Learning objectives:
- ☐ analyze how many operations are performed in simple algorithms
- ☐ characterize algorithm running time complexity using big-O notation

1 Stacking blocks

Say I give you a bunch of length 1 blocks (each with a mass of 1 kg) and ask you to stack them on the edge of the table. Can you think of a stacking sequence such that the projection of the top block onto the plane of the table will be entirely *off* the table? See Figure 1 for a description of what we are trying to achieve. Can you do this by only stacking one block per level?

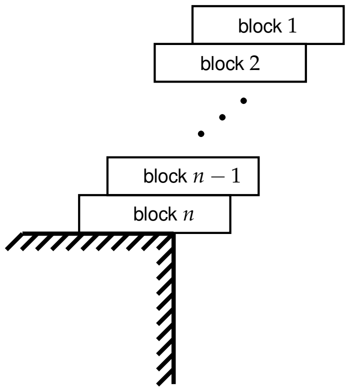

Figure 1: Stacking blocks off the edge of the table.

1.1 Optimal solution

In order for the stack of blocks to be stable (not fall off the table), we need the center of mass of all the blocks to lie above the bottom block. If we just have a single block, then we can stick it out by at most one half the length without falling over (see Figure 2): $h_1 = \frac{1}{2}$. If there are two blocks, then we can't stick out the bottom one by one half the length, because the center of mass of the top two blocks combined is a little further to the right. Say we stick out the bottom block by h_2. To maximize the overhang, we want the center of mass of the top two blocks to lie exactly at the edge of the table. Using the edge of the table as our origin, then

$$0 \times \underbrace{(1+1)}_{\text{mass of two blocks}} = h_2 \times 1 + \left(h_2 - \frac{1}{2}\right) \times 1.$$

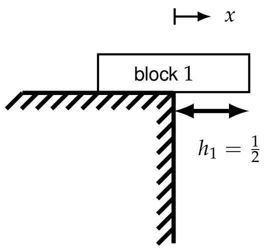

Figure 2: Maximum overhang with a single block.

Solving for h_2 gives $h_2 = \frac{1}{4}$. Okay, let's try with three blocks. We retain the configuration of the top two blocks (since this maximizes the overhang of those blocks) and need to figure out how much to stick out the bottom block, which we will denote as h_3. Again taking the origin as the edge of the table and letting the center of mass of the three blocks lie *exactly* at the edge of the table gives

$$0 \times \underbrace{(1+1+1)}_{\text{mass of three blocks}} = h_3 \times 1 + (h_3 - h_2) \times 1$$

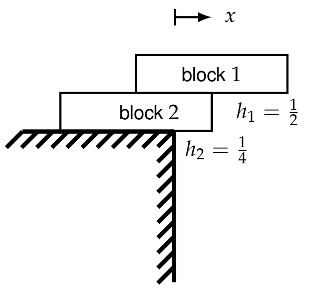

Figure 3: Maximum overhang with two blocks.

Now, solving for h_3 gives $h_3 = h_2/2$. See a pattern? In order for the center of mass to lie exactly on the edge of the table, the bottom block must stick out half the amount that the block above it sticks out (relative to this bottom block). So what's the total overhang of the top block? Well, we can just add up all the overhangs:

$$h_1 + h_2 + h_3 = \frac{1}{2} + \left(\frac{1}{2} \times \frac{1}{2}\right) + \left(\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2}\right) = 0.875.$$

We're getting close! We need this to be greater than 1 (a full block hangs over the edge). Do you see a pattern? In fact, the configuration for block n that maximizes the overhang (will still being stable) is $h_n = h_{n-1}/2$. Then the total overhang is:

$$\text{total overhang of top block} = \frac{1}{2}\sum_{i=1}^{n}\frac{1}{i}. \tag{1}$$

So you only need $n = 4$ blocks to hang the top block entirely over the edge of the table.

To infinity and beyond!

In fact, this sum, goes to ∞ as $n \to \infty$, so you can achieve any distance off the table with this method.

1.2 A more detailed solution

It's possible to approximate the solution for the maximum overhang by integrating Equation 1 and bounding it with a few terms in the sequence:

$$\text{total overhang of top block} = \frac{1}{2}\ln(n) + \frac{1}{2}\gamma + \frac{1}{4n} + \frac{1}{24n^2} + \frac{\delta(n)}{240n^4}$$

where $\gamma \approx 0.577215664$ is Euler's constant and $\delta(n)$ is some function that is between 0 and 1 for all nSo what does this mean? Well, as n gets big, the dominant term is $\frac{1}{2}\ln(n)$. All other terms get really really small. Does it matter that there's a $\frac{1}{2}$ in front? Not really. Theoretically, we can achieve *any* distance off the table with enough blocks. However, the maximum overhang grows with the dominant terms, which is $\ln(n)$ which is really slow, so you would need a lot of blocks! This leads into our discussion about asymptotic notation, which is a way to characterize function growth.

2 Estimating function growth

You may have seen something called "big-O" notation in CS 145/150. If not, that's okay, we'll formally define it here.

Definition 1. *Given functions f, g which map x from $\mathbb{R} \to \mathbb{R}$, then f is $O(g)$ if*

$$\lim_{x\to\infty}\left|\frac{f(x)}{g(x))}\right| < \infty. \tag{2}$$

Another way to think about this is that f grows at the same rate, or slower than g, when we ignore constant factors.

If you're not familiar with limits, the following definition of big-O might be more intuitive.

Definition 2. *Given $f, g\colon \mathbb{R} \to \mathbb{R}$, we say that $f(x)$ is $O(g(x))$ if-and-only-if there* ***exists*** *constants $c > 0$ and k such that $|f(x)| \le c \cdot |g(x)|$ for all $x \ge k$. Mathematically,*

$$f(x) \in O(g(x)) \iff \exists c > 0, \exists k : |f(x)| \le c \cdot |g(x)| \; \forall x \ge k. \tag{3}$$

Note that the role of k is because we only want to consider problems with large input sizes. The constant c is because we only care about the growth of f up to constant factors. Our job is then to find constants c and k that satisfy this definition.

Example 1:
Let $f(x) = x$ and $g(x) = 2x^2$. Prove that $f(x)$ is $O(g(x))$.

Solution:
We just need to check Definition 1. Computing the limit gives

$$\lim_{x\to\infty} \frac{x}{2x^2} = 0 < \infty, \tag{4}$$

thus verifying that $f(x)$ is $O(g(x))$. In this example, any $c > 2$ would work to satisfy Definition 2.

One of the difficulties with big-O notation is the concept of "constant factors." Is $100x^2 = O(x^2)$? Yes, because the limit of f/g as $x \to \infty$ is 100, which is finite. Is $x^2 = O(1000000000x)$? No, because even though 1000000000 is really really big, the limit of f/g as $x \to \infty$ is ∞.

Can I do this? $f(x) \ge O(g(x))$

No!! this is not correct notation. You can say $f(x)$ is $O(g(x))$ but relational operators don't mean much in the context of big-O notation.

Example 2:
Determine whether the following are true or false.

(a) is $10^6 x = O(x^2)$?

(b) is $x^{10} = O(e^x)$?

(c) is $4^x = O(2^x)$?

(d) is $10 = O(1)$?

Solution:

(a) Yes, $10^6 x$ is $O(x^2)$ because the limit of $f/g = 0$ as $x \to \infty$.

(b) Yes, x^{10} is $O(e^x)$ because the limit of $f/g = 0$ as $x \to \infty$ (you can prove this with l'Hôpital's rule).

(c) No, 4^x is not $O(2^x)$ because the limit of $f/g = \infty$ as $x \to \infty$.

(d) Yes, 10 is $O(1)$. In fact, any constant is $O(1)$.

3 Searching algorithms

Let's practice what we've learned with summations and asymptotic notation with some algorithms. For example, take the pseudocode for linear search in Algorithm 1. See if you can do a detailed worst-case analysis of the run-time of this algorithm.

```
linearSearch($v, x$)

    input: value $v$ and array $x$
    output: index of $v$ in $x$
1   $n \leftarrow$ length($x$)
2   for $i = 1, 2, \ldots, n$
3       if $x[i] == v$
4           return $i$
5   return $-1$
```

Algorithm 1: Linear Search.

Example 3:
What is the worst-case running time of the linear search algorithm (Algorithm 1)? First do a detailed calculation and then express your result with big-O notation.

Solution:
We need one operation to compute the length of the array and one to return the index. In the worst case, the body of the for-loop is executed n times. Within this loop we have one operation to increase the counter i, another one to check if we are still in the bounds of the loop, one operation to retrieve the i^{th} value in x, and another to compare it to v. This means 4 operations are performed within the loop, for a total of $4n$ executions. We also need to include the initialization of the counter i (setting $i = 1$ at the start of the loop). Therefore we have $4n + 3$ operations in the worst case. The running time of this algorithm is $O(n)$.

More efficient solution?

A more efficient method for searching for a value in an array is binary search. The idea is to perform the linear search algorithm on two arrays: one from index 1 to $n/2$ and another from $n/2 + 1$ to n. If you keep doing this recursively, then the number of times you need to break up the array into halves is equal to $\log_2(x)$. Therefore, the running time for binary search is $O(\log(x))$.

Although we did a detailed analysis here, we generally don't count operations like checking the bounds of the loop, incrementing counters, etc. They merely add constant factors which are really small compared to the terms that depend on the size of our inputs n. It's usually a good idea to keep track of these when you do your initial analysis, because you don't want to mistake some steps as incurring a constant number of operations, when they really depend on n. However, the ultimate goal here is to get a big-O bound on the number of operations performed by the algorithm, so it's okay if your constant factors are different than the solution above.

4 Other ways to characterize complexity (optional)

There are other ways to characterize function growth, which can be useful in computer science. You do not need to know these for this class! I'm simply including them here in case you are interested and come across them in the future.

name	**symbol**	**notation**	**definition**	**example**
tilde	$\sim$	$f(x) \sim g(x)$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$	$x^2 + x + 10 \sim x^2 + 5x$
little-o	o	$f(x) = o(g(x))$	$\lim_{x\to\infty} \left\lvert\frac{f(x)}{g(x)}\right\rvert = 0$	$\frac{x}{\ln(x)} = o(x)$
big-O	O	$f(x) = O(g(x))$	$\lim_{x\to\infty} \left\lvert\frac{f(x)}{g(x)}\right\rvert < \infty$	$x^2 = O(10x^2)$
little-omega	ω	$f(x) = \omega(g(x))$	$\lim_{x\to\infty} \left\lvert\frac{f(x)}{g(x)}\right\rvert = \infty$	$x^2 = \omega(x)$
big-omega	Ω	$f(x) = \Omega(g(x))$	$\lim_{x\to\infty} \left\lvert\frac{f(x)}{g(x)}\right\rvert > 0$	$2^x = \Omega(x^2)$
theta	Θ	$f(x) = \Theta(g(x))$	$\lim_{x\to\infty} \left\lvert\frac{f(x)}{g(x)}\right\rvert > 0$ **and** $< \infty$	$x^2 = \Theta(x^2)$