Recursion

The divide and conquer paradigm

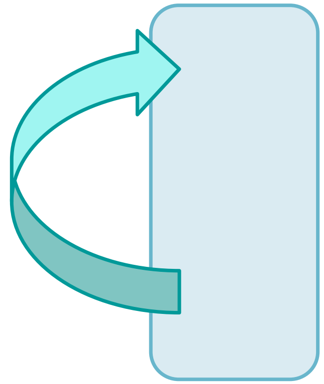
Stefano Quer

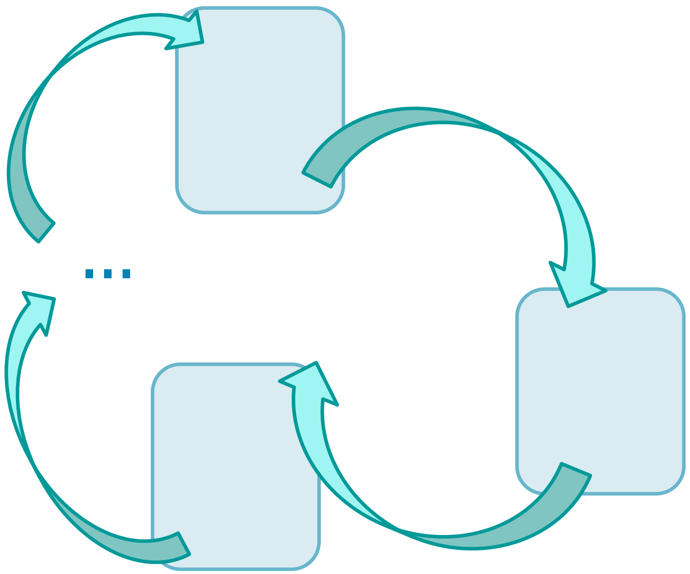
Dipartimento di Automatica e Informatica

Politecnico di Torino

Definition

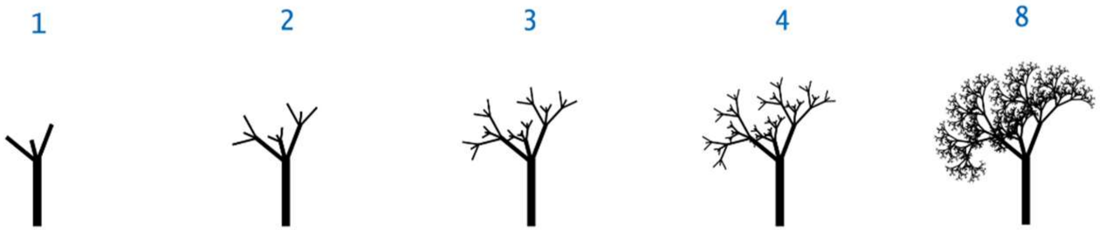
- Recursive procedure
  - **Direct** recursion
    - Inside its definition there is a call to the procedure itself
  - **Indirect** recursion
    - Inside its definition there is a call to at least one procedure that, directly or indirectly, calls the procedure itself

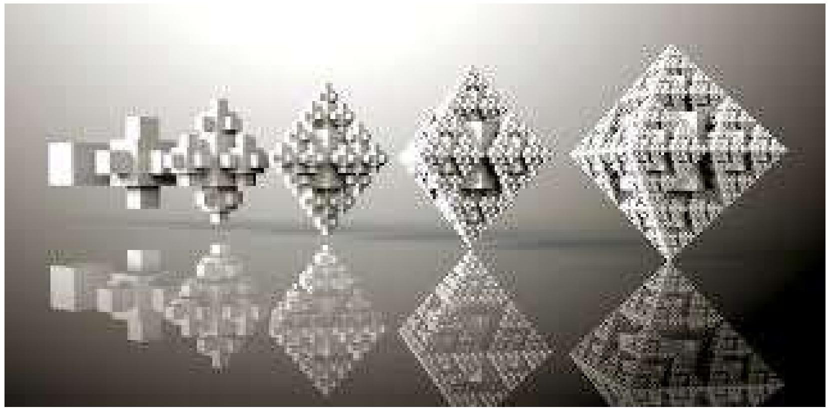




Definition

- Recursive algorithm
  - Based on recursive procedures





Definition

- The solution to a problem S applied to data D is recursive if we can express it as

Generic function (f) of …

D_{n-1} is simpler than D_n

$$S(D_n) = f\left(S(D_{n-1})\right) \qquad iff \quad D_n \neq D_0$$

$$S(D_0) = S_0 \qquad otherwise$$

Termination condition

The number of times we do apply f, i.e., $i = 0 \ldots n$ is the **recursion level**

Rationale

- Recursive solutions
  - Are mathematically elegant
  - Generate nice and neat procedures
- The nature of many problems is by itself recursive
  - Solution of many sub-problems may be similar to the initial one, though simpler and smaller
  - Combination of partial solutions may be used to obtain the solution of the initial problem
- Recursion is the basis for the problem-solving paradigm known as **divide and conquer**

The divide and conquer paradigm

- Recursion formulations should generate simpler and solvable sub-problems
  - We recur until
    - The sub-problems are trivial
    - All valid choices exhausted
- To reach this target, we apply the so called divide-and-conquer paradigm, which is based on 3 phases
  - Divide
  - Conquer
  - Combine

The divide and conquer paradigm

- Divide
  - Starting from a problem of size n
  - We partition it into $a \geq 1$ **independent** problems
  - Each of these problems has a size $\hat{n}$ smaller than n, i.e., $\hat{n} < n$
- Conquer
  - Solve an elementary problem
  - In this part of the algorithm we need a **termination condition**
    - All algorithms must eventually terminate
    - The recursion must be finite
- Combine
  - Build a global solution combining partial solutions

The divide and conquer paradigm

```
solve (problem){
  if (problem is elementary) {
    solution = solve_trivial (problem)
  } else {
    subproblem_{1,2,3,…,a} = divide (problem)

    for each s ∈ subproblem_{1,2,3,…,a}
      subsolution_s = solve (subproblem_s)

    solution = combine (subsolution_{1,2,3,…,a})
  }
  return solution
}
```

Termination condition

Conquer

Divide

Recursive call

Combine

a subproblems of size n'
Each subproblem is smaller than the original one (n'<n)

The divide and conquer paradigm

The else part is often avoided inserting one extra return

Conquer

```
solve (problem){
  if (problem is elementary) {
    solution = solve_trivial (problem)
    return (solution)
  }

  subproblem_{1,2,3,…,a} = divide (problem)

  for each s ∈ subproblem_{1,2,3,…,a}
    subsolution_s = solve (subproblem_s)
  solution = combine (subsolution_{1,2,3,…,a})

  return solution
}
```

Divide

Combine

Complexity Analysis

- A **recursion equation** expresses the time asymptotic cost $T(n)$ in terms of

$D(n)$	Cost of dividing the problem
$T(\hat{n})$	Cost of the execution time for smaller inputs (recursion phase)
$C(n)$	Cost of recombining the partial solutions
Cost of the terminal cases	We often assume unit cost for solving the elementary problems $\Theta(1)$

$$T(n) = D(n) + \sum T(\hat{n}) + C(n)$$

Complexity Analysis

$$T(n) = D(n) + \sum T(\hat{n}) + C(n)$$

We suppose that the number of subproblems of size $\hat{\boldsymbol{n}}$ is a.
If $a = 1$, we have linear recursion.
If $a > 1$, we have multi-way recursion.

Complexity Analysis

Case 1

$$T(n) = D(n) + \sum T(\hat{n}) + C(n)$$

We suppose that the number of subproblems of size $\hat{n}$ is a.
If $a = 1$ we have linear recursion.
If $a > 1$we have multi-way recursion.

The size of the original problem n and the generated ones $\hat{n}$ are related by a **constant factor** b in general the same for all subproblems
$b = {}^{n}/_{\hat{n}}$ and $\hat{n} = {}^{n}/_{b}$

1

Divide

Recur $T(\hat{n})$

Combine

$$T(n) = D(n) + a \cdot T\left({}^{n}/_{b}\right) + C(n) \qquad n > const$$
$$T(n) = \Theta(1) \qquad n \leq const$$

Conquer

If $b > 1$we have divide-and-conquer approach

Complexity Analysis

Case 2

$T(n) = D(n) + \sum T(\hat{n}) + C(n)$

We suppose that the number of subproblems of size $\hat{\boldsymbol{n}}$ is a.
If $a = 1$ we have linear recursion.
If $a > 1$we have multi-way recursion.

1

The size of the original problem $\boldsymbol{n}$ and the generated ones $\hat{\boldsymbol{n}}$ are related by a **constant value** k_i not always the same for all subproblems

$\hat{n} = n - k_i$

2

Divide

Recur $T(\hat{n})$

Combine

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n - k_i) + C(n) \qquad n > const$$

$$T(n) = \Theta(1) \qquad n \leq const$$

Conquer

If $b = 1$we have decrease-and-conquer approach

A first example: Array split

- Given an array of $n = 2^k$ integers

Simple case (complete tree of height k)

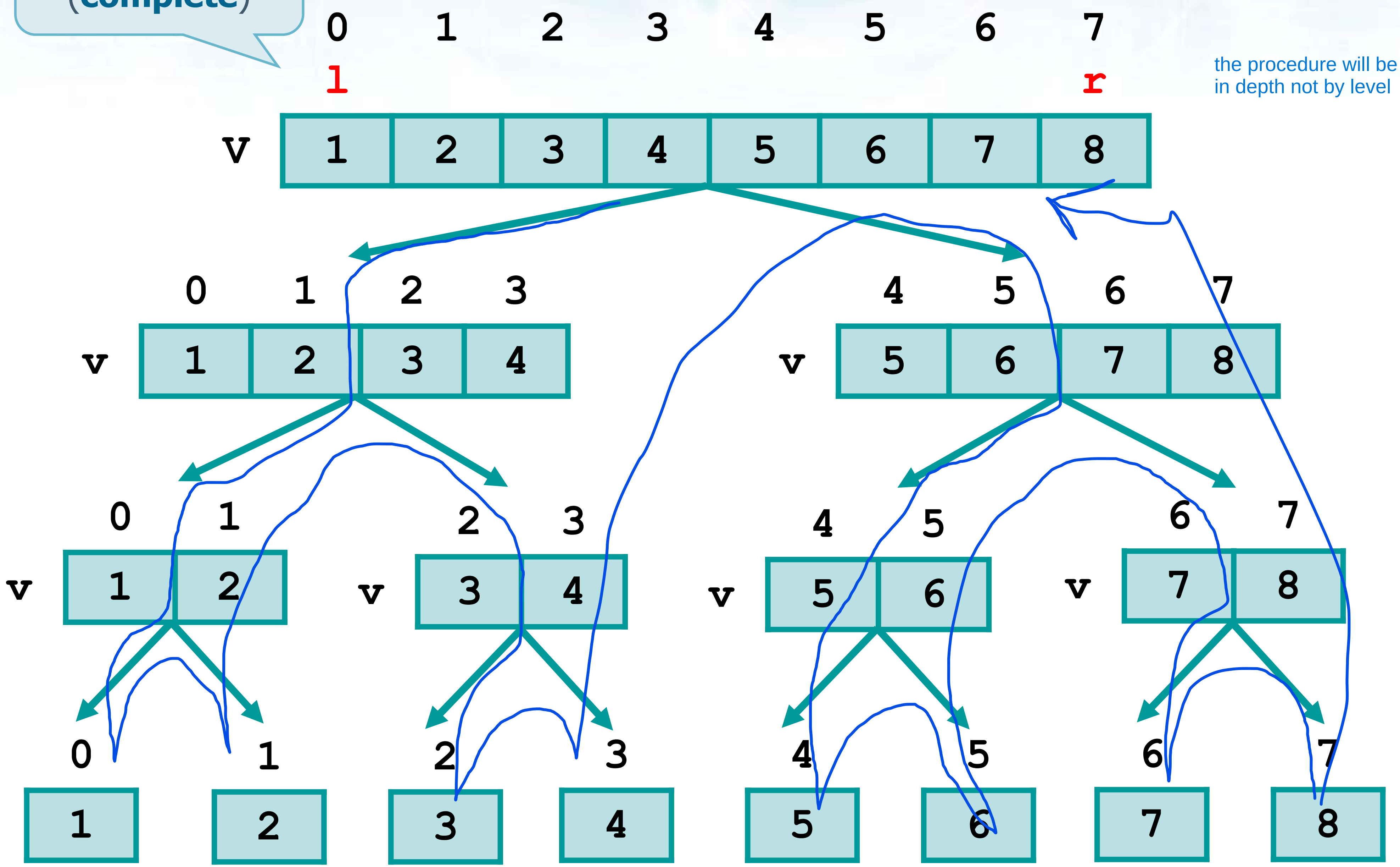
- Recursively partition it in sub-arrays half the size, until the termination condition is reached
  - The termination conditions is reached when sub-arrays have only 1 cell
- Print-out all generated partitions on standard output

Divide and conquer
At each step we generate a=2 subproblems
Each subproblem has a size equal to $\hat{n} = n/2$, i.e., $b = {}^{n}/_{\hat{n}} = 2$

Solution

Recursion tree (**complete**)

the procedure will be executed in depth not by level



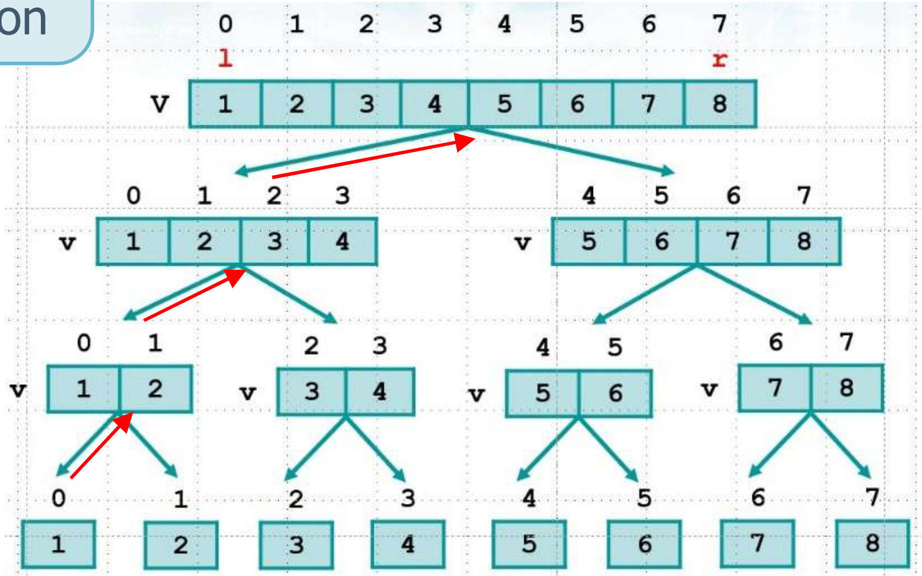
Solution 1

```
void show (
  int v[], int l, int r
) {
  int i, c;

  if (l >= r) {
    return;
  }

  c = (r+l)/2;
  show (v, l, c);
  show (v, c+1, r);

  return;
}
```

Termination condition

Recursion:
Left recursion
Right recursion

Recursion tree
(**visited depth-first**)



Solution 1

```
void show (
  int v[], int l, int r
) {
  int i, c;

  printf ("v = ");
  for (i=l; i<=r; i++)
    printf ("%d ", v[i]);
  printf ("\n");

  if (l >= r) {
    return;
  }

  c = (r+l)/2;
  show (v, l, c);
  show (v, c+1, r);

  return;
}
```

returns to the previous level, return will call again the function

divide

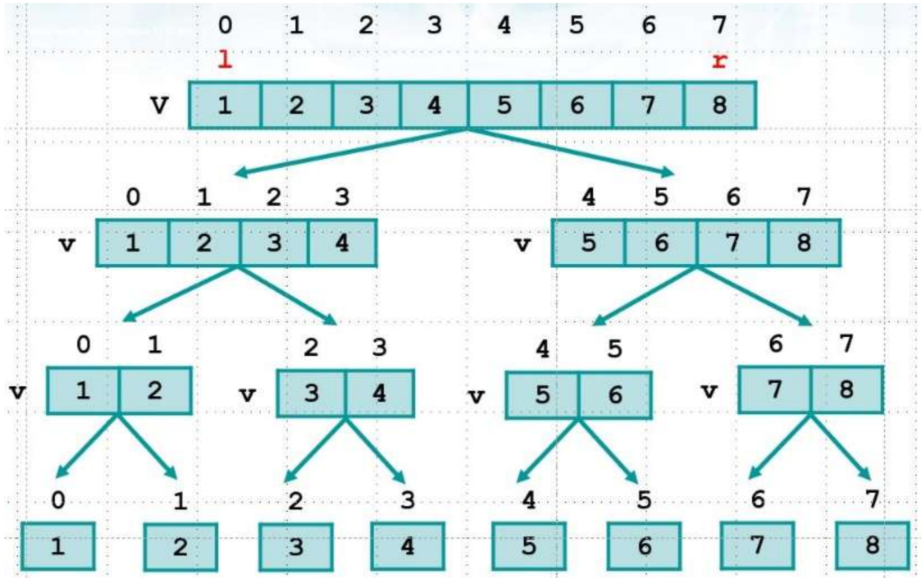
recursion/ combination

c+1 so element c is considered only once

combination

Array print (from element l to r)

Recursion tree (**visited depth-first**)



Solution 2

```
void show (
  int v[], int l, int r
) {
  int i, c;

  if (l >= r) {
    return;
  }

  printf (“v = ");
  for (i=l; i<=r; i++)
    printf ("%d ", v[i]);
  printf ("\n");

  c = (r+l)/2;
  show (v, l, c);
  show (v, c+1, r);

  return;
}
```

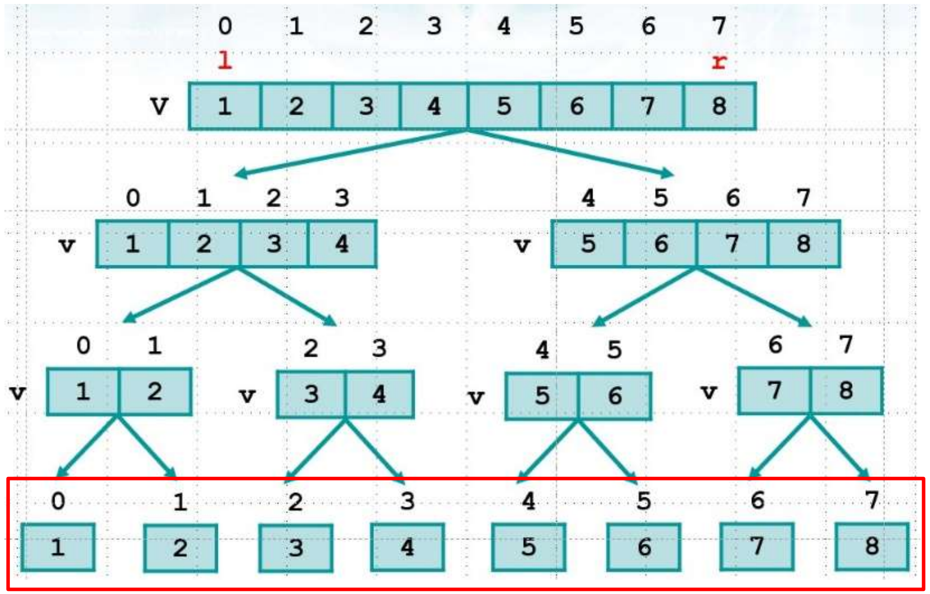
Array print (from element l to r)

left recursion

right recursion

If I check before I print, the last layer is not printed

Recursion tree (**visited depth-first**)



Not printed

Solution 2

```
void show (
  int v[], int l, int r
) {
  int i, c;

  if (l >= r) {
    return;
  }

  c = (r+l)/2;
  printf ("v = ");
  for (i=l; i<=c; i++)
    printf ...
  show (v, l, c);
  printf ("v = ");
  for (i=c+1; i<=r; i++)
    printf ...
  show (v, c+1, r);
  return;
}
```

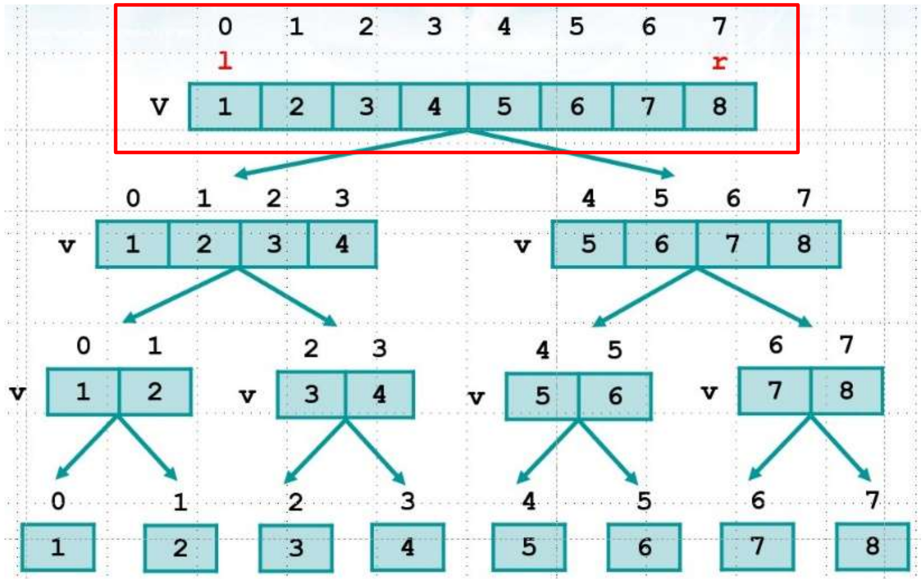
NOTE: A LIFO stack is used for storing the state of the function calls

If termination condition is not presented, stack overflow is triggered

If I recur before printing I miss the first layer

Recursion tree (**visited depth-first**)

Not printed



Example 1: Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = 2$
Reduction factor	$b = {}^{n}/_{\hat{n}} = 2$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$
\begin{cases}
T(n) = D(n) + a \cdot T\left({}^{n}/_{b}\right) + C(n) & n > 1 \\
T(n) = \Theta(1) & n \leq 1
\end{cases}
$$

```c
void show (
  int v[], int l, int r
) {
  int i, c;
  if (l >= r) {
    return;
  }
  c = (r+l)/2;
  show (v, l, c);
  show (v, c+1, r);
  return;
}
```

Example 1: Complexity Analysis

$$T(n) = D(n) + a \cdot T\left(\frac{n}{b}\right) + C(n) \quad n > 1$$
$$T(n) = \Theta(1) \quad n \leq 1$$

No cost for the combination phase

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 1 \quad n > 1$$
$$T(1) = 1 \quad n \leq 1$$

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

$$T\left(\frac{n}{2}\right) = 1 + 2 \cdot T\left(\frac{n}{4}\right)$$

$$T\left(\frac{n}{4}\right) = 1 + 2 \cdot T\left(\frac{n}{8}\right)$$

$$T\left(\frac{n}{8}\right) = 1 + 2 \cdot T\left(\frac{n}{16}\right)$$

$$\ldots$$

$$T(1) = 1$$

At the i-th step

Termination condition

$$\frac{n}{2^i} = 1$$
$$n = 2^i$$
$$i = log_2 n$$

Example 1: Complexity Analysis

$$T(n) = 1 + 2 \cdot T\left({}^{n}/_{2}\right)$$

$$T\left({}^{n}/_{2}\right) = 1 + 2 \cdot T\left({}^{n}/_{4}\right)$$

$$T\left({}^{n}/_{4}\right) = 1 + 2 \cdot T\left({}^{n}/_{8}\right)$$

$$T\left({}^{n}/_{8}\right) = 1 + 2 \cdot T\left({}^{n}/_{16}\right)$$

...

$$T(1) = 1$$

$i = log_2 n$
steps

$$T(n) = 1 + 2 \cdot T\left({}^{n}/_{2}\right)$$

$$T(n) = 1 + 2 + 4 \cdot T\left({}^{n}/_{4}\right)$$

$$T(n) = 1 + 2 + 4 + 8 \cdot T\left({}^{n}/_{8}\right)$$

$$T(n) = 1 + 2 + 4 + 8 + 16 \cdot T\left({}^{n}/_{16}\right)$$

...

$$T(n) = \sum_{i=0}^{\log n} 2^i = \frac{(2^{\log n +} \quad - 1)}{2 - 1} =$$

$$= 2 \cdot 2^{\log n} - 1 =$$

$$= 2n - 1 =$$

$$= O(n)$$

log2(n)

$$\sum_{i=0}^{k} x^i = \frac{(x^{k+1} - 1)}{(x - 1)}$$

sum of a geometric series

A second example: Maximum of an array

- Given an array of $n = 2^k$ integers
- Find its maximum and print it on standard output

	0	1	2	3	4	5	6	7
	l							r
v	11	1	5	6	9	36	13	4

36

Solution

- If the array size n is equal to 1 ($n = 1$)
  - Find maximum explicitly
- If the array size n is larger than 1 ($n > 1$)
  - Divide array in 2 subarrays, each being half the original array
  - Recursively search for maximum in the **left** subarray and **return** the maximum value in it
  - Recursively search for maximum in the **right** subarray and **return** the maximum value in it
  - **Compare** maximum values returned and return bigger one

Termination condition

Solution

```
result = max (v, 0, 3);
```

	0	1	2	3
v	10	3	40	6

Initial call
$l = 0, r = 3, n = 2^k$

Implementation

```
int max(int v[],int l,int r){
  int c, m1, m2;
  if (l >= r)
    return v[l];
  c = (l + r)/2;
  m1 = max (v, l, c);
  m2 = max (v, c+1, r);
  if (m1 > m2)
    return m1;
  else
    return m2;
}
```

it is enough to put l==r but like this it is safer

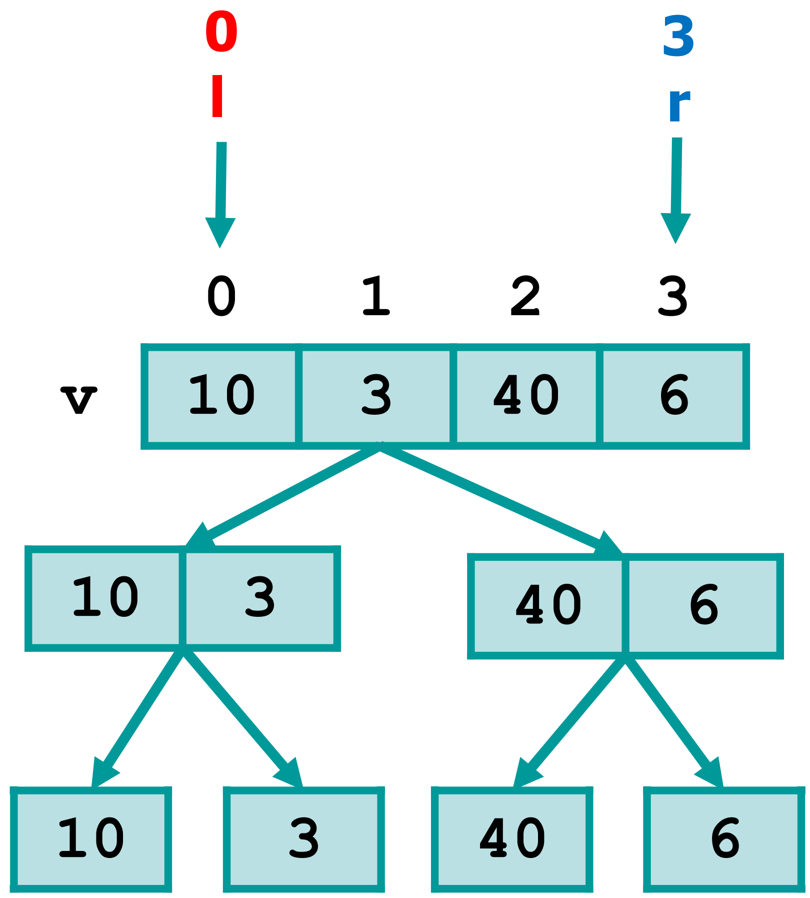
Solution

```
result = max (v, 0, 3);
```

Implementation

```
int max(int v[],int l,int r){
  int c, m1, m2;
  if (l >= r)
    return v[l];
  c = (l + r)/2;
  m1 = max (v, l, c);
  m2 = max (v, c+1, r);
  if (m1 > m2)
    return m1;
  else
    return m2;
}
```

Recursion tree (**visited depth-first**)



Example ~~1~~ 2: Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = 2$
Reduction factor	$b = {}^{n}/_{\hat{n}} = 2$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + a \cdot T\left({}^{n}/_{b}\right) + C(n) \qquad n > 1$$

$$T(n) = \Theta(1) \qquad n \le 1$$

```
int max(int v[],int l,int r){
  int c, m1, m2;
  if (l >= r)
    return v[l];
  c = (l + r)/2;
  m1 = max (v, l, c);
  m2 = max (v, c+1, r);
  if (m1 > m2)
    return m1;
  else
    return m2;
}
```

As for example 1 ...

$$T(n) = O(n)$$

Factorial

- The factorial of a number can be defined using an
  - Iterative definition

$$n! = \prod_{i=0}^{n-1}(n-i) = n \cdot (n-1) \cdot \ \ldots \cdot 2 \cdot 1$$

  - Recursive definition

termination condition

$$n! = n \cdot (n-1)! \qquad n \geq 1$$
$$0! = 1$$

  - Examples

$$3! = 6$$
$$5! = 120$$

An example

$5! = 5 \cdot \boxed{4!} = \mathbf{120}$

$4! = 4 \cdot \boxed{3!} = \mathbf{24}$

$3! = 3 \cdot \boxed{2!} = \mathbf{6}$

$2! = 2 \cdot \boxed{1!} = \mathbf{2}$

$1! = 1 \cdot \boxed{0!} = \mathbf{1}$

$0! = \mathbf{1}$

Recursion tree (**complete**)

$$n! = n \cdot (n-1) \qquad n \geq 1$$
$$0! = 1$$

Solution

Complete program (main and function)

```
#include <stdio.h>

                  long
long int fact(int n);

main() {
  long int n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d ! = %d\n",
    n, fact(n));
}

   long, to solve the problem of overflowing
long int fact (long int n)
{
  if (n == 0)
    return (1);
  return (n * fact(n-1));
}
```

Recursion

Alternative implementation

```
long int fact (long int n)
{
  long int f;
  if (n == 0)
    return (1);
  f = fact (n-1);
  return (n * f);
}
```

same but not recurring inside the return statement

Recursion

Example 1: Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = 1$
Reduction value	$k_i = 1$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-k_i) + C(n) \qquad n > 1$$

$$T(n) = \Theta(1) \qquad n \le 1$$

```
long int fact (long int n)
{
  long int f;
  if (n == 0)
    return (1);
  f = fact (n-1);
  return (n * f);
}
```

Example 1: Complexity Analysis

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-k_i) + C(n) \quad n > 1$$
$$T(n) = \Theta(1) \quad n \le 1$$

No cost for the combination phase

$$T(n) = 1 + T(n-1) \quad n > 1$$
$$T(1) = 1 \quad n \le 1$$

$$T(n) = 1 + T(n-1)$$

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

$$T(n-2) = 1 + T(n-3)$$

$$T(n-3) = 1 + T(n-4)$$

...

$$T(1) = 1$$

Termination condition
$$n - 1 = 1$$
$$i = n - 1$$

Example 1: Complexity Analysis

$$T(n) = 1 + T(n-1)$$

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

$$T(n-2) = 1 + T(n-3)$$

$$T(n-3) = 1 + T(n-4)$$

…

$$T(1) = 1$$

$i = n - 1$
steps

$$T(n) = 1 + T(n-1)$$

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

$$T(n) = 1 + 1 + 1 + T(n-3)$$

$$T(n) = 1 + 1 + 1 + 1 + T(n-4)$$

…

$$T(n) = 1 + 1 + 1 + 1 \ldots =$$

$$= \sum_{i=0}^{n-1} i =$$

$$= n = O(n)$$

Fibonacci Numbers

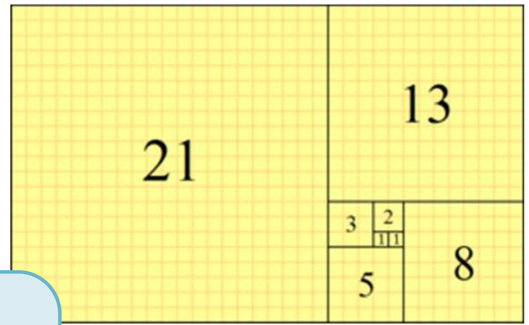
Leonardo Pisano
known as Fibonacci
(Pisa, 1170-1242)

- Fibonacci numbers
  - Iterative and recursive definition

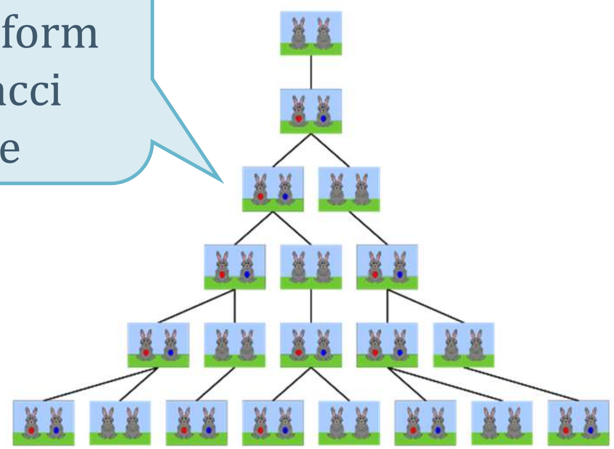
$$F(n) = F(n-2) + F(n-1) \qquad n > 1$$
$$F(0) = 0$$
$$F(1) = 1$$

  - Example

$$F(0) = 0$$
$$F(1) = 1$$
$$F(2) = 0 + 1 = 1$$
$$F(3) = 1 + 1 = 2$$
$$F(4) = 1 + 2 = 3$$
$$\ldots$$
$$0 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13 \;\; 21 \;\; 34 \;\; \ldots$$

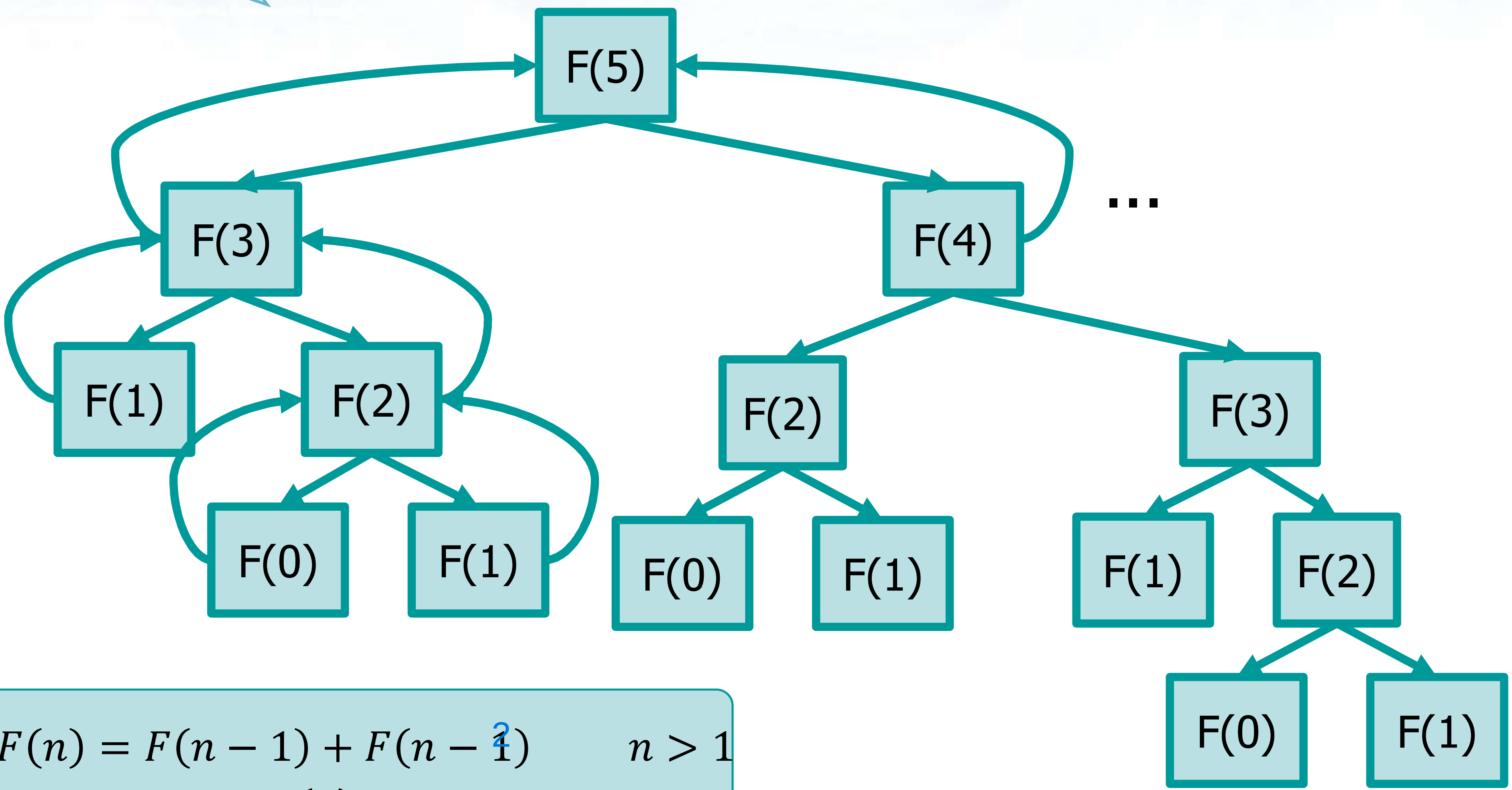


The number of rabbit pairs form the Fibonacci sequence



An Example: Computing F(5)

Recursion tree (**complete**)



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

Solution

```
#include <stdio.h>

long int fib(long int n);

main() {
  long int n;

  printf("Input n:  ");
  scanf("%d", &n);
  printf("Fibonacci of %d is: %d \n", n, fib(n));
}

long int fib (long int n) {
  if (n == 0 || n == 1)
    return (n);
  return (fib(n-2) + fib(n-1));
}
```

Solution

```
long int fib (long int n) {
  if (n == 0 || n == 1)
    return (n);
  return (fib(n-2) + fib(n-1));
}
```

Alternative implementation

```
long int fib (long int n) {
  long int f1, f2;

  if (n == 0 || n == 1)
    return (n);
  f1 = fib (n-2);
  f2 = fib (n-1)
  return (f1 + f2);
}
```

Example 1: Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = 2$
Reduction factor	$k_i = 1$ $k_{i-1} = 2$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n - k_i) + C(n) \qquad n > 1$$

$$T(n) = \Theta(1) \qquad n \leq 1$$

```
long int fib (long int n) {
  long int f1, f2;

  if (n == 0 || n == 1)
    return (n);
  f1 = fib (n-2);
  f2 = fib (n-1)
  return (f1 + f2);
}
```

Example 1: Complexity Analysis

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

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-k_i) + C(n) \quad n > 1$$
$$T(n) = \Theta(1) \quad n \leq 1$$

No cost for the combination phase

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

We can make a conservative approximation

$$T(n) = 1 + 2 \cdot T(n-1)$$
$$T(0) = 1$$
$$T(1) = 1$$

We replace $T(n-2)$ with $T(n-1)$

Example 1: Complexity Analysis

$$T(n) = 1 + 2 \cdot T(n-1)$$
$$T(n-1) = 1 + 2 \cdot T(n-2)$$
$$T(n-2) = 1 + 2 \cdot T(n-3)$$
$$\dots$$
$$T(0) = 1$$
$$T(1) = 1$$

Termination condition

$$\frac{n}{2^i} = 1$$
$$\mathrm{n} = 2^i$$
$$i = log_2 n$$

steps

$$T(n) = 1 + 2 \cdot T(n-1)$$
$$T(n) = 1 + 2 + 4 \cdot T(n-2)$$
$$T(n) = 1 + 2 + 4 + 8 \cdot T(n-3)$$
$$\dots$$
$$T(n) = \sum_{i=0}^{n-1} 2^i =$$
$$= 2^n - 1 =$$
$$= O(2^n)$$

$$\sum_{i=0}^{k} x^i = \frac{(x^{k+1}-1)}{(x-1)}$$

Not linear.
Why?

because we are doing the same job over and over again

Binary Search

- Binary search
  - Does key k belong to the sorted array v[n]?
  - Yes/No

Assumption
$n = 2^k$

- Approach
  - Start with (sub-)array of extremes l and r
  - At each step
    - Find middle element c=(int)((l+r)/2)
    - Compare k with middle element in the array
      - =: termination with success
      - <: search continues on left subarray
      - >: search continues on right subarray

Example

k = 8

	0	1	2	3	4	5	6	7	8	9
	l									r
v	2	4	6	8	10	12	14	16	18	20

	0	1	2	3						
	l			r						
v	2	4	6	8						

			2	3						
			l	r						
v			6	8						

				3						
				lr						
v				8						

k = key to search
l = leftmost index
r = rightmost index
c = index of the middle element

Solution

```
int bin_search (int v[], int l, int r, int k){
  int c;

  if (l > r)
    return(-1);

  c = (l+r) / 2;

  if (k < v[c])
    return(bin_search (v, l, c-1, k));
  if (k > v[c])
    return(bin_search (v, c+1, r, k));

  return c;
}
```

Termination condition

Skip ther element already checked

Example 1: Complexity Analysis

Decrease and conquer problem	
Number of subproblems	$a = 1$
Reduction factor	$b = {}^{n}/_{\hat{n}} = 2$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + a \cdot T\left({}^{n}/_{b}\right) + C(n) \qquad n > 1$$

$$T(n) = \Theta(1) \qquad n \leq 1$$

```
int bin_search (...){
  int c;
  if (l > r)
    return(-1);
  c = (l+r) / 2;
  if (k < v[c])
    return(bin_search (...));
  if (k > v[c])
    return(bin_search (...));
  return c;
}
```

Example 1: Complexity Analysis

$$T(n) = D(n) + a \cdot T\left(n/b\right) + C(n) \quad n > 1$$
$$T(n) = \Theta(1) \quad n \leq 1$$

No cost for the combination phase

$$T(n) = 1 + T\left(n/2\right) \quad n > 1$$
$$T(1) = 1 \quad n \leq 1$$

$$T(n) = 1 + T\left(n/2\right)$$

$$T\left(n/2\right) = 1 + T\left(n/4\right)$$

$$T\left(n/4\right) = 1 + T\left(n/8\right)$$

$$T\left(n/8\right) = 1 + T\left(n/16\right)$$

...

$$T(1) = 1$$

Termination condition

$$\frac{n}{2^i} = 1$$
$$\mathrm{n} = 2^i$$
$$i = log_2 n$$

Example 1: Complexity Analysis

$$T(n) = 1 + T({}^{n}/_{2})$$

$$T({}^{n}/_{2}) = 1 + T({}^{n}/_{4})$$

$$T({}^{n}/_{4}) = 1 + T({}^{n}/_{8})$$

$$T({}^{n}/_{8}) = 1 + T({}^{n}/_{16})$$

...

$$T(1) = 1$$

$i = log_2 n$ steps

$$T(n) = 1 + T({}^{n}/_{2})$$

$$T(n) = 1 + 1 + T({}^{n}/_{4})$$

$$T(n) = 1 + 1 + 1 + T({}^{n}/_{8})$$

$$T(n) = 1 + 1 + 1 + 1 + T({}^{n}/_{16})$$

...

$$T(n) = \sum_{i=0}^{\log} 1 =$$

$$= O(log_2(n)) = 1 + log_2(n)$$

$\sum_{i=0}^{k} x^i = \frac{(x^{k+1}-1)}{(x-1)}$

Reverse printing

- Read a string from standard input
- Print it in reverse order
  - Start printing from last character and move back to first one

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
s	T	h	i	s	I	s	A	S	t	r	i	n	g	\0

gnirtSAsIsihT

Solution

```
int main() {
  char str[max+1];
  printf ("Input string: ");
  scanf ("%s", str);
  printf ("Reverse string is: ");
  reverse_print (str);
}

void reverse_print (char *s) {
  if (*s == '\0') {
    return;
  }
  reverse_print (s+1);
  printf (“%c”, *s);
  return;
}
```

Example 1: Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = 1$
Reduction value	$k_i = 1$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n - k_i) + C(n) \quad n > 1$$

$$T(n) = \Theta(1) \quad n \leq 1$$

As for the factorial ...

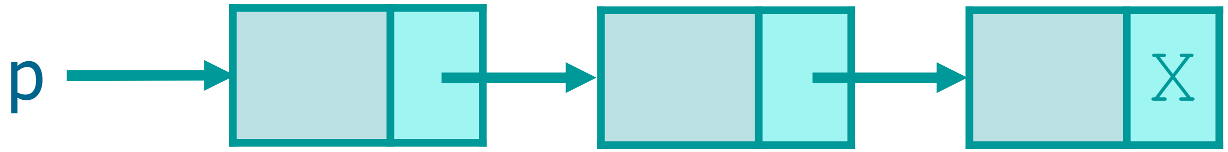
$$T(n) = O(n)$$

```
void reverse_print (char *s) {
  if (*s == '\0') {
    return;
  }
  reverse_print (s+1);
  printf ("%c", *s);
  return;
}
```

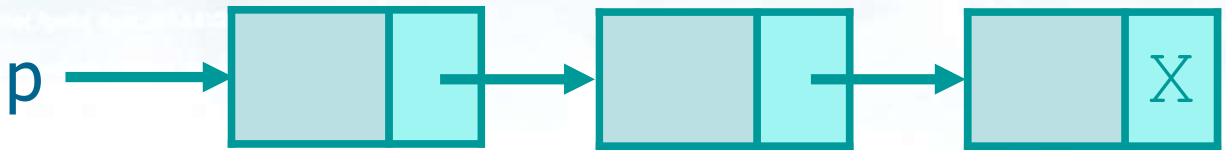
List processing

- Recursive list processing
  - Count the number of elements in a list
  - Traverse a list in order
  - Traverse a list in reverse order
  - Delete an element (of a given item) from the list

```
typedef struct list_s list_t;
struct node {
  int key;
  ...
  list_t *next;
};
```



Solution



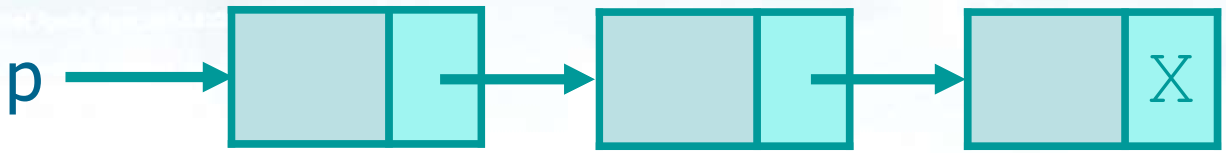

```
int count (list_t *p) {
  if (p == NULL)
    return 0;
  return (1 + count(p->next));
}
```

Count the number of elements

```
void traverse (list_t *p) {
  if (p == NULL)
    return;
  printf ("%d", p->key);
  traverse (p->next);
}
```

Traverse the list forward

Solution




```
void traverse_reverse (list_t *p) {
  if (p == NULL)
    return;
  traverse_reverse (p->next);
  printf ("%d", p->key);
}
```

Traverse the list backward

very interesting!!!

```
link delete(list_t *p, int val) {
  if (p==NULL) return NULL;
  if (p->key==val) {
    list_t *t=p->next; free(p);
    return t;
  }
  p->next = delete (p->next, val);
  return p;
}
```

Dispose an element of the list

Create (re-create) link on the way back

Complexity Analysis

for all those functions

Divide and conquer problem	
Number of subproblems	$a = 1$
Reduction value	$k_i = 1$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-k_i) + C(n) \quad n > 1$$

$$T(n) = \Theta(1) \quad n \leq 1$$

As for the factorial ...

$$T(n) = O(n)$$

```
int count (link x) {
  if (x == NULL)
    return 0;
  return (1 + count(x->next));
}
```

Greatest Common Divisor

- Given 2 non-zero integers x and y, the greatest common divisor $\gcd(x, y)$ is the greatest among the common divisors of x and y
  - Inefficient algorithm are based on decomposition in prime factors of x and y

$$x = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3} \cdot \ldots \cdot p_r^{e_r}$$

$$y = p_1^{f_1} \cdot p_2^{f_2} \cdot p_3^{f_3} \cdot \ldots \cdot p_r^{f_r}$$

Common factors with the minimum exponent

$$\gcd(x, y) = p_1^{\min(e_1, f_1)} \cdot p_2^{\min(e_2, f_2)} \cdot p_3^{\min(e_3, f_3)} \cdot \ldots \cdot p_r^{\min(e_r, f_r)}$$

  - More efficient methods are base on Euclid's algorithm

Euclid's Algorithm

- Version number 1
  - Based on subtraction
- Version number 2
  - Based on the remainder of integer divisions

```
if (x > y)
  gcd(x, y) = gcd(x-y, y)
else
  gcd(x, y) = gcd(x, y-x)
```

```
if (y > x)
  swap (x, y)

gcd (x, y) = gcd(y, x%y)
```

Termination condition

```
if (x == y)
  return x
```

```
if (y == 0)
  return x
```

Examples: Version 1

gcd (20, 8) =

= gcd (20-8, 8) = gcd (12, 8)

= gcd (12-8, 8) = gcd (4, 8)

= gcd (4, 8-4) = gcd (4, 4)

= 4 → return 4

gcd (600, 54) =

= gcd (600-54, 54) = gcd (546, 54)

= gcd (546-54, 54) = gcd (492, 54) …

= gcd (6,54) = gcd (6, 54-6) …

= gcd (6, 12) = gcd (6,6)

= 6→ return 6

very slow convergence

```
if (x > y)
  gcd(x, y) = gcd(x-y, y)
else
  gcd(x, y) = gcd(x, y-x)
```

Examples: Version 2

gcd (20, 8) =

= gcd (8, 20%8) = gcd (8, 4)

= gcd (4, 8%4) = gcd (4, 0)

= 4 → return 4

much faster convergence

gcd (600, 54) =

= gcd (54, 600%54) = gcd (54, 6)

= gcd (6, 54%6) = gcd (6, 0)

= 6 → return 6

```
if (y > x)
  swap (x, y)
gcd (x, y) = gcd(y, x%y)
```

Examples: Version 2

gcd (314159, 271828)=

= gcd (271828, 314159%271828) = gcd (271828,42331)

= gcd (42331, 271828%42331)= gcd(42331,17842)

= gcd (17842, 42331%17842) = gcd (17842, 6647)

= gcd (6647, 17842%6647) = gcd (6647, 4548)

= gcd (4548, 6647%4548) = gcd (4548, 2099)

= gcd (2099, 4548%2099) = gcd (2099, 350)

= gcd (350, 2099%350) = gcd (350, 349)

= gcd (349, 350%349), gcd (349, 1)

= gcd (1,349%1) = gcd (1, 0)

= 1 → return 1

In fact, 314159 and 271828 are mutually prime

```
if (y > x)
  swap (x, y)
gcd (x, y) = gcd(y, x%y)
```

Solution: Version 1

```
#include <stdio.h>

int gcd (int x, int y);

main() {
  int x, y;
  printf("Input x and y:  ");
  scanf("%d%d", &x, &y);
  printf("gcd of %d and %d: %d \n", x, y, gcd(x, y));
}

int gcd (int x, int y) {
  if (x == y)
    return (x);
  if (x > y)
    return gcd (x-y, y);
  else
    return gcd (x, y-x);
}
```

Solution: Version 2

```
#include <stdio.h>

int gcd (int m, int n);

main() {
  int m, n, r;
  printf("Input m and n:  ");
  scanf("%d%d", &m, &n);
  if (m>n)
    r = gcd(m, n);
  else
    r = gcd(n, m);
  printf("gcd of (%d, %d) = %d\n", m, n, r);
}

int gcd (int m, int n) {
  if(n == 0)
    return(m);
  return gcd(n, m % n);
}
```

Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = 1$
Reduction value	$k_i = variable$
Division cost	$D(x, y) = \Theta(1)$
Recombination cost	$C(x, y) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n - k_i) + C(n) \quad n > 1$$

$$T(n) = \Theta(1) \quad n \leq 1$$

Proof beyond the scope of this course...

$$T(n) = O(\log(y))$$

```
int gcd (int m, int n) {
  if(n == 0)
    return(m);
  return gcd(n, m % n);
}
```

Determinant

- Laplace's algorithm with unfolding on row i
  - Square matrix M $(n \; x \; n)$ with indices from 1 to n
- Computation

$$det(M) = \sum_{j=1}^{n} (-1)^{(i+j)} \cdot M[i][j] \cdot det(M_{minor\; i,j})$$

  - Where $M_{minor\; i,j}$ is obtained from M ruling-out row i and column j

Example

$$M = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix}$$

$$M_{minor\ 1,1} = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix}$$

$$M_{minor\ 1,2} = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 3 \\ 2 & -1 \end{bmatrix}$$

$$M_{minor\ 1,3} = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 2 & 0 \end{bmatrix}$$

$$det \begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix} = -1 - 0 = -1$$

$$det \begin{bmatrix} -1 & 3 \\ 2 & -1 \end{bmatrix} = 1 - 6 = -5$$

$$det \begin{bmatrix} -1 & 1 \\ 2 & 0 \end{bmatrix} = 0 - 2 = -2$$

$$\begin{aligned} \det(M) = {} & (-1)^{(1+1)} \cdot (-2) \cdot \det(M_{minor\ 1,1}) + (-1)^{(1+2)} \cdot (\ 2) \cdot \det(M_{minor\ 1,2}) \\ & + (-1)^{(1+3)} \cdot (-3) \cdot \det(M_{minor\ 1,3}) = \\ & = (1) \cdot (-2) \cdot (-1) + (-1) \cdot (2) \cdot (-5) + (1) \cdot (-3) \cdot (-2) = 18 \end{aligned}$$

Solution

- Recursive algorithm
  - If M has size n, indice ranges between 0 and n-1
- If n = 2
  - Compute the trivial solution

$$det(M) = M[0][1] \;\; M[1][1] - M[0][1] \;\; M[1][0]$$

- If n>2
  - With row=0 and column ranging from 0 and n-1
  - Store in tmp the minor $M_{minor\ 0,j}$
  - Recursively compute $\det(M_{minor\ 0,j})$
  - Store result results in

$$sum = sum + M[0][k] \cdot pow(-1, k) \cdot det(tmp, n-1)$$

Solution

```
int det (int m[][MAX], int n) {
  int sum, c;
  int tmp[MAX][MAX];
  sum = 0;

  if (n == 2)
    return (det2x2(m));

  for (c=0; c<n; c++) {
    minor (m, 0, c, n, tmp);
    sum = sum + m[0][c] * pow(-1,c) * det (tmp,n-1);
  }

  return (sum);
}
```

Create minor

Recur on minor computation

Solution

```
int det2x2(int m[][MAX]) {
  return(m[0][0]*m[1][1] - m[0][1]*m[1][0]);
}

void minor(
  int m[][MAX],int i,int j,int n,int m2[][MAX]
){
  int r, c, rr, cc;

  for (rr = 0, r = 0; r < n; r++)
    if (r != i) {
      for (cc = 0, c = 0; c < n; c++) {
        if (c != j) {
          m2[rr][cc] = m[r][c];
          cc++;
        }
        rr++;
      }
  }
}
```

Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = \mathrm{n}$
Reduction value	$k_i = 2n - 1$
Division cost	$D(x, y) = \Theta(1)$
Recombination cost	$C(x, y) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n - k_i) + C(n) \quad n > 1$$

$$T(n) = \Theta(1) \quad n \leq 1$$

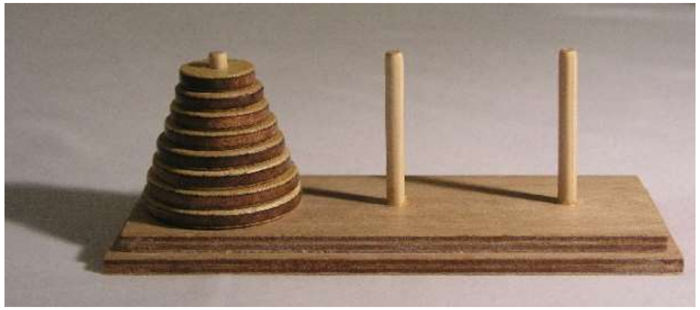
Proof beyond the scope of this course...

$$T(n) = O(n!)$$

```
int det (int m[][MAX], int n) {
  ...
  if (n == 2)
    return (det2x2(m));
  for (c=0; c<n; c++) {
    minor (m, 0, c, n, tmp);
    sum = sum + m[0][c] * ...;
  }
  return (sum);
}
```

Tower of Hanoi

- The Tower of Hanoi (also called The problem of Benares Temple or Tower of Brahma or Lucas' Tower) is a mathematica game consisting of three rods and a number of disks of various diameters

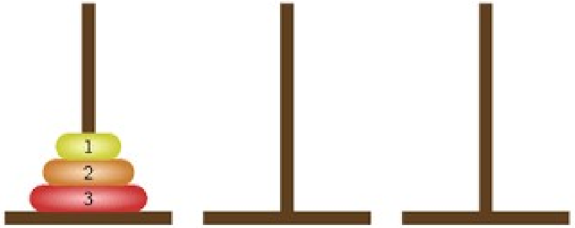


- It was firtlsy introduced the French mathematician Édouard Lucas in 1883

Édouard Lucas
(Paris, 1842-1891)

Tower of Hanoi

- Initial configuration
  - 3 pegs
  - 3 disks
  - Disks of decreasing size on first peg



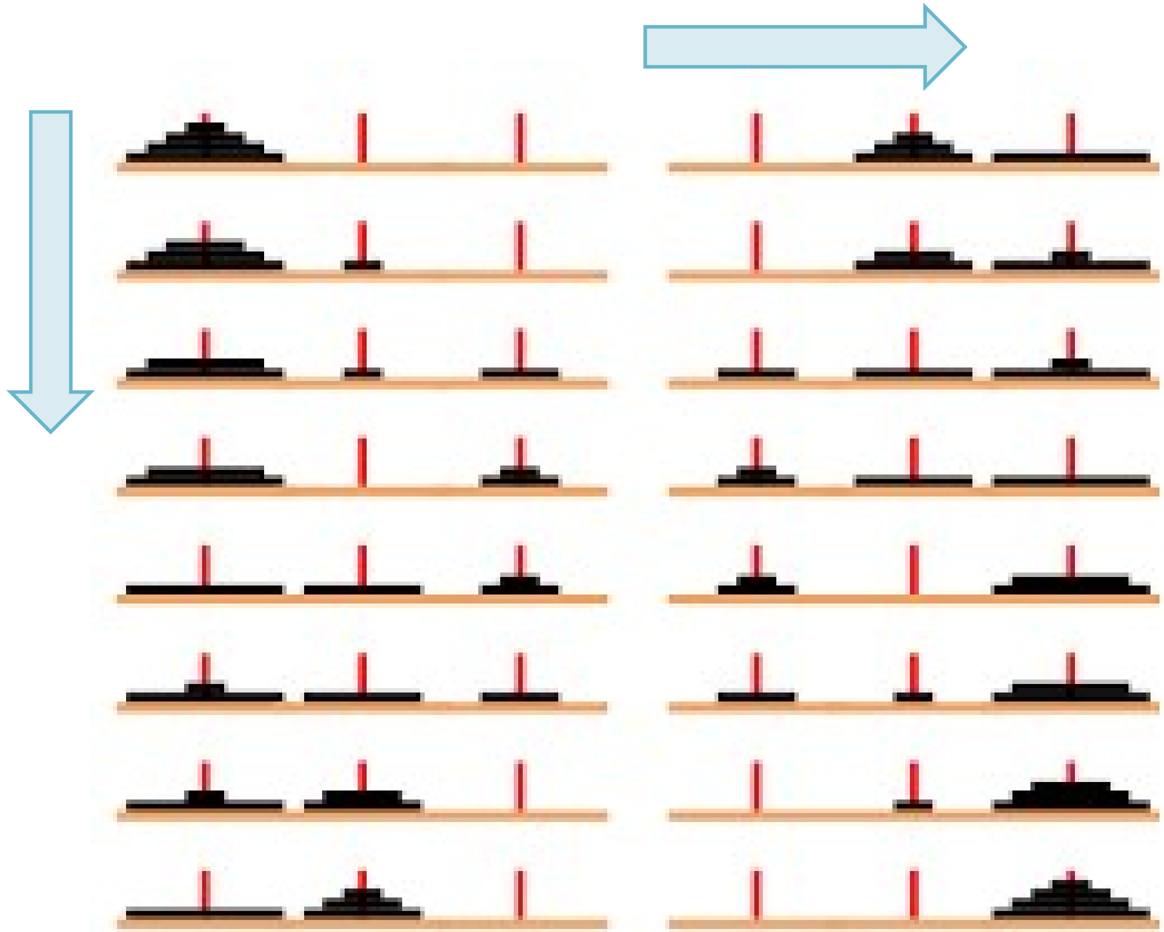
- Final configuration
  - Disks of decreasing size on third peg

- Rules
  - Access only to the top disk
  - On each disk overlap only smaller disks

Tower of Hanoi

- Generalization
  - Work with n disks (and 3 pegs)

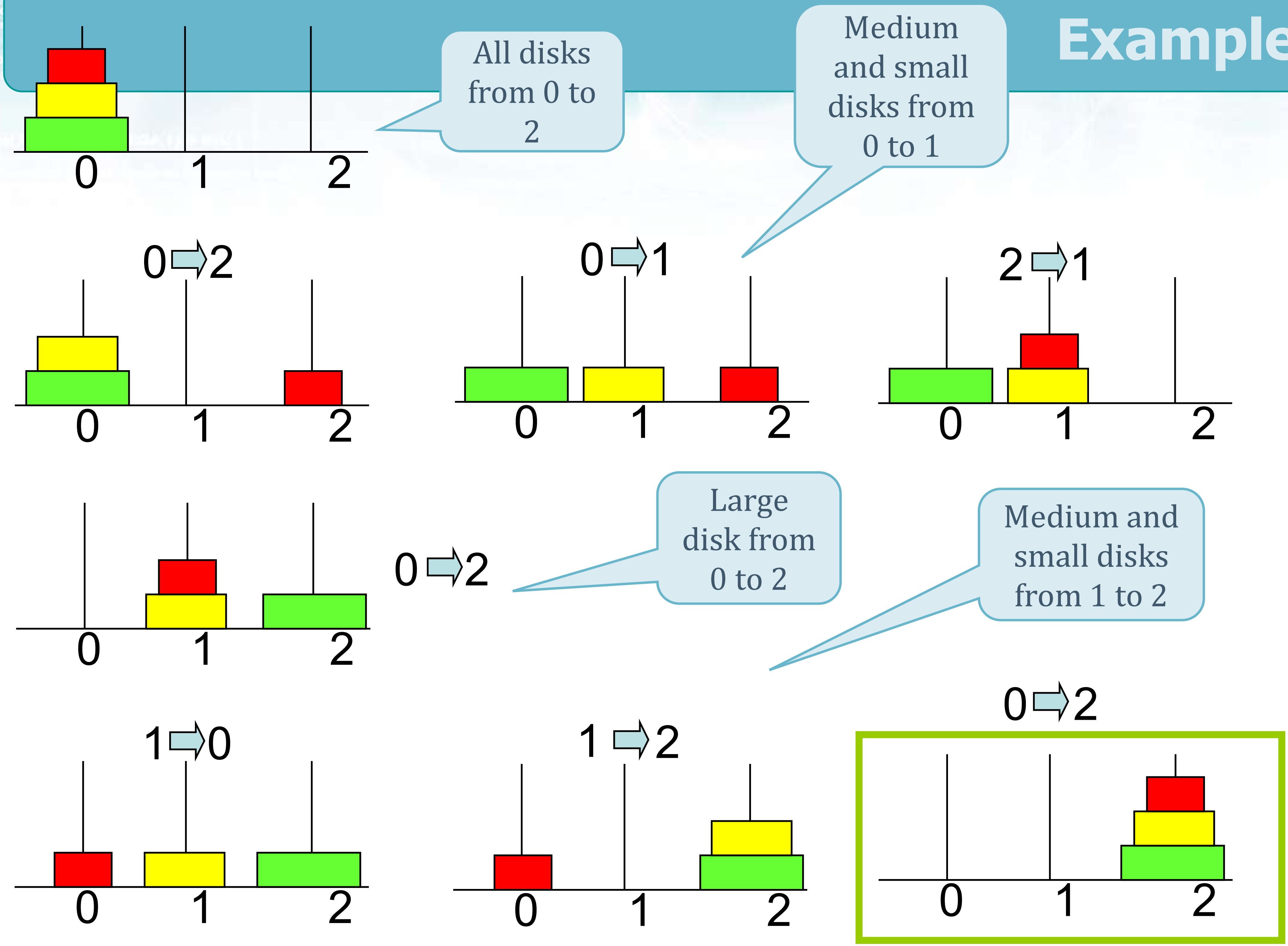
4 disks, 3 pegs
Sequence of correct moves



Solution

- Divide and Conquer strategy
  - Initial problem
    - Move n disks from 0 to 2
  - Reduction to subproblems
    - Move n-1 disks from 0 to 1, 2 temporary storage
    - Move last disk from 0 to 2
    - Move n-1 disks from 1 to 2, 0 temporary storage
  - Termination condition
    - Move just 1 disk

Example


All disks from 0 to 2
Medium and small disks from 0 to 1
0 ⇨ 2
0 ⇨ 1
2 ⇨ 1
0 ⇨ 2
Large disk from 0 to 2
Medium and small disks from 1 to 2
1 ⇨ 0
1 ⇨ 2
0 ⇨ 2
0 1 2

Solution

Move 3 disks from per 0 to peg 2

```
int main (void) {
  hanoi (3, 0, 2);
  return;
}
```

Output

```
Disk 1 from peg 0 to 2
Disk 2 from peg 0 to 1
Disk 1 from peg 2 to 1
Disk 3 from peg 0 to 2
Disk 1 from peg 1 to 0
Disk 2 from peg 1 to 2
Disk 1 from peg 0 to 2
```

```
void hanoi (int n, int src, int dest) {
  int aux;

  if (n <= 0) {
    return;
  }
  aux = 3 - (src + dest);
  hanoi (n-1, src, aux);
  printf("Disk %d from peg %c to %c\n", n,src,dest);
  hanoi (n-1, aux, dest);

  return;
}
```

Termination condition

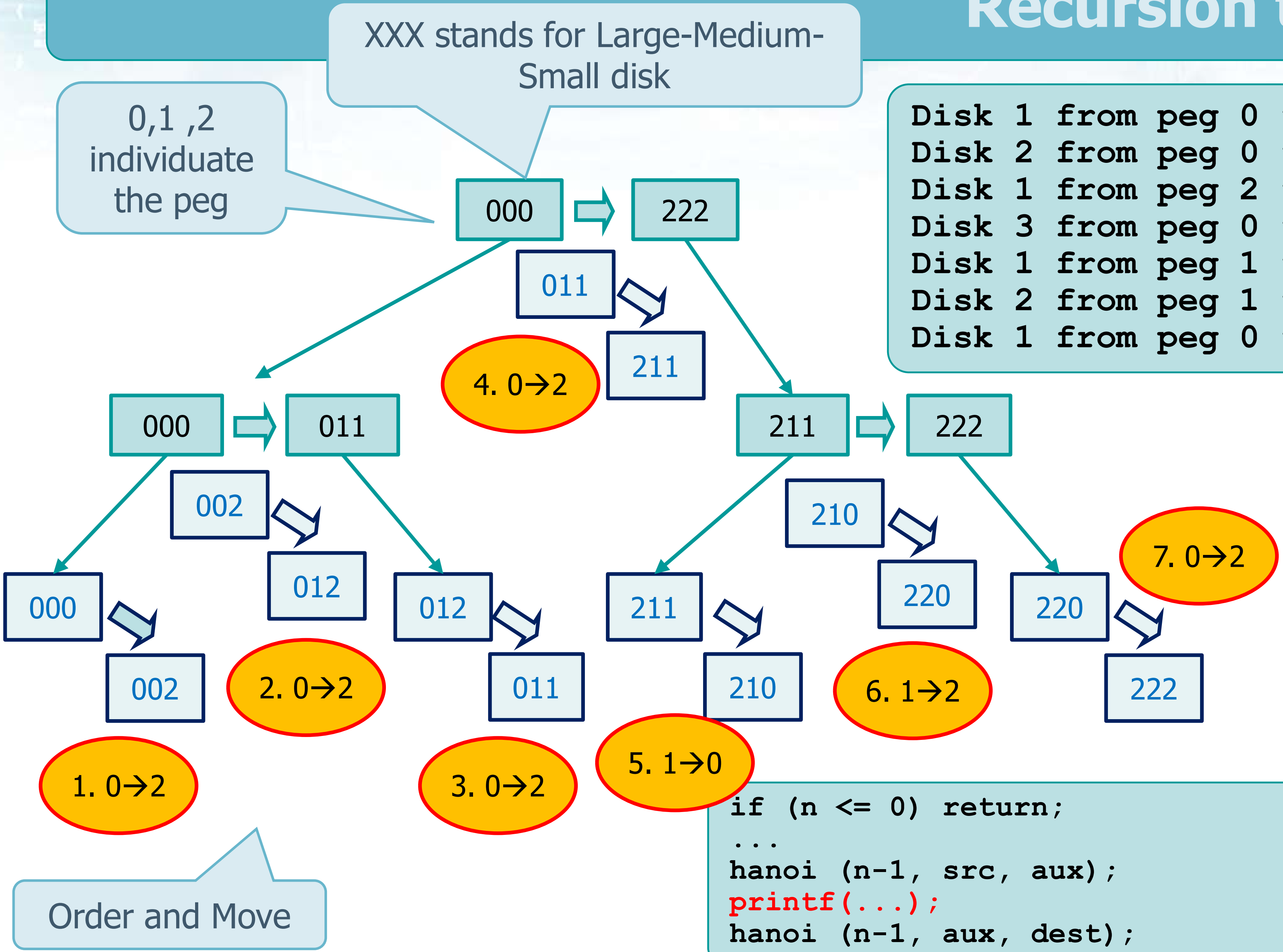
Divide

Move

Divide

Recursion tree

```
Disk 1 from peg 0 to 2
Disk 2 from peg 0 to 1
Disk 1 from peg 2 to 1
Disk 3 from peg 0 to 2
Disk 1 from peg 1 to 0
Disk 2 from peg 1 to 2
Disk 1 from peg 0 to 2
```




```
if (n <= 0) return;
...
hanoi (n-1, src, aux);
printf(...);
hanoi (n-1, aux, dest);
```

Complexity Analysis

Divide and conquer problem	
Number of subproblems	$a = 2$
Reduction factor	$k_i = 1$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-k_i) + C(n) \qquad n > 1$$

$$T(n) = \Theta(1) \qquad n \leq 1$$

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

As for Fibonacci ...

$$T(n) = O(2^n)$$

```
void hanoi(...) {
  int aux;
  aux = 3 - (src + dest);
  if (n == 1) {
    printf(...);
    return;
  }
  hanoi(n-1, src, aux);
  printf(...);
  hanoi(n-1, aux, dest);
  return;
}
```