CSC 225 - Summer 2019

Recursive Algorithm Analysis

Bill Bird

Department of Computer Science
University of Victoria

May 28, 2019

Operation Counting

We have seen how to analyse algorithms by counting primitive operations. We have also seen that it is usually unnecessary to count operations to determine asymptotic worst-case running time.

When comparing two algorithms with the same asymptotic complexity, or two implementations of the same algorithm, operation counting (and other metrics such as timings) can be useful for choosing the best version.

A restricted version of operation counting, which counts specific types of operations, is also a useful heuristic for analysing algorithms.

Iterative Factorial (1)

```
1: procedure FactorialIterative(n)
2:     result ← 1
3:     for i ← 2, 3, ..., n do
4:         result ← i · result
5:     end for
6:     return result
7: end procedure
```

- **Exercise**: Give a Big-Theta expression for the worst-case running time of FactorialIterative.

Iterative Factorial (2)

```
1: procedure FactorialIterative(n)
2:     result ← 1
3:     for i ← 2, 3, ..., n do
4:         result ← i · result
5:     end for
6:     return result
7: end procedure
```

- At this point, it should be easy to see that the algorithm is $\Theta(n)$.
- One way to justify the $\Theta(n)$ running time is to observe that the loop runs for $(n-1)$ iterations, and the contents of the loop require constant time.

Iterative Factorial (3)

```
1: procedure FactorialIterative(n)
2:     result ← 1
3:     for i ← 2, 3, ..., n do
4:         result ← i · result
5:     end for
6:     return result
7: end procedure
```

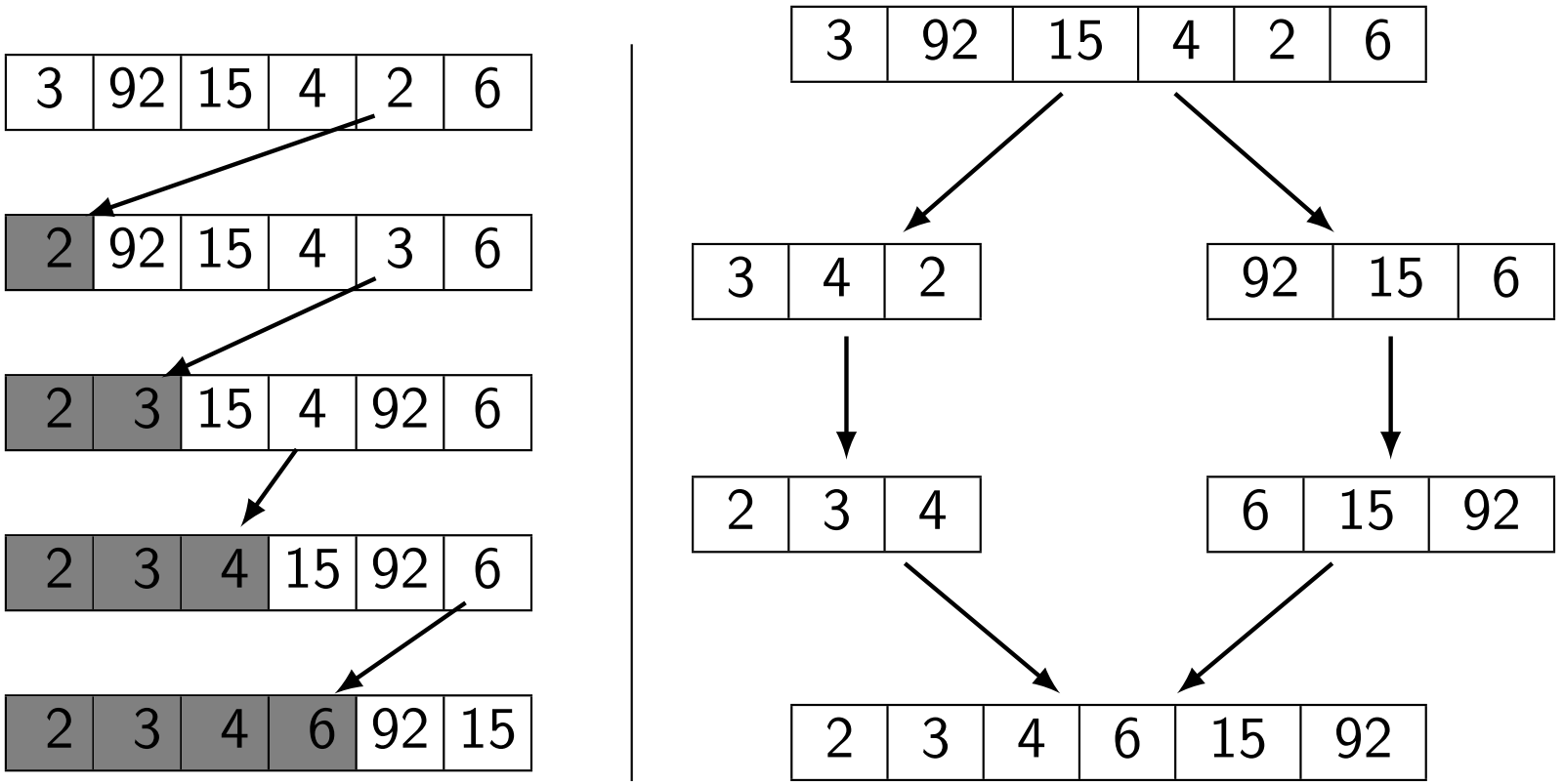
- Another option is to observe that every iteration of the loop requires one multiplication, and the function performs $\Theta(n)$ multiplications in total (due to the way that it computes the factorial).
- Adding a constant to account for lines 2 and 6, the total running time is $\Theta(n + c) = \Theta(n)$

Iterative Factorial (4)

```
1: procedure FactorialIterative(n)
2:     result ← 1
3:     for i ← 2, 3, ..., n do
4:         result ← i · result
5:     end for
6:     return result
7: end procedure
```

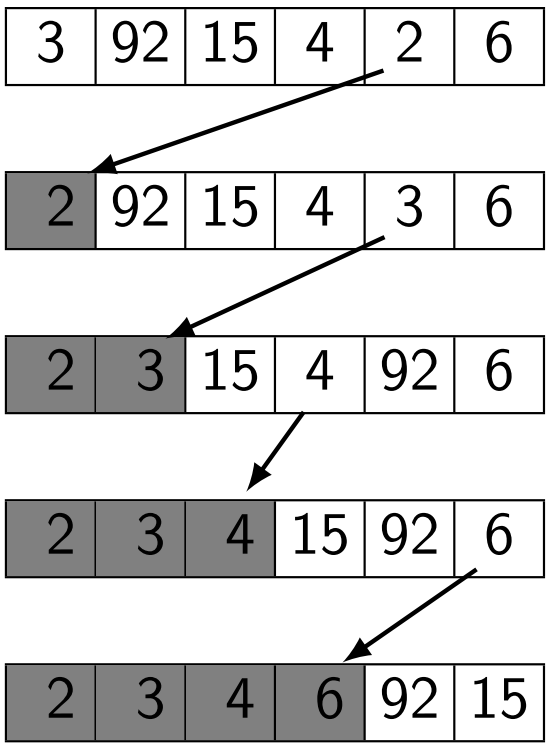
- It's often easy to analyse algorithms by arguing that a certain type of operation is significant, then counting only operations of that type.
- We call such an operation a 'proxy operation'.

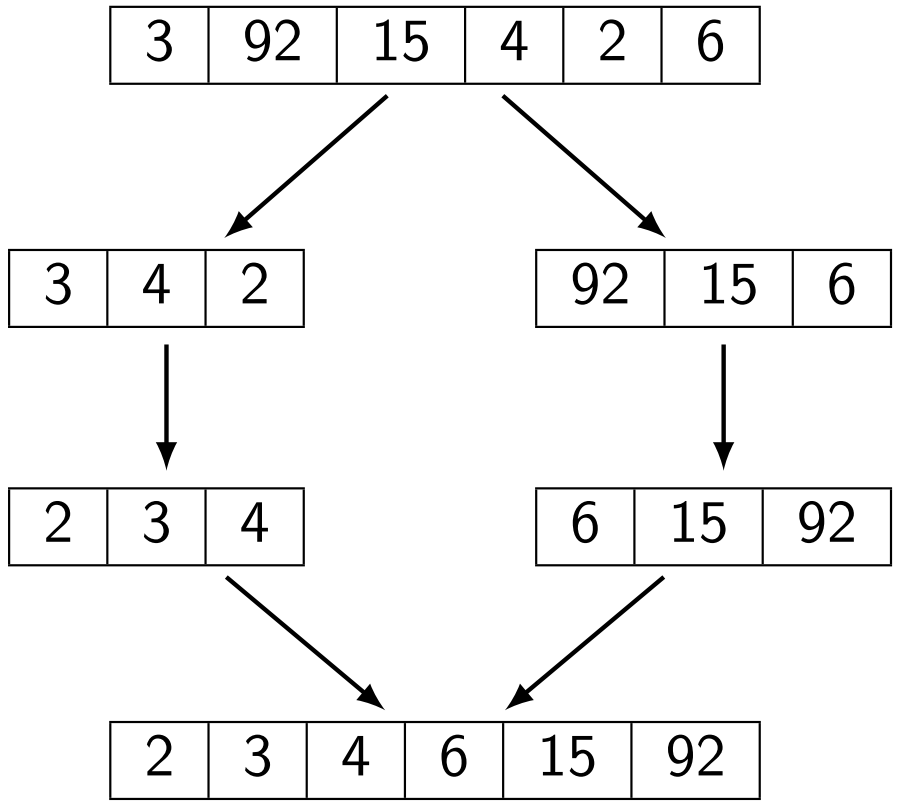
Counting Comparisons (1)



In CSC 226, a proof is given that any comparison sorting algorithm must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Counting Comparisons (2)





As a result, no comparison sorting algorithm can have $o(n \log n)$ running time.

Recursive Factorial (1)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- **Exercise**: Give a Big-Theta expression for the worst-case running time of FACTORIAL.

Recursive Factorial (2)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- Unlike the iterative version, there is no loop in the recursive function, so we can't make an argument about the number of iterations.
- **Idea**: Use multiplication as a proxy operation.

Recursive Factorial (3)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- If we count only multiplications, then the algorithm seems to require 0 operations when $n = 1$.
- **Different Idea**: Count multiplications and return statements.

Recursive Factorial (4)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- Multiplications and return statements are good choices for proxy operations, since they are present in all possible control paths.
- It turns out that only counting return statements would also work here.

Recursive Factorial (5)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- We will construct a function $T(n)$ describing the number of multiplications and return statements in the FACTORIAL function for $n \geq 1$.

Recursive Factorial (6)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- When $n = 1$, only one return statement is executed, so $T(1) = 1$.
- When $n \geq 2$, one multiplication and one return statement are executed, along with the recursive call, giving

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

Recursive Factorial (7)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

Therefore, the running time of FACTORIAL is described by the **recurrence relation**

$$\begin{aligned} T(n) &= 1 && \text{if } n = 1 \\ &= 2 + T(n-1) && \text{if } n \geq 2 \end{aligned}$$

Recursive Factorial (8)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

n	1	2	3	4	5	6
$T(n)$	1	3	5	7	9	11

Observation: For the values above, $n \leq T(n) \leq 2n$.

Recursive Factorial (9)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

n	1	2	3	4	5	6
$T(n)$	1	3	5	7	9	11

We can hypothesize that $T(n) \in \Theta(n)$, but how do we prove it?

Recursive Factorial (10)

$$\begin{aligned} T(n) &= 1 && \text{if } n = 1 \\ &= 2 + T(n-1) && \text{if } n \geq 2 \end{aligned}$$

Claim: $T(n) \leq 2n$ for all $n \geq 1$.

Proof: By induction.
Basis: When $n = 1$, $T(1) = 1$ (by definition), so the claim holds.
Induction Hypothesis: Suppose $T(n) \leq 2n$ for some $n \geq 1$.
Induction Step: Consider $n + 1$.
By the definition of the recurrence,

$$T(n+1) = 2 + T(n)$$

and by the induction hypothesis, $T(n) \leq 2n$, giving

$$T(n+1) \leq 2 + 2n = 2(n+1).$$

Therefore, the claim holds and by induction, $T(n) \leq 2n$ for all $n \geq 1$.

Recursive Factorial (11)

$$\begin{aligned} T(n) &= 1 && \text{if } n = 1 \\ &= 2 + T(n-1) && \text{if } n \geq 2 \end{aligned}$$

Claim: $T(n) \geq n$ for all $n \geq 1$.

Proof: By induction.
Basis: When $n = 1$, $T(1) = 1$ (by definition), so the claim holds.
Induction Hypothesis: Suppose $T(n) \geq n$ for some $n \geq 1$.
Induction Step: Consider $n + 1$.
By the definition of the recurrence,

$$T(n+1) = 2 + T(n)$$

and by the induction hypothesis, $T(n) \geq n$, giving

$$T(n+1) \geq 2 + n = (n+1) + 1 \geq n + 1.$$

Therefore, the claim holds and by induction, $T(n) \geq n$ for all $n \geq 1$.

Recursive Factorial (12)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- What if we counted all primitive operations instead of just multiplication and recursion?
- The recurrence relation would be

$$\begin{aligned} T(n) &= 2 && \text{if } n = 1 \\ &= 4 + T(n-1) && \text{if } n \geq 2 \end{aligned}$$

Recursive Factorial (13)

```
1: procedure FACTORIAL(n)
2:     if n = 1 then
3:         //1! = 1
4:         return 1
5:     else
6:         return n·FACTORIAL(n − 1)
7:     end if
8: end procedure
```

- We can prove that the alternate recurrence is also $\Theta(n)$.
- If you are asked to analyse a recursive algorithm, the exact recurrence you derive may vary, since different constants appear depending on which operations are counted.

ListAddK (1)

```
1:  procedure ListAddK(head, k)
2:      if head = NULL then
3:          //Adding k to an empty list produces an empty list.
4:          return NULL
5:      else
6:          new_node ← New list node
7:          new_node.value ← head.value + k
8:          new_node.next ← ListAddK(head.next, k)
9:          return new_node
10:     end if
11: end procedure
```

- **Exercise**: Determine a recurrence for the worst case running time of ListAddK (which adds a constant k to each element of a linked list).

ListAddK (2)

```
1:  procedure ListAddK(head, k)
2:      if head = NULL then
3:          //Adding k to an empty list produces an empty list.
4:          return NULL
5:      else
6:          new_node ← New list node
7:          new_node.value ← head.value + k
8:          new_node.next ← ListAddK(head.next, k)
9:          return new_node
10:     end if
11: end procedure
```

- Before anything else, we have to choose a parameter for the recurrence (since the function does not have an obvious value n).
- We will take n to be the size of the input list.

ListAddK (3)

```
1:  procedure ListAddK(head, k)
2:      if head = NULL then
3:          //Adding k to an empty list produces an empty list.
4:          return NULL
5:      else
6:          new_node ← New list node
7:          new_node.value ← head.value + k
8:          new_node.next ← ListAddK(head.next, k)
9:          return new_node
10:     end if
11: end procedure
```

- Other than the recursive call, every line of the algorithm requires constant time.
- We will derive a recurrence based on the number of return statements.

ListAddK (4)

```
1:  procedure ListAddK(head, k)
2:      if head = NULL then
3:          //Adding k to an empty list produces an empty list.
4:          return NULL
5:      else
6:          new_node ← New list node
7:          new_node.value ← head.value + k
8:          new_node.next ← ListAddK(head.next, k)
9:          return new_node
10:     end if
11: end procedure
```

- On an empty list, $n = 0$ and $T(0) = 1$.
- Otherwise (when $n \geq 1$), $T(n) = T(n-1) + 1$.

ListAddK (5)

```
1:  procedure ListAddK(head, k)
2:      if head = NULL then
3:          //Adding k to an empty list produces an empty list.
4:          return NULL
5:      else
6:          new_node ← New list node
7:          new_node.value ← head.value + k
8:          new_node.next ← ListAddK(head.next, k)
9:          return new_node
10:     end if
11: end procedure
```

- We can show that the resulting recurrence is $\Theta(n)$.

Solving Recurrences (1)

Evaluating the first few values of the recurrence

$$\begin{aligned} T(n) &= 1 && \text{if } n = 1 \\ &= 2 + T(n-1) && \text{if } n \geq 2 \end{aligned}$$

gives

n	1	2	3	4	5	6
$T(n)$	1	3	5	7	9	11

From the values in the table, it appears that $T(n) = 2n - 1$. We can prove this fact by induction. The expression

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

is a **closed form** for the recurrence, and is much easier to work with than the recursive form.

Solving Recurrences (2)

Evaluating the first few values of the recurrence

$$\begin{aligned} T(n) &= 1 && \text{if } n = 0 \\ &= T(n-1) + 1 && \text{if } n \geq 1 \end{aligned}$$

gives

n	0	1	2	3	4	5
$T(n)$	1	2	3	4	5	6

In this case, we can *guess* that $T(n) = n + 1$ based on the values above, then prove by induction that the closed form is correct.

Solving Recurrences (3)

Consider the recurrence

$$T(n) = 1 \qquad \text{if } n = 0$$

$$= T(n-1) + \frac{n(n-1)}{2} + 1 \qquad \text{if } n \geq 1$$

Evaluating the first few terms gives

$$T(0) = 1$$
$$T(1) = 2$$
$$T(2) = 4$$
$$T(3) = 8$$

Based on the data above, what can we conclude about the closed form of the recurrence?

Solving Recurrences (4)

Consider the recurrence

$$\begin{aligned} T(n) &= 1 && \text{if } n = 0 \\ &= T(n-1) + \frac{n(n-1)}{2} + 1 && \text{if } n \geq 1 \end{aligned}$$

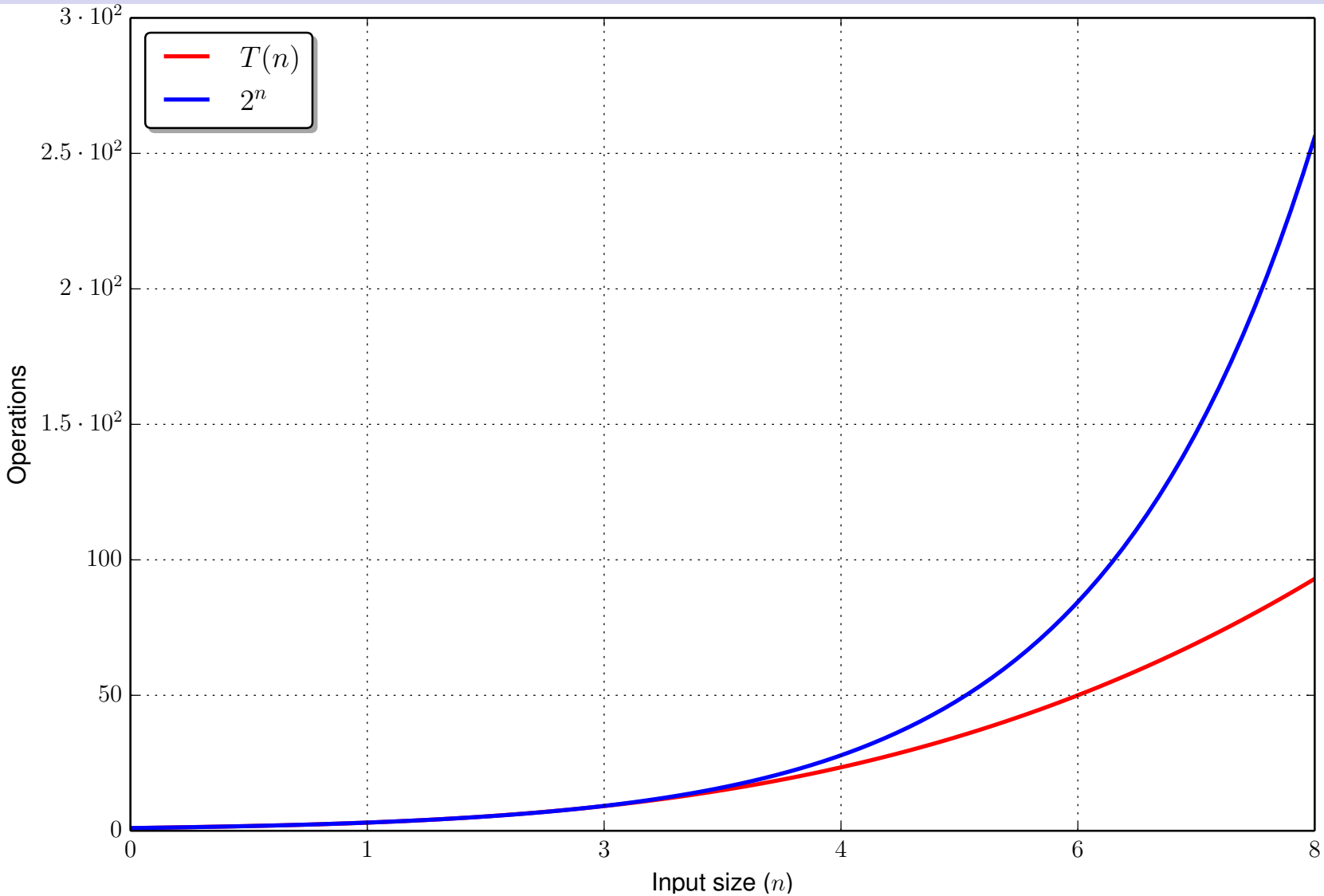
Evaluating the first few terms gives

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 2 \\ T(2) &= 4 \\ T(3) &= 8 \end{aligned}$$

Based on the data above, what can we conclude about the closed form of the recurrence?

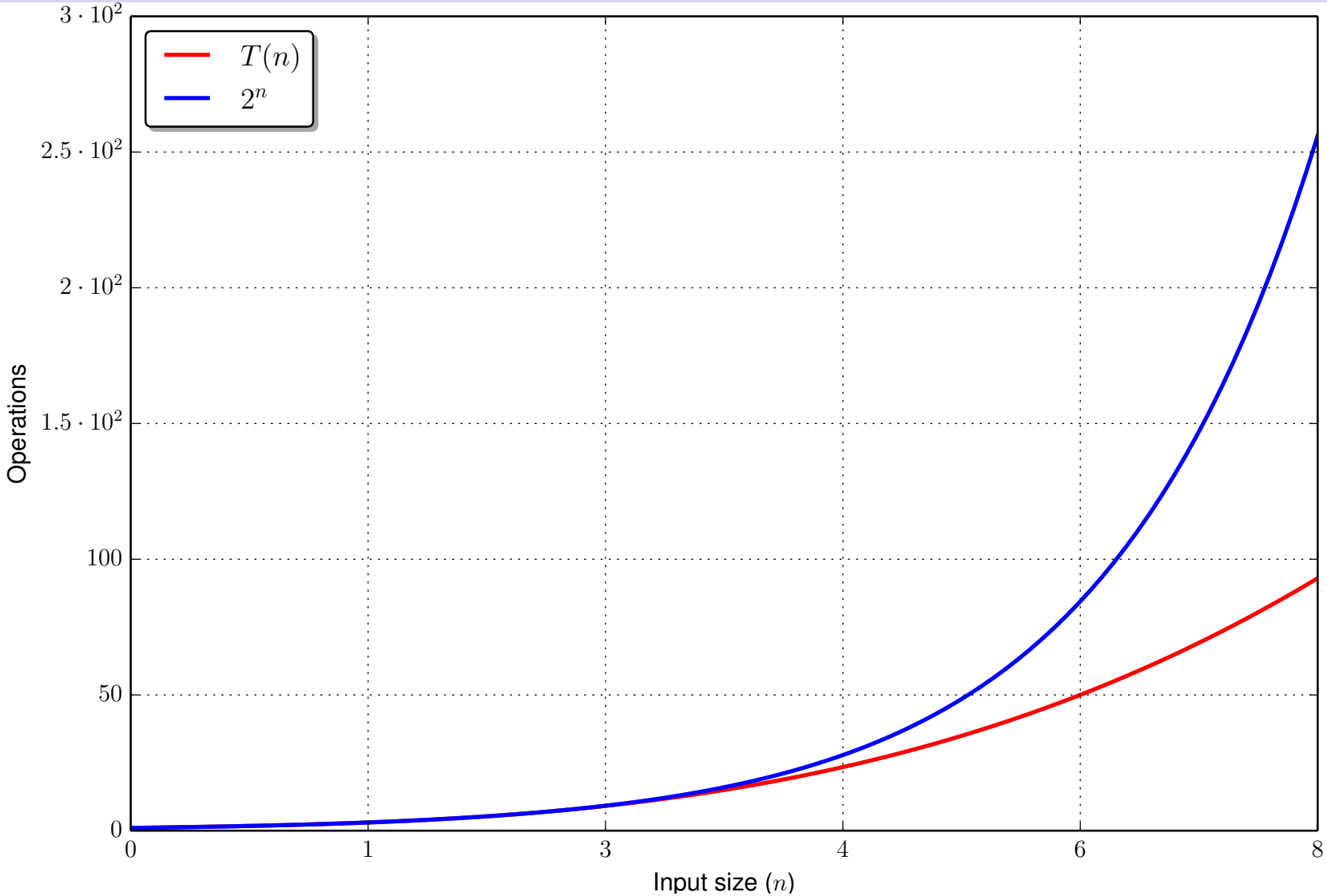
NOTHING.

Solving Recurrences (5)



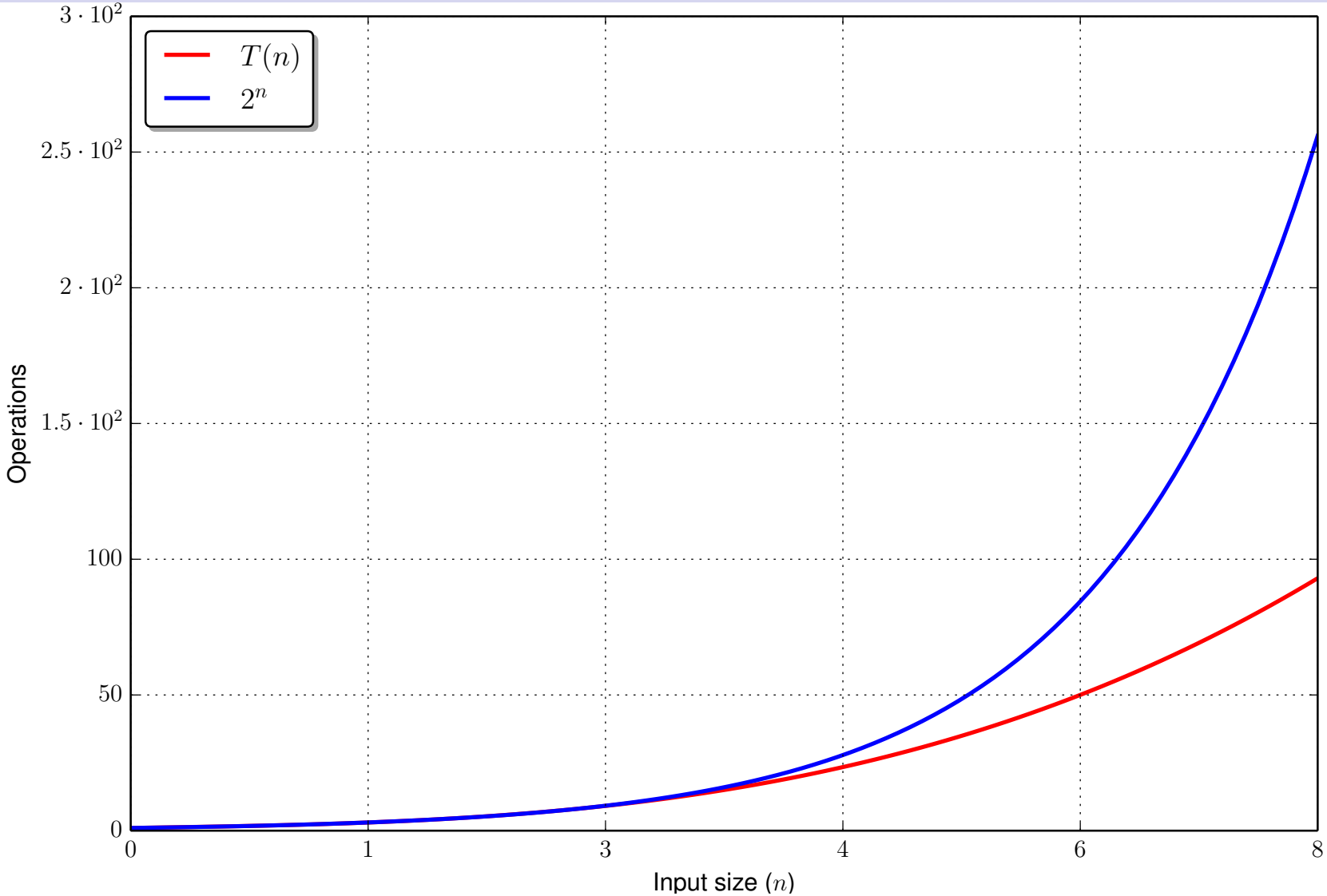
It is reasonable to hypothesize that $T(n) = 2^n$ based on the initial values.

Solving Recurrences (6)



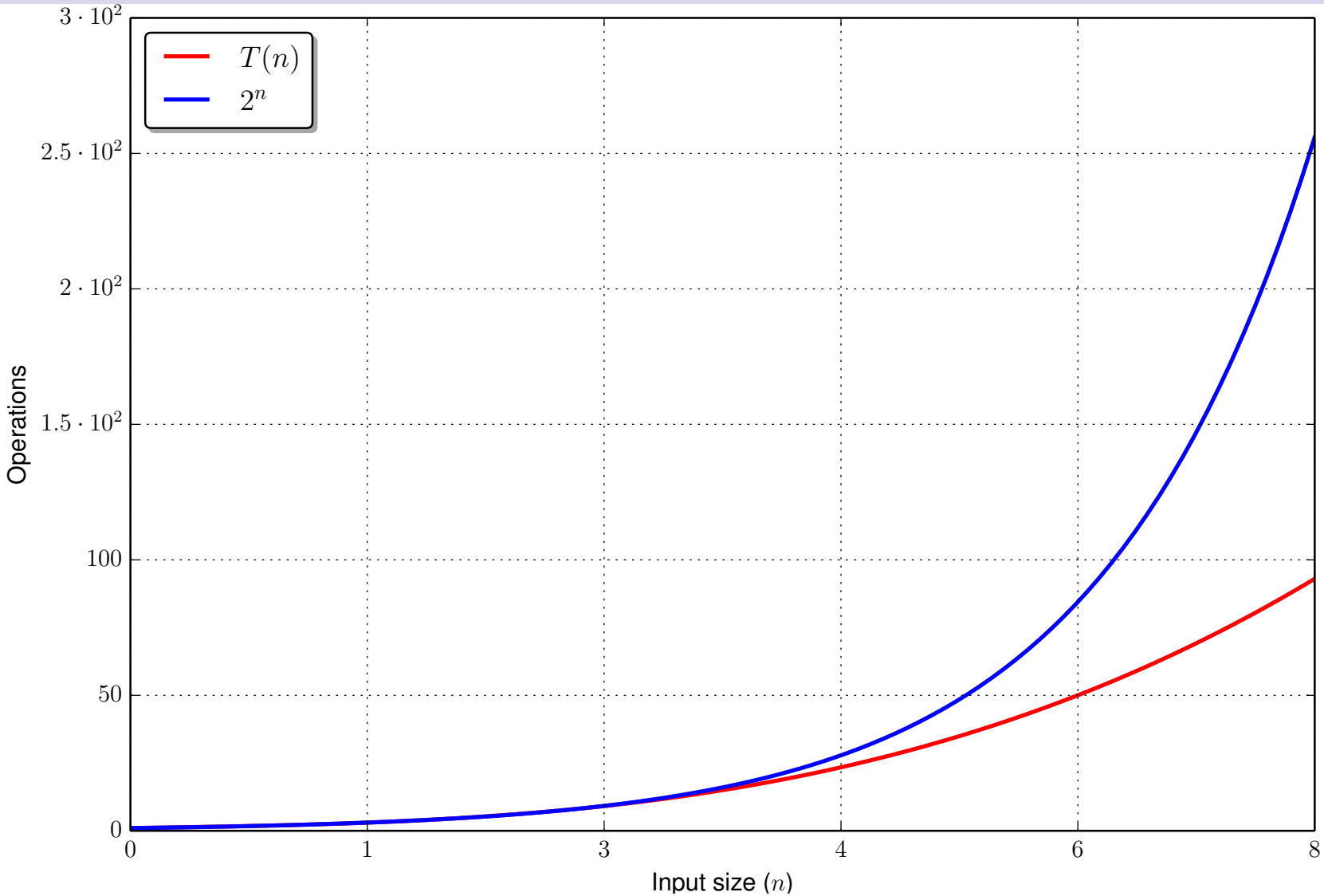
However, $T(n)$ diverges from 2^n when $n \geq 4$.

Solving Recurrences (7)



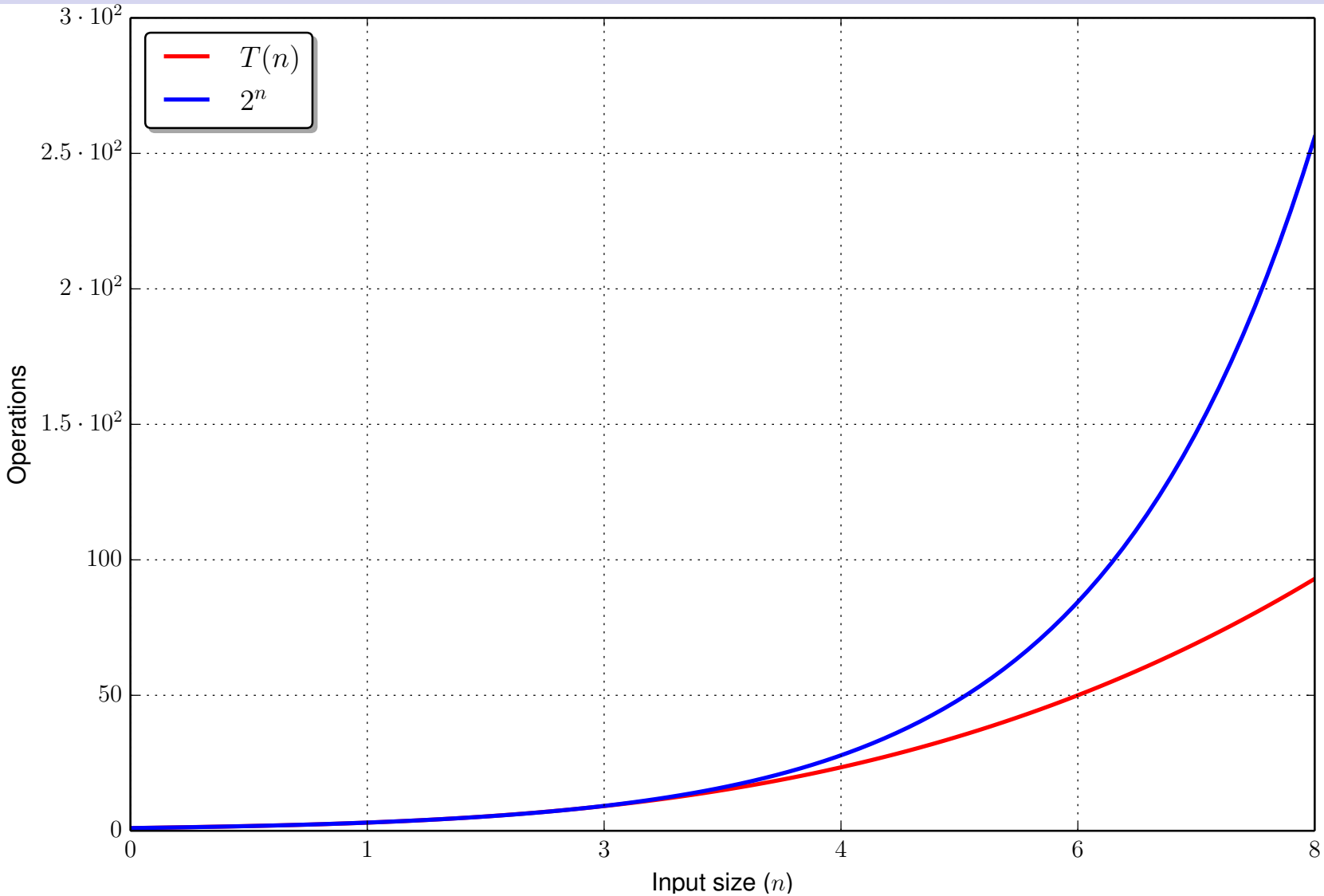
A sufficiently complicated recurrence could match 2^n for the first 10^6 terms, then diverge.

Solving Recurrences (8)



If you guess a closed form for $T(n)$ based on initial values, it is necessary to prove the closed form to be correct.

Solving Recurrences (9)



We will see a general technique for finding closed forms for recurrences in the next lecture.