what is the runtime of the following algorithm?

① function f indicating # of primitive operations on input n

② "best" (smallest) g s.t. $f = O(g)$.

```
for i = 1 to n do
  → if i < 3 do
      for j = 1 to n do
        sum = sum + 1
```

$O(1)$

$$\sum_{j=1}^{n} 1 = n$$

n times we do 3 operations

$$\sum_{i \in \{1,2\}} n = 2n$$

$f = 2n + n = 3n$

$f = O(n)$

Analysis of Recursive Algorithms

Factorial problem: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

input: $n \in \mathbb{Z}^{>0}$

output: $n! = n(n-1)(n-2) \cdots 3 \cdot 2 \cdot 1$

solution: an algorithm in pseudo code

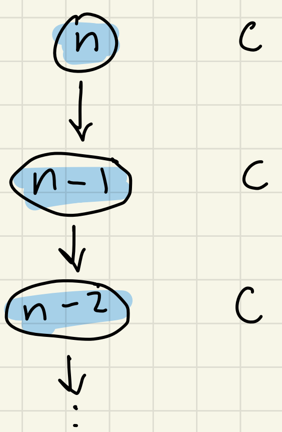
```
fact(n):
   if n = 1 then     } d primitive operations
      return 1       }
   else
      return n · fact(n-1)  } c prim. ops.
```

what is its worst-case runtime?

idea: look at the recursion tree.

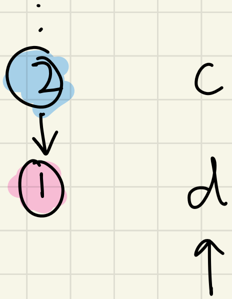
<u>Def</u> the <u>recursion tree</u> for a recursive algorithm A is a tree that shows all of the recursive calls spawned by A on an input of size n.

recursion tree for fact(n):



this recursion tree has n "rows" and 1 "column"

$$f = \underbrace{(n-1)c + d} = O(n)$$



idea #2: use recurrence relation.

Def A recurrence relation is a function $T(n)$ that is defined in terms of values of $T(k)$ for $k < n$.

$T(1) = d$, $T(n) = c + T(n-1)$

base case

How do we find the closed-form solution for $T(n)$?

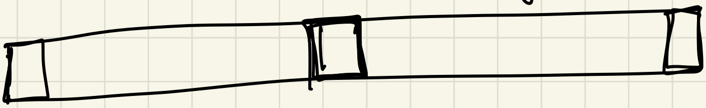
- iterate the solution a few times
- make a guess about the formula
- prove using induction

$T(1) = d$
$T(2) = c + T(1) = c + d$
$T(3) = c + T(2) = c + (c+d) = 2c + d$
$T(4) = c + T(3) = c + (2c+d) = 3c + d$

guess: $T(n) = (n-1)c + d$.

prove using induction. $(n-1)c + d = O(n)$

sorted array A $x \in A$

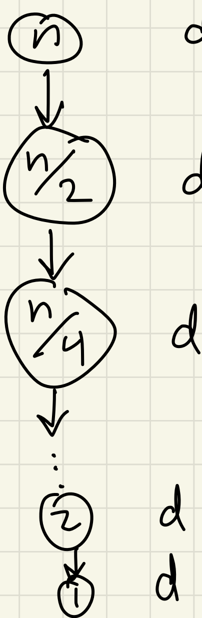


binarySearch(A, $loIndex$, $hiIndex$, x):

```
if loIndex > hiIndex then
    return False
middle := ⌊(loIndex + hiIndex)/2⌋
if A[middle] = x then
    return True
else if A[middle] > x then
    return binarySearch(A, loIndex, middle − 1, x)
else
    return binarySearch(A, middle + 1, hiIndex, x)
```

To avoid the inefficiency of copying portions of A wh
a recursive call is made, this code uses four paramet
instead of two: the array A, the left- and right-most
indices in A to search, and the sought element x. You
call the algorithm **binarySearch**($A[1\ldots n]$, 1, n, x)
start the recursive search for x in A.

recursion trees:



each call takes d work

$\log_2 n$ calls.

$d \log_2 n + c = O(\log n)$

0 c