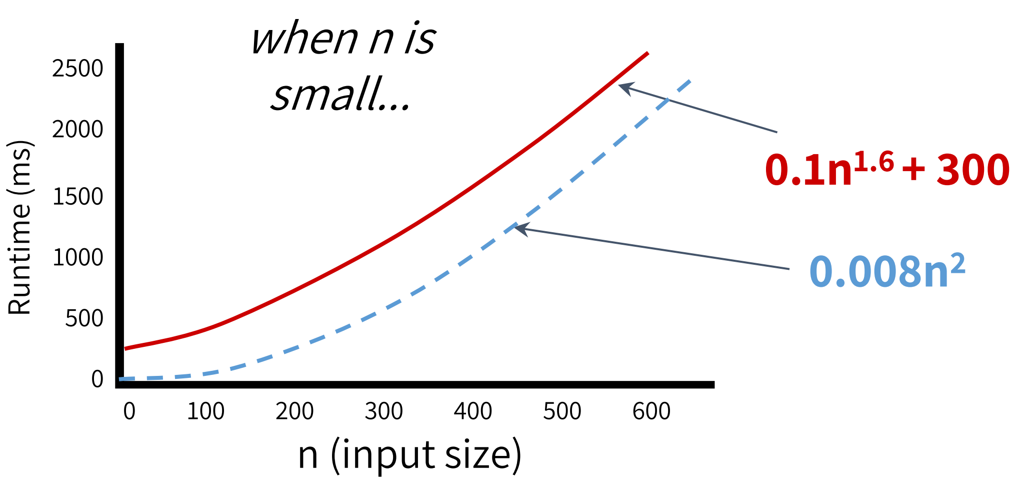
Too system dependent — Irrelevant when input size **n** is large

Asymptotic Analysis

The key point of Asymptotic Big-O notation is:

Ignore **constant factors** and **lower-order terms**

Too system dependent — Irrelevant when input size **n** is large

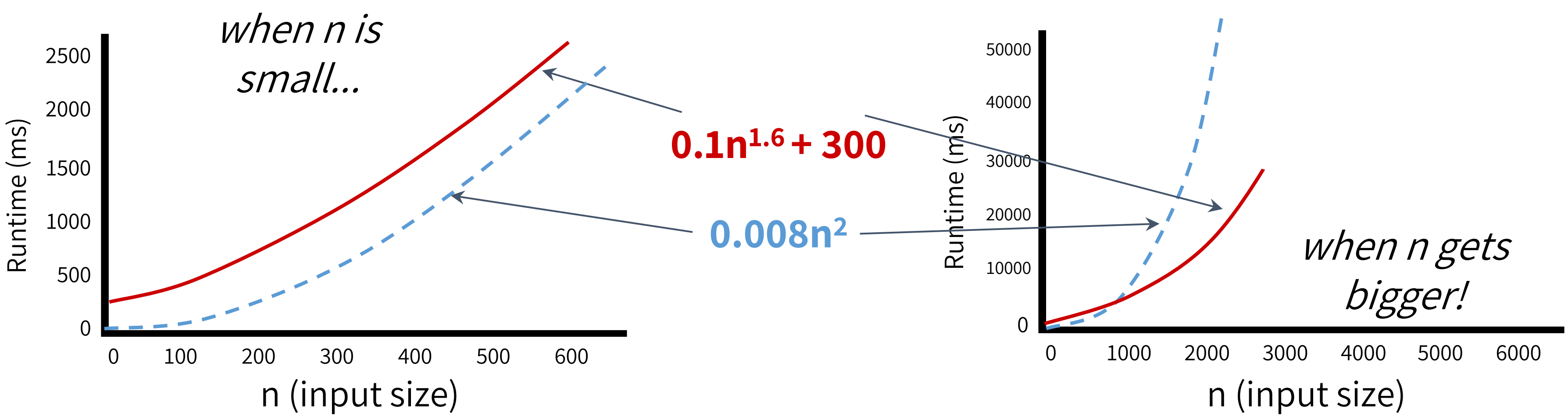


Asymptotic Analysis

The key point of Asymptotic Big-O notation is:

Ignore **constant factors** and **lower-order terms**

Too system dependent — Irrelevant when input size **n** is large

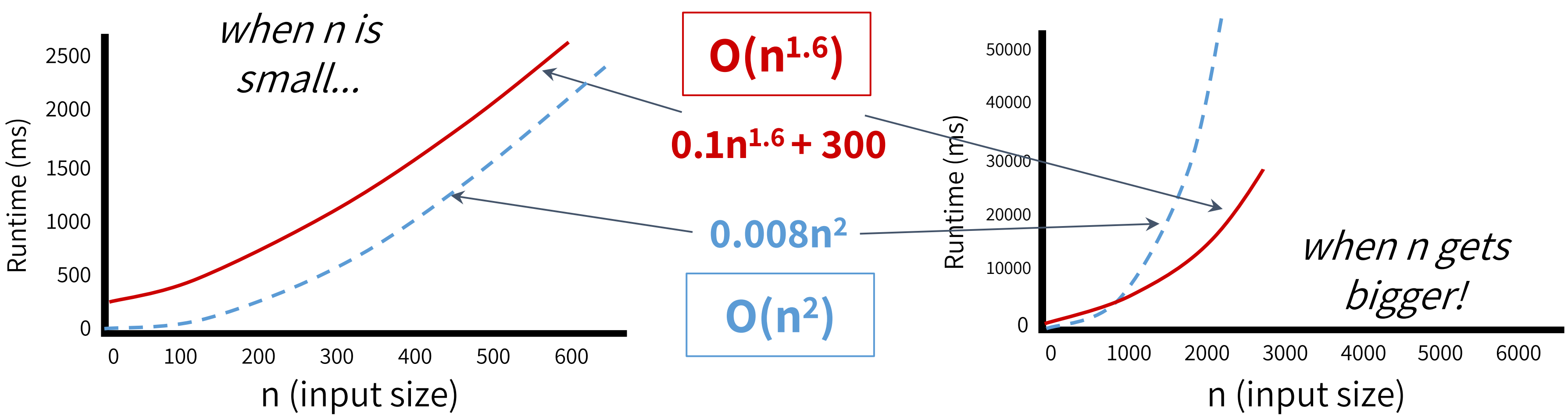


Asymptotic Analysis

The key point of Asymptotic Big-O notation is:

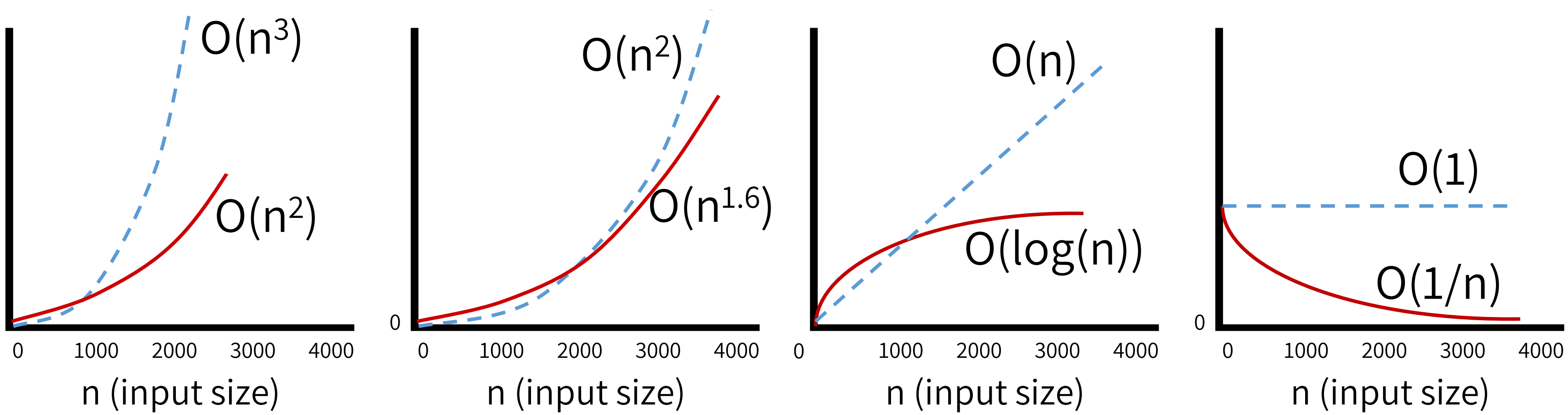
Ignore **constant factors** and **lower-order terms**

Too system dependent — Irrelevant when input size **n** is large



Asymptotic Analysis

- To compare algorithm runtimes, we compare their Big-O runtimes
  - Eg: a runtime of $O(n^2)$ is considered "better" than a runtime of $O(n^3)$
  - Eg: a runtime of $O(n^{1.6})$ is considered "better" than a runtime of $O(n^2)$
  - Eg: a runtime of $O(\log(n))$ is considered "better" than a runtime of $O(n)$
  - Eg: a runtime of $O(1/n)$ is considered "better" than $O(1)$



In all of the above figures, red lines are "**better**" than blue lines
Because red is **eventually** smaller than blue, i.e., **when n is sufficiently large.**
(This is what we mean by "**asymptotic**")

Back to Integer Multiplication

- We would say
  - The Sequential Search algorithm **"runs in time O(n)"**
  - The Binary Search algorithm **"runs in time O(log(n))"**
  - The Grade-school Multiplication algorithm **"runs in time $O(n^2)$"**
    - Informally, this means that the runtime of the algorithm "scales like" n^2

Can we multiply n-digit integers faster than $O(n^2)$?

5-Minute Break

Divide and Conquer

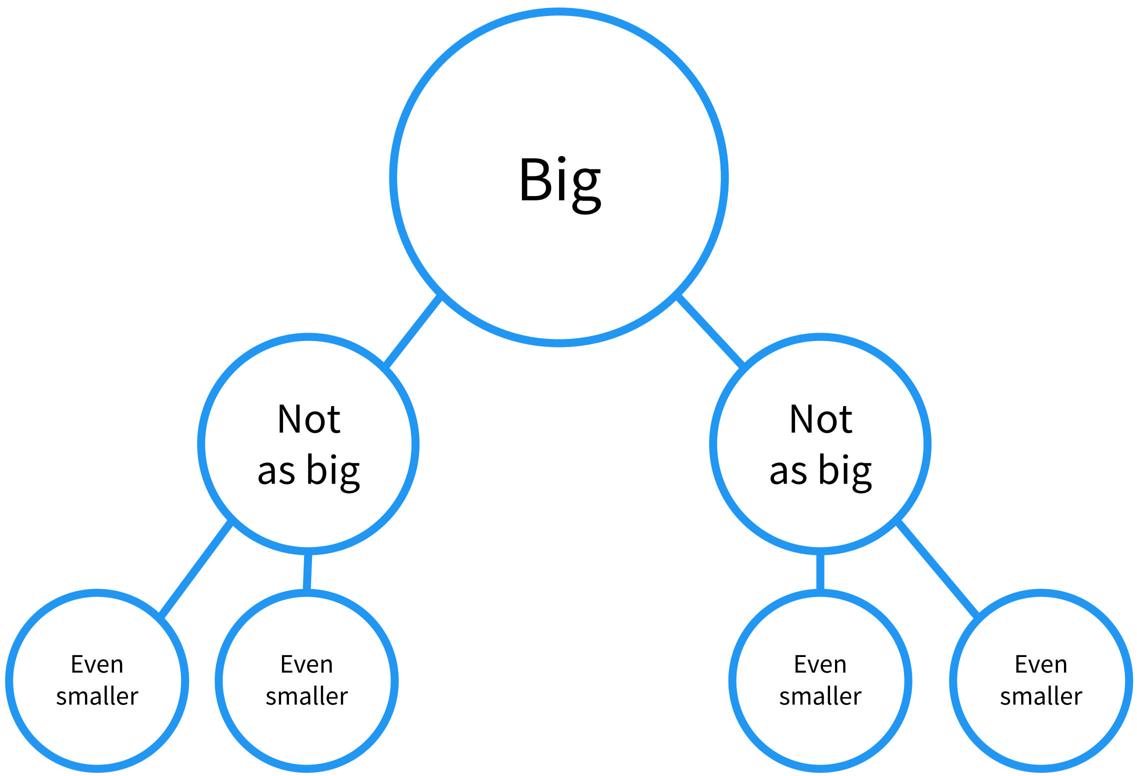
Our first paradigm for algorithm design

Divide and Conquer

An algorithm design paradigm

Divide: break current problem into smaller sub-problems.

Conquer: solve the smaller sub-problems recursively and collect the results to solve the current problem.



Multiplication Sub-Problems

- **Original large problem:** multiply two n-digit numbers
- **What are the subproblems?**

Multiplication Sub-Problems

- **Original large problem:** multiply two n-digit numbers
- **What are the subproblems?**

$$1234 \times 5678$$

$$= (12 \times 100 + 34) \times (56 \times 100 + 78)$$

$$= (12 \times 56)100^2 + (12 \times 78 + 34 \times 56)100 + (34 \times 78)$$

Multiplication Sub-Problems

- **Original large problem:** multiply two n-digit numbers
- **What are the subproblems?**

$$1234 \times 5678$$

$$= (12 \times 100 + 34) \times (56 \times 100 + 78)$$

$$= (\underbrace{12 \times 56}_{1})100^2 + (\underbrace{12 \times 78}_{2} + \underbrace{34 \times 56}_{3})100 + (\underbrace{34 \times 78}_{4})$$

One 4-digit problem → *Four 2-digit sub-problems*

Multiplication Sub-Problems

- **Original large problem:** multiply two n-digit numbers
- **What are the subproblems?** More generally:

$$[x_1 x_2 \ldots x_{n-1} x_n] \times [y_1 y_2 \ldots y_{n-1} y_n]$$

$$= (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$

$$= (a \times c)10^n + (a \times d + b \times c)10^{n/2} + (b \times d)$$

1 2 3 4

One n-digit problem → *Four n/2-digit sub-problems*

Pseudo-Code

```
algorithm multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac = multiply(a, c, n/2)
  set ad = multiply(a, d, n/2)
  set bc = multiply(b, c, n/2)
  set bd = multiply(b, d, n/2)

  return ac·10^n + (ad+bc)·10^(n/2) + bd
```

Pseudo-Code

```
algorithm multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac = multiply(a, c, n/2)
  set ad = multiply(a, d, n/2)
  set bc = multiply(b, c, n/2)
  set bd = multiply(b, d, n/2)

  return ac·10^n + (ad+bc)·10^(n/2) + bd
```

x, y are n-digit numbers

Note: we are making an assumption that n is a power of 2 just to make the pseudocode simpler

Pseudo-Code

```
algorithm multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac = multiply(a, c, n/2)
  set ad = multiply(a, d, n/2)
  set bc = multiply(b, c, n/2)
  set bd = multiply(b, d, n/2)

  return ac·10^n + (ad+bc)·10^(n/2) + bd
```

x, y are n-digit numbers

Base case: when x and y are 1-digit, we can directly return their product, e.g., by referencing the multiplication table

Pseudo-Code

```
algorithm multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac = multiply(a, c, n/2)
  set ad = multiply(a, d, n/2)
  set bc = multiply(b, c, n/2)
  set bd = multiply(b, d, n/2)

  return ac·10^n + (ad+bc)·10^(n/2) + bd
```

x, y are n-digit numbers

Base case: when x and y are 1-digit, we can directly return their product, e.g., by referencing the multiplication table

a, b, c, d are n/2-digit numbers

Pseudo-Code

```
algorithm multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac = multiply(a, c, n/2)
  set ad = multiply(a, d, n/2)
  set bc = multiply(b, c, n/2)
  set bd = multiply(b, d, n/2)

  return ac·10^n + (ad+bc)·10^(n/2) + bd
```

x, y are n-digit numbers

Base case: when x and y are 1-digit, we can directly return their product, e.g., by referencing the multiplication table

a, b, c, d are n/2-digit numbers

Call the algorithm recursively to get answers of the sub-problems

Pseudo-Code

```
algorithm multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac = multiply(a, c, n/2)
  set ad = multiply(a, d, n/2)
  set bc = multiply(b, c, n/2)
  set bd = multiply(b, d, n/2)

  return ac·10^n + (ad+bc)·10^(n/2) + bd
```

x, y are n-digit numbers

Base case: when x and y are 1-digit, we can directly return their product, e.g., by referencing the multiplication table

a, b, c, d are n/2-digit numbers

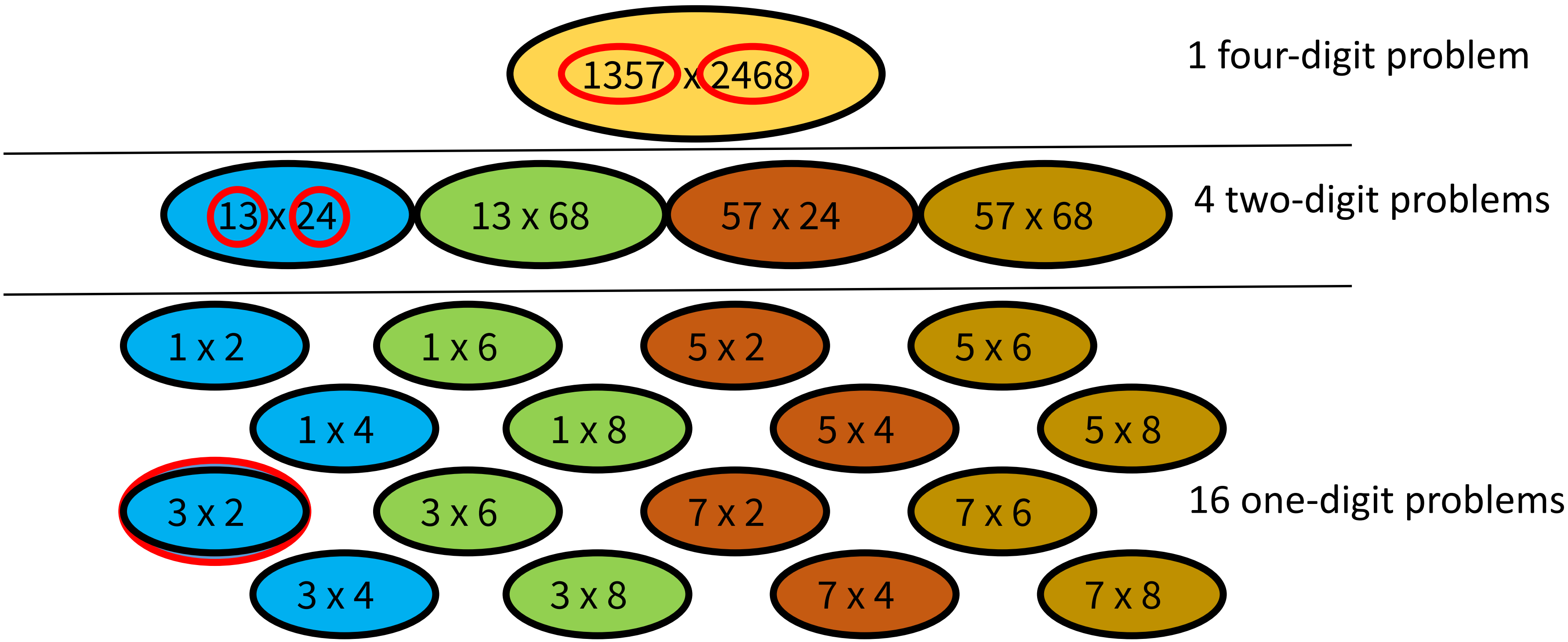
Call the algorithm recursively to get answers of the sub-problems

Add-up to get final answer

How Efficient is the Algorithm?

Let's start with a small case: If we're multiplying two 4-digit numbers, how many 1-digit multiplications does the algorithm perform?

- In other words, how many times do we reach the base case where we actually perform a "multiplication" (a.k.a. a table lookup)?

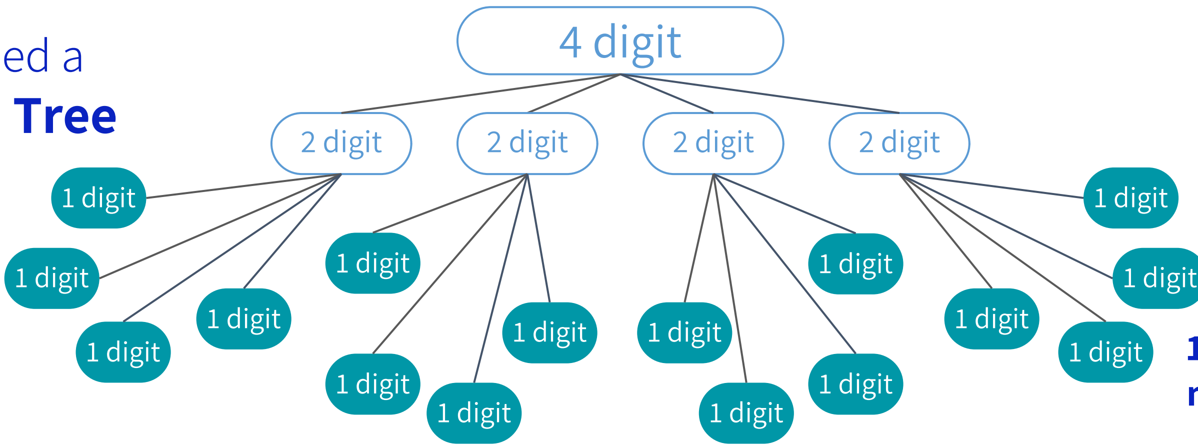


Recursion Tree Method

Let's start with a small case: If we're multiplying two 4-digit numbers, how many 1-digit multiplications does the algorithm perform?

- In other words, how many times do we reach the base case where we actually perform a "multiplication" (a.k.a. a table lookup)?

This is called a **Recursion Tree**

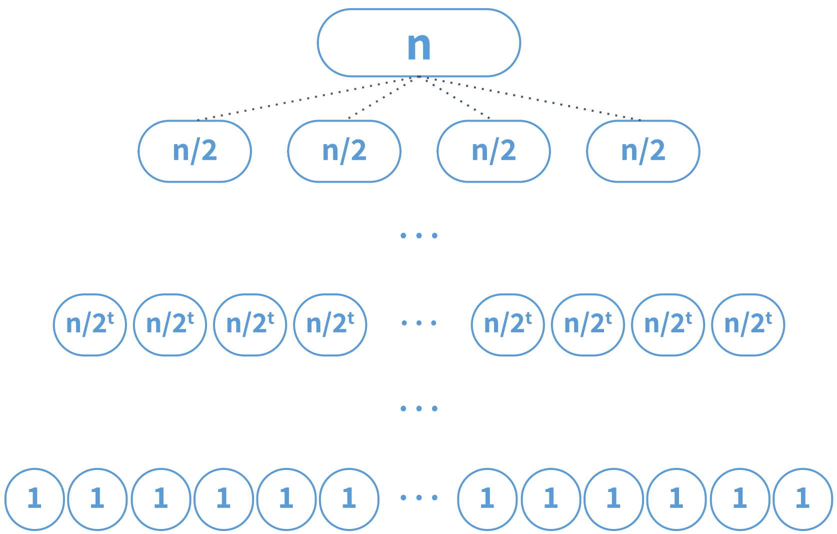


16 one-digit multiplications!

Recursion Tree Method

Now let's generalize to general cases: If we're multiplying two n-digit numbers, how many 1-digit multiplications does the algorithm perform?

Recursion Tree



Level 0: 1 problem of size n

Level 1: 4^1 problems of size n/2

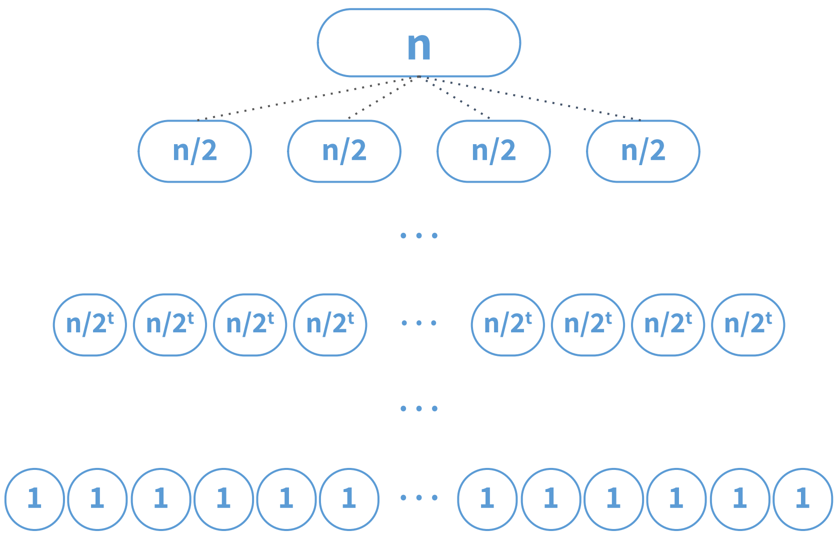
Level t: 4^t problems of size $n/2^t$

Level $\log_2 n$: ____ problems of size 1

Recursion Tree Method

Now let's generalize to general cases: If we're multiplying two n-digit numbers, how many 1-digit multiplications does the algorithm perform?

Recursion Tree



Level 0: 1 problem of size n

Level 1: 4^1 problems of size n/2

Level t: 4^t problems of size $n/2^t$

Level $\log_2 n$: $\underline{\;n^2\;}$ problems of size 1

- Why $\log_2 n$ levels? Because if $n/2^t=1$, we have $t=\log_2 n$
  - i.e., you need to cut n in half $\log_2 n$ times to get to size 1
- Why n^2 problems in the last level $\log_2 n$? Because $4^{\log_2 n} = n^{\log_2 4} = \mathbf{n^2}$

How Efficient is the Algorithm?

The running time of this Divide-and-Conquer multiplication algorithm is **at least $O(n^2)$**!

We know there are already n^2 multiplications happening at the bottom level of the recursion tree, so that's why we say "at least" $O(n^2)$

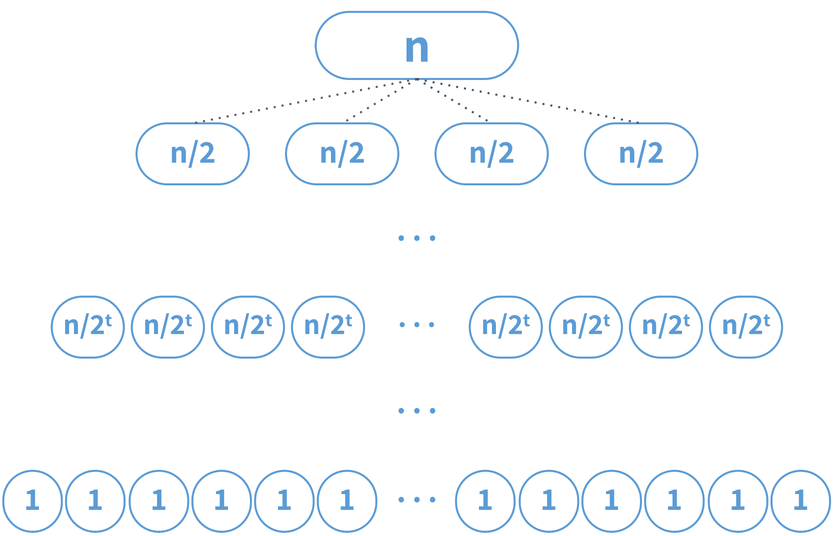
Level $\log_2 n$: $\underline{n^2}$ problems of size 1

How Efficient is the Algorithm?

The running time of this Divide-and-Conquer multiplication algorithm is **at least $O(n^2)$**!

We know there are already n^2 multiplications happening at the bottom level of the recursion tree, so that's why we say "at least" $O(n^2)$

More concretely, we add up the total computation in all levels



Level 0: 1 problem of size n

Level 1: 4^1 problems of size n/2

Level t: 4^t problems of size $n/2^t$

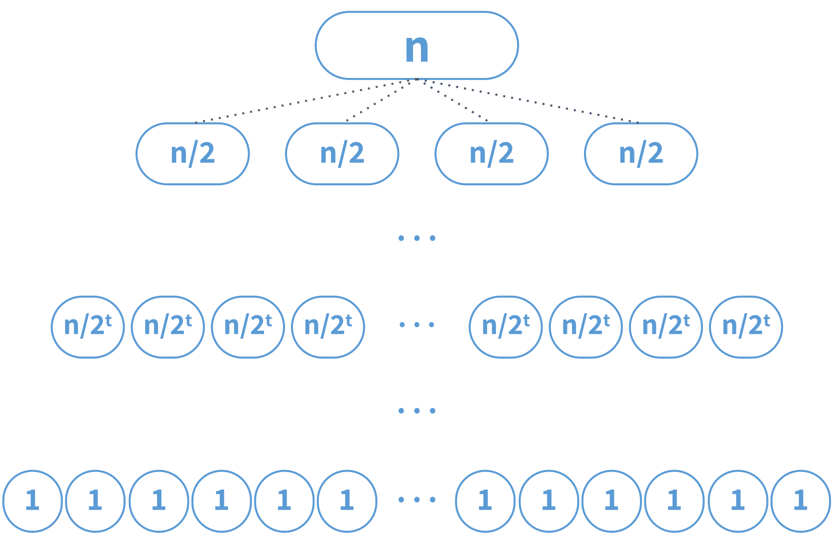
Level $\log_2 n$: $\underline{\;n^2\;}$ problems of size 1

How Efficient is the Algorithm?

The running time of this Divide-and-Conquer multiplication algorithm is **at least $O(n^2)$**!

We know there are already n^2 multiplications happening at the bottom level of the recursion tree, so that's why we say "at least" $O(n^2)$

More concretely, we add up the total computation in all levels



Level 0: 1 problem of size n $\quad 1 \times c\,n = cn$

Level 1: 4^1 problems of size n/2 $\quad 4 \times c\,n/2 = 2cn$

Level t: 4^t problems of size $n/2^t$ $\quad 4^t \times c\,n/2^t = 2^t cn$

Level $\log_2 n$: $\underline{n^2}$ problems of size 1 $\quad 4^{\log_2 n} \times 1 = 2^{\log_2 n} \times cn$

$$\left(1 + 2 + 2^2 + 2^3 + \cdots + 2^{\log_2 n}\right)cn = \mathbf{2cn^2 - cn} = O(n^2)$$

Computational Complexity: $O(n^2)$

How Efficient is the Algorithm?

The running time of this Divide-and-Conquer multiplication algorithm is **$O(n^2)$**!

$$\overbrace{\begin{array}{r} 4512345667809342058121733242 1 \\ \times\ 6378238419834775065209123642 3 \\ \hline \texttt{):} \end{array}}^{\boldsymbol{n}\ \textit{digits}}$$

However, our grade-school algorithm was already $O(n^2)$!

Is Divide-and-Conquer really useful?

Karatsuba Integer Multiplication

Three sub-problems instead of four!

Designing Sub-Problems Wisely

$$[x_1 x_2 \ldots x_{n-1} x_n] \times [y_1 y_2 \ldots y_{n-1} y_n]$$

$$= (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$

$$= (a \times c)10^n + (a \times d + b \times c)10^{n/2} + (b \times d)$$

The subproblems we choose to solve just need to provide these quantities:

ac   ad + bc   bd

Originally, we get these quantities by computing FOUR sub-problems: ac, ad, bc, bd.

Karatsuba's Trick

Final result = (ac)10^n + (ad + bc)$10^{n/2}$ + (bd)

ac & bd can be recursively computed as two subproblems

ad + bc is equivalent to **(a+b)(c+d) - ac - bd**

= (ac + ad + bc + bd) - ac - bd

= ad + bc

So, instead of computing ad & bc as two separate sub-problems, we can just compute one sub-problem **(a+b)(c+d)** instead! Because we can re-used **ac** and **bd**!

(a+b)(c+d) is still an n/2-digit subproblem, since both (a+b) and (c+d) are (approximately) n/2-digit numbers.

Our Three Sub-Problems

These *three* subproblems give us everything we need to compute our desired quantities:

1 ac

2 bd

3 $(a+b)(c+d)$

(a+b) and (c+d) are both going to be n/2-digit numbers!

⇩

This means we still have half-sized subproblems!

Compute our final result by combining these three subproblems:

$$\underbrace{(\ ac\ )}_{1}10^n + \underbrace{(\ ad + bc\ )}_{3\ -\ 1\ -\ 2}10^{n/2} + \underbrace{(\ bd\ )}_{2}$$

Pseudo-Code

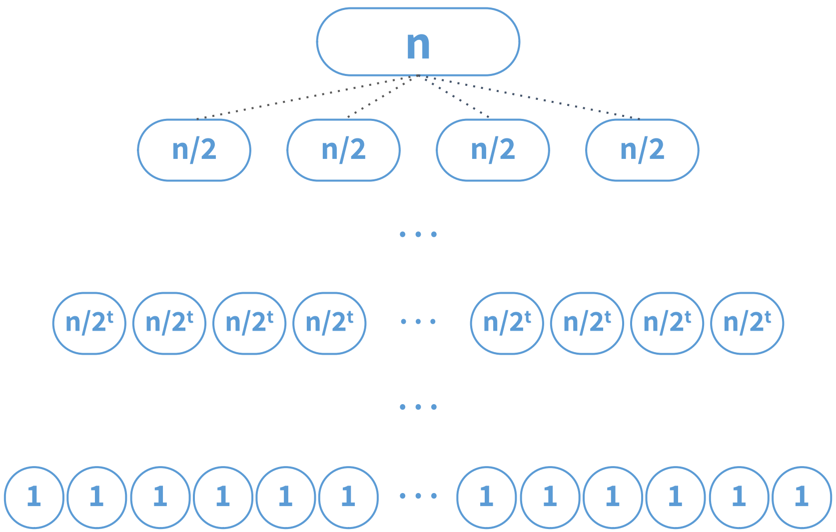
```
algorithm karatsuba_multiply(x, y, n):
  if n == 1: return x·y

  Rewrite x as a·10^(n/2) + b
  Rewrite y as c·10^(n/2) + d

  set ac   = multiply(a, c, n/2)
  set ad   = multiply(a, d, n/2)
  set abcd = multiply(a+b, c+d, n/2)

  return ac·10^n + (abcd-ac-bd)·10^(n/2) + bd
```

Only 3 n/2-digit sub-problems

Add-up to get final answer

How Efficient is the Algorithm?

This was the recursion tree analysis of our previous divide-and-conquer algorithm



Level 0: 1 problem of size n $\quad 1 \times c\ n = cn$

Level 1: 4^1 problems of size n/2 $\quad 4 \times c\ n/2 = 2cn$

Level t: 4^t problems of size $n/2^t$ $\quad 4^t \times c\ n/2^t = 2^t cn$

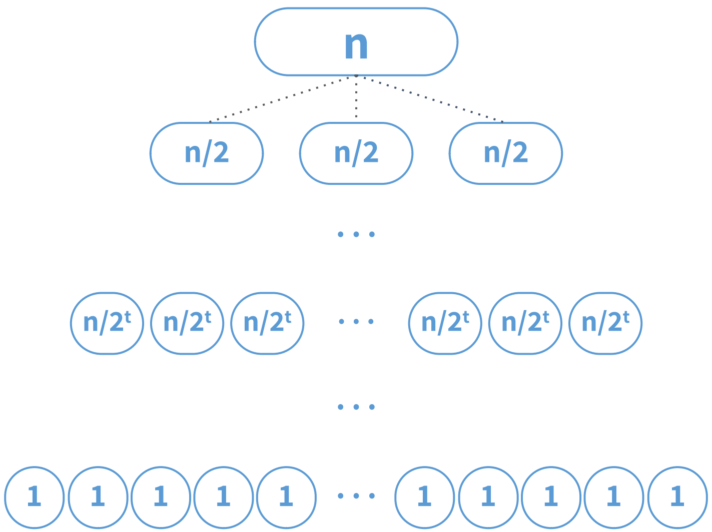
Level $\log_2 n$: $\underline{\ n^2\ }$ problems of size 1 $\quad 4^{\log_2 n} \times 1 = 2^{\log_2 n} \times cn$

$$\left(1 + 2 + 2^2 + 2^3 + \cdots + 2^{\log_2 n}\right)cn = \mathbf{2cn^2 - cn} = O(n^2)$$

Computational Complexity: $O(n^2)$

How Efficient is the Algorithm?

For the new algorithm, we replace branching factor of 4 to 3



Level 0: 1 problem of size n

Level 1: 3^1 problems of size n/2

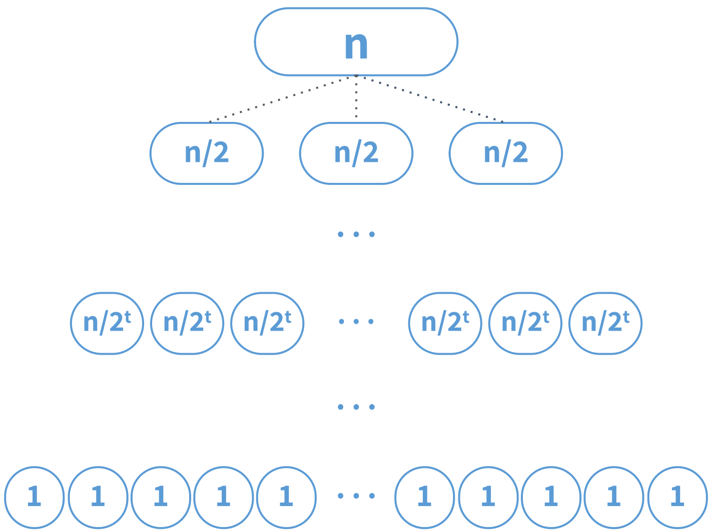
Level t: 3^t problems of size $n/2^t$

Level $\log_2 n$: $\underline{\mathbf{n^{1.6}}}$ problems of size 1

- Why $\log_2 n$ levels? Because if $n/2^t=1$, we have $t=\log_2 n$
  - i.e., you need to cut n in half $\log_2 n$ times to get to size 1
- Why $n^{1.6}$ problems in the last level $\log_2 n$? Because $3^{\log_2 n} = n^{\log_2 3} = \mathbf{n^{1.6}}$

How Efficient is the Algorithm?

For the new algorithm, we replace branching factor of 4 to 3



Level 0: 3^0 problem of size n — $1 \times c\, n = cn$

Level 1: 3^1 problems of size n/2 — $3 \times c\, n/2 = (3/2)cn$

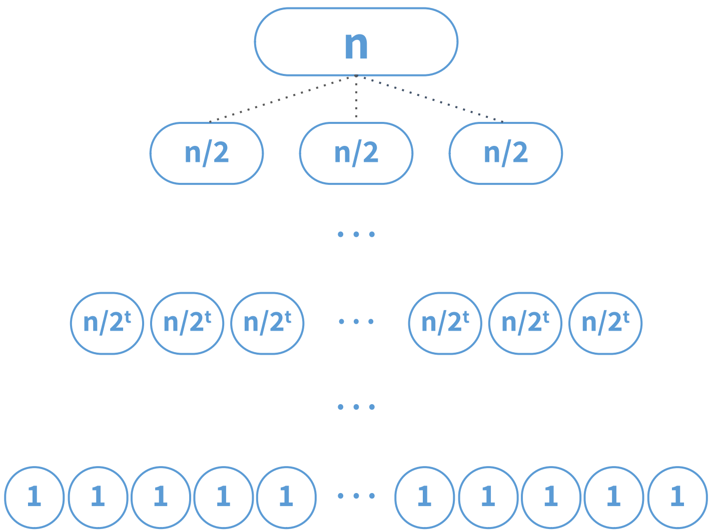
Level t: 3^t problems of size $n/2^t$ — $3^t \times c\, n/2^t = (3/2)^t cn$

Level $\log_2 n$: $n^{1.6}$ problems of size 1 — $3^{\log_2 n} \times 1 = (3/2)^{\log_2 n} \times cn$

$$\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3 + \cdots + \left(\frac{3}{2}\right)^{\log_2 n}\right) cn = \mathbf{3cn^{\log_2 3} - 2cn}$$
$$= \mathbf{3cn^{1.6} - 2cn}$$

How Efficient is the Algorithm?

For the new algorithm, we replace branching factor of 4 to 3



Level 0: 3^0 problem of size n $\quad 1 \times c\ n = cn$

Level 1: 3^1 problems of size n/2 $\quad 3 \times c\ n/2 = (3/2)cn$

Level t: 3^t problems of size $n/2^t$ $\quad 3^t \times c\ n/2^t = (3/2)^t cn$

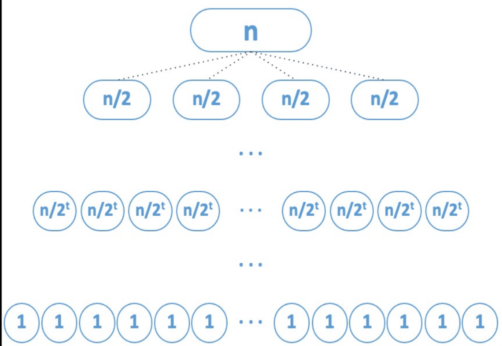
Level $\log_2 n$: $\underline{n^{1.6}}$ problems of size 1 $\quad 3^{\log_2 n} \times 1 = (3/2)^{\log_2 n} \times cn$

$$\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3 + \cdots + \left(\frac{3}{2}\right)^{\log_2 n}\right) cn = 3cn^{\log_2 3} - 2cn$$

$$= \cancel{3}cn^{1.6} - \cancel{2}cn = O(n^{1.6})$$

Computational Complexity: $O(n^{1.6})$

An Interesting Observation

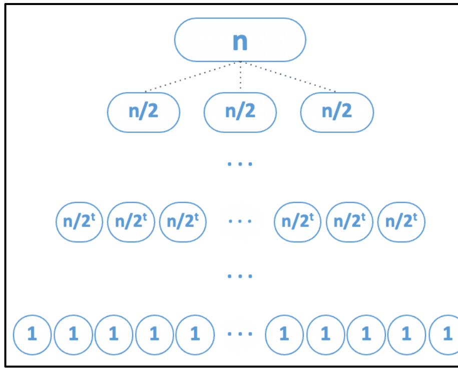


Level 0: 1 problem of size n

Level 1: 4^1 problems of size n/2

Level t: 4^t problems of size $n/2^t$

Level $\log_2 n$: $\underline{n^2}$ problems of size 1



Level 0: 3^0 problem of size n

Level 1: 3^1 problems of size n/2

Level t: 3^t problems of size $n/2^t$

Level $\log_2 n$: $\underline{n^{1.6}}$ problems of size

Computational Complexity: $O(n^2)$

Computational Complexity: $O(n^{1.6})$

For both algorithms:

Number of sub-problem in the last layer

=

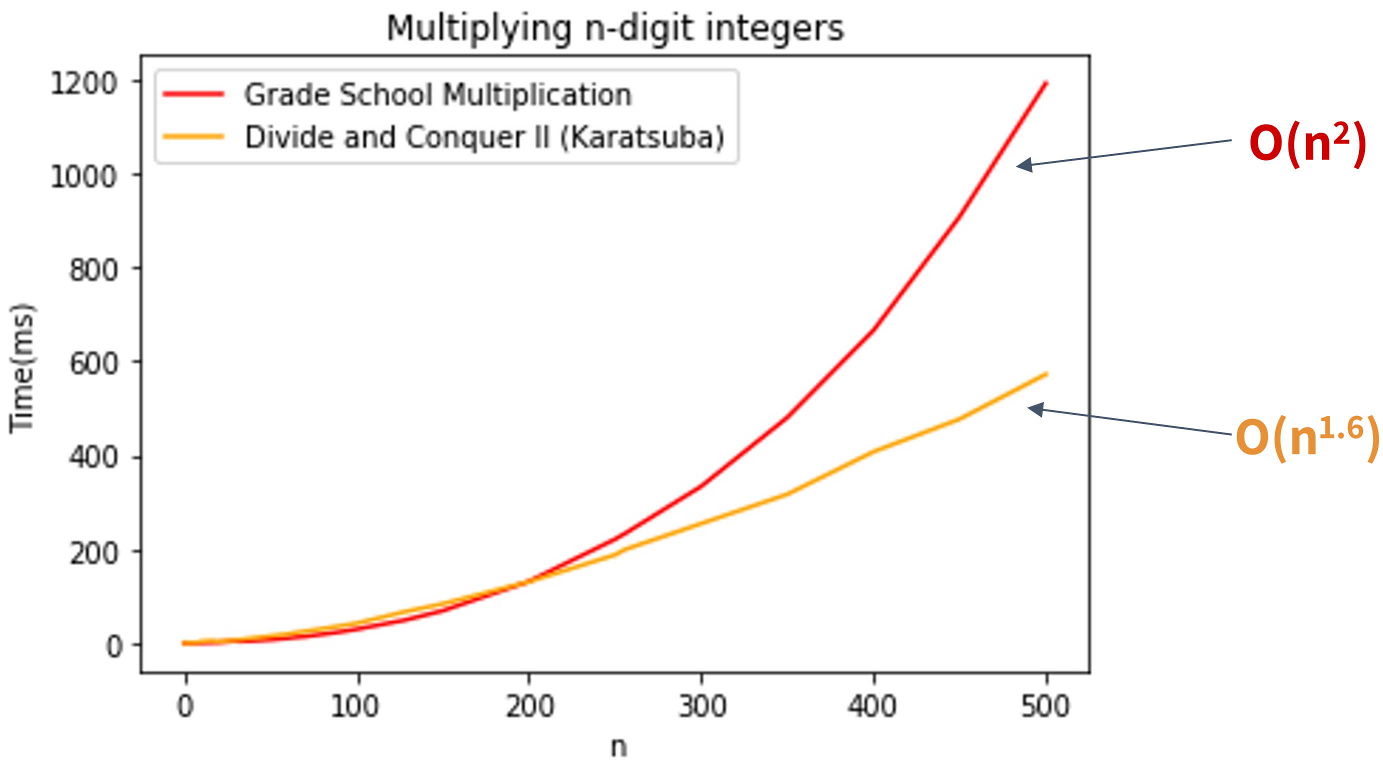
Final computational complexity

We will introduce this observation more formally later.

Generally, the work on the last level actually dominates the algorithm.

But only *in this particular recursion tree*. In other trees, result could be different.

It's Indeed Better in Practice



Can we do even Better?

- **Toom-Cook (1963):** another Divide & Conquer! Instead of breaking into three (n/2)-sized problems, break into five (n/3)-sized problems.
  - Runtime: $O(n^{1.465})$
- **Schönhage–Strassen (1971):** uses fast polynomial multiplications
  - Runtime: $O(n \log n \log \log n)$
- **Fürer (2007):** uses Fourier Transforms over complex numbers
  - Runtime: $O(n \log(n)\, 2^{O(\log^*(n))})$
- **Harvey and van der Hoeven (2019):** crazy stuff
  - Runtime: $O(n \log(n))$

Out of the scope of this class. But feel free to read the papers if you are interested in these (really exciting) algorithms.

5-Minute Break

Asymptotic Analysis

Big-O Notation and its relatives (Big-Ω and Big-Θ)

From Earlier Slides

ASYMPTOTIC ANALYSIS

- **The Key Idea:** we care about how the number of operations *scales* with the size of the input (i.e. the algorithm's *rate of growth*).

- We want some measure of algorithm efficiency that describes the nature of the algorithm, regardless of the environment that runs the algorithm, such as hardware, programming language, memory layout, etc.

The key point of Asymptotic Big-O notation is:

Ignore **constant factors** and **lower-order terms**

Too system dependent — Irrelevant when input size **n** is large

Different Ways to Analysis Runtime of an Algorithm

There are a few different ways to analyze the runtime of an algorithm:

Worst-case analysis:
What is the runtime of the algorithm on the *worst* possible input?

Best-case analysis:
What is the runtime of the algorithm on the *best* possible input?

Average-case analysis:
What is the runtime of the algorithm on the *average* input?

Different Ways to Analysis Runtime of an Algorithm

There are a few different ways to analyze the runtime of an algorithm:

We will mainly focus on worst case analysis since it tells us how fast the algorithm is on *any* kind of input.

Worst-case analysis:
What is the runtime of the algorithm on the *worst* possible input?

Best-case analysis:
What is the runtime of the algorithm on the *best* possible input?

Average-case analysis:
What is the runtime of the algorithm on the *average* input?

We will talk more about this when we introduce randomized algorithms.

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is O(f(n))"?

Language Definition

Picture Definition

Math Definition

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is O(f(n))"?

In Language

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

Picture Definition

Math Definition

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

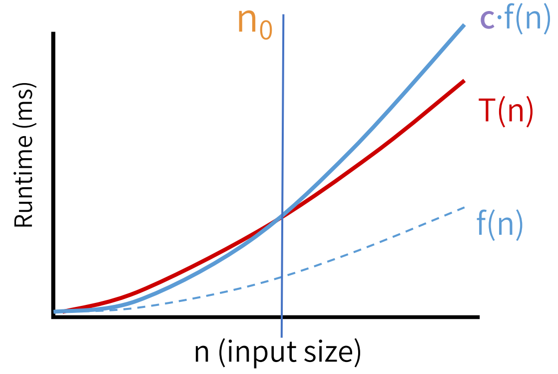
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is O(f(n))"?

In Language

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

In Pictures



Math Definition

Big-O Notation

Let $T(n)$ & $f(n)$ be functions defined on the positive integers.

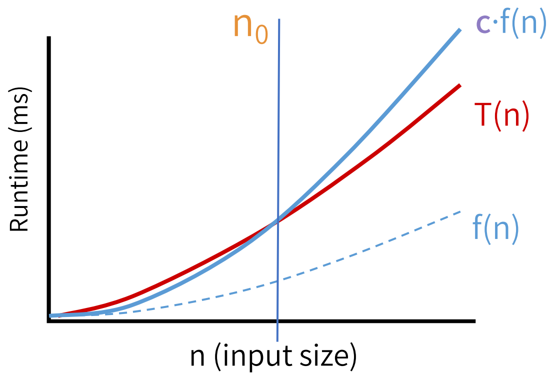
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is O(f(n))"?

In Language

$T(n) = O(f(n))$ if and only if $T(n)$ is *eventually* **upper bounded** by a constant multiple of $f(n)$

In Pictures



In Math

$T(n) = O(f(n))$ if and only if there exists positive **constants** **c** and $\mathbf{n_0}$ such that *for all* $n \geq n_0$

$$T(n) \leq c \cdot f(n)$$

Big-O Notation

Let $T(n)$ & $f(n)$ be functions defined on the positive integers.

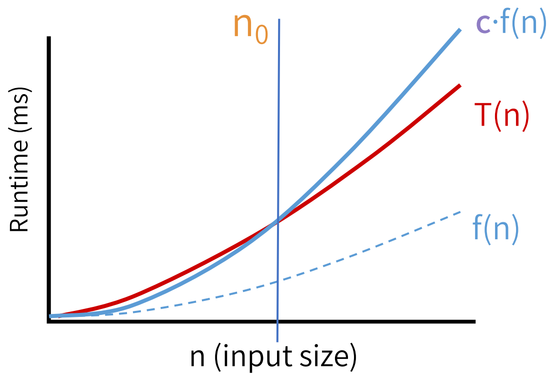
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is O(f(n))"?

In Language

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

In Pictures



In Math

$$T(n) = O(f(n))$$

$$\Leftrightarrow$$

$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$

$$\mathbf{T(n) \leq c \cdot f(n)}$$

Big-O Notation

Let T(n) & f(n) be functions defined on the positive integers.

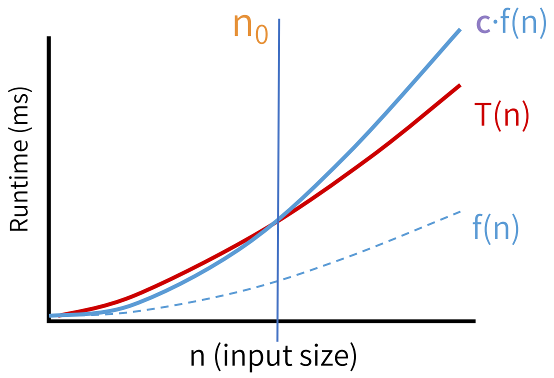
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is O(f(n))"?

In Language

T(n) = O(f(n)) if and only if T(n) is *eventually* **upper bounded** by a constant multiple of f(n)

In Pictures



In Math

$$T(n) = O(f(n))$$

"if and only if" $\Leftrightarrow$

$$\exists\, c,\, n_0 > 0 \;\text{ s.t. }\; \forall\, n \geq n_0,$$

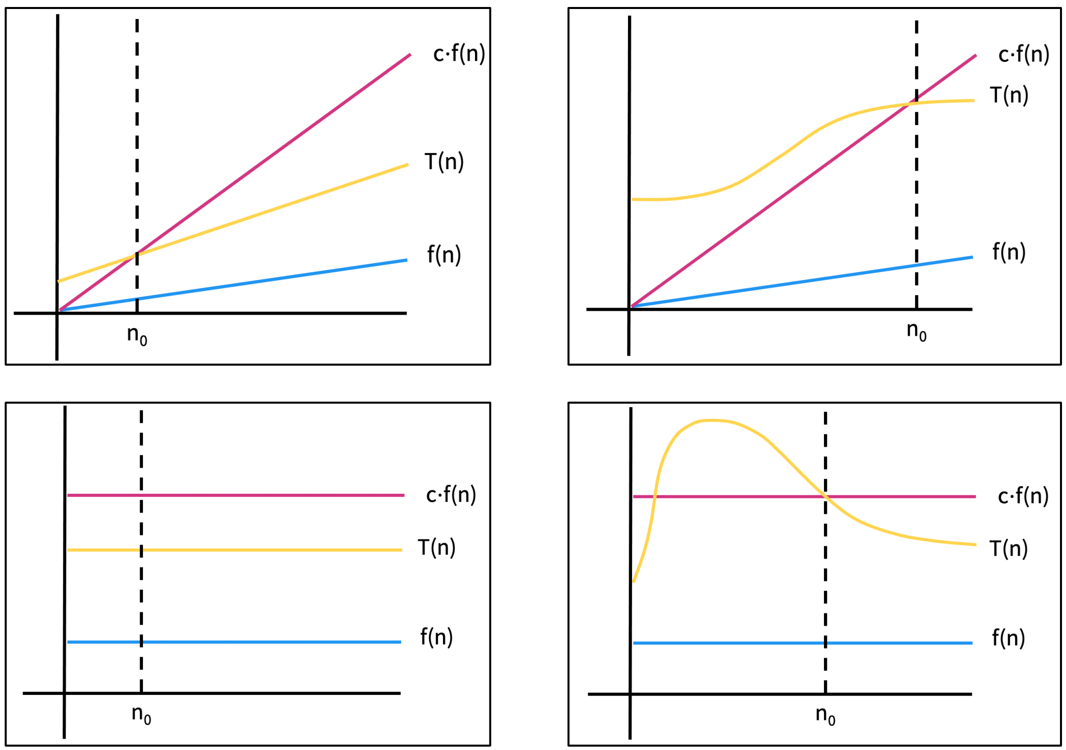
"there exists" "for all" "such that"

$$T(n) \leq c \cdot f(n)$$

Key Point: Asymptotic

When we say "T(n) is O(f(n))", we only care about cases when n is sufficiently large, i.e., $\forall\ n \geq n_0$

This is what we mean by "Asymptotic Analysis"



In all of the above four figures, T(n)=O(f(n))

Proving Big-O Bounds

If you're ever asked to formally prove that T(n) is O(f(n)), use the MATH definition:

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$
$$\mathbf{T(n) \leq c \cdot f(n)}$$

must be constants!
i.e. c & n_0 cannot
depend on n!

- To **prove** T(n) = O(f(n)), you need to announce your c & n_0 up front!
    - Play around with the expressions to find appropriate choices of c & n_0 (positive constants)
    - Then you can write the proof. Here is typically how to structure the start of the proof:

"Let c = __ and n_0 = __. We will show that T(n) ≤ c · f(n) for all n ≥ n_0."

……

Proving Big-O Bounds Example

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c,\ n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$
$$T(n) \leq c \cdot f(n)$$

Prove that $3n^2 + 5n = O(n^2)$.

Let $c = 4$ and $n_0 = 5$. We will now show that $3n^2 + 5n \leq c \cdot n^2$ for all $n \geq n_0$.

We know that for any $n \geq n_0$, we have:

$$5 \leq n$$
$$5n \leq n^2$$
$$5n + 3n^2 \leq n^2 + 3n^2$$
$$3n^2 + 5n \leq 4n^2$$

Using our choice of c and n_0, we have successfully shown that $3n^2 + 5n \leq c \cdot n^2$ for all $n \geq n_0$. From the definition of Big-O, this proves that $3n^2 + 5n = O(n^2)$. ■

Dis-Proving Big-O Bounds

If you are asked to formally disprove that T(n) is O(f(n)), use **proof by contradiction!**

Dis-Proving Big-O Bounds

If you are asked to formally prove that T(n) is not O(f(n)), use **proof by contradiction!**

For sake of contradiction, assume that T(n) is O(f(n)). In other words,
assume there does indeed exist a choice of c & n_0 s.t. $\forall$ $n \geq n_0$, **$T(n) \leq c \cdot f(n)$**

Dis-Proving Big-O Bounds

If you are asked to formally prove that T(n) is not O(f(n)), use **proof by contradiction!**

For sake of contradiction, assume that T(n) is O(f(n)). In other words,
assume there does indeed exist a choice of c & n_0 s.t. $\forall\ n \geq n_0$, $\mathbf{T(n) \leq c \cdot f(n)}$

Treating c & n_0 as variables, derive a contradiction!

Dis-Proving Big-O Bounds

If you are asked to formally prove that T(n) is not O(f(n)), use **proof by contradiction!**

For sake of contradiction, assume that T(n) is O(f(n)). In other words,
assume there does indeed exist a choice of c & n_0 s.t. $\forall$ $n \geq n_0$, **T(n) ≤ c · f(n)**

⇩

Treating c & n_0 as variables, derive a contradiction!

⇩

Conclude that the original assumption must be false, so **T(n) is *not* O(f(n))**.

Dis-Proving Big-O Bounds Example

$T(n) = O(f(n))$
$\Leftrightarrow$
$\exists$ $c, n_0 > 0$ s.t. $\forall$ $n \geq n_0$,
$T(n) \leq c \cdot f(n)$

Prove that $3n^2 + 5n$ is *not* O(n).

For sake of contradiction, assume that $3n^2 + 5n$ is $O(n)$. This means that there exists positive constants c & n_0 such that $3n^2 + 5n \leq c \cdot n$ for all $n \geq n_0$. Then, we would have the following:

$$3n^2 + 5n \leq c \cdot n$$
$$3n + 5 \leq c$$
$$n \leq (c - 5)/3$$

However, since $(c - 5)/3$ is a constant, we've arrived at a contradiction since n cannot be bounded above by a constant for all $n \geq n_0$. For instance, consider $n = n_0 + c$: we see that $n \geq n_0$, but $n > (c - 5)/3$, because $c > (c - 5)/3$. Thus, our original assumption was incorrect, which means that $3n^2 + 5n$ is not $O(n)$. ■

Frequently used Big-O Examples

$\log_2 n + 15 = O(\log_2 n)$

$3^n = O(4^n)$

Polynomials

Say $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree $k \geq 1$.

Then:

i. $p(n) = O(n^k)$
ii. $p(n)$ is **not** $O(n^{k-1})$ or $O(n^{k-2})$ or …

e.g., $6n^3 + 10n^2 + 5 = O(n^3)$

$\log_2 n = O(n)$

$6n^3 + n \log_2 n = O(n^3)$

$25 = O(1)$
[any constant] $= O(1)$

Frequently used Big-O Examples

lower order terms do not matter!

$$\log_2 n + 15 = O(\log_2 n)$$

remember, big-O is upper bound!

$$3^n = O(4^n)$$

Polynomials

Say $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree $k \geq 1$.

Then:

i. $p(n) = O(n^k)$
ii. $p(n)$ is **not** $O(n^{k-1})$ or $O(n^{k-2})$ or …

e.g., $6n^3 + 10n^2 + 5 = O(n^3)$

$$\log_2 n = O(n)$$

constant multipliers & lower order terms don't matter

$$6n^3 + n \log_2 n = O(n^3)$$

$$25 = O(1)$$

[any constant] = O(1)

Big-Ω Notation

Let T(n) & f(n) be functions defined on the positive integers.

(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is Ω(f(n))"?

Language Definition	Picture Definition	Math Definition

Big-Ω Notation

Let $T(n)$ & $f(n)$ be functions defined on the positive integers.

(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "$T(n)$ is $\Omega(f(n))$"?

In Language

$T(n) = \Omega(f(n))$ if and only if $T(n)$ is eventually **lower bounded** by a constant multiple of $f(n)$

Picture Definition

Math Definition

Big-Ω Notation

Let T(n) & f(n) be functions defined on the positive integers.

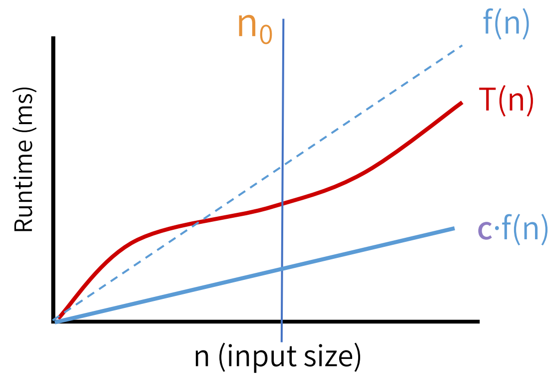
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is Ω(f(n))"?

In Language

T(n) = Ω(f(n)) if and only if T(n) is eventually **lower bounded** by a constant multiple of f(n)

In Picture



Math Definition

Big-Ω Notation

Let T(n) & f(n) be functions defined on the positive integers.

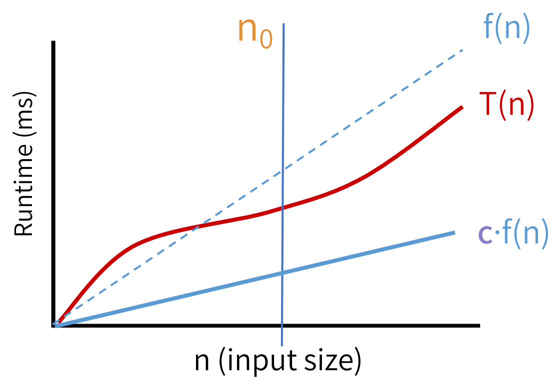
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is Ω(f(n))"?

In Language

T(n) = Ω(f(n)) if and only if T(n) is eventually **lower bounded** by a constant multiple of f(n)

In Picture



In Math

$$T(n) = \Omega(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c\ ,\ n_0 > 0\ \text{ s.t. } \forall\ n \geq n_0\ ,$$
$$\mathbf{T(n) \geq c \cdot f(n)}$$

inequality switches direction!

Big-Θ Notation

We say **"$T(n)$ is $\Theta(f(n))$"** if and only if both

$$\mathbf{T(n) = O(f(n))}$$

and

$$\mathbf{T(n) = \Omega(f(n))}$$

$$T(n) = \Theta(f(n))$$

$$\Leftrightarrow$$

$$\exists\ c_1, c_2, n_0 > 0 \ \text{ s.t. } \forall\ n \geq n_0,$$

$$\mathbf{c_1 \cdot f(n) \ \leq\ T(n) \ \leq\ c_2 \cdot f(n)}$$

Asymptotic Notation Summary

BOUND	DEFINITION (HOW TO PROVE)	WHAT IT REPRESENTS
$T(n) = O(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0$ s.t. $\forall\ n \geq n_0,\ \mathbf{T(n) \leq c \cdot f(n)}$	upper bound
$T(n) = \Omega(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0$ s.t. $\forall\ n \geq n_0,\ \mathbf{T(n) \geq c \cdot f(n)}$	lower bound
$T(n) = \Theta(f(n))$	$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$ $\exists\ c_1, c_2 > 0,\ \exists\ n_0 > 0$ s.t. $\forall\ n \geq n_0,\ c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$	tight bound

Summary

- You will learn how to **design**, **analyze**, and **communicate about** algorithms.
- We introduced you to Divide-and-Conquer!
- Karatsuba Integer Multiplication is a clever application of Divide-and-Conquer.
- Asymptotic Analysis (Big-O etc.) helps us to express the efficiency (runtime) of algorithms.

Summary

- You will learn how to **design**, **analyze**, and **communicate about** algorithms.
- We introduced you to Divide-and-Conquer!
- Karatsuba Integer Multiplication is a clever application of Divide-and-Conquer.
- Asymptotic Analysis (Big-O etc.) helps us to express the efficiency (runtime) of algorithms.

Acknowledgement: Part of the materials are adapted from Mary Wootter, Virginia Williams, David Eng and Karey Shi's lectures on algorithms. We appreciate their contributions.