Algorithms, Data Structures, and Problem Solving

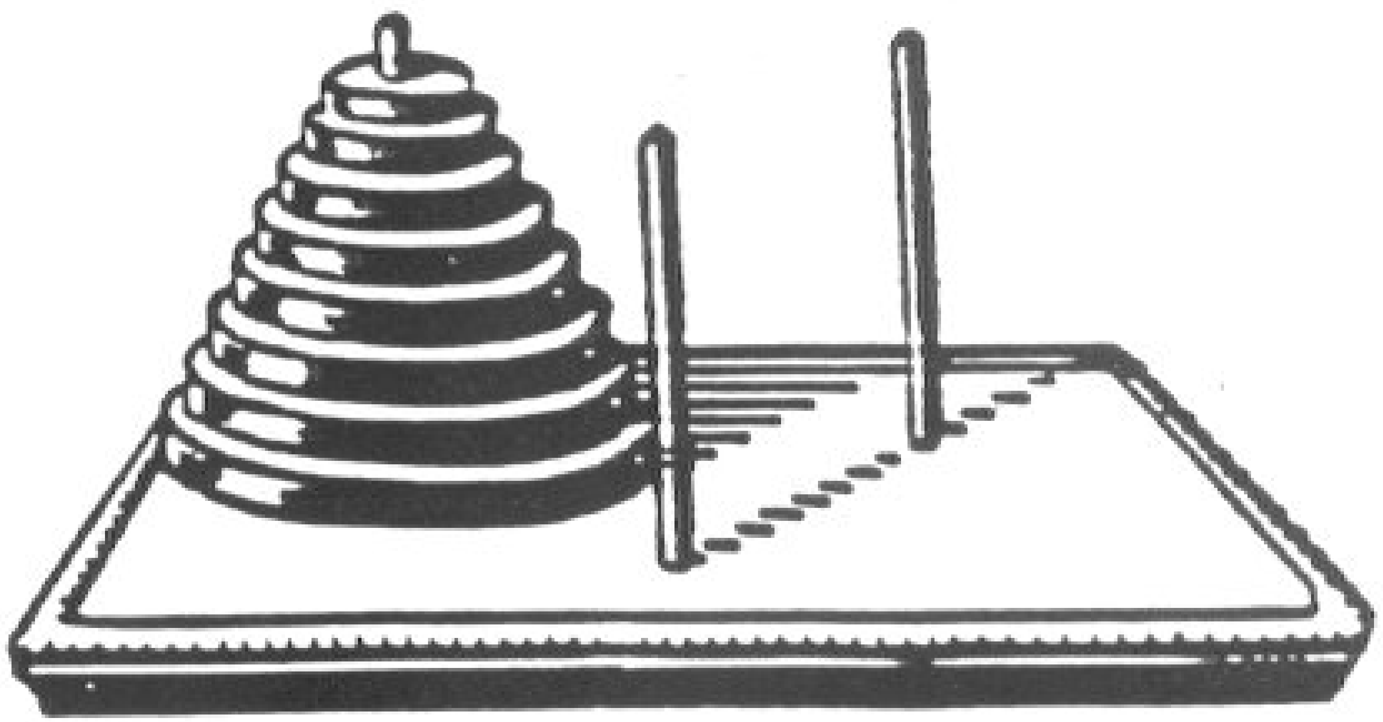
Masoumeh Taromirad

Hamlstad University

HÖGSKOLAN I HALMSTAD

DA4002, Fall 2016

Motivating Example:
Tower of Hanoi

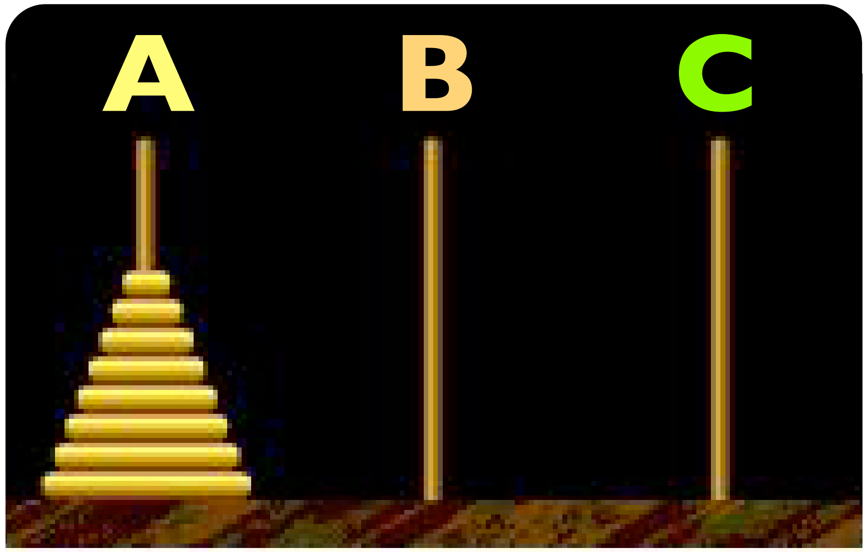


- move the stack of disks to another pole
- you may only place smaller disks on larger ones

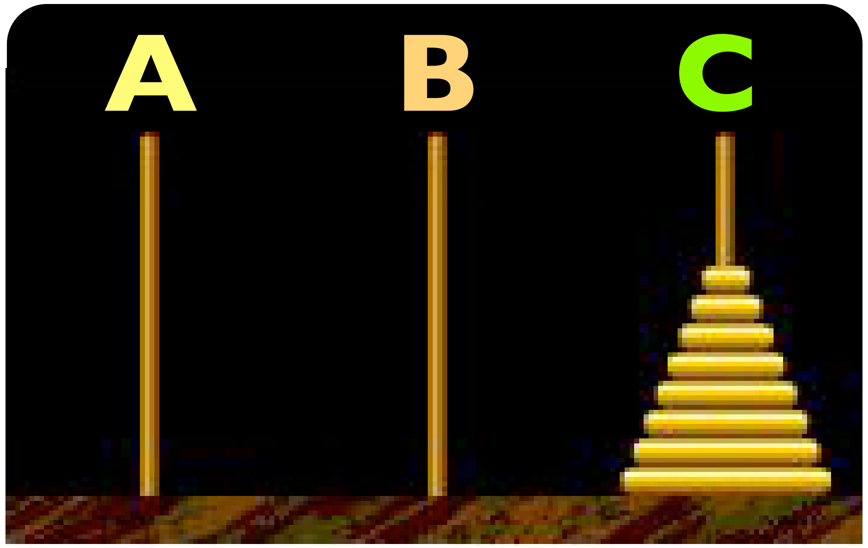
André Karwath http://en.wikipedia.org/wiki/File:Tower_of_Hanoi_4.gif

Motivating Example:
Tower of Hanoi

setup

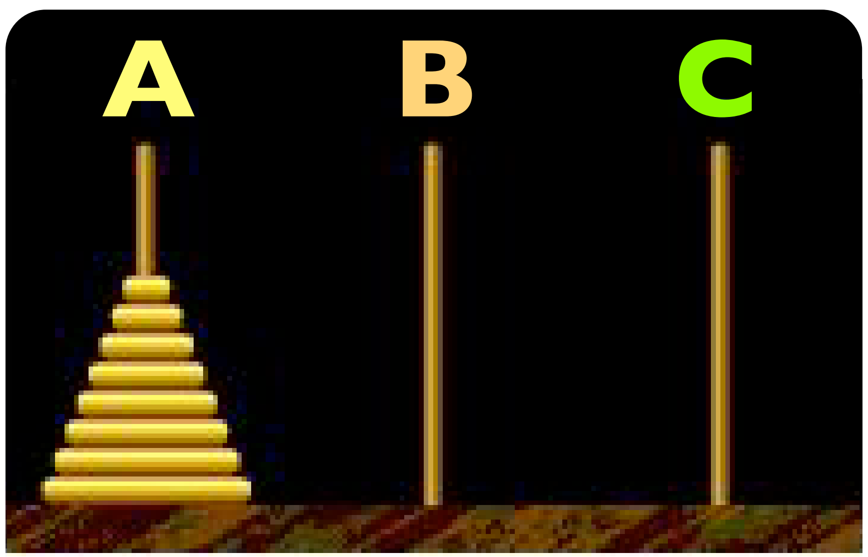


goal



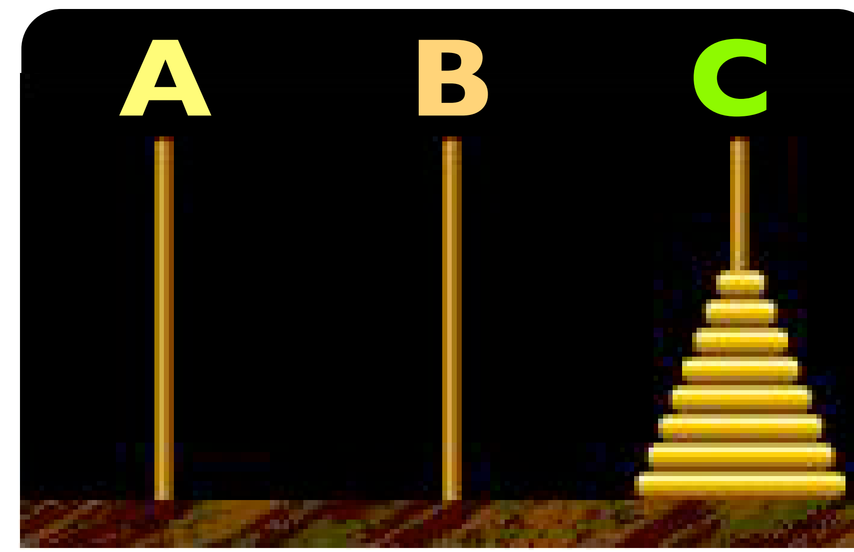
Motivating Example:
Tower of Hanoi

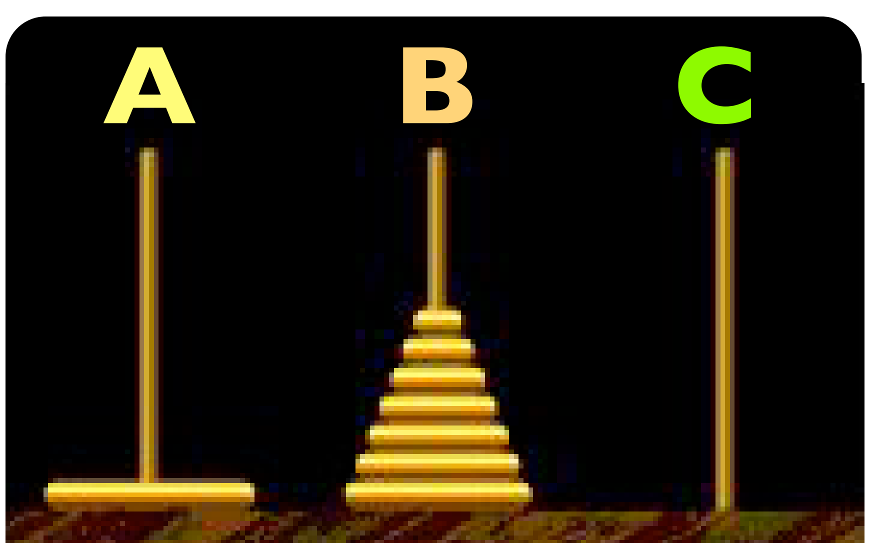
setup



step 1:
move **N-1** disks
from **A** to **B**

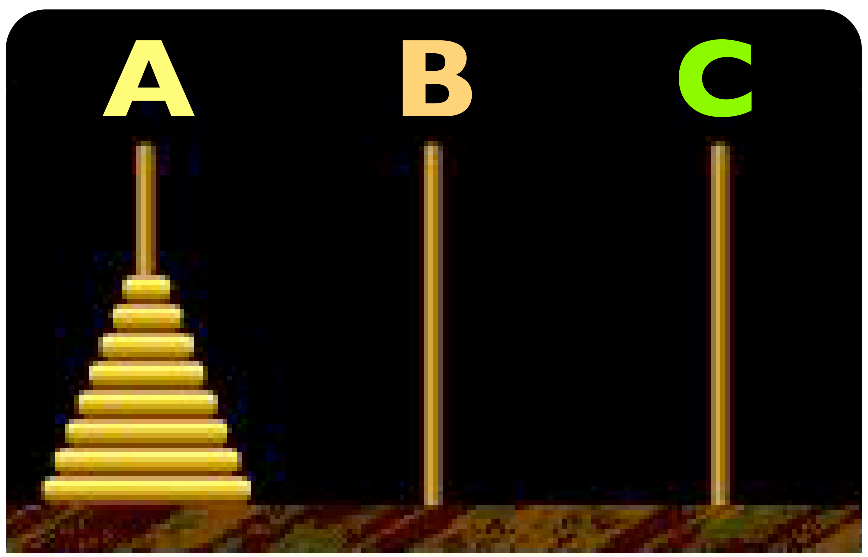
goal



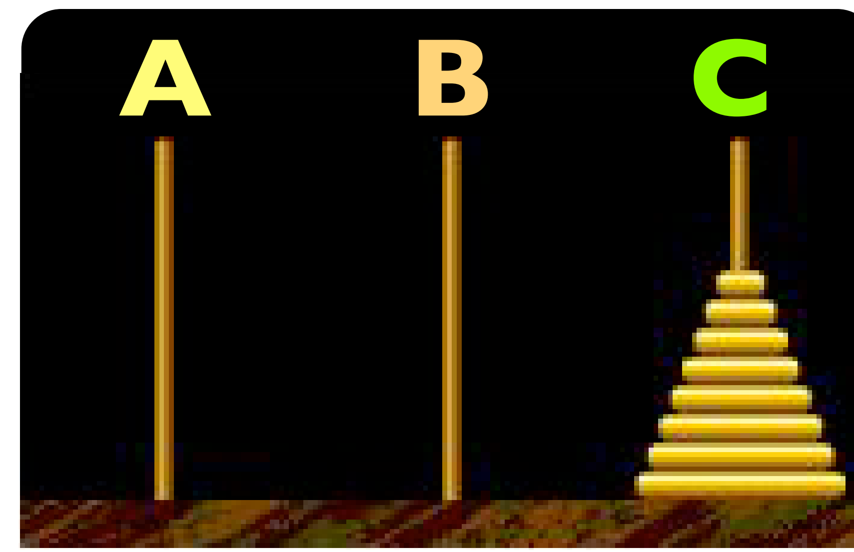


Motivating Example:
Tower of Hanoi

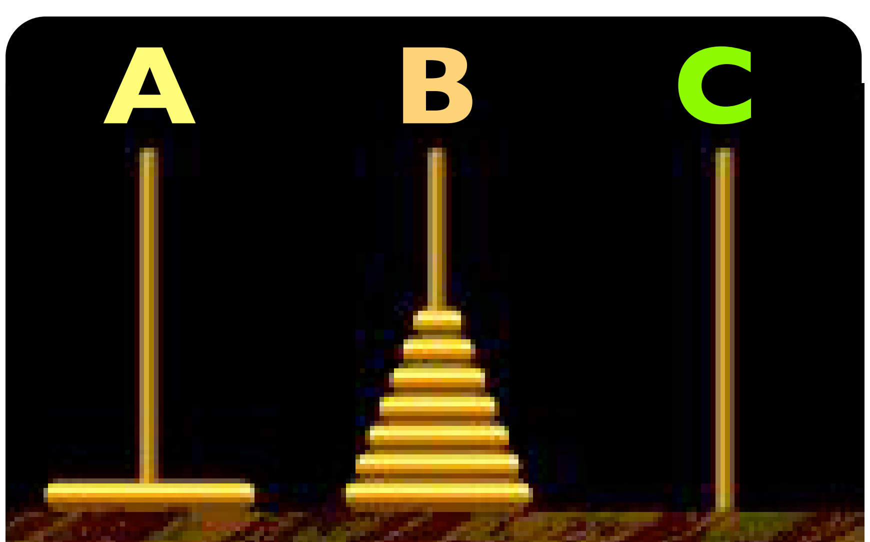
setup

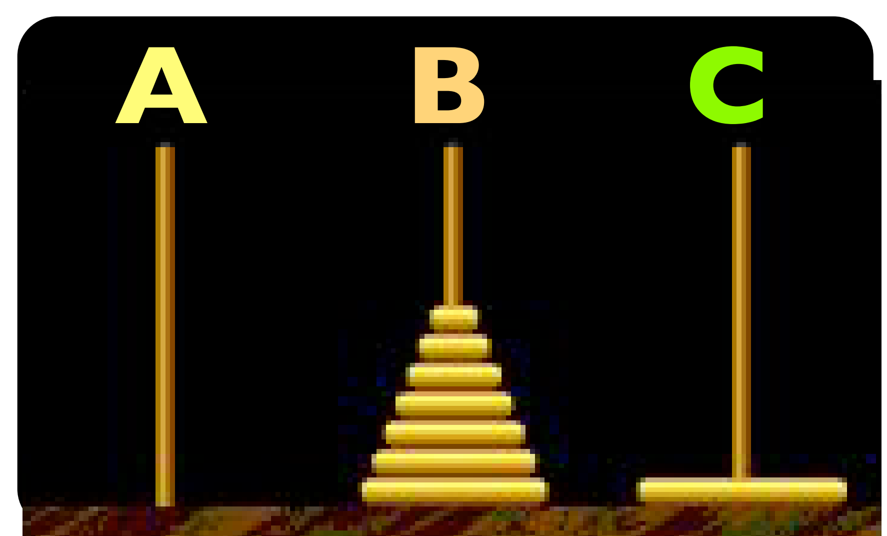


goal



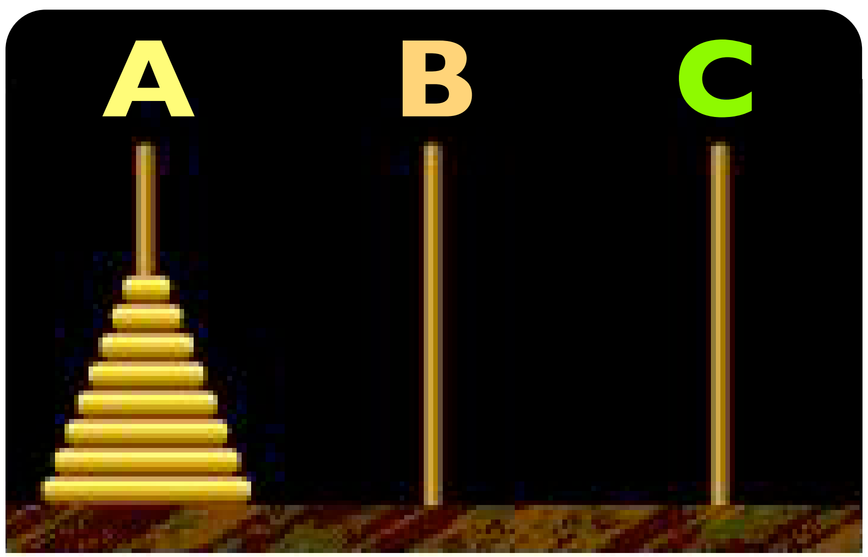
step 2:
move **biggest** disk from **A** to **C**



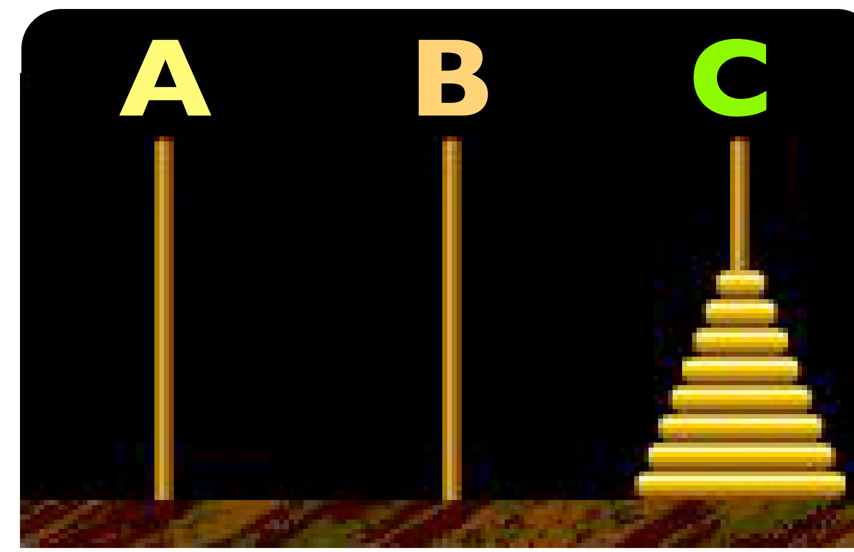


Motivating Example:
Tower of Hanoi

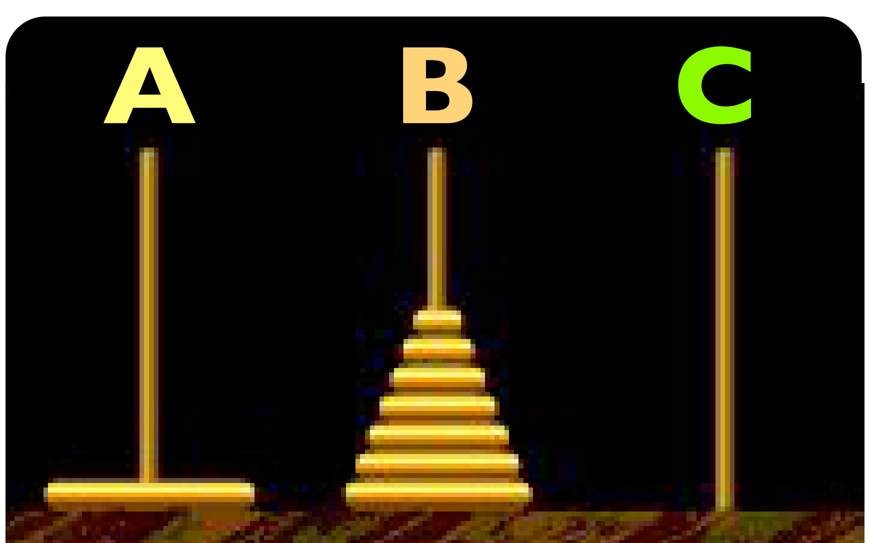
setup

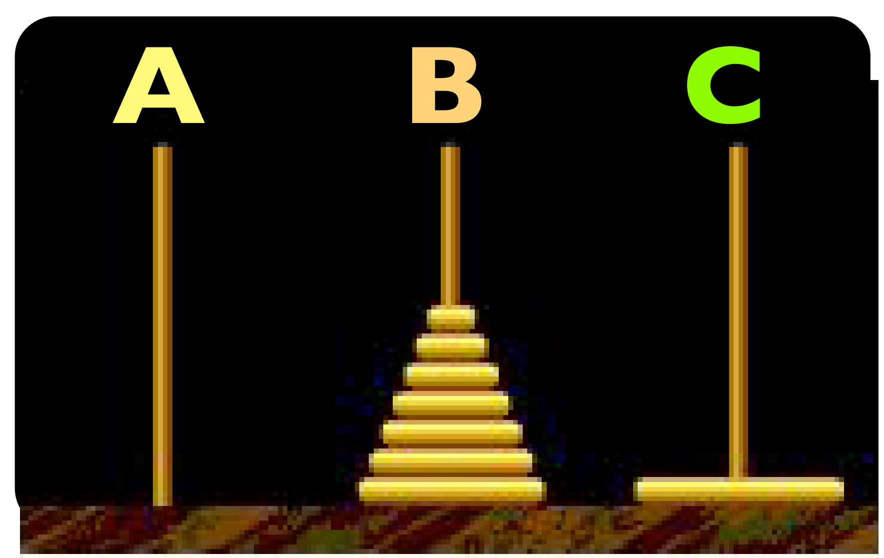


goal



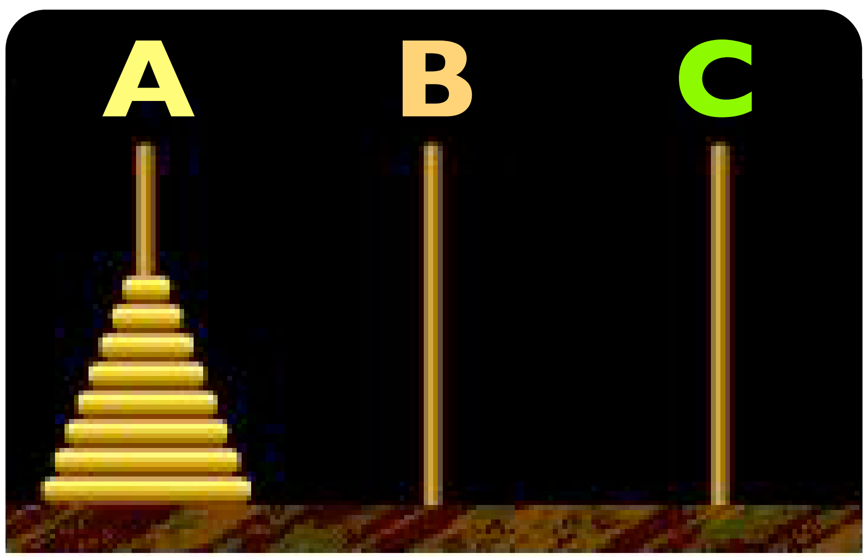
step 3:
move **N-1** disks
from **B** to **C**



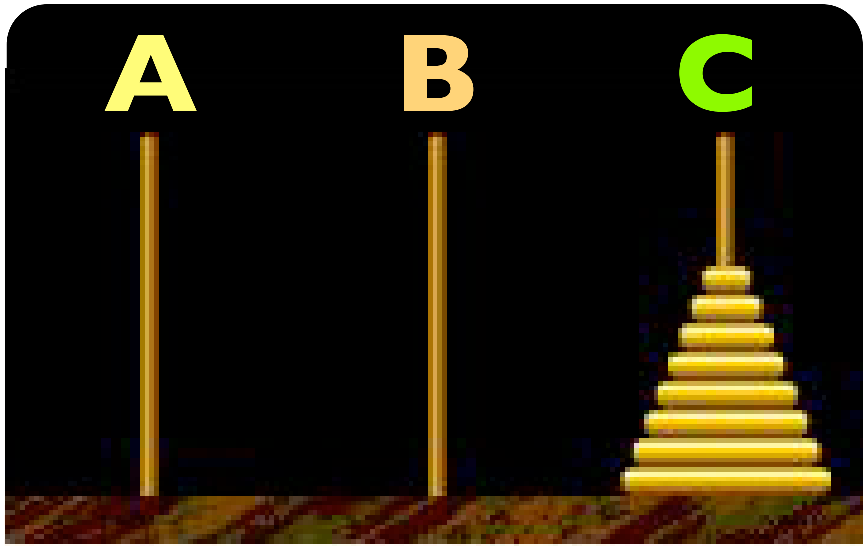


Motivating Example:
Tower of Hanoi

setup



goal



step 3:
move **N-1** disks
from **B** to **C**

smaller sub-problem:
use recursion to solve it,
reapplying the same method

Today's Lecture

- *detecting container types from code*
- exercise discussion
- *estimating runtimes with Big-Oh*
- *when is Big-Oh useful?*
- Divide & Conquer
    - *maximum subsequence*
    - closest pair of points
- memoization
    - *memoization for Fibonacci sequence*
    - outlook on Dynamic Programming

exercise discussion

- A: `T(N) = 150 N log N`
- B: `T(N) = N*N`
  - program A better for large N *(but not "always faster"!)*
  - program B better for small N *(but not "always faster"!)*
  - cannot answer about average performance

$$\left.\begin{array}{r} T(N) = cF(N) \\ T(N') = cF(N') \end{array}\right\} \Leftrightarrow \frac{T}{F(N)} = \frac{T'}{F(N')} \Leftrightarrow \begin{cases} T' & = \frac{F(N')}{F(N)} T \\ N' & = F^{-1}\left(\frac{T'}{T} F(N)\right) \end{cases}$$

Estimating Runtimes with Big-Oh

another good exam question...

When is Big-Oh Useful?

formulate advantages and disadvantages

Divide & Conquer
problem-solving methodology

To solve a problem with the Divide and Conquer methodology, do the following.

1. Identify (significantly) smaller sub-problems of the same type as the original problem.
2. Solve each sub-problem using recursion, terminating at trivially small sub-problems.
3. Combine sub-solutions into overall solution.

Divide & Conquer
problem-solving methodology

- D&C is a simple idea...
    - ...and also more of an art than a science.
- We look at some examples today.
- Beware of a common pitfall: overlapping subproblems.
    - easy answer: memoization *(today)*
    - better answer: Dynamic Programming *(next week)*

Example: Max Subsequence Sum

given a sequence of integers

$$\{A_i \in \mathbb{N}\} = \{A_1, A_2, \ldots, A_N\}$$

find the subsequence (from *i* to *j*)
which maximizes the sum

$$\sum_{k=i}^{j} A_k$$

(the sum is zero if all integers are negative)

Example: Max Subsequence Sum

```
maxSum = 0;
for (ii = 0; ii < length; ++ii) {
 for (jj = ii; jj < length; ++jj) {
  sum = 0;
  for (kk = ii; kk <= jj; ++kk) {
   sum += aa[kk];
  }
  if (sum > maxSum) {
   maxSum = sum;
   first = ii;
   last = jj;
  }
 }
}
```

$O(N^3)$

Example: Max Subsequence Sum

```
maxSum = 0;
for (ii = 0; ii < length; ++ii) {
 for (jj = ii; jj < length; ++jj) {
  sum = 0;
  for (kk = ii; kk <= jj; ++kk) {
   sum += aa[kk];
  }
  if (sum > maxSum) {
   maxSum = sum;
   first = ii;
   last = jj;
  }
 }
}
```

Example: Max Subsequence Sum

```
maxSum = 0;
for (ii = 0; ii < length; ++ii) {
 sum = 0;
 for (jj = ii; jj < length; ++jj) {
  sum += aa[jj];
  if (sum > maxSum) {
   maxSum = sum;
   first = ii;
   last = jj;
  }
 }
}
```

O(N²)

Group Activity

Divide and Conquer the Max Subsequence

apply D&C to a specific problem

Max Subsequence Revisited

```
maxSum = 0;
for (ii = 0; ii < length; ++ii) {
 sum = 0;
 for (jj = ii; jj < length; ++jj) {
  sum += aa[jj];
  if (sum > maxSum) {
   maxSum = sum;
   first = ii;
   last = jj;
  }
 }
}
```

we can eliminate this loop!

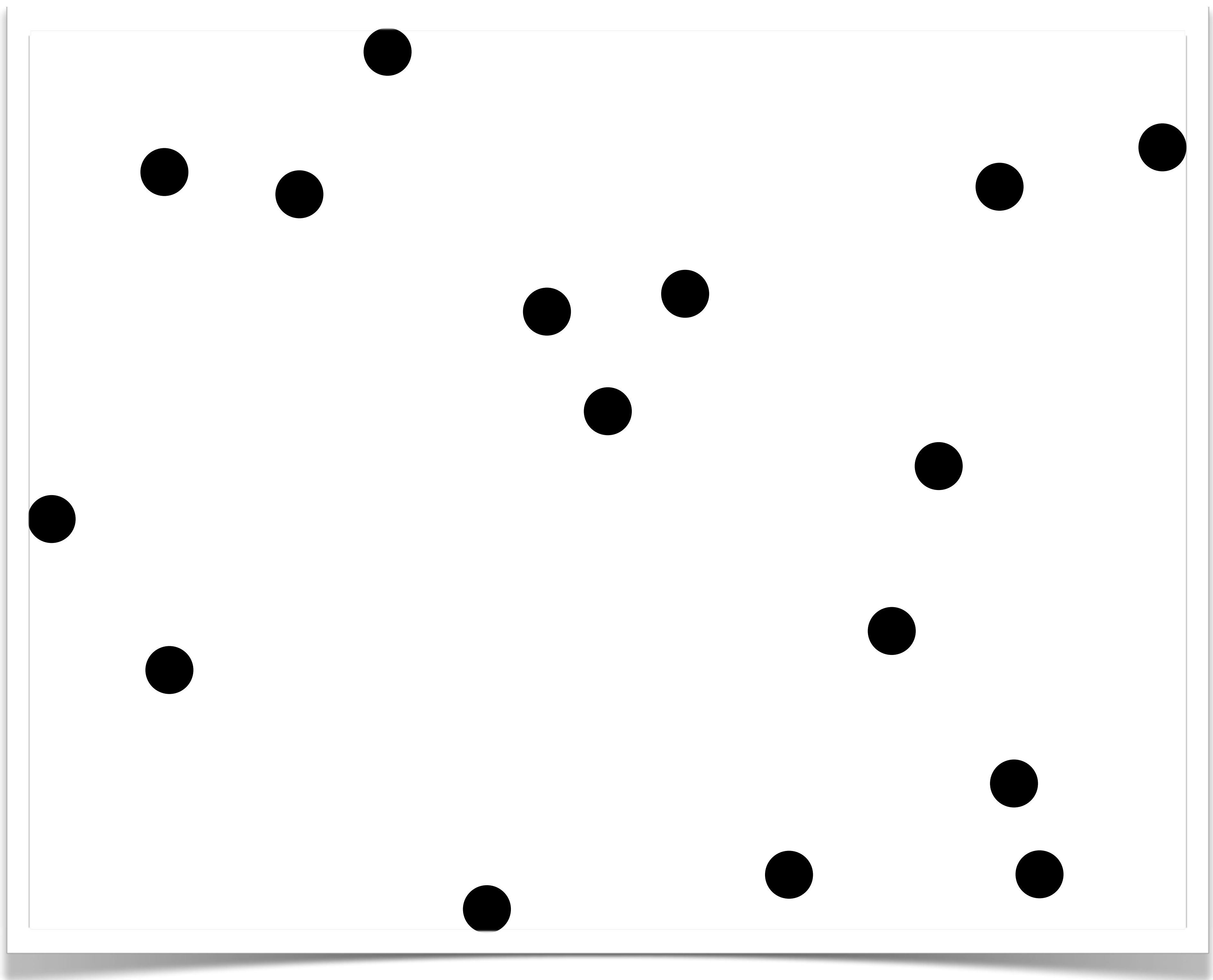
Max Subsequence Revisited

```
maxSum = 0;
sum = 0;
for (ii=0, jj=0; jj < length; ++jj) {
 sum += aa[jj];
 if (sum > maxSum) {
  maxSum = sum;
  first = ii;
  last = jj;
 }
 else if (sum < 0) {
  ii = jj + 1;
  sum = 0;
 }
}
```

The max subsequence never starts with a negative-sum sub-subsequence.

So we only scan for its end, resetting the beginning so it never contains negatives.

Closest Pair of Points



Subproblem Overlap

- Divide & Conquer *(and recursion generally)* is great, but it can also go wrong.

Subproblem Overlap Example:
Fibonacci Numbers

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

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

Subproblem Overlap Example:
Fibonacci Numbers

```
int fibRec(int nn) {
 if (2 > nn) {
  return 1;
 }
 return fibRec(nn - 2) + fibRec(nn - 1);
}
```

Subproblem Overlap Example: Fibonacci Numbers

```
int fibRec(int nn) {
 if (2 > nn) {
  return 1;
 }
 return fibRec(nn - 2) + fibRec(nn - 1);
}
```

solve smaller subproblems

Subproblem Overlap Example: Fibonacci Numbers

```
int fibRec(int nn) {
 if (2 > nn) {
  return 1;
 }
 return fibRec(nn - 2) + fibRec(nn - 1);
}
```

will call F(n-4) and F(n-3)

will call F(n-3) and F(n-2)

Subproblem Overlap Example: Fibonacci Numbers

```
int fibRec(int nn) {
 if (2 > nn) {
  return 1;
 }
 return fibRec(nn - 2) + fibRec(nn - 1);
}
```

duplicate!

will call F(n-4) and F(n-3) will call F(n-3) and F(n-2)

duplicate!

Subproblem Overlap Example:
Fibonacci Numbers

will call F(n-4) and F(n-3)

will call F(n-6) and F(n-5)

will call F(n-5) and F(n-4)

will call F(n-3) and F(n-2)

will call F(n-5) and F(n-4)

will call F(n-4) and F(n-3)

Subproblem Overlap Example:

Fibonacci Numbers

will call F(n-4) and F(n-3)

will call F(n-6) and F(n-5)

will call F(n-5) and F(n-4)

will call F(n-3) and F(n-2)

will call F(n-5) and F(n-4)

will call F(n-4) and F(n-3)

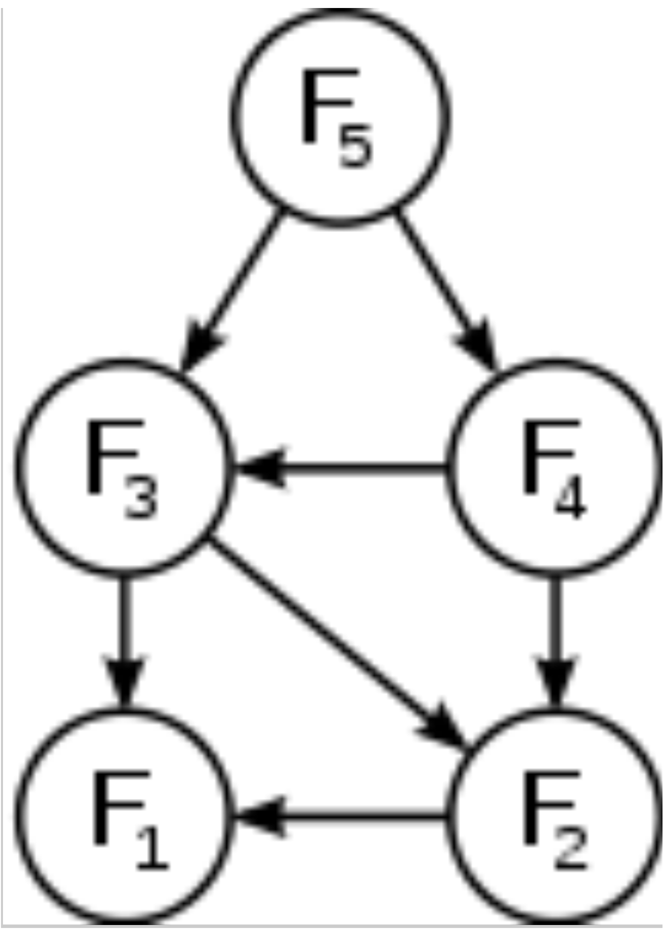
Subproblem Overlap Example: Fibonacci Numbers

- Execution trace for F(n=6)

```
fibRec(6)
 fibRec(4)
  fibRec(2)
   fibRec(0)
   fibRec(1)
  fibRec(3)
   fibRec(1)
   fibRec(2)
    fibRec(0)
    fibRec(1)
```

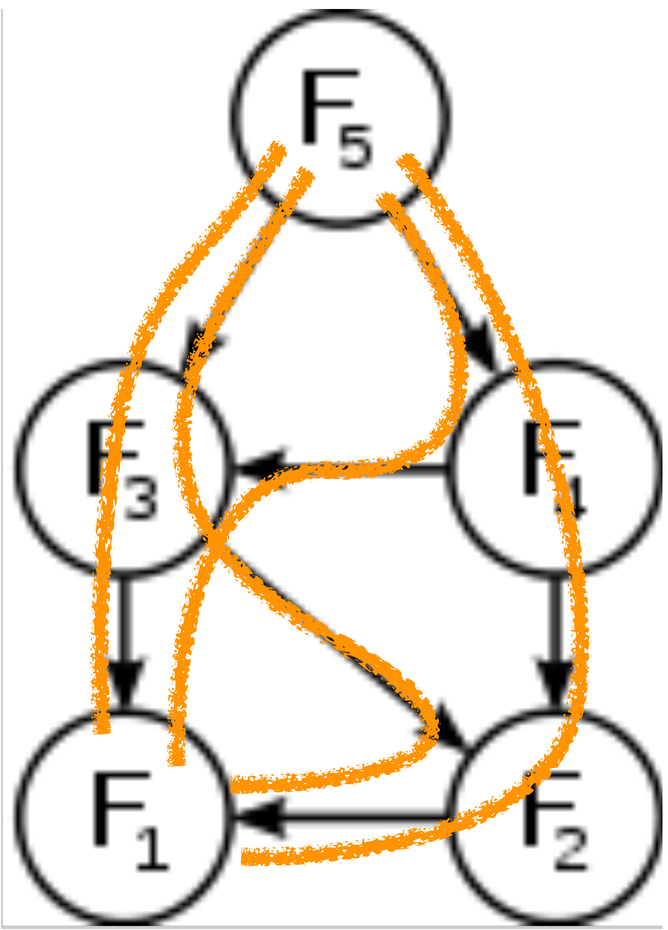
```
fibRec(5)
 fibRec(3)
  fibRec(1)
  fibRec(2)
   fibRec(0)
   fibRec(1)
 fibRec(4)
  fibRec(2)
   fibRec(0)
   fibRec(1)
```

Subproblem Overlap Example:
Fibonacci Numbers



The dependency structure is not a tree, but a graph.

Subproblem Overlap Example: Fibonacci Numbers



For example:
there are four paths from F(5) to F(1)
and **`fibRec`** ***walks each of them.***

So, how do we solve this problem?

Memoization

"[...] avoid repeating the calculation of results for previously processed inputs." (Wikipedia)

1. store new sub-solutions in a lookup table
2. reuse old sub-solutions when available

```c
Memo * memo_create ();
void   memo_destroy (Memo * memo);
int    memo_get (Memo * memo, int ii);
void   memo_set (Memo * memo, int ii, int fi);

int fibMemo (int ii) {
  static Memo * memo = NULL;
  int fi;
  if (NULL == memo) {
    memo = memo_create ();
  }
  fi = memo_get (memo, ii);
  if (0 < fi) {
    return fi;
  }
  fi = fibMemo (ii-1) + fibMemo (ii-2);
  memo_set (memo, ii, fi);
  return fi;
}
```

Fibonacci with Memoization

- **execution** trace for F(n=6)

```
fibRecMemo: compute(6)...
 fibRecMemo: compute(4)...
  fibRecMemo: compute(2)...
   fibRecMemo: lookup[0] = 1
   fibRecMemo: lookup[1] = 1
  fibRecMemo: compute(3)...
   fibRecMemo: lookup[1] = 1
   fibRecMemo: lookup[2] = 2
 fibRecMemo: compute(5)...
  fibRecMemo: lookup[3] = 3
  fibRecMemo: lookup[4] = 5
```

Group Activity

Detecting Container Types from Code

this makes a good exam question...

Group Activity

Memoization for Fibonacci

implement according to a given interface

Outlook on Dynamic Programming: Memoization is not the only way...

```
int fibIter(int nn) {
 int v1 = 1;
 int v2 = 1;
 int vv = 1;
 for (int ii = 2; ii <= nn; ++ii) {
  vv = v1 + v2;
  v2 = v1;
  v1 = vv;
 }
 return vv;
}
```

```c
int fibIter(int nn) {
 int v1 = 1;
 int v2 = 1;
 int vv = 1;
 for (int ii = 2;
      ii <= nn; ++ii) {
  vv = v1 + v2;
  v2 = v1;
  v1 = vv;
 }
 return vv;
}
```

```c
Memo * memo_create ();
void   memo_destroy (Memo * memo);
int    memo_get (Memo * memo, int ii);
void   memo_set (Memo * memo, int ii, int fi);

int fibMemo (int ii) {
  static Memo * memo = memo_create ();
  int fi;
  fi = memo_get (ii);
  if (0 < fi) {
    return fi;
  }
  fi = fibMemo (ii-1) + fibMemo (ii-2);
  memo_set (ii, fi);
  return fi;
}
```

How do these alternatives compare?

```c
int fibIter(int nn) {
 int v1 = 1;
 int v2 = 1;
 int vv = 1;
 for (int ii = 2;
      ii <= nn; ++ii) {
  vv = v1 + v2;
  v2 = v1;
  v1 = vv;
 }
 return vv;
}
```

this also takes effort to implement

```c
Memo * memo_create ();
void   memo_destroy (Memo * memo);
int    memo_get (Memo * memo, int ii);
void   memo_set (Memo * memo, int ii, int fi);
```

```c
int fibMemo (int ii) {
  static Memo * memo = memo_create ();
  int fi;
  fi = memo_get (ii);
  if (0 < fi) {
    return fi;
  }
  fi = fibMemo (ii-1) + fibMemo (ii-2);
  memo_set (ii, fi);
  return fi;
}
```

Dynamic Programming is a methodology to help find this kind of solution.

Take-Home Message

Divide and Conquer:

1. identify sub-problems
2. solve sub-problems recursively
3. combine sub-solutions

Memoization

- avoid duplicate computation due to sub-problem overlap by storing sub-solutions in a lookup table
- “shallow” but practical solution

Function Call Mechanism

A quick look into implementation

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9           cumul(2);
```

instruction pointer: i9

stack pointer: s0

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9             cumul(2);
```

instruction pointer: i9

stack pointer: s1

s1	argument **val**	2
	return value	?
	return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9           cumul(2);
```

instruction pointer: **i1**

stack pointer: **s1**

s1	argument **val**	**2**
	return value	?
	return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9            cumul(2);
```

instruction pointer: `i1`

stack pointer: `s1`

s1

argument **`val`**	2
return value	?
return address	`i8`

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9             cumul(2);
```

instruction pointer: i5

stack pointer: s1

s1

argument **val**	2
return value	?
return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9           cumul(2);
```

instruction pointer: `i5`

stack pointer: `s2`

s2

argument **`val`**	1
return value	?
return address	`i4`

s1

argument **`val`**	2
return value	?
return address	`i8`

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5          cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9            cumul(2);
```

instruction pointer: **i1**

stack pointer: **s2**

s2

argument **val**	1
return value	?
return address	i4

s1

argument **val**	2
return value	?
return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5          cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9            cumul(2);
```

instruction pointer: i1

stack pointer: s2

s2	argument **val**	1
	return value	?
	return address	i4
s1	argument **val**	2
	return value	?
	return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9            cumul(2);
```

instruction pointer: i2

stack pointer: s2

s2		
	argument **val**	1
	return value	1
	return address	i4

s1		
	argument **val**	2
	return value	?
	return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +    1
i5          cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9             cumul(2);
```

instruction pointer: i4

stack pointer: s1

s2	argument **val**	1
	return value	1
	return address	i4
s1	argument **val**	2
	return value	?
	return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5          cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9              cumul(2);
```

instruction pointer: i4

stack pointer: s1

s1

argument **val**	**2**
return value	?
return address	i8

i0 int cumul (int val) {
i1 if (1 >= val) {
i2   return 1;
i3 }
i4 return 2 + 1
i5   ~~cumul(val-1);~~
i6 }
i7 /* ...later... */
i8 int val =
i9   cumul(2);

instruction pointer: i4

stack pointer: s1

s1

argument **val**	2
return value	?
return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return  2  +   1
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val =
i9               cumul(2);
```

instruction pointer: i4

stack pointer: s1

s1	argument **val**	2
	return value	3
	return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5          cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val = 3
i9          cumul(2);
```

instruction pointer: i8

stack pointer: s0

s1	argument **val**	2
	return value	3
	return address	i8

```
i0 int cumul (int val) {
i1  if (1 >= val) {
i2   return 1;
i3  }
i4  return val +
i5         cumul(val-1);
i6 }
i7 /* ...later... */
i8 int val = 3
i9             cumul(2);
```

instruction pointer: i8

stack pointer: s0

- arguments are not modified in the call<u>ing</u> function

  the passed values live in the previous stack frame
- arguments are local variables in the call<u>ed</u> function

  they live in the current stack frame
- functions can "call themselves" without interference

  the stack keeps track of suspended computations