Recurrence Relations and Growth Rates

Solving recurrence relations, running times, and big-O notation

Last modified: 2026-11-12 01:23:00 (PST)

Source

The topics of this chapter follow the course calendar and Kameron Decker Harris's Fall 2026 lecture notes, which name recurrence relations and order of growth as the topics after recursive algorithms. One idea comes from those notes: the running time of a recursive function is the running time of its recursive call plus the work outside it.
Everything else is new: the exercises, the examples, the algorithms `pal_loop`, `table` and `pairs`, the Racket functions, the theorems and their proofs. The algorithms `pal` and `root` come from earlier chapters of these notes.

Recurrence relations

Exercise 1 (Double and add a power of 2)**.** A sequence (Definition: Sequence) starts with $b_0 = 0$, and for every integer (Definition: Integers) $n \geq 1$ the term b_n is twice the term before it, plus 2^n. Find b_1, b_2, b_3, b_4 and b_5.

Solution

Solution 1. Each term comes from the one before it:

$$\begin{aligned} b_1 &= 2 \cdot 0 + 2^1 = 2, \\ b_2 &= 2 \cdot 2 + 2^2 = 8, \\ b_3 &= 2 \cdot 8 + 2^3 = 24, \\ b_4 &= 2 \cdot 24 + 2^4 = 64, \\ b_5 &= 2 \cdot 64 + 2^5 = 160. \end{aligned}$$

Definition 1 (Recurrence relation)**.** A **recurrence relation** (Wikipedia contributors 2026e) is the recurrence of a recursively defined sequence (Definition: Recursively defined sequence), that is, its recursive rule (Definition: Recursive definition) written as an equation, which gives the term a_n in terms of the m terms $a_{n-1}, \ldots, a_{n-m}$ just before it, for every integer $n \geq n_0 + m$.

Example 1 (The sequence b_n, defined recursively)**.** The sequence of Exercise 1 is recursively defined (Definition: Recursively defined sequence) with $n_0 = 0$ and $m = 1$. Its initial term is $b_0 = 0$, and its recurrence relation (Definition 1) is

$$b_n = 2b_{n-1} + 2^n \quad \text{for every integer } n \geq 1.$$

In Racket, where `(expt 2 n)` is 2^n, the recurrence relation becomes a recursive function:

```
(define (b n)
  (if (= n 0)
      0
      (+ (* 2 (b (- n 1))) (expt 2 n))))
```

`(b 5)` leads to calls of `(b 4)`, `(b 3)`, `(b 2)`, `(b 1)` and `(b 0)` in turn, and returns b_5, as in Solution 1:

```
(b 5)
; 160
```

Definition 2 (Closed form)**.** A **closed form** (Wikipedia contributors 2026b) for a sequence $a_{n_0}, a_{n_0+1}, \ldots$ (Definition: Sequence) is a formula for a_n that uses only n and fixed numbers, combined by a fixed number of operations such as addition, multiplication, powers and floors (Definition: Floor), and that uses no other term of the sequence.

Example 2 (Closed forms and recurrence relations)**.** The double-minus-one sequence of Example: The double-minus-one sequence, defined recursively has the recurrence relation (Definition 1) $a_n = 2a_{n-1} - 1$, which is not a closed form (Definition 2), since it uses the term a_{n-1}. By Theorem: A formula for the double-minus-one sequence, $a_n = 2^n + 1$ is a closed form for the same sequence: it gives $a_{50} = 2^{50} + 1$ without finding a_1 through a_{49} first. Finding a closed form for a sequence from its recurrence relation is called *solving* the recurrence relation.

Solving recurrence relations by iteration

Exercise 2 (Using the recurrence relation three times)**.** Let $b_0, b_1, \ldots$ be the sequence of Example 1, and let $n \geq 3$. Starting from $b_n = 2b_{n-1} + 2^n$, replace b_{n-1} by $2b_{n-2} + 2^{n-1}$, and then replace b_{n-2} by $2b_{n-3} + 2^{n-2}$, to write b_n in terms of b_{n-3}.

Solution

Solution 2. Replacing b_{n-1} gives b_n in terms of b_{n-2}:

$$\begin{aligned} b_n &= 2(2b_{n-2} + 2^{n-1}) + 2^n \\ &= 4b_{n-2} + 2^n + 2^n \\ &= 4b_{n-2} + 2 \cdot 2^n. \end{aligned}$$

Replacing b_{n-2} then gives b_n in terms of b_{n-3}:

$$\begin{aligned} b_n &= 4(2b_{n-3} + 2^{n-2}) + 2 \cdot 2^n \\ &= 8b_{n-3} + 2^n + 2 \cdot 2^n \\ &= 8b_{n-3} + 3 \cdot 2^n. \end{aligned}$$

Definition 3 (Iteration (unrolling))**.** Solving a recurrence relation (Definition 1) by **iteration**, also called **unrolling**, finds a closed form (Definition 2) for its sequence in three steps:

1. use the recurrence relation again and again to write a_n in terms of a_{n-1}, then a_{n-2}, then a_{n-3}, and so on;
2. guess, from the pattern, a formula for a_n in terms of a_{n-k} for every number k of steps;
3. choose k so that a_{n-k} is an initial term, and put in its value.

Example 3 (Unrolling $b_n = 2b_{n-1} + 2^n$)**.** The recurrence relation and Solution 2 give the first steps of iteration (Definition 3) for the sequence of Example 1:

$$\begin{aligned} b_n &= 2b_{n-1} + 1 \cdot 2^n, \\ b_n &= 4b_{n-2} + 2 \cdot 2^n, \\ b_n &= 8b_{n-3} + 3 \cdot 2^n. \end{aligned}$$

After k steps the factor in front of b_{n-k} is 2^k, and the multiple of 2^n is k, so the pattern suggests

$$b_n = 2^k b_{n-k} + k \cdot 2^n \quad \text{for } k = 1, 2, \dots, n.$$

The initial term is b_0, which is b_{n-k} when $k = n$. With $k = n$ and $b_0 = 0$,

$$\begin{aligned} b_n &= 2^n b_0 + n \cdot 2^n \\ &= n \cdot 2^n. \end{aligned}$$

This agrees with Solution 1: for example, $5 \cdot 2^5 = 160 = b_5$.

Remark 1 (Iteration gives a guess). In Example 3, the formula for general k came from the three cases $k = 1, 2, 3$. Nothing proved that the pattern continues, so the closed form $b_n = n \cdot 2^n$ is still a guess. Induction (Definition: Proof by induction) turns such a guess into a theorem, with the recurrence relation in the inductive step.

Checking a closed form by induction

Exercise 3 (Checking a guess)**.** A sequence starts with $e_0 = 1$, and has the recurrence relation (Definition 1)

$$e_n = 3e_{n-1} + 2^n \quad \text{for every integer } n \geq 1.$$

Someone guesses the closed form (Definition 2) $e_n = 3^{n+1} - 2^{n+1}$. Find e_1, e_2 and e_3 from the recurrence relation, and check the guess for $n = 0, 1, 2, 3$.

Solution

Solution 3. From the recurrence relation, $e_1 = 3 \cdot 1 + 2 = 5$, $e_2 = 3 \cdot 5 + 4 = 19$ and $e_3 = 3 \cdot 19 + 8 = 65$.

n	0	1	2	3
e_n	1	5	19	65
3^{n+1}	3	9	27	81
2^{n+1}	2	4	8	16
$3^{n+1} - 2^{n+1}$	1	5	19	65

The guess is right for $n = 0, 1, 2, 3$. Four cases do not prove it for every n.

Theorem 1 (A closed form for $b_n = 2b_{n-1} + 2^n$)**.** *Let $b_0, b_1, \ldots$ be the sequence with $b_0 = 0$ and $b_n = 2b_{n-1} + 2^n$ for every integer $n \geq 1$ (Example 1). Then $b_n = n \cdot 2^n$ for every non-negative integer (Definition: Non-negative integers) n.*

Proof

Proof. We use induction (Definition: Proof by induction), following Remark: The shape of a proof by induction with $n_0 = 0$. Let $P(n)$ be the equation $b_n = n \cdot 2^n$.
Base case. $b_0 = 0 = 0 \cdot 2^0$.
Inductive step. Let $k \geq 0$ be an integer, and suppose $P(k)$: $b_k = k \cdot 2^k$. The goal $P(k+1)$ is $b_{k+1} = (k+1) \cdot 2^{k+1}$. Since $k + 1 \geq 1$, the recurrence relation gives b_{k+1}, and

$$\begin{aligned} b_{k+1} &= 2b_k + 2^{k+1} && \text{(recurrence relation)} \\ &= 2 \cdot k \cdot 2^k + 2^{k+1} && \text{(induction hypothesis)} \\ &= k \cdot 2^{k+1} + 2^{k+1} \\ &= (k+1) \cdot 2^{k+1}. \end{aligned}$$

So $P(k+1)$ holds, and by induction $b_n = n \cdot 2^n$ for every non-negative integer n. □

Theorem 2 (A closed form for $e_n = 3e_{n-1} + 2^n$)**.** *Let $e_0, e_1, \ldots$ be the sequence with $e_0 = 1$ and $e_n = 3e_{n-1} + 2^n$ for every integer $n \geq 1$. Then $e_n = 3^{n+1} - 2^{n+1}$, as guessed in Exercise 3, for every non-negative integer (Definition: Non-negative integers) n.*

Proof

Proof. We use induction (Definition: Proof by induction), following Remark: The shape of a proof by induction with $n_0 = 0$. Let $P(n)$ be the equation $e_n = 3^{n+1} - 2^{n+1}$.
Base case. $e_0 = 1 = 3 - 2 = 3^1 - 2^1$.
Inductive step. Let $k \geq 0$ be an integer, and suppose $P(k)$: $e_k = 3^{k+1} - 2^{k+1}$. The goal $P(k+1)$ is $e_{k+1} = 3^{k+2} - 2^{k+2}$. Since $k + 1 \geq 1$, the recurrence relation gives e_{k+1}, and

$$\begin{aligned} e_{k+1} &= 3e_k + 2^{k+1} && \text{(recurrence relation)} \\ &= 3(3^{k+1} - 2^{k+1}) + 2^{k+1} && \text{(induction hypothesis)} \\ &= 3^{k+2} - 3 \cdot 2^{k+1} + 2^{k+1} \\ &= 3^{k+2} - 2 \cdot 2^{k+1} \\ &= 3^{k+2} - 2^{k+2}. \end{aligned}$$

So $P(k+1)$ holds, and by induction $e_n = 3^{n+1} - 2^{n+1}$ for every non-negative integer n. □

Example 4 (A guess that fails)**.** The sequence of Exercise 3 starts $e_0 = 1$, $e_1 = 5$, which are 5^0 and 5^1. So one might guess $e_n = 5^n$. The guess fails at $n = 2$, since $e_2 = 19$ and $5^2 = 25$.
An attempt to prove the guess by induction (Definition: Proof by induction) shows where it goes wrong. Supposing $e_k = 5^k$, the recurrence relation gives

$$\begin{aligned} e_{k+1} &= 3e_k + 2^{k+1} \\ &= 3 \cdot 5^k + 2^{k+1}, \end{aligned}$$

and the inductive step needs this to equal $5^{k+1} = 5 \cdot 5^k$, which happens only when $2^{k+1} = 2 \cdot 5^k$, that is, only when $k = 0$.

Remark 2 (Finding a closed form, then proving it). Iteration (Definition 3) and checking small cases, as in Exercise 3, find a closed form (Definition 2) to try. Induction proves it, as in Theorem 1 and Theorem 2, or shows that it is wrong, as in Example 4. A proof by induction needs the closed form before it starts: it can check a formula, but it cannot find one.

Running times of recursive algorithms

Exercise 4 (How many calls does `pal` make?)**.** Let `pal` be the algorithm of Exercise: Is it a palindrome?. How many calls of `pal`, including the first, does `pal(abbaabba)` make?

Solution

Solution 4. `pal(abbaabba)` calls `pal(bbaabb)`, which calls `pal(baab)`, which calls `pal(aa)`, which calls `pal` on the empty string, which returns true. That is 5 calls, on strings of 8, 6, 4, 2 and 0 symbols: each call removes the first and last symbols.

Remark 3 (The size of an input and a step). To count the work an algorithm (Definition: Algorithm) does, we first fix two things:

- the *size* of an input, such as the number of symbols in a string or the number of entries in an array;
- what one *step* is, such as one comparison, one arithmetic operation or one call of a function.

Definition 4 (Running time)**.** Fix, for an algorithm (Definition: Algorithm), the size of an input and what one step is (Remark 3). The **running time** (Wikipedia contributors 2026f) of the algorithm is the function T (Definition: Function, map, mapping, transformation) whose value $T(n)$ is the largest number of steps that the algorithm takes on any input of size n.

Example 5 (The running time of `pal`)**.** Take the size (Remark 3) of the input of `pal` (Exercise: Is it a palindrome?) to be the number n of symbols in the string, and count each call of `pal` as one step, since apart from its recursive call each call does at most two comparisons. This assumes that the call on `s[2..n-1]` passes the shorter string as a pair of positions in `s`, which takes a fixed number of operations; copying its $n-2$ symbols into a new string would add about n steps to every call. Let $T(n)$ be the running time (Definition 4). The running time of a call is the running time of its recursive call plus the work outside it.

- When $n \leq 1$, `pal` returns without calling itself, so $T(0) = T(1) = 1$.
- When $n \geq 2$, the call that takes longest is one whose first and last symbols match, since otherwise `pal` returns at once. It calls `pal` on a string of $n-2$ symbols, which can be any such string, so $T(n) = T(n-2) + 1$.

Iteration (Definition 3) gives

$$\begin{aligned} T(n) &= T(n-2) + 1 \\ &= T(n-4) + 2 \\ &= T(n-6) + 3, \end{aligned}$$

and in general $T(n) = T(n-2k) + k$, as long as $n - 2k \geq 0$. To reach an initial term, take $k = \lfloor n/2 \rfloor$ (Definition: Floor): then $n-2k$ is 0 when n is even and 1 when n is odd, and either way $T(n-2k) = 1$. So the guess is

$$T(n) = \lfloor n/2 \rfloor + 1,$$

which gives $T(8) = 5$, as in Solution 4. Theorem 3 proves it.

Theorem 3 (A closed form for the running time of `pal`)**.** *Let $T(0) = T(1) = 1$ and $T(n) = T(n-2) + 1$ for every integer $n \geq 2$, as in Example 5. Then $T(n) = \lfloor n/2 \rfloor + 1$ (Definition: Floor) for every non-negative integer (Definition: Non-negative integers) n.*

Proof

Proof. The recurrence relation uses $T(n-2)$, not $T(n-1)$, so we use strong induction (Definition: Proof by strong induction). Let $n \geq 0$ be an integer, and suppose $T(m) = \lfloor m/2 \rfloor + 1$ for every integer m with $0 \leq m < n$.
If $n \leq 1$, then $T(n) = 1 = 0 + 1 = \lfloor n/2 \rfloor + 1$.
If $n \geq 2$, then $0 \leq n-2 < n$, and $\lfloor (n-2)/2 \rfloor = \lfloor n/2 - 1 \rfloor = \lfloor n/2 \rfloor - 1$, since subtracting the integer 1 from a real number subtracts 1 from its floor. So

$$\begin{aligned} T(n) &= T(n-2) + 1 && \text{(recurrence relation)} \\ &= \lfloor (n-2)/2 \rfloor + 1 + 1 && \text{(strong induction hypothesis)} \\ &= \lfloor n/2 \rfloor - 1 + 2 \\ &= \lfloor n/2 \rfloor + 1. \end{aligned}$$

So the formula holds for n, and by strong induction it holds for every non-negative integer n. □

Two recursive calls

Exercise 5 (A function that calls itself twice)**.** This Racket function computes the double-minus-one sequence of Example: The double-minus-one sequence, defined recursively, but it finds $2a_{n-1}$ as $a_{n-1} + a_{n-1}$, with two recursive calls:

```
(define (dm n)
  (if (= n 0)
      2
      (- (+ (dm (- n 1)) (dm (- n 1))) 1)))
```

Write down every call of `dm` that running `(dm 2)` involves, counting `(dm 2)` itself, and the value that each call returns.

Solution

Solution 5. `(dm 2)` calls `(dm 1)` twice, and each of those calls `(dm 0)` twice:

```
(dm 2)
  (dm 1)
    (dm 0)  returns 2
    (dm 0)  returns 2
  returns 2 + 2 - 1 = 3
  (dm 1)
    (dm 0)  returns 2
    (dm 0)  returns 2
  returns 2 + 2 - 1 = 3
returns 3 + 3 - 1 = 5
```

That is 7 calls, counting `(dm 2)` itself, and 4 of them are calls of `(dm 0)`. `(dm 2)` returns 5, which is a_2. The two calls of `(dm 1)` do exactly the same work.
Racket returns the same value:

```
(dm 2)
; 5
```

Theorem 4 (Two calls of size $n-1$ give 2^n base cases)**.** *For every non-negative integer (Definition: Non-negative integers) n, the calls that `(dm n)` of Exercise 5 makes include exactly 2^n calls of `(dm 0)`, counting `(dm 0)` itself when $n = 0$. So, counting each call as one step, the running time (Definition 4) of `dm` satisfies $T(n) \geq 2^n$.*

Proof

Proof. Let $B(n)$ be the number of calls of `(dm 0)` among the calls that `(dm n)` makes. Every call of `(dm n)` does the same work, so $B(n)$ depends only on n. When $n \geq 1$, `(dm n)` is not itself a call of `(dm 0)`, and it calls `(dm (- n 1))` twice, as in Solution 5, so

$$B(0) = 1, \qquad B(n) = 2B(n-1) \quad \text{for } n \geq 1.$$

We use induction (Definition: Proof by induction), following Remark: The shape of a proof by induction with $n_0 = 0$, with $P(n)$ the equation $B(n) = 2^n$.
Base case. $B(0) = 1 = 2^0$.
Inductive step. Let $k \geq 0$ be an integer, and suppose $B(k) = 2^k$. Then $B(k+1) = 2B(k) = 2 \cdot 2^k = 2^{k+1}$.
So $B(n) = 2^n$ for every non-negative integer n. Each of these calls is a step, so `(dm n)` takes at least 2^n steps. □

Remark 4 (One call instead of two). The function `dm` of Exercise 5 finds the same value a_{n-1} twice. Writing the recursive rule of Example: The double-minus-one sequence, defined recursively directly, with one recursive call,

```
(define (dm1 n)
  (if (= n 0)
      2
      (- (* 2 (dm1 (- n 1))) 1)))
```

returns the same values, as the first six show:

```
(map dm (range 6))
; '(2 3 5 9 17 33)
(map dm1 (range 6))
; '(2 3 5 9 17 33)
```

`(range 6)` is the list `'(0 1 2 3 4 5)`. Each call of `dm1` makes only one recursive call, so its calls form a single chain instead of a tree that doubles at every level. By Theorem 4, `(dm 30)` makes more than a billion calls, since $2^{30} = 1073741824$.

Asymptotic notation

Exercise 6 (Which running time is better?)**.** One algorithm takes $100n$ steps on an input of size n, and another takes n^2 steps. Compare $100n$ with n^2 for $n = 10, 50, 100, 200, 1000$. Which algorithm takes fewer steps on small inputs, and which on large ones?

Solution

Solution 6.

n	10	50	100	200	1000
$100n$	1000	5000	10000	20000	100000
n^2	100	2500	10000	40000	1000000

The second algorithm takes fewer steps when $n < 100$, and the two take the same number at $n = 100$. For every $n > 100$, $n^2 = n \cdot n > 100n$, so the first algorithm takes fewer steps, and the ratio $n^2/(100n) = n/100$ grows without bound. The factor 100 decides which is better on small inputs; the powers n and n^2 decide which is better on large ones.

Definition 5 (Big-O)**.** Let f and g be functions from the natural numbers to the real numbers. Then $f(n)$ is **big-O** of $g(n)$ (Wikipedia contributors 2026a), written $f(n) = O(g(n))$, when there exist a real number $c > 0$ and a natural number N such that, for every natural number $n \geq N$,

$$|f(n)| \leq c\,|g(n)|.$$

In words, from some point N on, the absolute value of $f(n)$ is at most a fixed multiple c of the absolute value of $g(n)$. The "there exist" is an existential quantifier, and the "for every" is a universal quantifier.

Definition 6 (Witnesses)**.** A real number $c > 0$ and a natural number N for which the inequality in Definition 5 holds for every natural number (Definition: Natural numbers (our convention)) $n \geq N$ are a pair of **witnesses** for $f(n) = O(g(n))$. Witnesses for $f(n) = \Omega(g(n))$ are defined in the same way, from Definition 7. Some books write n_0 for N.

Example 6 (The running time of `pal` is $O(n)$)**.** By Example 5, `pal` takes $T(n) = \lfloor n/2 \rfloor + 1$ steps on a string of n symbols. For every natural number $n \geq 2$,

$$\begin{aligned} |T(n)| &= \lfloor n/2 \rfloor + 1 && (T(n) > 0) \\ &\leq n/2 + 1 && (\text{definition of the floor}) \\ &\leq n/2 + n/2 && (1 \leq n/2, \text{ since } n \geq 2) \\ &= n, \end{aligned}$$

so $c = 1$ and $N = 2$ are witnesses (Definition 6) for $T(n) = O(n)$ (Definition 5). At $n = 1$, also $T(1) = 1 \leq 1$, so $c = 1$ and $N = 1$ are witnesses too. Witnesses are not unique: any larger c or N works as well.

Example 7 (Finding witnesses for $2n^3 + 40 = O(n^3)$)**.** To prove $2n^3 + 40 = O(n^3)$ (Definition 5), we look for witnesses (Definition 6) c and N. Since $2n^3 + 40 > 2n^3$, the term $2n^3$ needs $c > 2$. With $c = 3$, the inequality $2n^3 + 40 \leq 3n^3$ is the same as $40 \leq n^3$, which holds when $n \geq 4$, since $4^3 = 64$, but not when $n = 3$, since $3^3 = 27$. So for every natural number $n \geq 4$,

$$\begin{aligned} \left|2n^3 + 40\right| &= 2n^3 + 40 && (2n^3 + 40 > 0) \\ &\leq 2n^3 + n^3 && (40 \leq 64 \leq n^3, \text{ since } n \geq 4) \\ &= 3\left|n^3\right|, \end{aligned}$$

and $c = 3$, $N = 4$ are witnesses. A larger c allows a smaller N: since $40 \leq 40n^3$ for every $n \geq 1$, the pair $c = 42$, $N = 1$ also works.

Definition 7 (Big-Omega)**.** Let f and g be functions from the natural numbers to the real numbers. Then $f(n)$ is **big-Omega** of $g(n)$ (Wikipedia contributors 2026a), written $f(n) = \Omega(g(n))$, when there exist a real number $c > 0$ and a natural number N such that, for every natural number $n \geq N$,

$$|f(n)| \geq c\,|g(n)|.$$

In words, from some point N on, the absolute value of $f(n)$ is at least a fixed positive multiple c of the absolute value of $g(n)$. Dividing by c shows that $f(n) = \Omega(g(n))$ exactly when $g(n) = O(f(n))$ in the sense of Definition 5.

Example 8 ($2n^3 + 40 = \Omega(n^3)$)**.** For every natural number $n \geq 1$,

$$\begin{aligned} \left|2n^3 + 40\right| &= 2n^3 + 40 && (2n^3 + 40 > 0) \\ &\geq 2n^3 && (40 \geq 0) \\ &= 2\left|n^3\right|, \end{aligned}$$

so $c = 2$ and $N = 1$ are witnesses (Definition 6) for $2n^3 + 40 = \Omega(n^3)$ (Definition 7).

Definition 8 (Big-Theta)**.** Let f and g be functions from the natural numbers to the real numbers. Then $f(n)$ is **big-Theta** of $g(n)$ (Wikipedia contributors 2026a), written $f(n) = \Theta(g(n))$, when both $f(n) = O(g(n))$ (big-O) and $f(n) = \Omega(g(n))$ (big-Omega). In words, $f(n)$ and $g(n)$ are within fixed multiples of each other, from some point on.

Example 9 ($2n^3 + 40 = \Theta(n^3)$)**.** By Example 7, $2n^3 + 40 = O(n^3)$, and by Example 8, $2n^3 + 40 = \Omega(n^3)$. So $2n^3 + 40 = \Theta(n^3)$ by Definition 8. The constant 40 and the factor 2 do not change how fast $2n^3 + 40$ grows.

Theorem 5 (Sums of big-O functions)**.** *Let f, g and h be functions (Definition: Function, map, mapping, transformation) from the natural numbers (Definition: Natural numbers (our convention)) to the real numbers (Definition: Real numbers). If $f(n) = O(h(n))$ and $g(n) = O(h(n))$ (Definition 5), then $f(n) + g(n) = O(h(n))$.*

Proof

Proof. Let c_1, N_1 be witnesses (Definition 6) for $f(n) = O(h(n))$, and c_2, N_2 witnesses for $g(n) = O(h(n))$. Let N be the larger of N_1 and N_2. The triangle inequality (Wikipedia contributors 2026g) says that $|x + y| \leq |x| + |y|$ for all real numbers x and y. For every natural number $n \geq N$, both $n \geq N_1$ and $n \geq N_2$ hold, so

$$\begin{aligned} |f(n) + g(n)| &\leq |f(n)| + |g(n)| && \text{(triangle inequality)} \\ &\leq c_1|h(n)| + c_2|h(n)| && (n \geq N_1 \text{ and } n \geq N_2) \\ &= (c_1 + c_2)|h(n)|. \end{aligned}$$

So $c_1 + c_2$ and N are witnesses for $f(n) + g(n) = O(h(n))$. □

Disproving a bound

Exercise 7 (Is $n \cdot 2^n$ big-O of 2^n?)**.** By Theorem 1, the sequence of Example 1 has $b_n = n \cdot 2^n$. Show that $n \cdot 2^n = O(2^n)$ (Definition 5) is false.

Solution

Solution 7. We use proof by contradiction (Definition: Proof by contradiction). Suppose $n \cdot 2^n = O(2^n)$, with witnesses (Definition 6) $c > 0$ and N. Then $n \cdot 2^n \leq c \cdot 2^n$ for every natural number $n \geq N$, and dividing by $2^n > 0$ gives $n \leq c$ for every natural number $n \geq N$.
Let $n = N + \lfloor c \rfloor + 1$, where $\lfloor c \rfloor$ is the floor (Definition: Floor) of c. Then n is a natural number with $n \geq N$, so $n \leq c$. But $\lfloor c \rfloor + 1 > c$ by the definition of the floor, and $N \geq 1$, so $n > c$. This is a contradiction, so $n \cdot 2^n = O(2^n)$ is false.

Remark 5 (How to disprove a big-O bound). The statement $f(n) = O(g(n))$ (Definition 5) says that there exist $c > 0$ and N such that every $n \geq N$ has $|f(n)| \leq c|g(n)|$. Its negation says that for every $c > 0$ and every N, some natural number $n \geq N$ has $|f(n)| > c|g(n)|$, by the rules for negating the existential (Definition: Existential quantifier) and universal (Definition: Universal quantifier) quantifiers. So a disproof cannot pick c and N: like Solution 7, it takes any c and N and finds such an n, which depends on c and N. Checking that the inequality fails for a few chosen c and N is not a disproof.

Common growth rates

Exercise 8 (Seven functions at $n = 16$)**.** Here $\log_2 n$ is the power to which 2 must be raised to give n, so $\log_2 16 = 4$ since $2^4 = 16$, and $n!$ is the product $1 \cdot 2 \cdots n$. Find the values at $n = 16$ of

- 1,
- $\log_2 n$,
- n,
- $n \log_2 n$,
- n^2,
- 2^n,
- $n!$,

and put the values in increasing order.

Solution

Solution 8.

1	$\log_2 n$	n	$n \log_2 n$	n^2	2^n	$n!$
1	4	16	64	256	65536	20922789888000

The values are already in increasing order. For $n!$, $1 \cdot 2 \cdots 16 = 20922789888000$, about 2×10^{13}.

Definition 9 (Logarithm)**.** Let $b > 1$ be a real number (Definition: Real numbers). For every real number $x > 0$, the **logarithm** of x to the base b (Wikipedia contributors 2026d), written $\log_b x$, is the real number y with $b^y = x$ (Definition: Power with a real exponent).

Example 10 (Some logarithms)**.** By Definition 9,

- $\log_2 16 = 4$, since $2^4 = 16$;
- $\log_2 1 = 0$, since $2^0 = 1$;
- $\log_{10} 1000 = 3$, since $10^3 = 1000$;
- $\log_2 2^n = n$ for every integer n.

The logarithm $\log_2 n$ grows by 1 each time n doubles: $\log_2(2n) = \log_2 n + 1$, since $2^{\log_2 n+1} = 2 \cdot 2^{\log_2 n} = 2n$.

Remark 6 (The base of a logarithm is a constant factor). For every real number $x > 0$, let $y = \log_{10} x$ (Definition 9). Then $x = 10^y = (2^{\log_2 10})^y = 2^{y \log_2 10}$, so $\log_2 x = \log_2 10 \cdot \log_{10} x$. The factor $\log_2 10$, about 3.32, does not depend on x, so $\log_2 n = \Theta(\log_{10} n)$ (Definition 8), with $c = \log_2 10$ and $N = 1$ as witnesses (Definition 6) for both the big-O and the big-Omega bound. For this reason big-O bounds often write $\log n$ without a base.

Definition 10 (Factorial)**.** For a non-negative integer n, the **factorial** of n (Wikipedia contributors 2026c), written $n!$, is $1 \cdot 2 \cdots n$, the natural numbers from 1 to n multiplied together. For $n = 0$ there is nothing to multiply, and $0! = 1$.

Example 11 (Some factorials)**.** By Definition 10, $1! = 1$, $3! = 1 \cdot 2 \cdot 3 = 6$ and $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. The number of orders in which 5 different books can stand on a shelf is $5! = 120$: 5 choices for the first, then 4 for the second, and so on.

Theorem 6 (A chain of growth rates)**.** *For every natural number (Definition: Natural numbers (our convention)) $n \geq 4$,*

$$1 \leq \log_2 n < n \leq n \log_2 n < n^2 \leq 2^n \leq n!,$$

where $\log_2$ is the logarithm to the base 2 (Definition 9) and $n!$ is the factorial (Definition 10).

Proof

Proof. We use that $2^y < 2^z$ for real numbers $y < z$. Let $n \geq 4$ be a natural number, and let $y = \log_2 n$, so that $2^y = n$.
$1 \leq \log_2 n$**.** If $y < 1$, then $n = 2^y < 2^1 = 2$, which is false.
$\log_2 n < n$**.** First, $m < 2^m$ for every natural number m, by induction (Definition: Proof by induction).
Base case. $1 < 2 = 2^1$.
Inductive step. If $k < 2^k$ for an integer $k \geq 1$, then $k + 1 \leq 2^k < 2^k + 2^k = 2^{k+1}$.
So $n < 2^n$. If $y \geq n$, then $n = 2^y \geq 2^n$, which is false.
$n \leq n \log_2 n$ **and** $n \log_2 n < n^2$**.** Multiply $1 \leq \log_2 n < n$ by $n > 0$.
$n^2 \leq 2^n$**.** This is Example: An induction that starts at 4, since $n \geq 4$.
$2^n \leq n!$**.** By induction, following Remark: The shape of a proof by induction with $n_0 = 4$.
Base case. $2^4 = 16 \leq 24 = 4!$.
Inductive step. If $2^k \leq k!$ for an integer $k \geq 4$, then

$$\begin{aligned} 2^{k+1} &= 2 \cdot 2^k \\ &\leq 2 \cdot k! && \text{(induction hypothesis)} \\ &< (k+1) \cdot k! && (2 < k+1) \\ &= (k+1)!. \end{aligned}$$

□

Remark 7 (The common growth rates). By Theorem 6, each function in the chain

$$1, \quad \log_2 n, \quad n, \quad n \log_2 n, \quad n^2, \quad 2^n, \quad n!$$

is big-O (Definition 5) of the next, with witnesses (Definition 6) $c = 1$ and $N = 4$. In the other direction, none is big-O of the one before it. Solution 7 shows how to prove such a statement for $n \cdot 2^n$ and 2^n, and the same argument, dividing by n instead of 2^n, shows that $n^2 = O(n)$ is false. A running time that is Θ (Definition 8) of one of these functions is called, in order,

- *constant*, for 1;
- *logarithmic*, for $\log_2 n$;
- *linear*, for n;
- $n \log n$, for $n \log_2 n$;
- *quadratic*, for n^2;
- *exponential*, for 2^n;
- *factorial*, for $n!$.

Other powers of n fit between n and 2^n. For natural numbers $a \leq b$, $n^a \leq n^b$ for every natural number n, so $n^a = O(n^b)$. Every power n^a is also $O(2^n)$, though the witness N

grows with a: for example, $n^4 \leq 2^n$ holds for every $n \geq 16$, where $16^4 = 65536 = 2^{16}$, but not for $n = 15$, since $15^4 = 50625 > 32768 = 2^{15}$. We do not prove the general statement here.

Counting the steps of loops

Exercise 9 (Printing a multiplication table)**.** The algorithm `table`, in the pseudocode of Remark: How to read the pseudocode in this chapter, prints a multiplication table:

```
procedure table(n)       // n is a natural number
    for i := 1 to n
        for j := 1 to n
            print i * j
```

Here `print x` writes the value of `x` as output. How many times does the line `print i * j` run, as a function of n?

Solution

Solution 9. For each value of i, the inner loop runs `print i * j` once for each of $j = 1, 2, \ldots, n$, that is, n times. The outer loop makes n passes, one for each of $i = 1, 2, \ldots, n$, so the line runs $n + n + \cdots + n = n \cdot n = n^2$ times, once for each entry of an n by n table. For example, when $n = 4$ it runs $4 \cdot 4 = 16$ times.

Remark 8 (Counting the steps of loops). To find the running time (Definition 4) of an algorithm with loops:

- a loop whose body takes the same number s of steps on each of p passes takes $p \cdot s$ steps, so the steps of nested loops multiply, as in Solution 9;
- parts of an algorithm that run one after the other take the sum of their steps, and Theorem 5 turns big-O bounds for the parts into a big-O bound for the sum.

Each test of a loop's guard (Definition: Guard of a loop) and each line of setup is also a step, but counting them too changes the count by at most a fixed multiple plus a fixed number, which does not change a Θ bound (Definition 8) on a count that is at least 1.

Example 12 (A loop, then a nested loop)**.** The algorithm `pairs` takes an array `A[1..n]` of numbers, where $n \geq 1$:

```
procedure pairs(A, n)    // A[1..n] is an array of numbers, n >= 1
    s := 0
    for i := 1 to n
        s := s + A[i]
    t := 0
    for i := 1 to n
        for j := 1 to n
            t := t + A[i] * A[j]
    return s + t
```

Count each run of an assignment inside a loop as one step. The first loop takes n steps, and the nested loops take $n \cdot n = n^2$ steps, as in Solution 9, so the algorithm takes $f(n) = n + n^2$ steps. Since $n \leq n^2$ and $n^2 \leq n^2$ for every natural number n, both $n = O(n^2)$ and $n^2 = O(n^2)$ (Definition 5) with witnesses (Definition 6) $c = 1$, $N = 1$, so $f(n) = O(n^2)$ by Theorem 5. Also $f(n) \geq n^2$, so $f(n) = \Omega(n^2)$ (Definition 7) with $c = 1$, $N = 1$. So $f(n) = \Theta(n^2)$ (Definition 8): the nested loops decide the growth, and the first loop does not.

Example 13 (A loop with the same recurrence as `pal`)**.** The algorithm `pal_loop` tests whether a string is a palindrome (Definition: Palindrome) with a loop instead of recursion:

```
procedure pal_loop(s)    // s is a string s[1] s[2] ... s[n] of n symbols
    i := 1
    j := n
    while i < j and s[i] = s[j]
        i := i + 1
        j := j - 1
    return i >= j
```

The test `s[i] = s[j]` is made only when `i < j` is true. Let $L(n)$ be the largest number of passes through the body on a string of n symbols. When $n \leq 1$, the guard is false at once, so $L(0) = L(1) = 0$. When $n \geq 2$, the most passes come when the first and last symbols match; after the first pass, the loop then runs exactly as it would on the string $s[2..n-1]$ of $n-2$ symbols, with every index one larger. So $L(n) = L(n-2) + 1$, the same recurrence relation (Definition 1) as for `pal` in Example 5, with initial terms one smaller.

So $L(n) = \lfloor n/2 \rfloor$ (Definition: Floor), by strong induction (Definition: Proof by strong induction) as in the proof of Theorem 3: if $n \leq 1$, then $L(n) = 0 = \lfloor n/2 \rfloor$, and if $n \geq 2$

and the formula holds for $n-2$, then $L(n) = \lfloor (n-2)/2 \rfloor + 1 = \lfloor n/2 \rfloor - 1 + 1 = \lfloor n/2 \rfloor$. The guard is tested once more than the body runs, $\lfloor n/2 \rfloor + 1$ times, which is the number of calls of `pal` by Theorem 3, and is $O(n)$ by Example 6.

Example 14 (`root` takes $\Theta(\sqrt{n})$ steps)**.** The algorithm `root` of Exercise: Tracing a square-root loop returns $r = \lfloor \sqrt{n} \rfloor$ (Definition: Square root), the largest integer whose square is at most n, and its body adds 1 to `r` on each pass, starting from 0. So on input n the body runs exactly $\lfloor \sqrt{n} \rfloor$ times. Let $f(n) = \lfloor \sqrt{n} \rfloor$, for natural numbers n.

- $f(n) \leq \sqrt{n}$, so $f(n) = O(\sqrt{n})$ (Definition 5) with witnesses (Definition 6) $c = 1$, $N = 1$.
- $f(n) \geq 1$ since $n \geq 1$, and $\sqrt{n} < f(n) + 1 \leq 2f(n)$ by the definition of the floor (Definition: Floor), so $f(n) > \frac{1}{2}\sqrt{n}$, and $f(n) = \Omega(\sqrt{n})$ (Definition 7) with $c = \frac{1}{2}$, $N = 1$.

So $f(n) = \Theta(\sqrt{n})$ (Definition 8). Since $\sqrt{n} \leq n$ for $n \geq 1$, the body of `root` runs no more times than the body of a loop with n passes: on $n = 10^6$, its body runs 1000 times.

References

Wikipedia contributors. 2026a. *Big O Notation — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Big_O_notation.

Wikipedia contributors. 2026b. *Closed-Form Expression — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Closed-form_expression.

Wikipedia contributors. 2026c. *Factorial — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Factorial.

Wikipedia contributors. 2026d. *Logarithm — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Logarithm.

Wikipedia contributors. 2026e. *Recurrence Relation — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Recurrence_relation.

Wikipedia contributors. 2026f. *Time Complexity — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Time_complexity.

Wikipedia contributors. 2026g. *Triangle Inequality — Wikipedia, the Free Encyclopedia.* https://en.wikipedia.org/wiki/Triangle_inequality.