CSC236 Week 4

Larry Zhang

Announcements

- PS2 due on Friday
- This week’s tutorial: Exercises with big-Oh
- PS1 feedback
    - People generally did well
    - Writing style need to be improved. This time the TAs are lenient, next time they will be more strict.
- Read the feedback post on Piazza.

Recap: Asymptotic Notations

- **Algorithm Runtime**: we describe it in terms of the **number of steps** as a **function** of **input size**
  - Like **n^2, nlog(n), n, sqrt(n), ...**
- **Asymptotic notations** are for describing the **growth rate** of **functions**.
  - constant factors don't matter
  - only the highest-order term matters

Big-Oh, Big-Omega, Big-Theta

O(f(n)): The set of functions that grows **no faster** than **f(n)**

- asymptotic **upper-bound** on growth rate

Ω(f(n)): The set of functions that grows **no slower** than **f(n)**

- asymptotic **lower-bound** on growth rate

Θ(f(n)): The set of functions that grows **no faster and no slower** than f(n)

- asymptotic **tight-bound** on growth rate

growth rate ranking of typical functions

grow fast

$$f(n) = n^n$$

$$f(n) = 2^n$$

$$f(n) = n^3$$

$$f(n) = n^2$$

$$f(n) = n \log n$$

$$f(n) = n$$

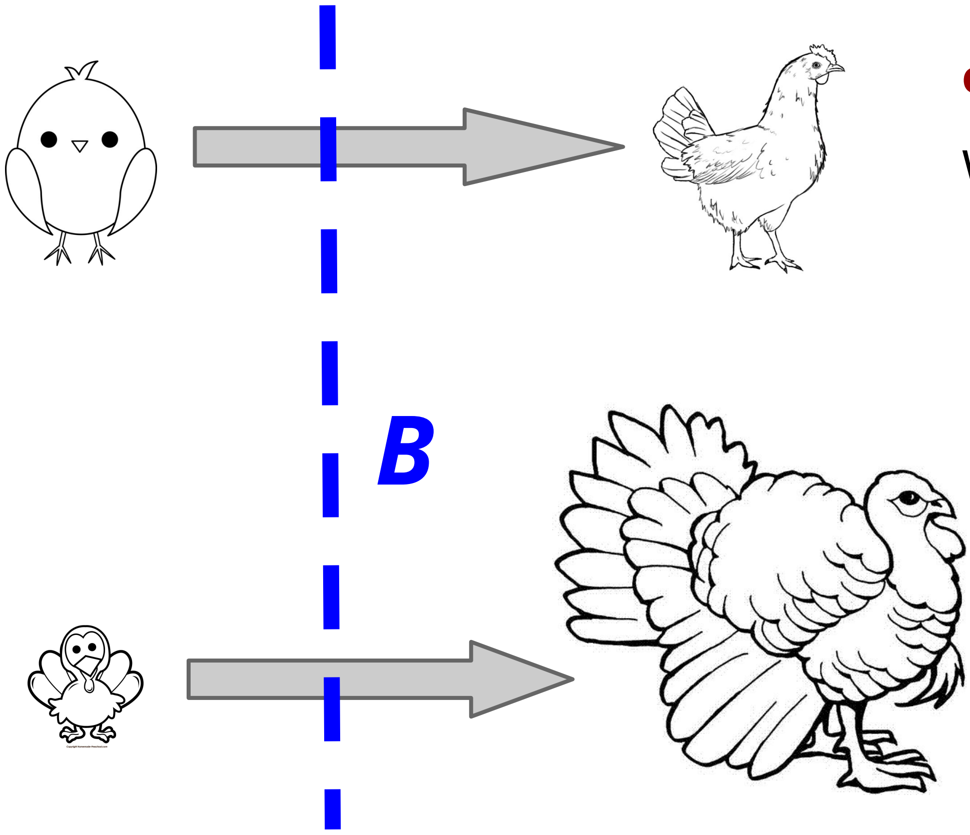
$$f(n) = \sqrt{n}$$

$$f(n) = \log n$$

$$f(n) = 1$$

grow slowly

The formal mathematical definition of big-Oh



Chicken **grows slower** than turkey, or

chicken size is in O(turkey size).

What it really means:

- Baby chicken might be larger than baby turkey at the beginning.
- But after certain "**breakpoint**", the chicken size will be **surpassed** by the turkey size.
- From the **breakpoint on**, the chicken size will **always** be smaller than the turkey size.

Definition of big-Oh

Function $f(n) = O(g(n))$ ("f is big oh of g") iff

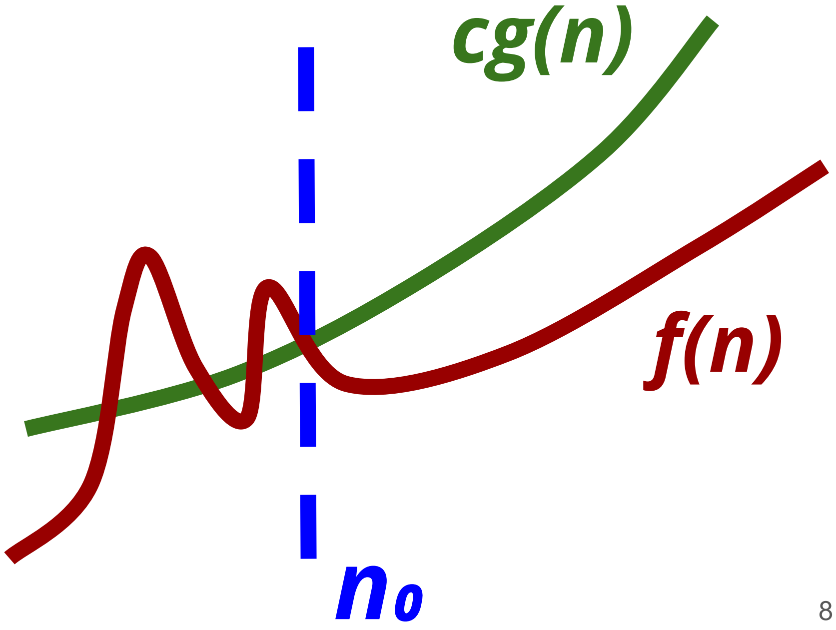
(i) There is some positive $n_0 \in N$

(ii) There is some positive $c \in R$

such that

$\forall n \geq n_0, f(n) \leq cg(n)$

*Beyond the **breakpoint n_0**, **f(n)** is **upper-bounded** by **cg(n)**, where **c** is some wisely chosen constant multiplier.*



Side Note

Both ways below are fine

- $f(n) \in O(g(n))$, i.e., $f(n)$ is **in** $O(g(n))$
- $f(n) = O(g(n))$, i.e., $f(n)$ is $O(g(n))$

Both means the same thing, while the latter is a slight abuse of notation.

Function $f(n) = O(g(n))$ ("f is big oh of g") iff

(i) There is some positive $n_0 \in N$

(ii) There is some positive $c \in R$

such that

$\forall n \geq n_0, f(n) \leq cg(n)$

Knowing the definition,

now we can write proofs for big-Oh.

The key is finding n_0 and c

Example 1

Prove that $100n + 10000$ is in $O(n^2)$

Need to find the n_0 and c such that $100n + 10000$ can be upper-bounded by n^2 multiplied by some c.

under-estimate and simplify

$c\, n^2$

large

Function $f(n) = O(g(n))$ ("f is big oh of g") iff

(i) There is some positive $n_0 \in N$

(ii) There is some positive $c \in R$

such that

$\forall n \geq n_0, f(n) \leq cg(n)$

Pick c = 10100

$10100n^2$

Pick n_0 = 1

$100n^2 + 10000n^2$

coz n >= 1

$100n^2 + 10000$

coz n >= 1

$100n + 10000$

over-estimate and simplify

small

Proof that $100n + 10000$ is in $O(n^2)$

Write up the proof

Function $f(n) = O(g(n))$ ("f is big oh of g") iff

(i) There is some positive $n_0 \in N$

(ii) There is some positive $c \in R$

such that

$\forall n \geq n_0, f(n) \leq cg(n)$

Proof:

Choose $n_0=1$, $c = 10100$,

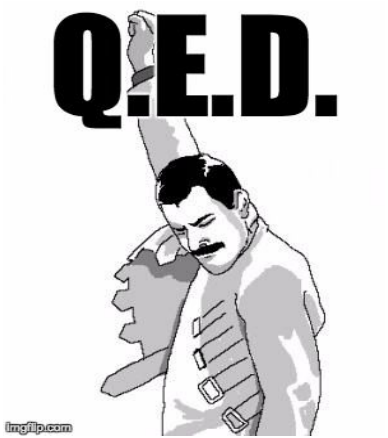
then for all $n >= n_0$,

$100n + 10000 <= 100n^2 + 10000n^2$ **# because n >= 1**

$= 10100n^2$

$= cn^2$

Therefore by definition of big-Oh, $100n + 10000$ is in $O(n^2)$



Quick note

The choice of n_0 and c is not unique.

There can be many (actually, infinitely many) different combinations of n_0 and c that would make the proof work.

It depends on what inequalities you use while doing the upper/lower-bounding.

Example 2

Prove that $5n^2 - 3n + 20$ is in $O(n^2)$

under-estimate and simplify

over-estimate and simplify

$c\ n^2$

Pick c = 25

$25n^2$

$5n^2 + 20n^2$

\# coz n >= 1

$5n^2 + 20$

\# remove a negative term

$5n^2 - 3n + 20$

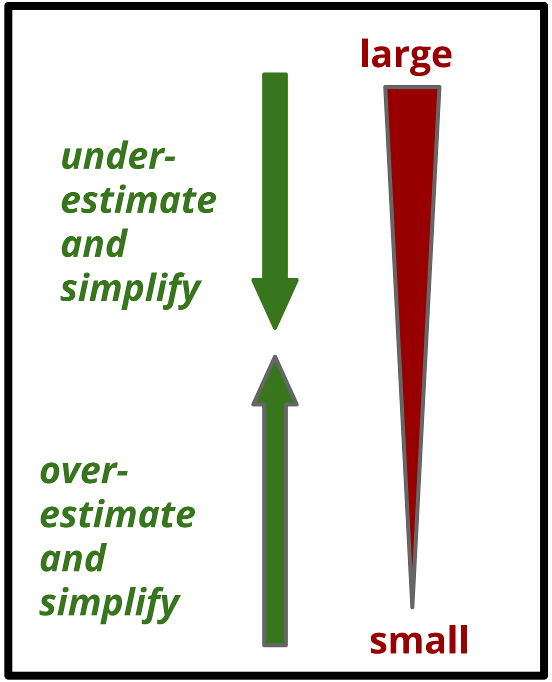
large

small

Pick n_0 = 1

Takeaway

choose some breakpoint (**n_0=1 often works**), then we can assume **n ≥ 1**



under-estimation tricks

➔ **remove** a **positive** term

◆ $3n^2 + 2n \geq 3n^2$

➔ **multiply** a **negative** term

◆ $5n^2 - n \geq 5n^2 - n*n = 4n^2$

over-estimation tricks

➔ **remove** a **negative** term

◆ $3n^2 - 2n \leq 3n^2$

➔ **multiply** a **positive** term

◆ $5n^2 + n \leq 5n^2 + n*n = 6n^2$

After simplification, **choose a *c*** that connects both sides.

The formal mathematical definition of big-**Omega**

Definition of big-Omega

Function $f(n) = \Omega(g(n))$ iff

(i) There is some positive $n_0 \in N$

(ii) There is some positive $c \in R$

such that

$\forall n \geq n_0, \boxed{cg(n) \leq f(n)}$

The only difference

This means that $g(n)$ is a **lower bound** on $f(n)$.

Example 3

Prove that $2n^3 - 7n + 1 = \Omega(n^3)$

Prove that $2n^3 - 7n + 1 = \Omega(n^3)$

under-estimate and simplify

over-estimate and simplify

large

$2n^3 - 7n + 1$

$n^3 + n^3 - 7n + 1$

$n^3 + 1$

$n^3 - 7n > 0$

coz $n >= 3$

n^3

Pick $c = 1$

$c\ n^3$

small

Pick $n_0 = 3$

Prove that $2n^3 - 7n + 1$ is in $\Omega(n^3)$

Write up the proof

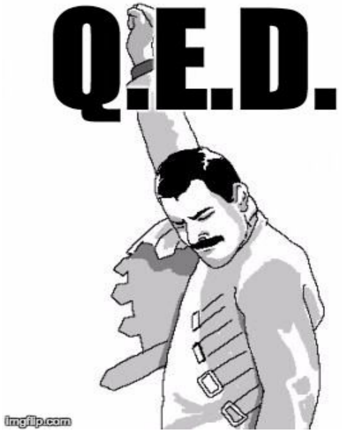
Proof:

Choose $n_0 = 3$, $c = 1$,

then for all $n \geq n_0$,

$$
\begin{aligned}
2n^3 - 7n + 1 &= n^3 + (n^3 - 7n) + 1 \\
&\geq n^3 + 1 \quad \textbf{\# because } \boldsymbol{n \geq 3} \\
&\geq n^3 = cn^3
\end{aligned}
$$

Therefore by definition of big-Omega, $2n^3 - 7n + 1$ is in $\Omega(n^3)$



Takeaway

Additional trick learned

- Splitting a higher order term
- Choose n_0 to however large you need it to be

$$\boxed{n^3 + n^3 - 7n + 1}$$

The formal mathematical definition of big-**Theta**

Definition of big-Theta

Function $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
This means that $g(n)$ is a **tight bound** on $f(n)$.

In other words, if you want to prove big-Theta, just prove **both** big-Oh and big-Omega, separately.

Exercise for home

Prove that $2n^3 - 7n + 1 = \Theta(n^3)$

Summary of Asymptotic Notations

- We use **functions** to describe algorithm runtime
    - Number of steps as a function of input size
- Big-Oh/Omega/Theta are used for describing function growth rate
- A proof for big-Oh and big-Omega is basically a chain of inequality.
    - Choose the n_0 and c that makes the chain work.

Now we can analyze algorithms more like a pro

```
def foo(lst):
1   result1 = 1
2   Result2 = 6
3   for i in range(len(lst)):
4       for j in range(len(lst)):
5           result1 += i*j
6           result2 = lst[j] + i
7   return
```

- Let n be the length of lst
- The outer loop iterates n times
- For each iteration of outer loop, the inner loop iterates n times
- Each iteration of inner loops takes 2 steps.
- Line 1 and 2 does some constant work
- So the overall runtime is $2 + n * n * 2 = 2n^2 + 2 =$ $\mathbf{\Theta(n^2)}$

NEW TOPIC

Recursion

To really understand the math of recursion, and to be able to analyze runtimes of recursive programs.

Recursively Defined Functions

The functions that describe
the runtime of recursive programs

Definition of functions

- The usual way: define a function using a **closed-form**.

$$T_0(n) = 2^n - 1$$

- Another way: using a **recursive** definition

$$T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1) + 1, & \text{if } n > 1 \end{cases}$$

You will get this when analyzing the runtime of recursive programs.

A function defined in terms of **itself**.

Let's work out a few values for $T_0(n)$ and $T_1(n)$

$$T_0(n) = 2^n - 1$$

$$T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1)+1, & \text{if } n > 1 \end{cases}$$

Maybe $T_0(n)$ and $T_1(n)$ are equivalent to each other.

	$T_0(n)$	$T_1(n)$
n=1	**1**	**1**
n=2	**3**	**3**
n=3	**7**	**7**
n=4	**15**	**15**
n=5	**31**	**31**

$$T_0(n) = 2^n - 1 \qquad T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1) + 1, & \text{if } n > 1 \end{cases}$$

Every **recursively** defined function has an equivalent **closed-form** function.

And we like closed-form functions.

- It is easier to evaluate
  - Just need **n** to know **f(n)**
  - Instead of having to know **f(n-1)** to know **f(n)**
- It is easier for telling the growth rate of the function
  - $T_0(n)$ is clearly $\Theta(2^n)$, while $T_1(n)$ it not clear.

Our Goal:

Given a **recursively** defined function,
find its equivalent **closed-form** function

Method #1
Repeated Substitution

Repeated substitution: Steps

Step 1: **Substitute** a few time to find a pattern

Step 2: **Guess** the recurrence formula after k substitutions (in terms of k and n)

For each base case:

Step 3: **solve** for k

Step 4: **Plug** k back into the formula (from Step 2) to find a **potential** closed form. ("Potential" because it might be wrong)

Step 5: **Prove** the potential closed form is equivalent to the recursive definition using induction.

Example 1

Step 1

$$T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1) + 1, & \text{if } n > 1 \end{cases}$$

Substitute a few times to find a pattern.

$k = 1$ $\quad T(n) = 2T(n-1) + 1$

Substitute T(n-1) with T(n-2): $k = 2$ $\quad T(n) = 2(2T(n-2) + 1) + 1$

$$= 4T(n-2) + 3$$

Substitute T(n-2) with T(n-3): $k = 3$ $\quad T(n) = 4(2T(n-3) + 1) + 3$

$$= 8T(n-3) + 7$$

Substitute T(n-3) with T(n-4): $k = 4$ $\quad T(n) = 8(2T(n-4) + 1) + 7$

$$= 16T(n-4) + 15$$

Step 2:

Guess the recurrence formula after ***k*** substitutions

$$
\begin{aligned}
k = 1 \qquad & T(n) = 2T(n-1) + 1 \\
k = 2 \qquad & T(n) = 2(2T(n-2) + 1) + 1 \\
& \quad\;\; = 4T(n-2) + 3 \\
k = 3 \qquad & T(n) = 4(2T(n-3) + 1) + 3 \\
& \quad\;\; = 8T(n-3) + 7 \\
k = 4 \qquad & T(n) = 8(2T(n-4) + 1) + 7 \\
& \quad\;\; = 16T(n-4) + 15
\end{aligned}
$$

The guess: $T(n) = 2^k T(n-k) + 2^k - 1$

$$T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1) + 1, & \text{if } n > 1 \end{cases}$$

Step 3: Consider the base case

What we have now: $T(n) = 2^k \underline{T(n-k)} + 2^k - 1$

We want this to be T(1), because we know clearly what T(1) is.

So, let n - k = 1, and solve for k

k = n - 1

This means you need to make n-1 substitutions to reach the base case n=1

$$T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1)+1, & \text{if } n > 1 \end{cases}$$

Step 4: plug ***k*** back into the guessed formula

Guessed formula: $T(n) = 2^k T(n-k) + 2^k - 1$

For Step 3, we got **k = n - 1**

Substitute k back in, we get ...

$$\begin{aligned} T(n) &= 2^k T(n-k) + 2^k - 1 \\ &= 2^{n-1} T(1) + 2^{n-1} - 1 \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 \end{aligned}$$

This is the "potential" closed-form of T(n).

Step 5: Drop the word "potential" by proving it

Prove $T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1) + 1, & \text{if } n > 1 \end{cases}$ **is equivalent to** $T_0(n) = 2^n - 1$

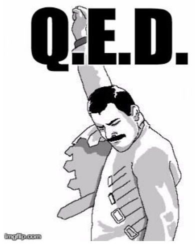
Use induction!

Define predicate: P(n): $T_1(n) = T_0(n)$

Base case: n=1 $T(1) = 1 = 2^1 - 1$

Induction step: suppose that $n > 1$ and that $T(n-1) = 2^{n-1} - 1$

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 && \text{\# by def of T(n)} \\
&= 2(2^{n-1} - 1) + 1 && \text{\# by I.H.} \\
&= 2^n - 2 + 1 \\
&= 2^n - 1
\end{aligned}
$$

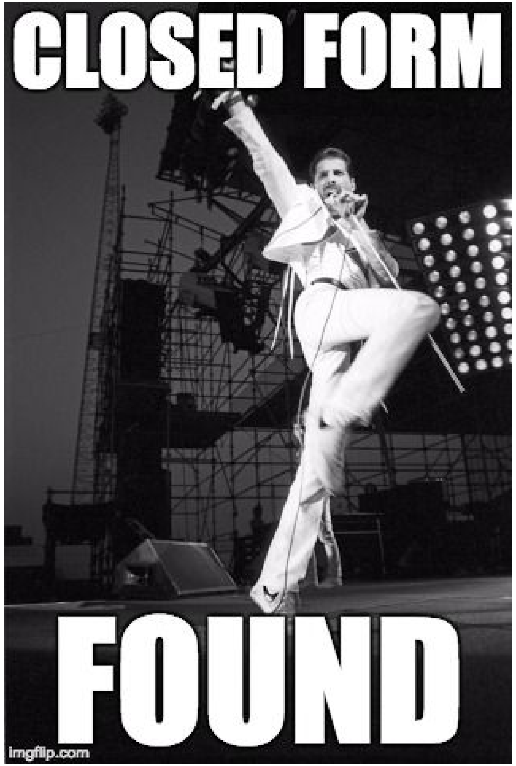


We have officially found that the closed-form of the recursively defined function

$$T_1(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T_1(n-1) + 1, & \text{if } n > 1 \end{cases}$$

is

$$T_0(n) = 2^n - 1$$



Repeated substitution: Steps

Step 1: **Substitute** a few time to find a pattern

Step 2: **Guess** the recurrence formula after k substitutions (in terms of k and n)

For each base case:

Step 3: **solve** for k

Step 4: **Plug** k back into the formula (from Step 2) to find a **potential** closed form. ("Potential" because it might be wrong)

Step 5: **Prove** the potential closed form is equivalent to the recursive definition using induction.

Example 2

Find the closed form of ...

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2T(n-2), & \text{if } n > 2 \end{cases}$$

Something new: there are **two** base cases, they may give us “**two**” closed forms!

Step 1: Repeat substitution

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2T(n-2), & \text{if } n > 2 \end{cases}$$

$k = 1$ $$T(n) = 2T(n-2)$$

$k = 2$ $$T(n) = 2(2T(n-4))$$
$$= 4T(n-4)$$

$k = 3$ $$T(n) = 4(2T(n-6))$$
$$= 8T(n-6)$$

Step 2: Guess the formula after k substitutions

$$T(n) = 2^k T(n-2k)$$

Step 3: Solve for k, **for each base case**

Base case #1: n = 1, we want to see T(1), so

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2T(n-2), & \text{if } n > 2 \end{cases}$$

Let $n - 2k = 1$

$$k = \frac{n-1}{2}$$

$$T(n) = 2^k T(n - 2k)$$

Step 4: Plug k back into the formula

$$\begin{aligned} T(n) &= 2^k T(n-2k) \\ &= 2^{\frac{n-1}{2}} T(1) \\ &= 0 \end{aligned}$$

Potential closed form #1

Step 3 again

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2T(n-2), & \text{if } n > 2 \end{cases}$$

Base case #2: n = 2, we want to see T(2), so

$$T(n) = 2^k T(n-2k)$$

Let $n - 2k = 2$ $\quad k = \frac{n-2}{2}$

Step 4 again: Plug k back into the formula

$$\begin{aligned} T(n) &= 2^k T(n-2k) \\ &= 2^{\frac{n-2}{2}} T(2) \\ &= 2^{\frac{n-2}{2}} 2 \\ &= 2^{\frac{n}{2}} \end{aligned}$$

Potential closed form #2

What we have so far

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2T(n-2), & \text{if } n > 2 \end{cases}$$

Starting from n=1, add by 2 each time, we have T(n) = 0

Starting from n=2, add by 2 each time, we have T(n) = 2^{n/2}

In other words,

$$T(n) = \begin{cases} 0 & \text{if } n \text{ is odd} \\ 2^{n/2} & \text{if } n \text{ is even} \end{cases}$$

This is the complete potential closed form.

Step 5: Prove it.

Try it yourself!

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 2T(n-2), & \text{if } n > 2 \end{cases}$$

is equivalent to

$$T(n) = \begin{cases} 0 & \text{if } n \text{ is odd} \\ 2^{n/2} & \text{if } n \text{ is even} \end{cases}$$

Summary

Find the closed form of a recursively defined function

- Method #1: Repeated substitution
  - It’s nothing tricky, just follow the steps!
  - Take care of all base cases!
  - It’s a bit tedious sometimes, we will learn faster methods later.