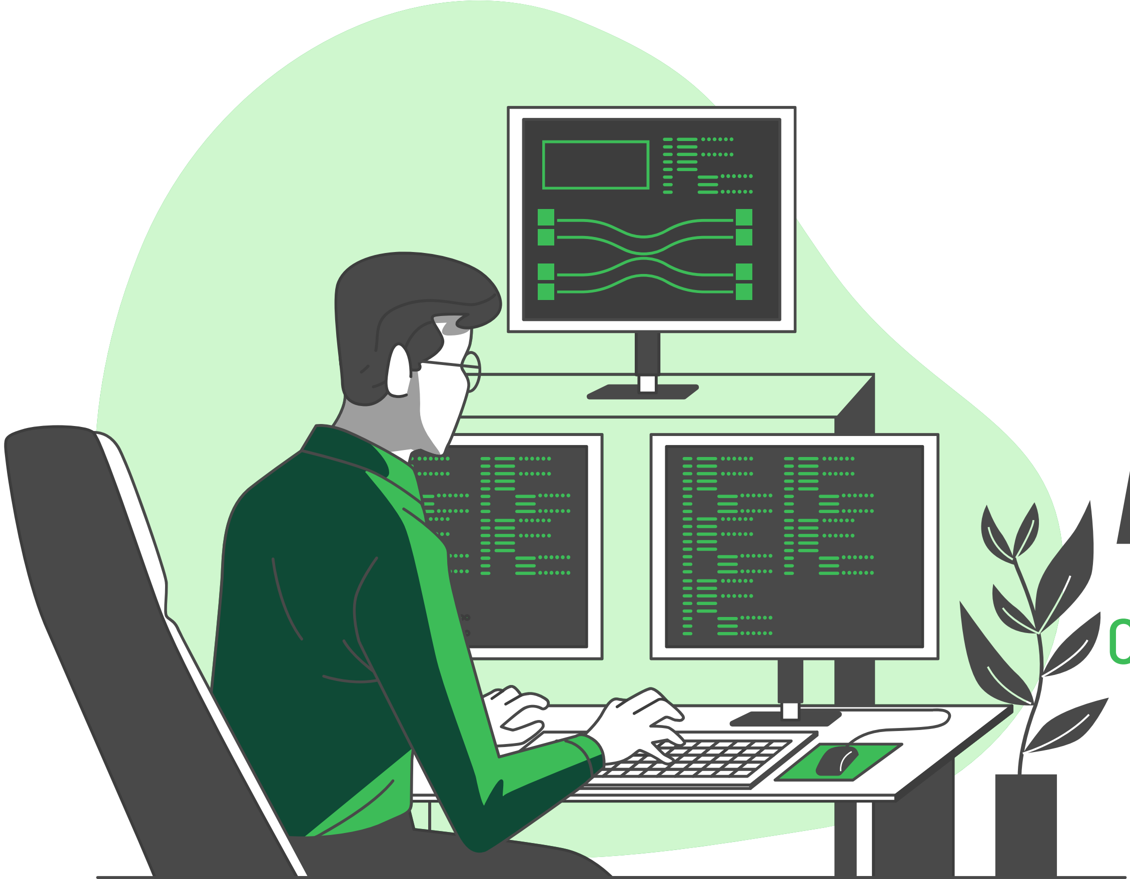
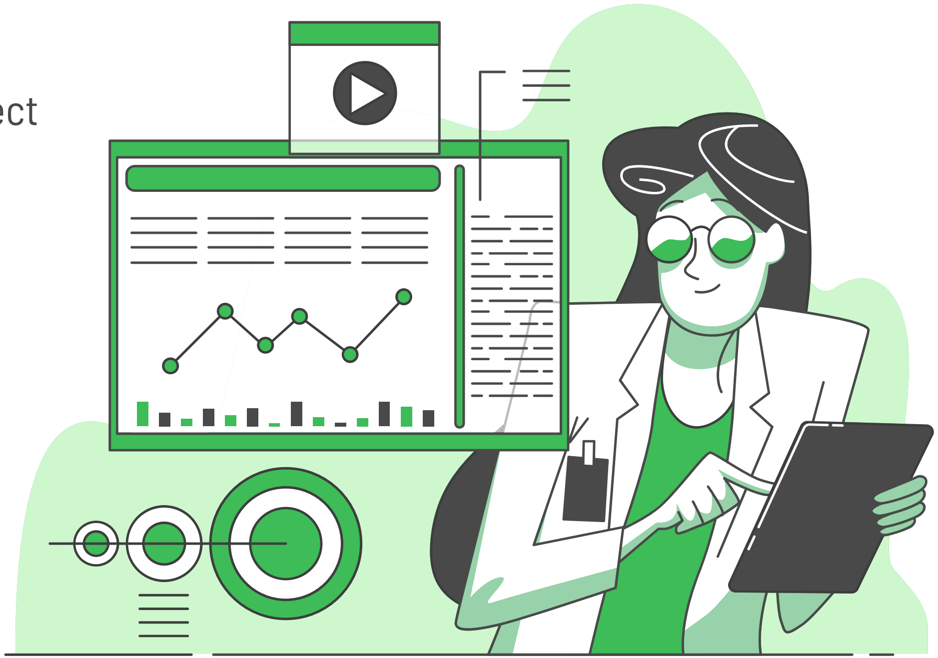


Recursive Algorithms

CS 251 - Data Structures and Algorithms

Note:
Slides complement the discussion in class

Table of Contents



01

Recursive Algorithms

A refresher

Recursive Algorithms

- A recursive algorithm solves a problem by calling a copy of itself to work on a smaller problem.
- A call to itself is known as a recursion step.
- Eventually, the algorithm reaches the smallest problem to deal with, for which it knows how to solve it.
- The solution to the smallest problem is known as the base case.
- We borrow the idea from recurrence relations to build recursive algorithms.

Traditional Recursive Algorithm Examples

Factorial

```
algorithm fact(n:ℤ≥0) → ℤ+
   if n ≤ 1 then
      return 1
   end if
   return n * fact(n-1)
end algorithm
```

Fibonacci

```
algorithm fib(n:ℤ≥0) → ℤ≥0
   if n ≤ 1 then
      return n
   end if
   return fib(n-1) + fib(n-2)
end algorithm
```

Binary Search (recursive)

```
algorithm BinarySearch(A:array, X:item, l:ℤ, r:ℤ) → ℤ

   if r < l then
      return -1
   end if

   m ← (l + r) / 2

   if A[m] = X then
      return m
   end if

   if A[m] > X then
      return BinarySearch(A, X, l, m - 1)
   end if

   return BinarySearch(A, X, m + 1, r)

end algorithm
```

First call:

```
let n be the length of A
index ← BinarySearch(A, X, 0, n-1)
```

Binary Search (iterative)

```
algorithm BinarySearch(A:array, X:item) → ℤ

   let n be the length of A
   l ← 0
   r ← n - 1

   while l <= r do

      m ← (l + r) / 2

      if A[m] = X then
         return m
      end if

      if A[m] > X then
         r ← m - 1
      else
         l ← m + 1
      end if
   end while

   return -1
end algorithm
```

Why binary search works?

02

Proof of Correctness

Show your algorithm is correct
without running it

Proof of Correctness

- A **formal** and **mathematical** demonstration that asserts the algorithm's correctness with respect to its specification. The purpose is to **ascertain that the algorithm will produce the correct output for any valid input**, showing that it consistently and accurately solves the problem it is intended to address.

- The proof of correctness ensures that the algorithm does not have logical errors, and it behaves as expected in all possible scenarios.

OK, but How?

01 Loop invariant

Condition maintained through the loops

02 Induction

Base case, inductive case, next case

03 Contradiction

Assume the algorithm is wrong, which leads to a contradiction

04 Assertions

Checkpoints at certain points of the algorithm

Example: Recursive Fibonacci

```
algorithm Fibonacci(n:ℤ≥0) → ℤ≥0

   if n ≤ 1 then
      return n
   end if

   return Fibonacci(n-1) + Fibonacci(n-2)

end algorithm
```

The Fibonacci sequence is mathematically defined as follows:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

```
algorithm Fibonacci(n:ℤ≥0) → ℤ≥0

   if n ≤ 1 then
      return n
   end if

   return Fibonacci(n-1) + Fibonacci(n-2)

end algorithm
```

Proof of correctness by induction (part 1):

Base Cases:

Case $n = 0$:
When $n = 0$, the algorithm returns 0. This is consistent with the definition $F(0) = 0$.

Case $n = 1$:
When $n = 1$, the algorithm returns 1. This is consistent with the definition $F(1) = 1$.

```
algorithm Fibonacci(n:ℤ≥0) → ℤ≥0

   if n ≤ 1 then
      return n
   end if

   return Fibonacci(n-1) + Fibonacci(n-2)

end algorithm
```

Proof of correctness by induction (part 2):

Inductive Step: Assume that the algorithm correctly computes the Fibonacci numbers for all values up to some arbitrary k. That is, assume that the algorithm correctly returns $F(i)$ for all i such that $0 \leq i \leq k$. We must show that the algorithm correctly computes $F(k+1)$.

According to the Fibonacci sequence definition:

$$F(k+1) = F(k) + F(k-1)$$

By our inductive hypothesis, we know that our algorithm correctly computes $F(k)$ and $F(k-1)$ since both k and $k-1$ are less than $k+1$.

When the algorithm is called with argument $k+1$, it recursively calculates:

$$\text{Fibonacci}(k) + \text{Fibonacci}(k-1)$$

Given our inductive assumption, this is precisely $F(k) + F(k-1)$, which matches the definition of $F(k+1)$.

Therefore, by the principle of mathematical induction, our recursive algorithm for the Fibonacci sequence is correct.

Example: Recursive Binary Search

```
algorithm BinarySearch(A:array, X:item, l:ℤ, r:ℤ) → ℤ

   if r < l then
      return -1
   end if

   m ← (l + r) / 2

   if A[m] = X then
      return m
   end if

   if A[m] > X then
      return BinarySearch(A, X, l, m - 1)
   end if

   return BinarySearch(A, X, m + 1, r)

end algorithm
```

Proof of correctness by loop invariant:
The correctness of Binary Search relies on the invariant that X, the item we are searching for, if it exists, is always within the search range defines by l and r. Initially, this range covers the entire array. Each recursive step maintains this invariant by narrowing down the range.

Initialization: At the start, $l = 0$ and $r = n - 1$, where n is the length of the array, ensuring the entire array is being considered.

Maintenance: If X is not at the middle index m, the algorithm eliminates half of the search range:

- If $A[m] > X$, then X, if present, must be in the left half of the array. Thus, we set $r = m - 1$.
- If $A[m] < X$, then X, if present, must be in the right half of the array. Thus, we set $l = m + 1$.
- Correctness of Search: if $A[m] = X$, the algorithm returns m, which is the correct index of X.

Termination: The loop terminates when $r < l$, at which point we can conclude that X is not in the array. If X is found earlier, the function returns immediately with the index.

Idea

Proof of Correctness for Recursive Algorithms Using Induction

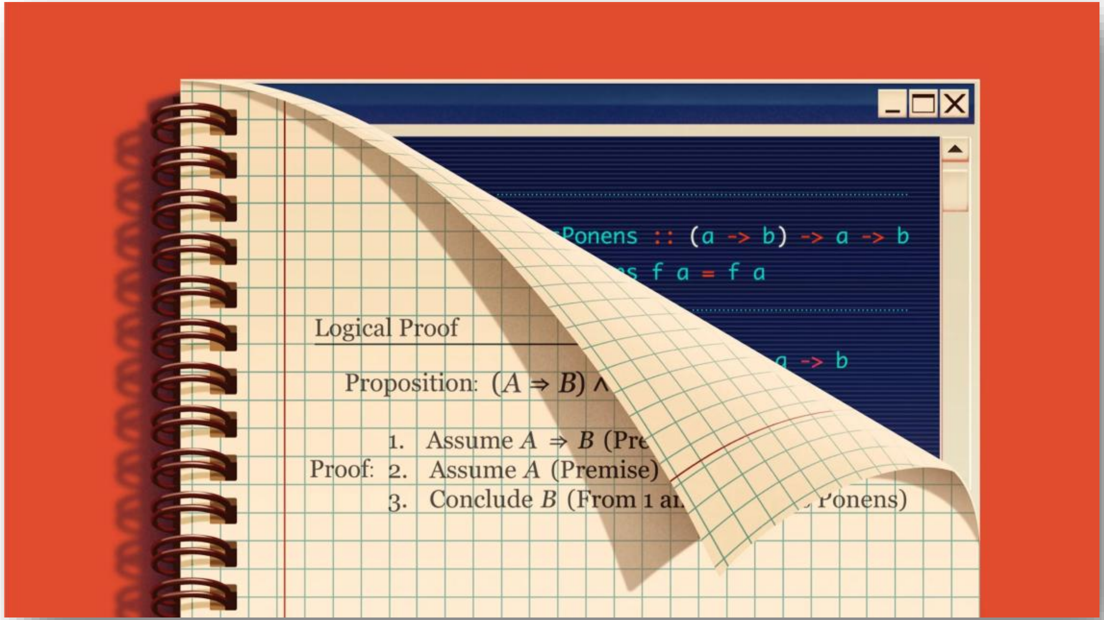
You can! It will be a combination of **induction** and **invariant**.

Prove the **invariant** holds for the base case(s) and the inductive step(s).

Then, prove the **termination** of the algorithm for the base case(s) and inductive step(s).

...

Curry–Howard Correspondence



"Mathematical logic and the code of computer programs are, in an exact way, mirror images of each other."
- Sheon Han. Contributing Writer for Quanta Magazine
[The Deep Link Equating Math Proofs and Computer Programs]

What about the $T(n)$?

03
Recursive Runtimes

How do we deal with them?

Iterations (Substitutions)

Idea: iterate the recurrence relation until a pattern for a k-th iteration becomes evident.

Let's use it to find a closed-form expression for the recursive expression $a_n = a_{n-1} + n$ with $a_0 = 4$.

$a_n = a_{n-1} + n$
$a_n = \left(a_{n-2} + (n-1)\right) + n = a_{n-2} + n + (n-1)$
$a_n = \left(a_{n-3} + (n-2)\right) + n + (n-1) = a_{n-3} + n + (n-1) + (n-2)$
…

$$a_n = a_{n-k} + \sum_{i=0}^{k-1}(n-i) = a_{n-k} + nk - \frac{(k-1)k}{2} = a_{n-k} + \frac{1}{2}(2nk - k^2 + k)$$

We reach the base case a_0 when $n - k = 0 \rightarrow n = k$.

$$a_n = a_{n-n} + \frac{1}{2}(2n^2 - n^2 + n) = a_0 + \frac{1}{2}(n^2 + n) = 4 + \frac{1}{2}(n^2 + n)$$

http://discrete.openmathbooks.org/dmoi3/sec_recurrence.html

```
algorithm BinarySearch(A:array, X:item, l:ℤ, r:ℤ) → ℤ

   if r < l then
      return -1
   end if

   m ← (l + r) / 2

   if A[m] = X then
      return m
   end if

   if A[m] > X then
      return BinarySearch(A, X, l, m - 1)
   end if

   return BinarySearch(A, X, m + 1, r)

end algorithm
```

Closed-form $T(n)$ for the cost of a successful search of X in terms of the input size n.

Let c_1 be the cost of the operations run by the successful base case, and c_2 be the cost of the operations run by the recursive case:

$$T(1) = c_1$$
$$T(n) = T\left(\frac{n}{2}\right) + c_2$$

Strategy: Use iterations until there is an identifiable pattern for a k-th recursive call.

$$T(n) = T\left(\frac{n}{2}\right) + c_2$$

$$T(n) = \left(T\left(\frac{n}{4}\right) + c_2\right) + c_2 = T\left(\frac{n}{4}\right) + 2c_2$$

$$T(n) = \left(T\left(\frac{n}{8}\right) + c_2\right) + 2c_2 = T\left(\frac{n}{8}\right) + 3c_2$$

$$T(n) = \left(T\left(\frac{n}{16}\right) + c_2\right) + 3c_2 = T\left(\frac{n}{16}\right) + 4c_2$$

…

$$T(n) = T\left(\frac{n}{2^k}\right) + kc_2$$

Last recursive call: $\frac{n}{2^k} = 1 \rightarrow n = 2^k \rightarrow k = \log_2(n)$

$$T(n) = T\left(\frac{n}{2^k}\right) + kc_2 = T\left(\frac{n}{2^{\log_2(n)}}\right) + \log_2(n)\, c_2$$

$$T(n) = T(1) + \log_2(n)c_2$$

$$T(n) = \log_2(n)c_2 + c_1$$

A Better ThreeSum Algorithm

Brute Force ThreeSum: A cubic problem!

$$T(n) = \frac{1}{2}n^3 - \frac{3}{2}n^2 + n$$

Better approach:
First, sort the numbers
Them, for each pair (A[i], A[j]), call
BinarySearch(A, -(A[i]+A[j]), j+1, n-1)

Runtime:
Sorting numbers: $\approx n\log(n)$ using a decent sorting algorithm (e.g., Merge Sort)
Generating pairs: $\approx n^2$.
Binary Search: $\approx \log_2(n)$ per pair.

$$T(n) \approx n\log(n) + n^2\log_2(n)$$

Iterations may not work for all recurrence relations

Example: Recursive Fibonacci

```
algorithm Fibonacci(n:ℤ≥0) → ℤ≥0

   if n ≤ 1 then
      return n
   end if

   return Fibonacci(n-1) + Fibonacci(n-2)

end algorithm
```

Base steps: $T(0) = 0, T(1) = 1$
Recursive step: $T(n) = T(n-1) + T(n-2)$

Warning: Solving this $T(n)$ using iterations is a bad idea!

Observations:

$$T(n) \leq T(n-1) + T(n-1)$$

$$T(n) \geq T(n-2) + T(n-2)$$

Let's use the first observation to find an upper bound and the second to find a lower bound.

Recursive Fibonacci: Upper Bound

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

Use iterations to solve the recurrence relation:

$T(n) \leq 2T(n-1)$
$T(n) \leq 2\big(2T(n-2)\big) = 2^2 T(n-2)$
$T(n) \leq 2^2\big(2T(n-3)\big) = 2^3 T(n-3)$
…
$T(n) \leq 2^k T(n-k)$

Last recursive call when $n - k = 1 \rightarrow k = n - 1$ (warning: using $T(0) = 0$ makes everything to be 0).

$$T(n) \leq 2^{n-1} T(n-n+1) = 2^{n-1} T(1) = \frac{1}{2} 2^n$$

Recursive Fibonacci: Lower Bound

$T(n) \geq T(n-2) + T(n-2) = 2T(n-2)$

Use iterations to solve the recurrence relation:

$T(n) \geq 2T(n-2)$
$T(n) \geq 2\big(2T(n-4)\big) = 2^2T(n-4)$
$T(n) \geq 2^2\big(2T(n-6)\big) = 2^3T(n-6)$
…
$T(n) \geq 2^kT(n-2k)$

Last recursive call when $n - 2k = 1 \rightarrow k = \frac{n-1}{2}$ (warning: using $T(0) = 0$ makes everything to be 0).

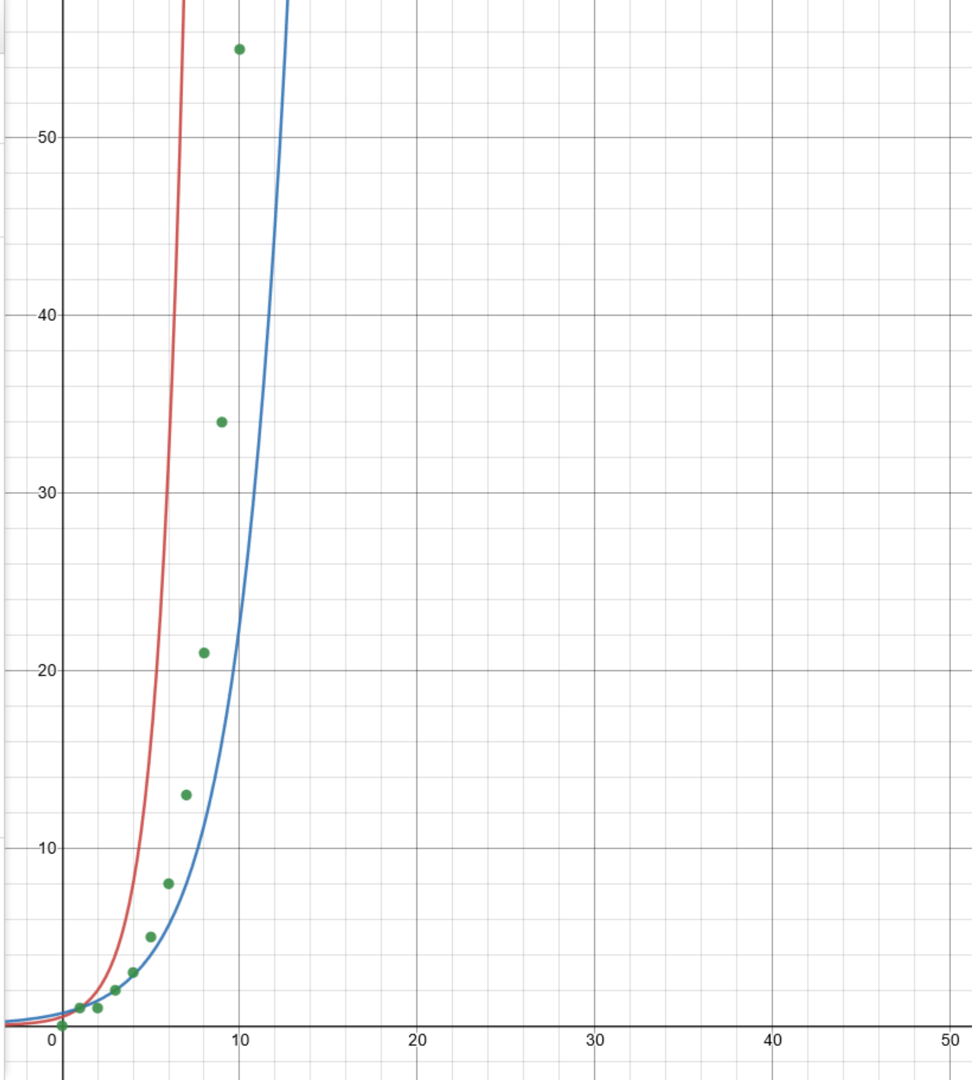
$$T(n) \geq 2^{(n-1)/2}T\left(n - 2\frac{n-1}{2}\right) = 2^{(n-1)/2}T(1) = \frac{1}{\sqrt{2}}\sqrt{2}^n$$

$$T(n) = \frac{1}{2} 2^n$$

$$T(n) = \frac{1}{\sqrt{2}} \sqrt{2}^n$$

x_1	y_1
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89

powered by desmos



Until next time

Do you have any questions?

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, infographics & images by Freepik and illustrations by Stories