Recurrence & Recursion

CISC 2210 — CUNY Brooklyn College
Lecture Notes

Comparing Function Growth: Intro

In computer science, we often need to compare the growth rates of functions such as $f(n) = n^2$ and $g(n) = 2^n - 1$ (both having the domain of $\mathbb{N}$.)

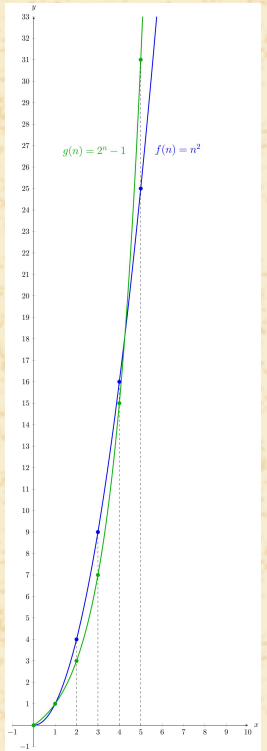
When designing an **algorithm**, which is a recipe-like method of solving a certain computational problem, we care about how fast its running time grows as the input (e.g., a set) of size n increases. That is, which computer program runs faster: one that does the work of the size of n^2, or one that does the work of the size of $2^n - 1$, both reaching the same correct solution anyways?

Initially, for small values of n, the function $f(n) = n^2$ may be larger than $g(n) = 2^n - 1$, but as n grows, $g(n)$ eventually surpasses $f(n)$: check out the diagram on slide 3.

This leads us to ask: How can we compare the long-term behavior of two functions in a mathematically precise way?

To answer this question, we will explore a few math tools: proof by induction, loop invariants, and asymptotic analysis using Big-O notation. We'll define these terms as we go through each tool.

Comparing Function Growth: Intro



$f(n)$ and $g(n)$. CC BY-NC 4.0

```
% Code for creating the graph of f(n) = n^2 and g(n) = 2^n - 1.
\documentclass{standalone}
\usepackage{pgfplots} % Plotting library
\usetikzlibrary{intersections}

\begin{document}

\begin{tikzpicture}
    \begin{axis}[
        xlabel = $x$,
        ylabel = $y$,
        y = 1cm,
        x = 1cm,
        xmin = -1, xmax = 10, ymin = -1, ymax = 33,
```

Loop Invariants

A **loop invariant** is a condition that holds true before and after each iteration of a loop and after the loop exits in an algorithm.

For a computer scientist, the confirmed presence of a loop invariant will prove that the loop will (1) eventually terminate/exit (and won't become an infinite loop), and (2) produce the correct result.

Here is a detailed description of each of the stages in which the invariant must be true:

1. **Initialization**: In the code right before the loop (the code that prepares the program to run the loop.)
2. **Maintenance**: After the execution of each of the iterations of the loop.
3. **Termination**: After the last iteration exited; that is, after the loop ended.

Let's see a couple examples of loops and their invariants.

Loop Invariants

Example #1: Consider the task of computing the sum of the first n positive integers using a loop. The pseudocode of this task is:

```
sum = 0;
for (i = 1; i <= n; i++)
    sum = sum + i;
```

We claim that the loop invariant is: "At the start of the ith iteration, `sum` contains the value $\sum_{k=1}^{i-1} k$ (= the sum of the first i positive integers seen so far.)" This invariant is indeed true because:

1. Before the loop starts, `sum` is 0, so the invariant holds because the sum of zero elements is 0.
2. Each time through the loop, we add the current value of i to `sum`, which maintains the invariant correctly.
3. When the loop finishes, we have $i = n + 1$, so the sum is $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$, which is the correct result.

Loop Invariants

Example #2: Suppose we want to reverse an array `A` of size n by swapping elements from both ends toward the center. Pseudocode:

```
for (i = 0; i < n/2; i++)
    swap(A[i], A[n-1-i]);
```

The loop invariant here is: "At the start of the ith iteration, the subarrays `A[0..i-1]` and `A[n-i..n-1]` are reversed." This invariant is true because:

1. Initially, no elements are swapped, and the invariant holds trivially.
2. In each iteration, the current pair of elements is swapped, extending the reversed section while preserving correctness.
3. When the loop ends, every pair is swapped exactly once, meaning the entire array is correctly reversed.

Loop Invariants

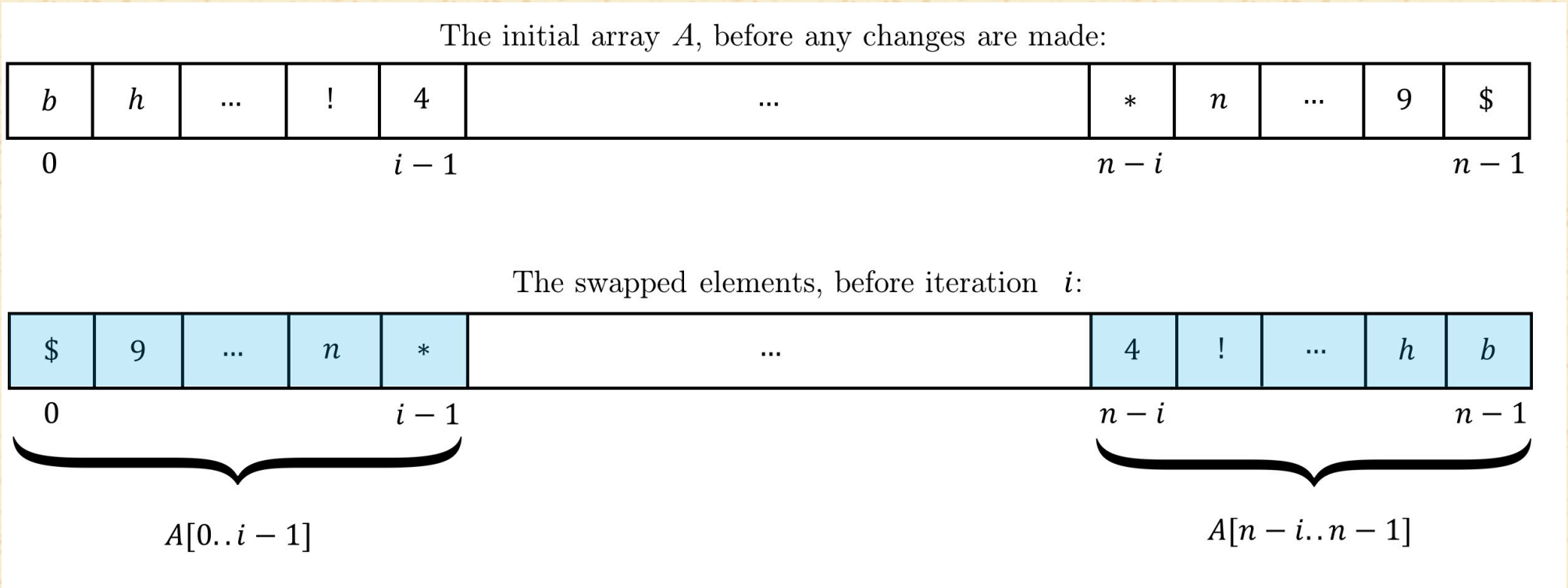


Illustration of the loop invariant of the array element swapping. Miriam Briskman, CC BY-NC 4.0

Exercises

Fun Class Activities:

1. Consider the loop below, and identify a correct loop invariant and justify it.
```
sum = 0;
for (i = 0; i < n; i++)
   sum += i;
```
2. Given an array `A`, and a loop that swaps `A[i]` and `A[n-1-i]` for `i = 0` to `n/2`, write a loop invariant and prove it informally.
3. Write a pseudocode that multiplies two numbers `a` and `b` using repeated addition, and state a suitable loop invariant.
4. In a loop that finds the maximum element in an array, what loop invariant should hold before and after each iteration?
5. Analyze the loop invariant in this pseudocode. What does the invariant guarantee about `product`?
```
product = 1;
for (i = 1; i <= n; i++)
    product *= i;
```

Exercises

Any questions?

What is Proof by Induction?

Mathematical induction is a powerful proof technique used to prove statements about all natural numbers. It works by proving a base case and then showing that if the statement holds for n, it must also hold for all integers bigger than n.

This is similar to toppling an infinite row of dominoes: push the first one, and each falls onto the next.

The structure of an inductive proof includes 3 parts/steps:

1. The **base case**: a short proof that the statement is true for one or a couple initial/small values,
2. The **inductive hypothesis**: stating an assumption that the statement is true for some integer k, and
3. The **inductive step**: a proof that the statement also holds for $k+1$. The inductive step proof is a bit more involved than the base case proof, and also uses the assumption of the inductive step in the proof.

If both the base case and the inductive step are proven, the statement is true for all $n \in \mathbb{N}$. Induction is often used to prove formulas, properties of algorithms, and combinatorial identities.

Induction

Example #1: Using induction, we'll prove that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (we presented this formula on slide 62 of Topic 2.)

Proof:

1. **Base case**: The first number that we are summing is 1, so the base case will show that the formula is true for $n = 1$. By plugging $n = 1$ into both sides of the formula, we get: $\sum_{i=1}^{n} i = \sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.
2. **Inductive hypothesis**: We assume that the formula is true for some $n = k$. That is, we assume that $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$. Note that the inductive hypothesis part doesn't need any proof: it's just an assumption that we explicitely make.
3. **Inductive step**: We'll now prove that the formula is true also for $n = k + 1$. We notice that $\sum_{i=1}^{k+1} i$ $= \left(\sum_{i=1}^{k} i\right) + (k+1) = \frac{k(k+1)}{2} + (k+1)$ [According to the inductive hypothesis] $= \frac{(k+1)(k+2)}{2}$.

According to our proof by induction above, we showed that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ is true for any $n \geq 1$. ■

Induction

Example #2: Using induction, we'll prove that a set S with $|S| = n$ elements has 2^n subsets (covered on slide 16 of Topic 2.)

Proof. [The size of a set S's power set is $2^{|S|} = 2^n$.]

1. **Base case**: For $n = 0$, the empty set has $2^0 = 1$ subset: the empty set $\emptyset$ itself.
2. **Inductive hypothesis**: We assume that any set with k elements has 2^k subsets.
3. **Inductive step**: We'll now prove that a set with $k + 1$ elements has 2^{k+1} subsets. Let's add a new element to a k-element set. Every existing subset either (1) contains the new element or (2) doesn't contain the new element. This doubles the number of subsets: $2 \cdot 2^k = 2^{k+1}$.
   Example: For $S_1 = \{a, b\}$, we have $\mathcal{P}(S_1) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$, while for $S_2 = \{a, b, c\}$, we have $\mathcal{P}(S_2) = \{\emptyset, \underline{\{c\}}, \{a\}, \underline{\{a, c\}}, \{b\}, \underline{\{b, c\}}, \{a, b\}, \underline{\{a, b, c\}}\} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

According to our proof by induction above, a set with $|S| = n \geq 0$ elements has 2^n subsets. ■

Induction

Example #3: Returning to our example from slides 2 – 3, we'll now prove that $2^n - 1 > n^2$ for $n \geq 5$ using induction.

Proof:

1. **Base case**: For $n = 5$, we confirm that $2^n - 1 = 2^5 - 1 = 32 - 1 = 31 > 25 = 5^2 = n^2$.
2. **Inductive hypothesis**: We assume that, for some $k \in \mathbb{P}$ for which $k \geq 5$, we have $2^k - 1 > k^2$.
3. **Inductive step**: We'll now prove that, for $k + 1$, we have $2^{k+1} - 1 > (k+1)^2$. Note that $2^{k+1} - 1 = 2 \cdot 2^k - 1 > 2 \cdot 2^k - 2 = 2(2^k - 1) > 2(k^2)$ [The latter inequality based on the inductive hypothesis]. Now, we need to show that $2(k^2) > (k+1)^2 = k^2 + 2k + 1$, or that $k^2 > 2k + 1$ in other words. Indeed, note that plugging any value $k \geq 5$ will make k^2 larger than $2k + 1$.

According to our proof by induction above, $2^n - 1 > n^2$ for all $n \geq 5$. ■

Exercises

Fun Class Activities:

1. Prove by induction that for all $n \geq 1$, the following formula is true:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}.$$

2. Use induction to show that for all $n \geq 1$, we have $2^n \geq n+1$.
3. Prove by induction that $3^n > n^2$ for all integers $n \geq 5$.
4. Show by induction that for all $n \geq 1$, the product

$$\prod_{k=1}^{n} \left(1 + \frac{1}{k}\right) = n+1.$$

Exercises

Any questions?

Function Growth: Big-O Notation

Big-O notation allows us to compare how fast functions grow as n becomes very large, which helps us compare algorithms based on their efficiency. The comparison rules are:

- **Big-O:** $f(n) = \mathcal{O}(g(n))$ (`$f(n) = \mathcal{O}(g(n))$`) means that $f(n)$ grows no faster than $g(n)$ up to constant factors. Examples: $n^2 = \mathcal{O}(n^2)$, $100n^2 = \mathcal{O}(n^2)$, $n^2 + n + 1 = \mathcal{O}(n^2)$, $n^2 = \mathcal{O}(n^3)$, and $n^2 = \mathcal{O}(2^n)$.
- **Little-o:** $f(n) = o(g(n))$ (`$f(n) = o(g(n))$`) means $f(n)$ grows strictly slower than $g(n)$. Examples: $n^2 = o(n^3)$, $1000n^2 + 500n + 2 = o(n^3)$, $n^2 = o(n^4)$, $n^2 = o(2^n)$, $100 = o(\log_2 n)$, $\log_2 n = o(n)$, and $n = o(n^2)$.
- **Big-Omega:** $f(n) = \Omega(g(n))$ (`$f(n) = \Omega(g(n))$`) means $f(n)$ grows at least as fast as $g(n)$. Examples: $n^2 = \Omega(n^2)$, $100n^2 = \Omega(n^2)$, $n^2 + n + 1 = \Omega(n^2)$, $n^3 = \Omega(n^2)$, and $2^n = \Omega(n^2)$.
- **Little-omega:** $f(n) = \omega(g(n))$ (`$f(n) = \omega(g(n))$`) means $f(n)$ grows strictly faster than $g(n)$. Examples: $n^5 = \omega(n^2)$, $n^4 + 300 = \omega(n)$, $2^n = \omega(n^2)$, $\log_2 n = \omega(1)$, $n = \omega(\log_2 n)$, and $n^2 = \omega(n)$.
- **Theta:** $f(n) = \Theta(g(n))$ (`$f(n) = \Theta(g(n))$`) means $f(n)$ and $g(n)$ grow at the same rate, up to constants. Examples: $1000 = \Theta(1)$, $n^2 + 4n + 4 = \Theta(n^2)$, $n^5 + n^3 = \Theta(n^5)$, $e^x + n^2 = \Theta(e^x)$, and $\log_{10} n = \Theta(\log_2 n)$.

Function Growth: Big-O Notation

The previous slide taught us that not only are the values of $2^n - 1$ larger than those of n^2 for large values of n (specifically, $n \geq 5$), but also that the function $f(n) = 2^n - 1$ growth at a much bigger rate than $g(n) = n^2$. That is, even if we multiply $g(n) = n^2$ by a huge constant, the function $f(n) = 2^n - 1$ will still keep surpassing $g(n) = n^2$ for a large enough n.

A couple more fun facts:

- On the previous slide, we said that $\log_{10} n = \Theta(\log_2 n)$. This is true because $\log_{10} n$ and $\log_2 n$ are constant multiples of one another! We know this because of the "change of base" logarithms formula: $\log_b a = \frac{\log_d a}{\log_d b}$.

  In our case, we have: $\log_{10} n = \frac{\log_2 n}{\log_2 10} \Rightarrow \log_2 10 \cdot \log_{10} n = \log_2 n \approx 3.3219 \cdot \log_{10} n = \log_2 n$.
- On the other hand, 2^n and e^n ($e \approx 2.72$ and is called "Euler's number") are <u>not</u> constant multiples. This means that e^n will grow strictly faster than 2^n since $e > 2$. In other words, the following statements are all true: $2^n = \mathcal{O}(e^n)$, $2^n = o(e^n)$, $e^n = \Omega(2^n)$, and $e^n = \omega(2^n)$.

Function Growth: Big-O Notation

Several more facts that will be useful when you take Brooklyn College's CISC 3220: Analysis of Algorithms course:

- If you know that $f(n) = \mathcal{O}(g(n))$ or $f(n) = o(g(n))$, then we call $g(n)$ an **upper bound** for $f(n)$ (we also say that $g(n)$ is **assymptotically larger than** $f(n)$) because $g(n)$ will surpass $f(n)$ for large inputs of n.
- If you know that $f(n) = \Omega(g(n))$ or $f(n) = \omega(g(n))$, then we call $g(n)$ a **lower bound** for $f(n)$ (we also say that $g(n)$ is **assymptotically smaller than** $f(n)$) because $f(n)$ will surpass $g(n)$ for large inputs of n.
- If you know that $f(n) = \Theta(g(n))$, then $g(n)$ is both an **upper bound** and a **lower bound** for $f(n)$.
- A function may have many upper bounds and many lower bounds. Some have no upper bound, or no lower bound.
- The **least (smallest) upper bound** (also called **supremum**) is the closest upper bound to the function.
- The **greatest lower bound** (also called **infimum**) is the closest lower bound to the function.
- If your current program runs at a speed of $g(n)$, and then you discover an algorithm $f(n) = \mathcal{O}(g(n))$, you should replace the $g(n)$ code with the $f(n)$ code. This is because $g(n)$ grows faster than $f(n)$, so a $g(n)$ program will run slower than a $f(n)$ one. Conclusion: code corresponding to a function that grows slower will result in an algorithm that runs faster!

Exercises

Fun Class Activities:

1. Find the smallest positive integer n such that $2^n > n^3$.
2. Compare the growth of $f(n) = n \log n$ and $g(n) = n^2$. Which one grows faster as n keeps growing?
3. Determine whether $f(n) = 5n^3 + 4n \log n$ is $O(n^3)$, $\Omega(n^3)$, or $\Theta(n^3)$.
4. Is it true that $f(n) = n^4$ is $o(n^5)$? Explain why or why not.
5. Suppose $f(n) = 3^n$ and $g(n) = 2^n$. Is $f(n) = O(g(n))$? Why or why not?
6. Which of the following functions is asymptotically smallest?
   - $f_1(n) = \log n$
   - $f_2(n) = n$
   - $f_3(n) = n \log n$
   - $f_4(n) = n^2$

Exercises

Any questions so far?

Recursion

Recursion is a problem-solving technique in which a function calls itself multiple times until the full solution is computed. Every recursive function must contain a **base case** that halts further recursion and a **recursive case** that reduces the input size or complexity, ensuring eventual termination.

In both mathematics and computer science, recursive definitions allow us to express infinite (or finite) computations using a finite set of rules, which makes them powerful tools for formalizing and solving problems.

Using recursion can result in simpler and cleaner code, especially when solving problems like tree (graph) traversal, factorial computation, or sequence generation, where the recursive structure is natural and intuitive.

Unlike loops, recursion provides a direct reflection of mathematical definitions, which makes recursions easier to code. However, recursion often consumes more memory than iteration due to repeated function calls and stack usage, unless the language or compiler performs tail-call optimization. As such, programmers must be careful to ensure that the number of function calls to itself is finite to prevent infinite recursion and stack overflows.

Recursion

Example #1: The **factorial**: $n!$ is the product of all positive integers less than or equal to n and has a natural recursive definition in mathematics. This function is defined by two rules: the 2 base cases $0! = 1$ and $1! = 1$ and the recursive step $n! = n \cdot (n-1)!$, which reduces the input size by 1 at each call.

For instance, to compute $4!$, we evaluate $4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Although this function can also be computed iteratively with a loop, the recursive version mirrors the mathematical definition more directly and is easier to reason about for small inputs. Nevertheless, for large n, recursive calls may lead to performance issues or stack overflow unless optimized using techniques like memoization or tail recursion.

Here is the recursive formula/definition: $n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$

($$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$)

Recursion

Example #2: Another common recursive function is the sum of all positive integers up to n, defined as $\sum n = 1 + 2 + \cdots + n$.

Like factorial, this can be defined recursively: $\sum n = n + \sum(n-1)$, with the base case $\sum 0 = 0$.

For example, $\sum 3 = 3 + \sum 2 = 3 + 2 + \sum 1 = 3 + 2 + 1 + \sum 0 = 3 + 2 + 1 + 0 = 6$.

While this can be computed quickly using formula $\frac{n(n+1)}{2}$ which we've already learned, the recursive version is useful pedagogically to understand recursion's mechanism.

Here is the definition: $\sum n = \begin{cases} 0 & \text{if } n = 0 \\ n + \sum(n-1) & \text{if } n > 0. \end{cases}$

```
$$\sum n = \begin{cases} 0 & \text{if } n = 0 \\ n + \sum(n - 1) & \text{if } n > 0 \end{cases}$$
```

Example #3: The **Fibonacci sequence** is a classic example of a recursively defined sequence in mathematics, where each term is the sum of the two preceding terms.

It is defined by the base cases $F(0) = 0$ and $F(1) = 1$, and the recurrence $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$.

For example, $F(4) = F(3) + F(2) = F(2) + F(1) + F(1) + F(0) = F(1) + F(0) + F(1) + F(1) + F(0) = 1 + 0 + 1 + 1 + 0 = 3$.

Here is the definition: $F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2. \end{cases}$

```
$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n - 1) + F(n - 2) & \text{if } n \geq 2 \end{cases}$$
```

Recursion

Beyond the Fibonacci sequence, many arithmetic and geometric sequences can also be defined recursively using simple rules and base cases.

- An arithmetic sequence with initial term a and difference d can be defined recursively as $A(0) = a$, $A(n) = A(n-1) + d$.
- A geometric sequence with initial term a and ratio r can be defined as $G(0) = a$, $G(n) = G(n-1) \cdot r$, which models exponential growth or decay.

Recursive definitions clarify how each term depends directly on the previous one, making the structure of the sequence explicit.

Although closed formulas like $A(n) = a + nd$ or $G(n) = ar^n$ are more efficient, recursive definitions offer insight into the logic of step-by-step construction.

Any questions so far?

Recurrences

A **recurrence relation** is an equation that defines each term of a sequence based on one or more previous terms, typically with specified initial values. Recurrence relations are common in both mathematics and computer science, especially in the analysis of recursive algorithms such as mergesort or binary search.

They describe how a problem of size n is reduced into smaller subproblems, which helps estimate time or space complexity of divide-and-conquer solutions. Many recurrence relations can be **solved** (= expressed just in terms of the variable n.)

For example, a relation like $T(n) = 2T(n/2) + n$ models a recursive algorithm that splits a problem into two halves and does work of n steps at each level. As you'll learn in the Analysis of Algorithms course, the solution to this recurrence is $T(n) = c \cdot n \log n$. We use the letter T, which represents an algorithm's runTime as a generic function name.

Summary so far: **Recursion** is the process of a function calling itself, while a **recurrence relation** is the formula for the computations done by the recursive function.

Recurrences

Many **divide-and-conquer** algorithms, which are ones that take a large task, split it into many small ones, solve them, and then merge the results of those tiny tasks together, naturally lead to recurrence relations. Examples:

- **Mergesort**, which is a type of a sorting algorithm, recursively splits the array in half and recursively sorts each half, doing $O(n)$ work to merge: $T(n) = 2T(n/2) + n$. [Solution: $T(n) = c \cdot n \log n$.]
- **Binary search**, whose purpose is to find an element in a sorted array, only explores one half of the array, performing constant-time comparisons: $T(n) = T(n/2) + 1$. [Solution: $T(n) = c \cdot \log n$.]
- **Strassen's matrix multiplication** reduces matrix multiplication to 7 recursive calls on half-size matrices: $T(n) = 7T(n/2) + cn^2$. [Solution: $T(n) = c \cdot n^{\log_2 7} \approx c \cdot n^{2.8074}$.]

These recurrence relations can be solved using various math techniques that you'll cover in the Analysis of Algorithms course.

Solutions to recurrence relations tells what the precise growth rates of the algorithms are, and helps us in choosing the fastest algorithm for our program.

Recurrences

There are several techniques to solve recurrence relations, each suited to different forms and complexities of the equation.

- The **iteration method**, also known as **backward substitution** expands the recurrence step-by-step, revealing a pattern that can be summed and then closed into a direct formula.
- The **recursion tree method** visualizes recursive calls as a tree and estimates the total work at each level, summing them to find a bound.
- The **master theorem** provides a shortcut for divide-and-conquer relations of the form $T(n) = aT(n/b) + f(n)$ when a, b are constants and $f(n)$ is some expression of n.

All of these approaches will be introduced, exemplified, and practiced with during the Analysis of Algorithms course you'll take.

The following lecture slides from an Algorithms course in University of Tennessee will give you a taste (+ examples) of how these methods work: (1) slides (2) slides. [Our course's final exam won't ask questions about these linked slides.]

Exercises

Fun Class Activities:

1. Given the recursive definition of the sequence $A(n) = \begin{cases} 2 & \text{if } n = 0 \\ A(n-1) + 3 & \text{if } n > 0 \end{cases}$
   compute the first 6 terms and derive the closed formula.
2. Given the recursive definition of the sequence $A(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ A(n-1) + 2A(n-2) & \text{if } n > 1 \end{cases}$
   compute the terms $A(2)$, $A(3)$, and $A(4)$ of this sequence.
3. Write the recursive definition for $\sum_{i=1}^{5} i^2$ (both the base case and the recurrence relation.)
4. Write the recursive definition for $\sum_{i=0}^{\infty} \frac{1}{3^i}$ (both the base case and the recurrence relation.)

Exercises

Any questions?

Recurrence & Recursion: Sources

Aspnes, James. (2022). Chapters 5, 6, and 7. *Notes on Discrete Mathematics*. License: CC BY-SA: Attribution-ShareAlike.

"CS202: Discrete Structures | Unit 6." *Saylor Academy*. License: CC BY: Attribution.

Doerr, Al and Levasseur, Ken. (2024). Chapter 8 and Appendix A. *Applied Discrete Structures*, https://discretemath.org/. License: CC BY-NC-SA: Attribution-NonCommercial-ShareAlike.

Jamaloodee, Mohamed, et al. Chapters 7, 8, and 11. *Discrete Math*. Georgia Gwinnett College. License: CC BY-NC: Attribution-NonCommercial.