CSCM12: software concepts and efficiency
Introducing recursion

Cécilia Pradic

February 13th 2025

What is recursion?

In general/informally

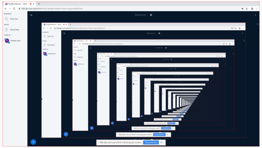
Self-referential notions

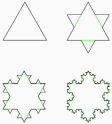
Some example/related concepts:

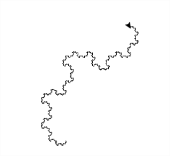
- Recursive definitions/characterizations $\quad F_0 = 0 \quad F_1 = 1 \quad F_{n+2} = F_{n+1} + F_n$

$$\big(\text{ancestor of } x\big) \quad = \quad (\text{parent}) \quad \text{or} \quad (\text{parent of some ancestor of } x)$$

- Fractals
- ...







(credit: wikipedia users)

Where is recursion in programming?

More specifically, in **Java**? (applicable to most procedural/**functional** programming languages)

In function definitions:

```java
static int fibo(int n)
{
  if (n <= 1)
    return n;
  else
    return fibo(n-2) + fibo(n-1);
}
```

In class definitions:

```java
class LinkedList<T>
{
    T head;
    LinkedList<T> tail;
}
```

Plan for today

Today: **only recursive functions**

(recursive type definitions will be introduced in later lecture on datastructures)

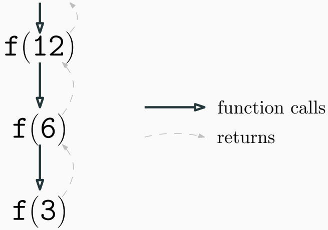
1. Recursive functions in **Java** (through examples)
   - How do they run?
   - Comparison with looping constructs (`for`, `while`)
   - Scopes of variable, mutual recursion
2. When it can useful (NB: not exhaustive!)
   - Use: recursion vs iteration?
   - Concrete use-cases in problem solving
3. Estimating the complexity of (some) recursive functions

Recursive functions in **Java**

A simple example

```
static int f(int n)
{
    if (n%2 != 0 || n == 0)
      return n;
    else
      return f(n/2);
}
```

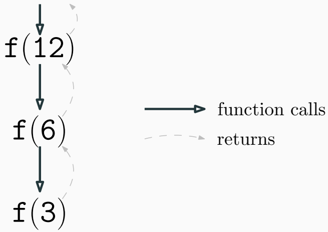
Output of `f(12)`: 3



A simple example

```
static int f(int n)
{
   if (n%2 != 0 || n == 0)
     return n;
   else
     return f(n/2);
}
```

Output of f(12): 3



Termination: the absolute value n decreases across calls.

A more involved example

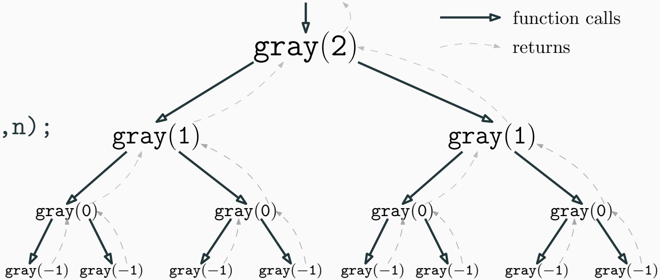
```
static void gray(int n)
{
   if (n < 0)
     return;
   gray(n-1);
   System.out.printf("%d ",n);
   gray(n-1);
}
```

How is, say, `gray(2)` executed?

A more involved example

```
static void gray(int n)
{
   if (n < 0)
     return;
   gray(n-1);
   System.out.printf("%d ",n);
   gray(n-1);
}
```

Output: 0 1 0 2 0 1 0

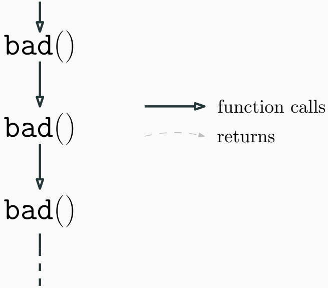


How is, say, `gray(2)` executed?

Infinite recursion (1/2)

```
int bad()
{
   return bad()+1;
}
```

- Will most likely lead to a "`stack overflow`" error
  (low-level: a stack structure is typically used at the CPU level to model a path in the call tree)



Infinite recursion (2/2)

```
static int advertise(char* product)
{
   Scanner sc = new Scanner(System.in);
   System.out.printf("\n Buy %s!\n", product);
   if (sc.nextByte() == 'y')
     return 0;
   else
     return advertise(product);
}

... advertise("data") ...
```

```
do {
  Scanner sc = new Scanner(System.in);
  System.out.printf("\n Buy data!\n");
} while (sc.NextByte() != 'y');
```

advertise("trinket")

advertise("trinket")

advertise("trinket")

function calls

returns

Recursion vs iteration

Use-cases of recursion: similar to those of iteration constructs `for` and `while`

```
int facto_rec(int n)
{
  if (n == 0)
    return 1;
  else
    return n * facto_rec(n-1);
}
```

```
static int facto_iter(int n)
{
  int r = 1;
  for (; n != 0; n--)
    r *= n;
  return r;
}
```

In theory, one can always pick one or the other without loss of generality.

Comments

- Mutable variables: required for meaningful iterations, not necessarily for recursion
  ($\leadsto$ sometimes easier to reason about recursive functions)
- Hard to translate recursive functions into iterative ones (easier the other way around)

Scoping of variables

Variables are local to one callsite of the function

To maintain state across calls, use `static` or global variables

```
static void f()
{
  int i = 2;
  i--;
  if(i > 0)
    f();
}

static void f1()
{
  int i1 = 2;
  i1--;
  if(i1 > 0)
    f();
}
```

```
//f,f1: same behaviour
//no guarantee of termination

static void g()
{
  static int i = 2;
  i--;
  if(i > 0)
    g();
}
//i is initialized once in the
//whole program
//g always terminate
```

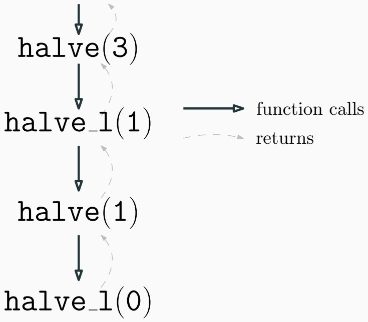
Mutual recursion

One can introduce a system of mutually recursive functions

```
static int halve_l(int);

static int halve(int n)
{
  if (n == 0)
    return 0;
  else
    return 1 + halve_l(n-1);
}

static int halve_l(int n)
{
  if (n == 0)
    return 0;
  else
    return halve(n-1);
}
```



Using recursive functions

High-level considerations

Why use recursive functions over iterations?

Cons:

- Arguably less idomiatic in procedural languages like **Java**
- Harder to compile away function calls (so maybe less intuitive *at first*)
- Performances losses (minor)

Pros:

- Meaningful procedures w/o mutable variables — In previous slides: where you can put `final`s?

$\rightsquigarrow$ Easier to reason about — Can be thought of mathematical functions w/o side effects

- Allow to express easily more complicated control flow — Think of `gray`

Also, later, for traversing complex datastructure

Morality

Focus on writing correct code…

…so don't hesitate to use recursive functions when it helps

Our first problem to be solved by recursion

Problem

If I give you n undistinguishable socks, how many ways P_n do you have to group them pairwise?

For $n = 3$?

Our first problem to be solved by recursion

Problem

If I give you n undistinguishable socks, how many ways P_n do you have to group them pairwise?

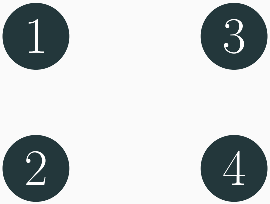
For $n = 3$? $\quad P_3 = 0$

Our first problem to be solved by recursion

Problem

If I give you n undistinguishable socks, how many ways P_n do you have to group them pairwise?

For $n = 3$? $P_3 = 0$ $n = 4$?

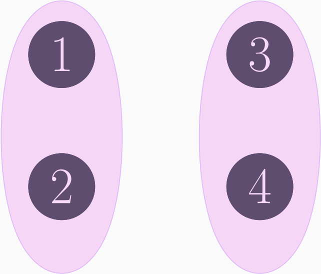


Our first problem to be solved by recursion

Problem

If I give you n undistinguishable socks, how many ways P_n do you have to group them pairwise?

For $n = 3$? $P_3 = 0$ $n = 4$?



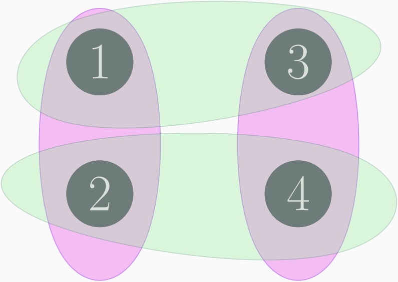
$\{\{1,2\},\{3,4\}\}$

Our first problem to be solved by recursion

Problem

If I give you n undistinguishable socks, how many ways P_n do you have to group them pairwise?

For $n = 3$? $P_3 = 0$ $n = 4$?



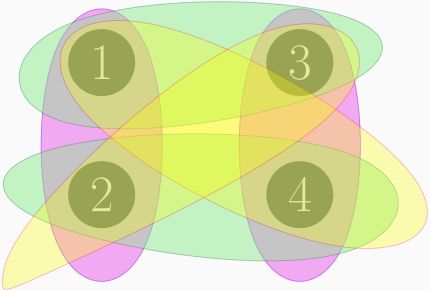
$\{\{1,2\},\{3,4\}\}$ $\{\{1,3\},\{2,4\}\}$

Our first problem to be solved by recursion

Problem

If I give you n undistinguishable socks, how many ways P_n do you have to group them pairwise?

For $n = 3$? $P_3 = 0$ $n = 4$?



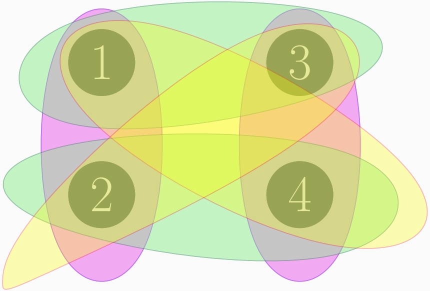
$\{\{1,2\},\{3,4\}\}$ $\{\{1,3\},\{2,4\}\}$ $\{\{1,4\},\{2,3\}\}$

Our first problem to be solved by recursion

Problem

If I give you n undistinguishable socks, how many ways P_n do you have to group them pairwise?

For $n = 3$? $P_3 = 0$ $n = 4$?



$\{\{1,2\},\{3,4\}\}$ $\{\{1,3\},\{2,4\}\}$ $\{\{1,4\},\{2,3\}\}$

$\rightarrow P_4 = 3$

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

- There is a single way of pairing 0 socks together and no way of pairing a single sock with no socks.

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

- There is a single way of pairing 0 socks together and no way of pairing a single sock with no socks.
- If I want to pair n socks $\{1, \dots n\}$

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

- There is a single way of pairing 0 socks together and no way of pairing a single sock with no socks.
- If I want to pair n socks $\{1, \ldots n\}$
  - I must necessarily pair n with another sock $k \in \{1, \ldots, n-1\}$

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

- There is a single way of pairing 0 socks together and no way of pairing a single sock with no socks.
- If I want to pair n socks $\{1, \dots n\}$
  - I must necessarily pair n with another sock $k \in \{1, \dots, n-1\}$

    $\rightarrow n-1$ ways of picking such a sock

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

- There is a single way of pairing 0 socks together and no way of pairing a single sock with no socks.
- If I want to pair n socks $\{1, \dots n\}$
  - I must necessarily pair n with another sock $k \in \{1, \dots, n-1\}$
    $\rightarrow n-1$ ways of picking such a sock
  - Then I can pair the remaining $n-2$ socks in $\{1, \dots, n-1\} - \{k\}$ arbitrarily

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

- There is a single way of pairing 0 socks together and no way of pairing a single sock with no socks.
- If I want to pair n socks $\{1, \dots n\}$
    - I must necessarily pair n with another sock $k \in \{1, \dots, n-1\}$
      $\rightarrow n - 1$ ways of picking such a sock
    - Then I can pair the remaining $n - 2$ socks in $\{1, \dots, n-1\} - \{k\}$ arbitrarily
      $\rightarrow P_{n-2}$ ways of doing that

Thinking this through with recursion

- Do we have an easy solution for inputs of size 0/small sizes?
- Can we obtain a solution for size n if I know the solutions for $k < n$?

- There is a single way of pairing 0 socks together and no way of pairing a single sock with no socks.
- If I want to pair n socks $\{1, \ldots n\}$
  - I must necessarily pair n with another sock $k \in \{1, \ldots, n-1\}$
    $\rightarrow n-1$ ways of picking such a sock
  - Then I can pair the remaining $n-2$ socks in $\{1, \ldots, n-1\} - \{k\}$ arbitrarily
    $\rightarrow P_{n-2}$ ways of doing that

Putting everything together

$$P_0 = 1 \qquad P_1 = 0 \qquad P_{n+2} = (n+1) \times P_n$$

In code

Easy to translate directly:

```
static int numberPairings(int n)
{
  switch(n)
  {
    case 0: return 1;
    case 1: return 0;
    default: return (n-1) * numberPairings(n-2);
  }
}
```

Complexity

$c_{n+2} = \mathcal{O}(1) + c_n \qquad c_0 = \mathcal{O}(1) \qquad c_1 = \mathcal{O}(1)$

In code

Easy to translate directly:

```
static int numberPairings(int n)
{
  switch(n)
  {
    case 0: return 1;
    case 1: return 0;
    default: return (n-1) * numberPairings(n-2);
  }
}
```

Complexity

$c_{n+2} = \mathcal{O}(1) + c_n \qquad c_0 = \mathcal{O}(1) \qquad c_1 = \mathcal{O}(1)$

So here $c_n = \mathcal{O}(n)$ (exponential complexity (the size of n is $\mathcal{O}(\log_2(n))$))

Computing the complexity of simple recursive functions

Typically, if we use recursion to reduce an input of size n to size $n-1$, we have a complexity satifying

$$u_{n+1} = a \times u_n + b \qquad \text{for } a, b, u_0 \geq 1$$

General recipe

- $u_n = \Theta(a^n)$ if $a > 1$
- $u_n = \Theta(n)$ if $a = 1$

Computing the complexity of simple recursive functions

Typically, if we use recursion to reduce an input of size n to size $n-1$, we have a complexity satifying

$$u_{n+1} = a \times u_n + b \qquad \text{for } a, b, u_0 \geq 1$$

General recipe

- $u_n = \Theta(a^n)$ if $a > 1$
- $u_n = \Theta(n)$ if $a = 1$

Proof for $a = 1$

Assume $m \leq a, b, u_0 \leq M$.
By induction, $mn \leq u_n \leq M(n+1)$.

Computing the complexity of simple recursive functions

Typically, if we use recursion to reduce an input of size n to size $n-1$, we have a complexity satifying

$$u_{n+1} = a \times u_n + b \qquad \text{for } a, b, u_0 \geq 1$$

General recipe

- $u_n = \Theta(a^n)$ if $a > 1$
- $u_n = \Theta(n)$ if $a = 1$

Proof for $a = 1$

Assume $m \leq a, b, u_0 \leq M$.

By induction, $mn \leq u_n \leq M(n+1)$.

Proof for $a > 1$

By induction $a^n m \leq u_n \leq M a^{n+1}$

Computing the complexity of simple recursive functions

Typically, if we use recursion to reduce an input of size n to size $n-1$, we have a complexity satifying

$$u_{n+1} = a \times u_n + b \qquad \text{for } a, b, u_0 \geq 1$$

General recipe

- $u_n = \Theta(a^n)$ if $a > 1$
- $u_n = \Theta(n)$ if $a = 1$

Proof for $a = 1$

Assume $m \leq a, b, u_0 \leq M$.
By induction, $mn \leq u_n \leq M(n+1)$.

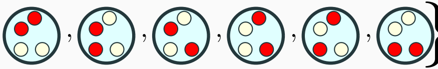
Proof for $a > 1$

By induction $a^n m \leq u_n \leq M a^{n+1}$

Maths exercise: exact solutions (Hint for $a \neq 1$: compute first $u_n - \ell$ for $\ell = a\ell + b$)

Problem

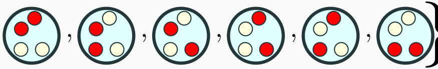
Compute the number of ways $\binom{n}{k}$ to pick k elements among n.

$\binom{4}{2} = \#\{$  $\}$

Extended example: binomial (1/4)

Problem

Compute the number of ways $\binom{n}{k}$ to pick k elements among n.

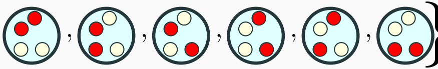
$\binom{4}{2} = \#\{$  $\}$

$$\binom{n}{k} = \#\{X \subseteq \{1, \ldots, n\} \mid \#X = k\} = \frac{n!}{k!(n-k)!}$$

Problem

Compute the number of ways $\binom{n}{k}$ to pick k elements among n.

$$\binom{4}{2} = \#\{$$



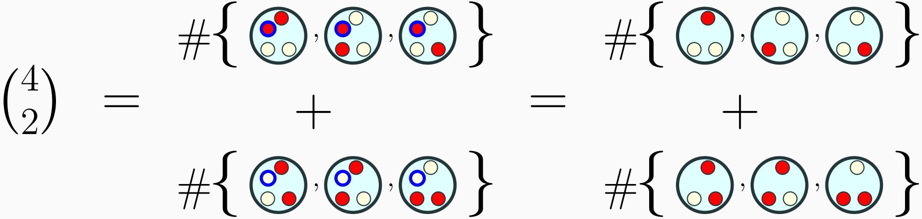
$$\}$$

$$\binom{n}{k} = \#\{X \subseteq \{1, \ldots, n\} \mid \#X = k\} = \frac{n!}{k!(n-k)!}$$

Issue with the closed formula: $n!$ overflows fast while $\binom{k}{n}$ is polynomial if $k = O(1)$.
Alternative way of computing?

Extended example: binomial (2/4)

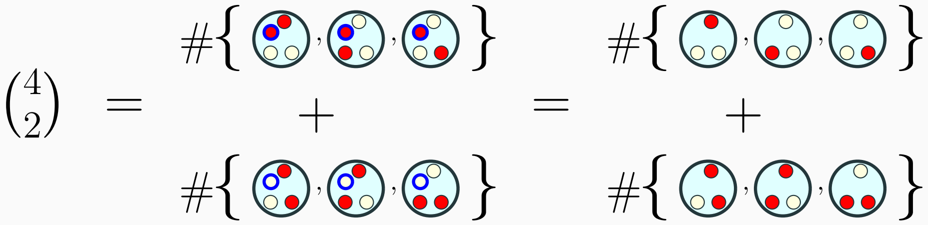
Decomposition by fixing an element and asking whether it is picked or not.



$$= \binom{3}{1} + \binom{3}{2}$$

Extended example: binomial (2/4)

Decomposition by fixing an element and asking whether it is picked or not.



$$= \binom{3}{1} + \binom{3}{2}$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Extended example: binomial (3/4)

```
int binom(int k, int n)
{
  if (k > n)
    return 0;
  if (k == 0 )
    return 1;
  else
    return binom(k-1,n-1) + binom(k,n-1);
}
```

Proof of termination: by induction over n.

Complexity

$u_{k,n} = \mathcal{O}(1)$ when $k > n$ or $k = 0$

$u_{k+1,n+1} = u_{k,n} + u_{k+1,n} + \mathcal{O}(1) \leq 2u_{k+1,n} + \mathcal{O}(1)$

Complexity

$u_{k,n} = \mathcal{O}(1)$ when $k > n$ or $k = 0$

$u_{k+1,n+1} = u_{k,n} + u_{k+1,n} + \mathcal{O}(1) \leq 2u_{k+1,n} + \mathcal{O}(1)$

So $u_{n,k} = \mathcal{O}(2^{n-k})$.

Complexity

$u_{k,n} = \mathcal{O}(1)$ when $k > n$ or $k = 0$

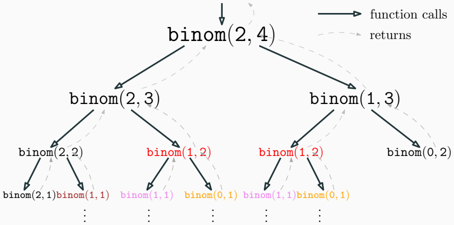
$u_{k+1,n+1} = u_{k,n} + u_{k+1,n} + \mathcal{O}(1) \leq 2u_{k+1,n} + \mathcal{O}(1)$

So $u_{n,k} = \mathcal{O}(2^{n-k})$.

(keeping in mind that the size of an integer n is $\log(n)$, this is double exponential complexity!)

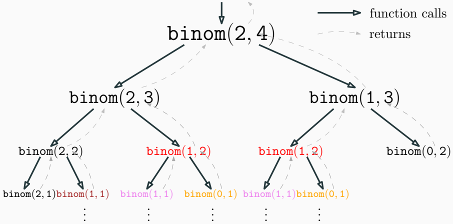
Extended example: binomial (3/5)

Issue: exponential number of calls (inefficient)



Extended example: binomial (3/5)

Issue: exponential number of calls (inefficient)



But there are redundant calls! Two ways of adressing this:

- Caching the common subcomputation a.k.a. (dynamic programming or memoization)
- Translating to an iterative program

Extended example: binomial (4/5)

- Assume `N` and `K` are sufficiently large for our needs.

Otherwise: bureaucratic memory management with `ArrayList`

```
final int N = 100;
final int K = 20;

final int[][] cache = new Array[K][N];
//assume that main() initializes cache with -1

static int binom(int k, int n)
{
  if (cache[k][n] != -1)
    return cache[k][n];
  if (k > n)
    return cache[k][n] = 0;
  if (k == 0)
    return cache[k][n] = 1;
  else
    return cache[k][n] = binom(k-1,n-1) + binom(k,n-1);
}
```

Extended example: binomial (4/5)

- Assume N and K are sufficiently large for our needs.

Otherwise: bureaucratic memory management with `ArrayList`

```
final int N = 100;
final int K = 20;

final int[][] cache = new Array[K][N];
//assume that main() initializes cache with -1

static int binom(int k, int n)
{
  if (cache[k][n] != -1)
    return cache[k][n];
  if (k > n)
    return cache[k][n] = 0;
  if (k == 0)
    return cache[k][n] = 1;
  else
    return cache[k][n] = binom(k-1,n-1) + binom(k,n-1);
}
```

Complexity?

Extended example: binomial (4/5)

- Assume N and K are sufficiently large for our needs.

Otherwise: bureaucratic memory management with `ArrayList`

```
final int N = 100;
final int K = 20;

final int[][] cache = new Array[K][N];
//assume that main() initializes cache with -1

static int binom(int k, int n)
{
  if (cache[k][n] != -1)
    return cache[k][n];
  if (k > n)
    return cache[k][n] = 0;
  if (k == 0)
    return cache[k][n] = 1;
  else
    return cache[k][n] = binom(k-1,n-1) + binom(k,n-1);
}
```

Complexity? Hint: bound the number of recursive calls

Extended example: binomial (4/5)

- Assume `N` and `K` are sufficiently large for our needs.

Otherwise: bureaucratic memory management with `ArrayList`

```
final int N = 100;
final int K = 20;

final int[][] cache = new Array[K][N];
//assume that main() initializes cache with -1

static int binom(int k, int n)
{
  if (cache[k][n] != -1)
    return cache[k][n];
  if (k > n)
    return cache[k][n] = 0;
  if (k == 0)
    return cache[k][n] = 1;
  else
    return cache[k][n] = binom(k-1,n-1) + binom(k,n-1);
}
```

Complexity? Hint: bound the number of recursive calls $\mathcal{O}(k \times n)$

Extended example: binomial (5/5)

```
final int N = 100;
final int K = 20;

int binom[K][N];
binom[0][0] = 1;
for(int n = 1; n < N; n++)
{
  binom[0][n] = 1;
  for(int k = 1; k <= min(n,K); k++)
    binom[k][n] = binom[k][n-1] + binom[k-1][n-1];
}
```

- The proof of correctness is slightly more subtle

  Need to reason about the mutable values of `binom`[`k`][`n`]

- The recursive variant is easier to write and an acceptable naive first implementation!
- Fill all the values upfront

  (the other method is better for incremental computation)

Extended example: binomial (5/5)

```
final int N = 100;
final int K = 20;

int binom[K][N];
binom[0][0] = 1;
for(int n = 1; n < N; n++)
{
  binom[0][n] = 1;
  for(int k = 1; k <= min(n,K); k++)
    binom[k][n] = binom[k][n-1] + binom[k-1][n-1];
}
```

- The proof of correctness is slightly more subtle

  Need to reason about the mutable values of `binom`[`k`][`n`]
- The recursive variant is easier to write and an acceptable naive first implementation!
- Fill all the values upfront

  (the other method is better for incremental computation)

Complexity?

Extended example: binomial (5/5)

```
final int N = 100;
final int K = 20;

int binom[K][N];
binom[0][0] = 1;
for(int n = 1; n < N; n++)
{
  binom[0][n] = 1;
  for(int k = 1; k <= min(n,K); k++)
    binom[k][n] = binom[k][n-1] + binom[k-1][n-1];
}
```

- The proof of correctness is slightly more subtle

  Need to reason about the mutable values of `binom`[`k`][`n`]

- The recursive variant is easier to write and an acceptable naive first implementation!
- Fill all the values upfront

  (the other method is better for incremental computation)

Complexity? $\mathcal{O}(K \times N)$

Exercise from last week to revisit

```java
/* Assumptions: arr contains an increasing
                sequence of values
                arr[mi] <= 0 and arr[ma] >=0*/
static int dicho_rec(int[] arr, int mi, int ma)
{
   if (ma <= mi)
      return mi;
   final int mid = (ma+mi)/2;
   if (arr[mid] <= 0)
     return dicho_rec(arr,mid,ma);
   else
     return dicho_rec(arr,mi,mid);
}
```

```java
static int dicho_iter(int[] arr, int mi, int ma)
{
   while (ma > mi)
   {
     int mid = (ma+mi)/2;
     if (arr[mid] <= 0)
       mi = mid;
     else
       ma = mid;
   }
   return mi;
}
```

In a nutshell

Recursion

- Seemingly circular definitions, but productive because you define a task in terms of smaller tasks
- Can seamlessly be used in most programming languages
- Might be harder to trace executions but…
- …very intuitive abstraction for seemingly stateless computations and problem-solving

Next

- Two other paradigmatic case of recursion
    - greedy algorithms
    - divide-and-conquer
- One class of motivating examples: sorting algorithms
- A bit more of dynamic programming/memoization

Important

No systematic way of coming up with efficient algorithms

$\rightarrow$ Practice is key!