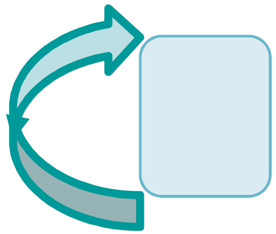
Recursion

The divide and conquer paradigm

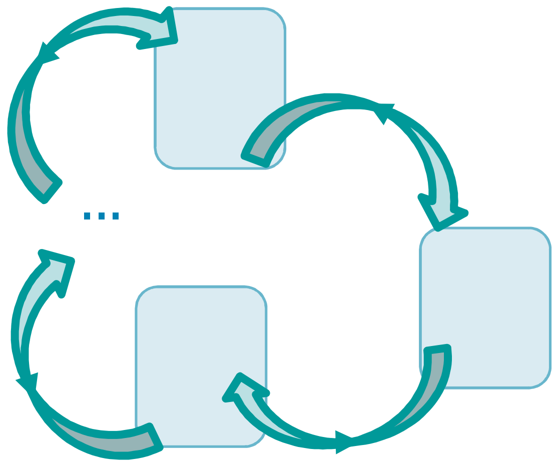
Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Definition

- Recursive procedure
  - **Direct** recursion
    - Inside its definition there is a call to the procedure itself

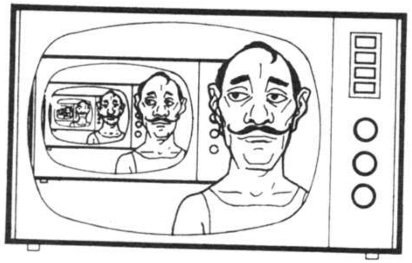


  - **Indirect** recursion
    - Inside its definition there is a call to at least one procedure that, directly or indirectly, calls the procedure itself



Definition

- Recursive algorithm
  - Based on recursive procedures



- 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} simpler than D_n

```
S(Dn)  = f(S(Dn-1))      iff  Dn-1 ≠ D0

S(D0) = S0                   otherwise
```

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

$$S(D_0) = S_0 \quad \text{otherwise}$$

Termination condition

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

- The divide and conquer paradigm is based on 3 phases
  - Divide
    - The recursion should generate simpler and solvable sub-problems, until the sub-problems are
      - Trivial
      - Valid choices exhausted
    - Process
      - Starting from a problem of size **n**
      - We partition it into **a≥1 independent** problems
      - Each of these problems has a smaller size **n’**
        - n’ < n

The divide and conquer paradigm

- Conquer
  - Solve an elementary problem
  - This part is the algorithm 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

The else part is often avoided inserting one more return

Termination condition

Conquer

Divide

Recursive call

```
solve (problem){
  if (problem is elementary) {
    solution = solve_trivial (problem)
  } else {
    subproblem1,2,3,…,a = divide (problem)

    for each s ∈ subproblem1,2,3,…,a
      subsolutions = solve (subproblems)

    solution = combine (subsolution1,2,3,…,a)
  }
  return solution
}
```

Combine

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

The divide and conquer paradigm

- Given
  - The original problem size **n**
  - The number of subproblems **a** of size **n'**

we may define

  - Linear recursion
    - a = 1
  - Multi-way recursion
    - a > 1

The divide and conquer paradigm

- The size of
  - The original problem **n**
  - The generated ones **n'**

may be related by

- A **constant factor** b, in general the same for all subproblems
  - $b = n / n'$ and $n' = n / b$
- A **constant value** k, not always the same for all subproblems
  - $n' = n - k$
- A **variable quantity** β, often difficult to estimate
  - $n' = n - \beta$

The divide and conquer paradigm

- When the reduction is a **constant factor**
  - $b = n / n'$

  the following terminology can be used
  - Divide and conquer
    - $b>1$
  - Decrease and conquer
    - $b=1$
    - With (in general) a constant reduction value k_i
      - $n' = n - k_i$

Complexity Analysis

- A **recursion equation** expresses the time asymptotic cost T(n) in terms of
  - D(n)
    - Cost of dividing the problem
  - T(n')
    - Cost of the execution time for smaller inputs (recursion phase)
  - C(n)
    - Cost of recombining the partial solutions
  - The cost of the teminale cases
    - We often assume unit cost for solving the elementary problems $\Theta(1)$

Complexity Analysis

- When we have a constant factor **b**
  - **a** is the number of subproblems originating from the "divide" phase
  - **b** is the reduction factor, thus **n' = n/b** is the size of each generated subproblem
  - The recurrence equation has the following form
    - $T(n) = \overbrace{D(n)}^{\text{Divide}} + a \cdot \overbrace{T(n/b)}^{\text{Recur } T(n')} + \overbrace{C(n)}^{\text{Combine}}$ $\quad n > \text{const}$
    - $T(n) = \underbrace{\Theta(1)}_{\text{Conquer}}$ $\quad n \le \text{const}$

Complexity Analysis

- When we have a constant value $\mathbf{k_i}$
  - **a** is the number of subproblems originating from the "divide" phase
  - Reduction amounts to $\mathbf{k_i}$, an amount that may vary at each step
  - The recurrence equation has the following form

Divide
Recur T(n')
Combine

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

Conquer

A first example: Array split

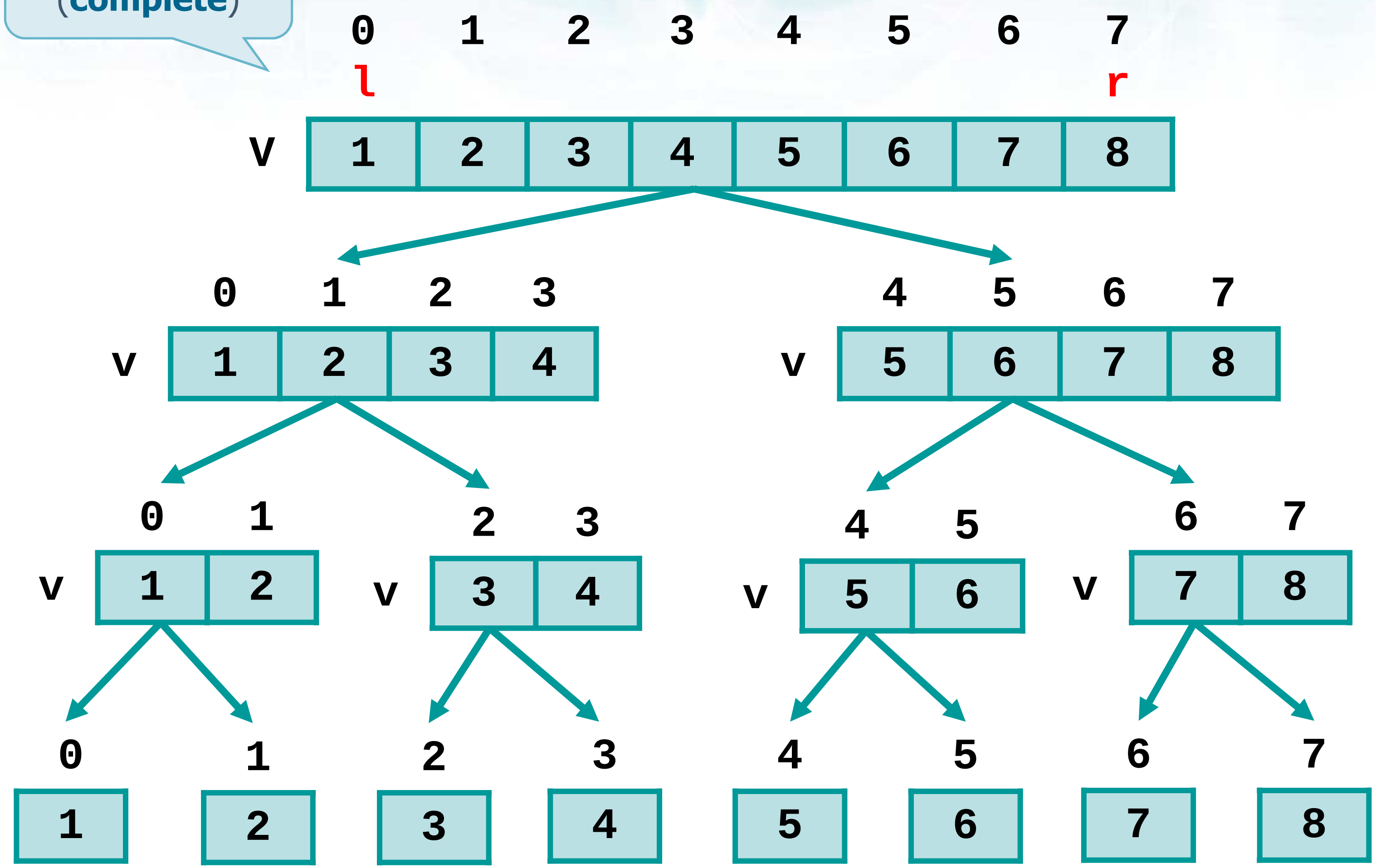
- Specifications
  - Given an array of $n=2^k$ integers
  - Recursively partition it in sub-arrays half the size, until the termination condition is reached
    - The termination conditions is reached when sub-arrays have only 1 cell
  - Print-out all generated partitions on standard output

Simple case
(complete tree of height k)

Divide and conquer
At each step we generate a=2 subproblems
Each subproblem has a size equal to n'=n/2, i.e., b=n/n'=2

Solution

Recursion tree (**complete**)



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;
}
```

Array print
(from element l to r)

Termination
condition

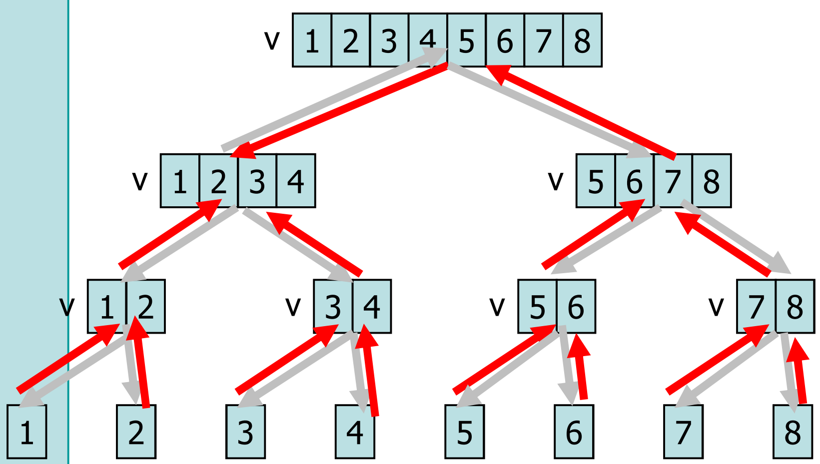
Recursion:
Left recursion
Right recursion

Solution 1

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

  printf (“v = ");
  ...

  if (l >= r) {
    return;
  }

  c = (r+l)/2;

  show (v, l, c);
  show (v, c+1, r);

  return;
}
```

Array print
(from element l to r)

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



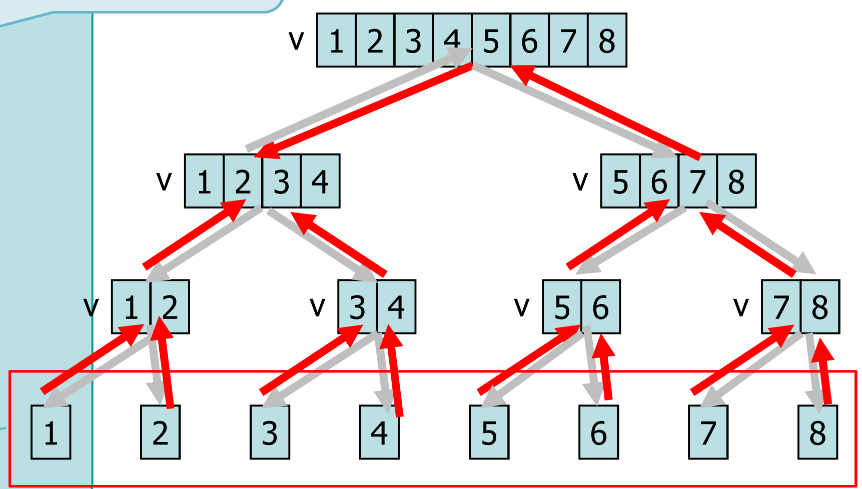
Solution 2

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

  if (l >= r) {
    return;
  }

  printf (“v = ");
  ...

  c = (r+l)/2;

  show (v, l, c);
  show (v, c+1, r);

  return;
}
```

Termination condition

Array print
(from element l to r)

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



Not printed

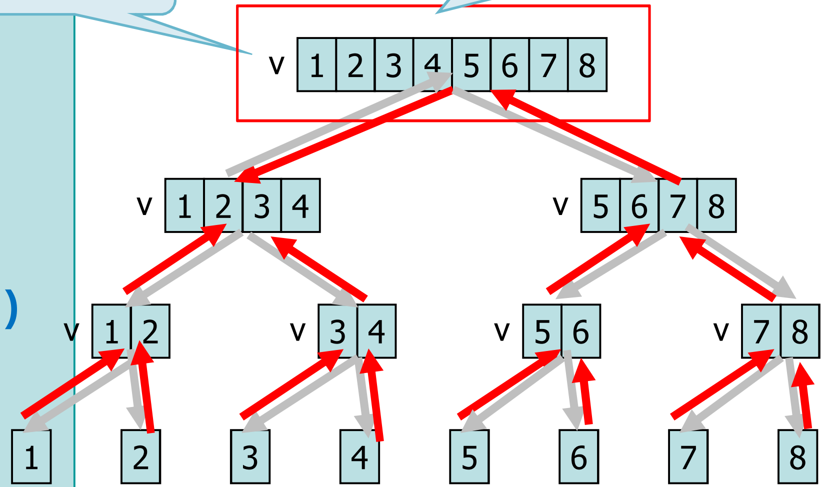
Solution 2

```
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;
}
```

Termination condition

Not printed

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



Example 1: Complexity Analysis

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

```
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;
}
```

Example 1: Complexity Analysis

- Recurrence equation
  - $T(n) = D(n) + a \cdot T(n/b) + C(n)$

Divide, conquer, combine

- That is
  - $T(n) = 2 \cdot T(n/2) + 1 \qquad n > 1$
  - $T(1) = 1 \qquad n = 1$

No cost for the combination phase

```
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

- Resolution by unfolding
  - T(n) = 1 + 2·T(n/2)
  - T(n/2) = 1 + 2·T(n/4)
  - T(n/4) = 1 + 2·T(n/8)
  - …

Termination condition

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

$$i = log_2 n$$

Example 1: Complexity Analysis

- We replace T(n/2) in T(n)
  - T(n) = 1 + 2 + 4 · T(n/4)

then we replace T(n/4) in T(n/2)

  - T(n) = 1 + 2 + 4 + 23 · T(n/8)

etc.

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

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

A second example: Maximum of an array

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

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 (a, 0, 3);
```

Implementation

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

	0	1	2	3
a	10	3	40	6

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

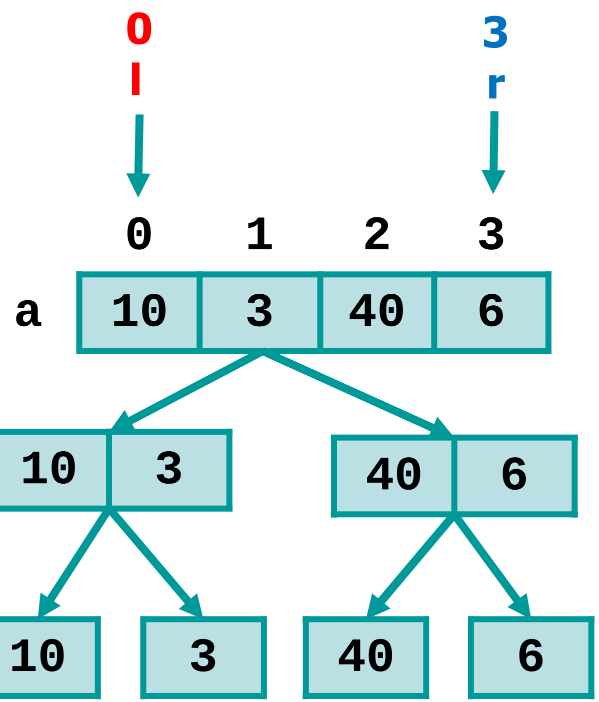
Solution

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

Implementation

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

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



Example 2: Complexity Analysis

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

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

Example 2: Complexity Analysis

- Recurrence equation
  - $T(n) = D(n) + a \cdot T(n/b) + C(n)$

Divide, conquer, combine

- That is
  - $T(n) = 2 \cdot T(n/2) + 1 \qquad n > 1$
  - $T(1) = 1 \qquad n = 1$

As for example 1 ...

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

- Time complexity
  - $T(n) = O(n)$

Factorial

- Factorial
  - Iterative definition
    - $n! = \prod_{i=0}^{n-1}(n-i) = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$
  - Recursive definition
    - $n! = n \cdot (n-1)! \qquad n \geq 1$
    - $0! = 1$
  - Examples
    - $3! = 6$
    - $5! = 120$

An example

Recursion tree (**complete**)

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

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

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

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

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

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

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

$$0! = 1$$

Solution

Complete program (main and function)

```
#include <stdio.h>

long int fact(int n);

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

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);
}
```

Recursion

Complexity Analysis

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

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

Complexity Analysis

- Recurrence equation
  - $T(n) = D(n) + \sum_{i=0}^{a-1} T(N - ki) + C(n)$
- That is
  - $T(n) = 1 + T(n-1)$ $\quad n > 1$
  - $T(1) = 1$ $\quad n = 1$

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

Complexity Analysis

- Resolution by unfolding
  - T(n) = 1 + T(n-1)
  - T(n-1) = 1 + T(n-2)
  - T(n-2) = 1 + T(n-3)
  - …
- Replacing in T(n)
  - T(n) = 1 + 1 + 1 + T(n-3)

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

$$= 1 + 1 + 1 + \ldots$$

$$= n$$

$$= O(n)$$

Termination
n-i = 1
i = n-1

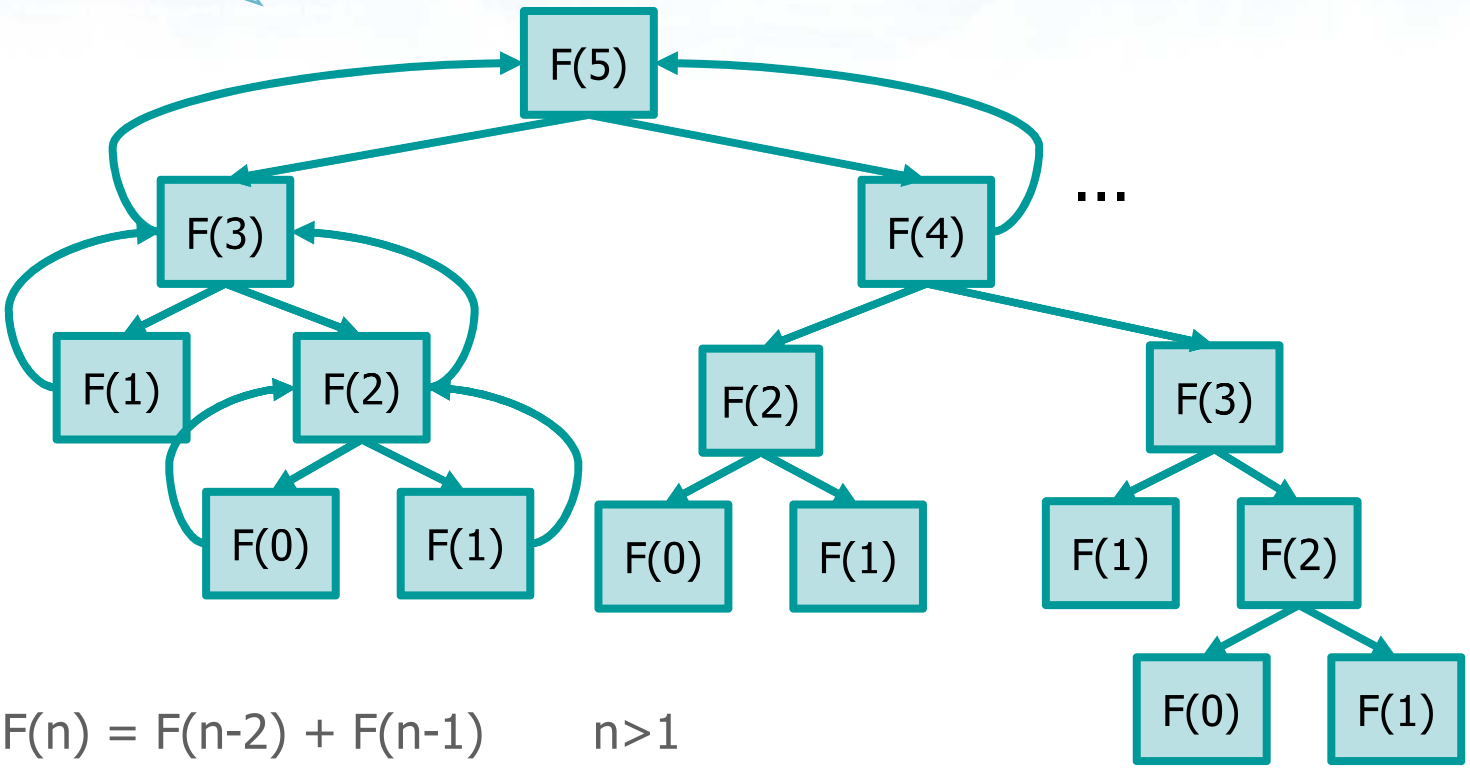
```
long int fact (long int n) {
  if (n == 0)
    return (1);
  return (n * fact(n-1));
}
```

Fibonacci Numbers

- Fibonacci numbers
  - Iterative and recursive definition
    - F(n) = F(n-2) + F(n-1)   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
    - etc.
    - That is
      - 0 1 1 2 3 5 8 13 21 34 …

An Example: Computing F(5)

Recursion tree (**complete**)



F(n) = F(n-2) + F(n-1)   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);
}
```

Complexity Analysis

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

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

Complexity Analysis

- Recurrence equation
  - $T(n) = D(n) + \sum_{i=0}^{a-1} T(N - ki) + C(n)$
- That is
  - $T(n) = 1 + T(n-1) + T(n-2) \qquad n > 1$
  - $T(0) = 1$
  - $T(1) = 1$

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

Complexity Analysis

- We can make the following conservative approximation
  - $T(n-2) \leq T(n-1)$
- Thus, we can replace T(n-2) with T(n-1), and we obtain
  - $T(n) = 1 + 2 \cdot T(n-1)$ $\quad$ $n > 1$
  - $T(n) = 1$ $\quad$ $n = 1$

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

Complexity Analysis

- Resolution by unfolding
  - $T(n) = 1 + 2 \cdot T(n-1)$
  - $T(n-1) = 1 + 2 \cdot T(n-2)$
  - $T(n-2) = 1 + 2 \cdot T(n-3)$
  - …
- Replacing in T(n)
  - $T(n) = 1 + 2 + 4 \cdot T(n-2)$

    $= 1 + 2 + 4 + 2^3 \cdot T(n-3)$

    $= \sum_{i=0}^{n-1} 2^i$

    $= 2^n - 1$

    $= O(2^n)$

Termination
n-i = 1
i = n-1

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

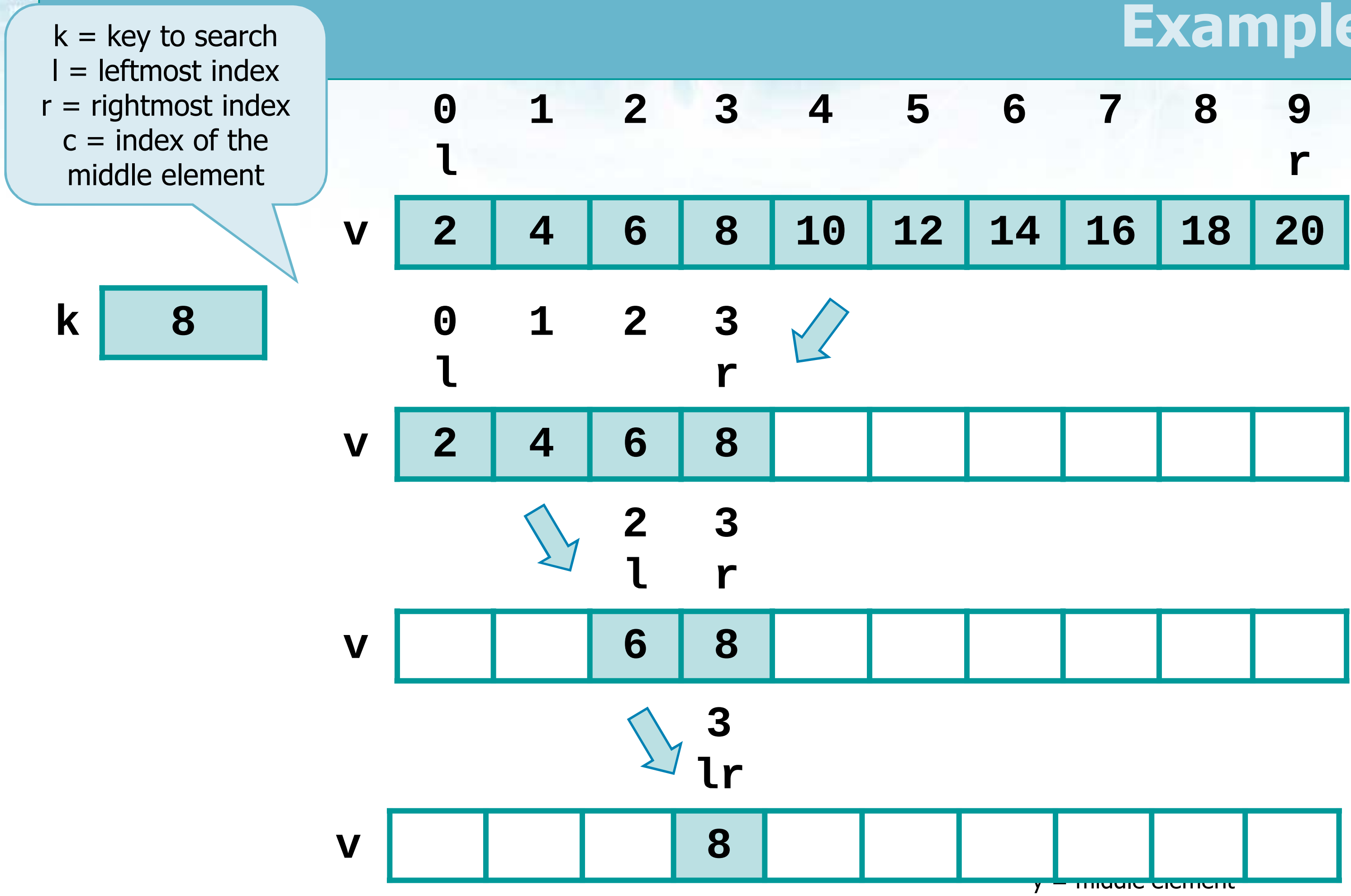
Not linear.
Why?

Binary Search

Assumption
$n = 2^p$

- Binary search
  - Does key k belong to the sorted array v[n]?
  - Yes/No
- 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 = key to search
l = leftmost index
r = rightmost index
c = index of the middle element
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

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

Complexity Analysis

- Decrease and conquer problem with
  - Number of subproblems
    - $a = 1$
  - Reduction factor
    - $b = n/n' = 2$
  - Division cost
    - $D(n) = \Theta(1)$
  - Recombination cost
    - $C(n) = \Theta(1)$

```
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;
}
```

Complexity Analysis

- Recurrence equation
  - $T(n) = D(n) + a \cdot T(n/b) + C(n)$
- That is
  - $T(n) = T(n/2) + 1 \qquad n > 1$
  - $T(1) = 1 \qquad n = 1$

```
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;
}
```

Complexity Analysis

- Resolution by unfolding
  - T(n/2) = T(n/4) +1
  - T(n/4) = T(n/8) +1
  - T(n/8) = …
- Replacing in T(n)
  - T(n) = 1 + 1 + 1 + T(n/8)

    $= \sum_{i=0}^{\log_2 n} 1$

    $= 1 + \log_2 n$
  - T(n) = O(log n)

Termination condition
$n/2^i = 1$
$i = \log_2 n$

```
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;
}
```

Reverse printing

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

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;
}
```

Complexity Analysis

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

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

Complexity Analysis

- Recurrence equation
  - $T(n) = D(n) + \sum_{i=0}^{a-1} T(N-ki) + C(n)$
- That is
  - $T(n) = 1 + T(n-1)$ $n > 1$
  - $T(1) = 1$ $n = 1$

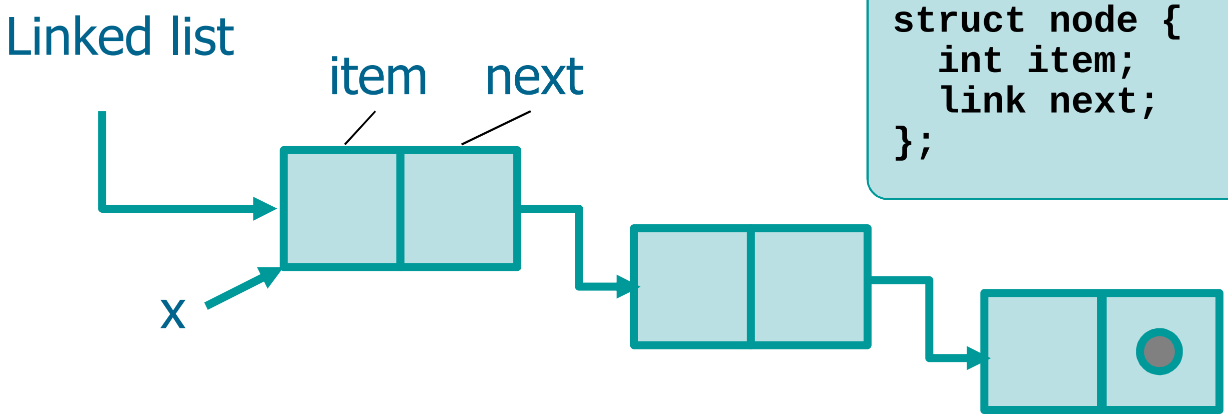
As for the factorial …

- Time complexity
  - $T(n) = O(n)$

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

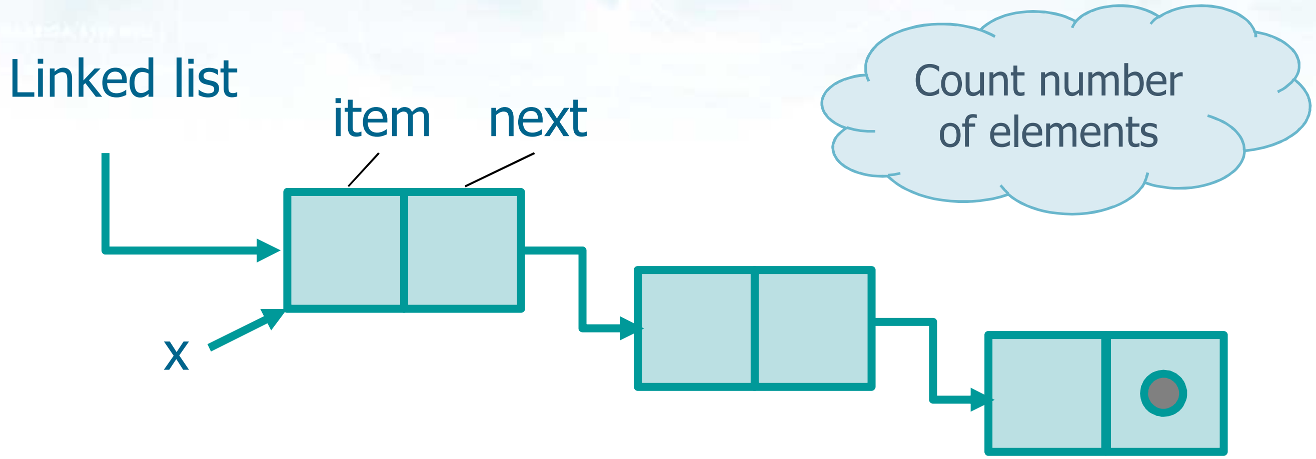
List processing

- 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



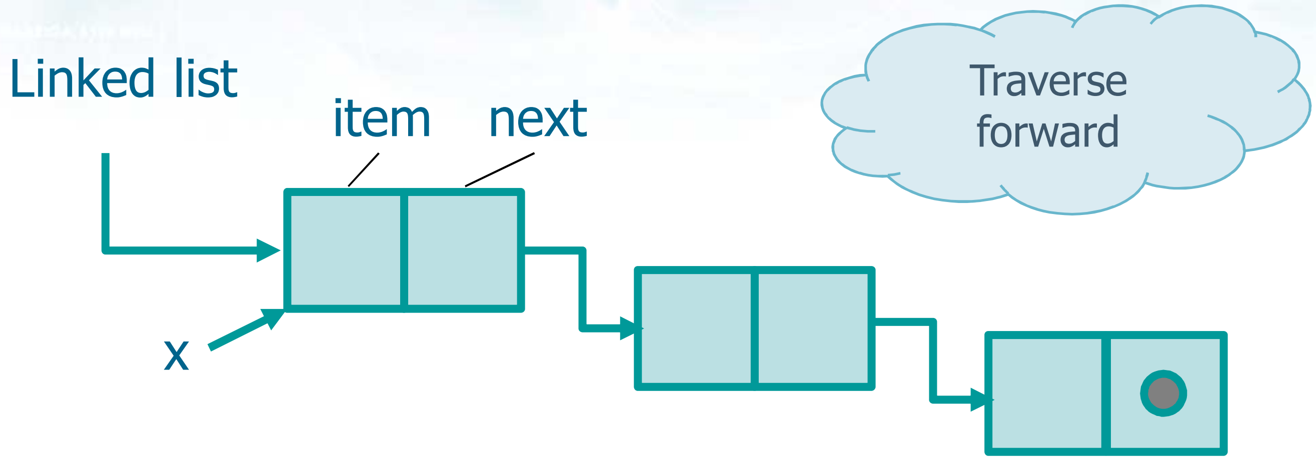

```
typedef struct node *link;
struct node {
  int item;
  link next;
};
```

Solution



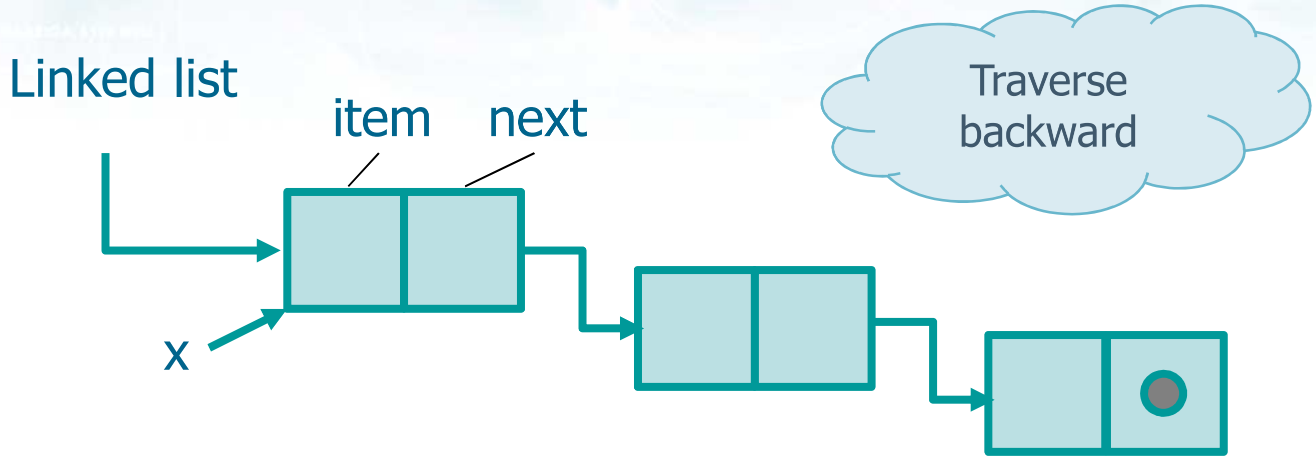

```
int count (link x) {
  if (x == NULL)
    return 0;
  return (1 + count(x->next));
}
```

Solution



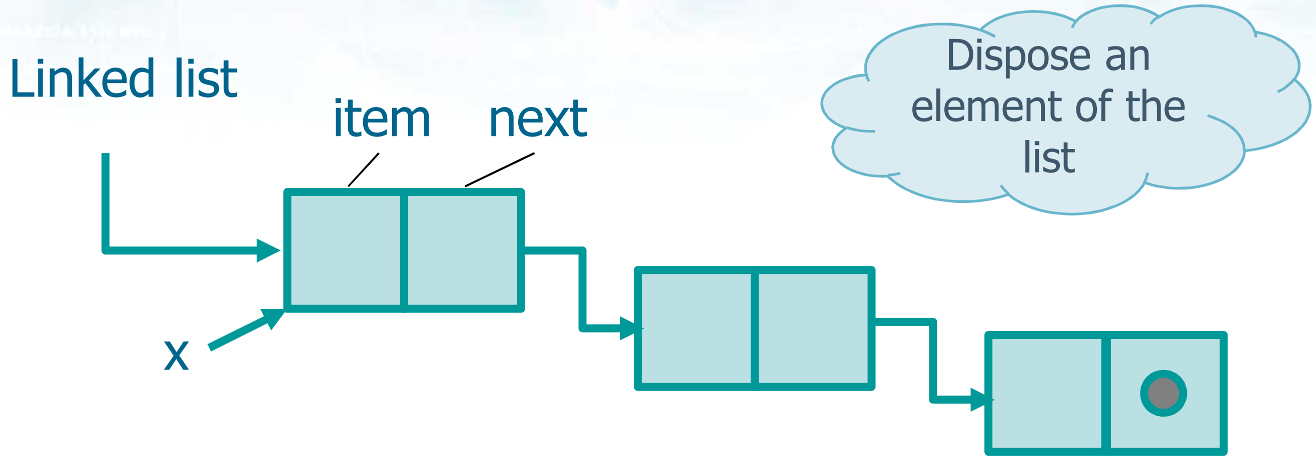

```
void traverse (link h) {
  if (h == NULL)
    return;
  printf ("%d", h->item);
  traverse (h->next);
}
```

Solution




```
void traverse_reverse (link h) {
  if (h == NULL)
    return;
  traverse_reverse (h->next);
  printf ("%d", h->item);
}
```

Solution




```
link delete(link x, Item v) {
  if (x == NULL)
    return NULL;
  if (x->item == v) {
    link t = x->next;
    free(x);
    return t;
  }
  x->next = delete (x->next, v);
  return x;
}
```

Create (re-create) link on the way back

Complexity Analysis

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

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

Complexity Analysis

- Recurrence equation
  - T(n) = D(n) + $\sum_{i=0}^{a-1} T(N - ki)$ + C(n)
- That is
  - T(n) = 1 + T(n-1) n > 1
  - T(1) = 1 n = 1

As for the factorial ...

- Time complexity
  - T(n) = O(n)

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

Greatest Common Divisor

- The greatest common divisor **gcd** of 2 non 0 integers x and 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

Common factors with the minimum exponent

$$\mathbf{x} = p_1^{e1} \cdot p_2^{e2} \cdot \ . \ . \ p_r^{er}$$
$$\mathbf{y} = p_1^{f1} \cdot p_2^{f2} \cdot \ . \ . \ p_r^{fr}$$
$$\mathbf{gcd(x,y)} = p_1^{\min(e1,f1)} \cdot p_2^{\min(e2,f2)} \cdot \ . \ . \ p_r^{\min(er,fr)}$$

- More efficient methods are base on Euclid's algorithm

Euclid's Algorithm: Version 1

- Version number 1 is based on subtraction

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

- Termination

```
if x == y
  return x
```

Euclid's Algorithm: Version 1

- Examples
  - 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

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

Solution 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);
}
```

Euclid's Algorithm: Version 2

- Version number 2 is based on the remainder of integer divisions

```
if y > x
  swap (x, y)
  // that is; tmp=x; x=y; y=tmp;

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

- Termination

```
if y == 0
  return x
```

Euclid’s Algorithm: Version 2

- Examples
  - gcd (20, 8) =
    = gcd (8, 20%8) = gcd (8, 4)
    = gcd (4, 8%4) = gcd (4, 0)
    = 4 → return 4
  - 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)
```

Euclid's Algorithm: 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 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

- Decrease and conquer problem with
  - Number of subproblems
    - a = 1
  - Reduction value
    - k_i variable
  - Division cost
    - D(x,y) = Θ(1)
  - Recombination cost
    - C(x,y) = Θ(1)
- Demonstration beyond the scope of this course
  - T(n) = O(log y)

Determinant

- Laplace Algorithm with unfolding on row I
  - Square matrix M (n·n) with indices from 1 to n
- Computation

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

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

Example

- Given the matrix

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

- Compute its determinant as

$$\begin{aligned} \boldsymbol{det}(\boldsymbol{M}) = \ & (-\mathbf{1})^{(1+1)} \cdot (-\mathbf{2}) \cdot \boldsymbol{det}(\boldsymbol{M}_{\boldsymbol{minor}1,1}) \\ & + (-\mathbf{1})^{(1+2)} \cdot (\ \mathbf{2}) \cdot \boldsymbol{det}(\boldsymbol{M}_{\boldsymbol{minor}\,1,2}) \\ & + (-\mathbf{1})^{(1+3)} \cdot (-\mathbf{3}) \cdot \boldsymbol{det}(\boldsymbol{M}_{\boldsymbol{minor}\,1,3}) \end{aligned}$$

Example

- Minor computation

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

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

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

Example

- Termination condition (terminal case)
  - Square matrix M 2x2
    - det(M) = M[0][0] · M[1][1] - M[0][1] · M[1][0]
  - That is
    - $\det \left(\begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix} \right) = -1 - 0 = -1$
    - $\det \left(\begin{bmatrix} -1 & 3 \\ 2 & -1 \end{bmatrix} \right) = 1 - 6 = -5$
    - $\det \left(\begin{bmatrix} -1 & 1 \\ 2 & 0 \end{bmatrix} \right) = 0 - 2 = -2$

Example

- Then

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

$$\begin{aligned} \boldsymbol{det(M)} = {} & (-\mathbf{1})^{(\mathbf{1+1})} \cdot (-\mathbf{2}) \cdot \boldsymbol{det(M_{minor1,\,1})} \\ & + (-\mathbf{1})^{(\mathbf{1+2})} \cdot (\ \mathbf{2}) \cdot \boldsymbol{det(M_{minor\,1,\,2})} \\ & + (-\mathbf{1})^{(\mathbf{1+3})} \cdot (-\mathbf{3}) \cdot \boldsymbol{det(M_{minor\,1,\,3})} \end{aligned}$$

$$\boldsymbol{det(M)} = (\mathbf{1}) \cdot (-\mathbf{2}) \cdot (-\mathbf{1}) + (-\mathbf{1}) \cdot (\mathbf{2}) \cdot (-\mathbf{5}) + (\mathbf{1}) \cdot (-\mathbf{3}) \cdot (-\mathbf{2}) = \mathbf{18}$$

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][0] · 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($M_{minor\ i,\ j}$)
  - Store result results in
    - sum = sum + M[0][k] · pow (-1,k) · 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

- Decrease and conquer problem with
  - Number of subproblems
    - $a = n$
  - Reduction value
    - $k_i = 2 \cdot n-1$
  - Division cost
    - $D(n) = \Theta(1)$
  - Recombination cost
    - $C(n) = \Theta(1)$
- Demonstration beyond the scope of this course
  - $T(n) = O(n!)$

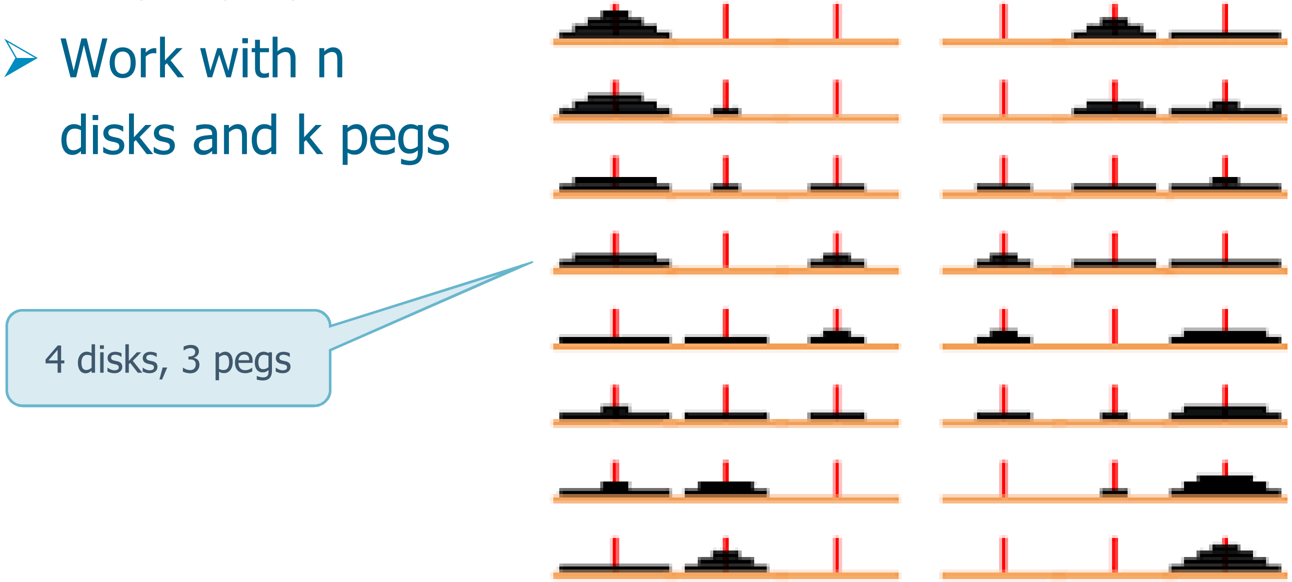
Tower of Hanoi

- By the French mathematician Édouard Lucas (1883)
- Initial configuration
  - 3 pegs
    - Pegs are identified with 0, 1, 2
  - 3 disks
  - Disks of decreasing size on first peg

- Final configuration
  - 3 disks on third peg

Tower of Hanoi

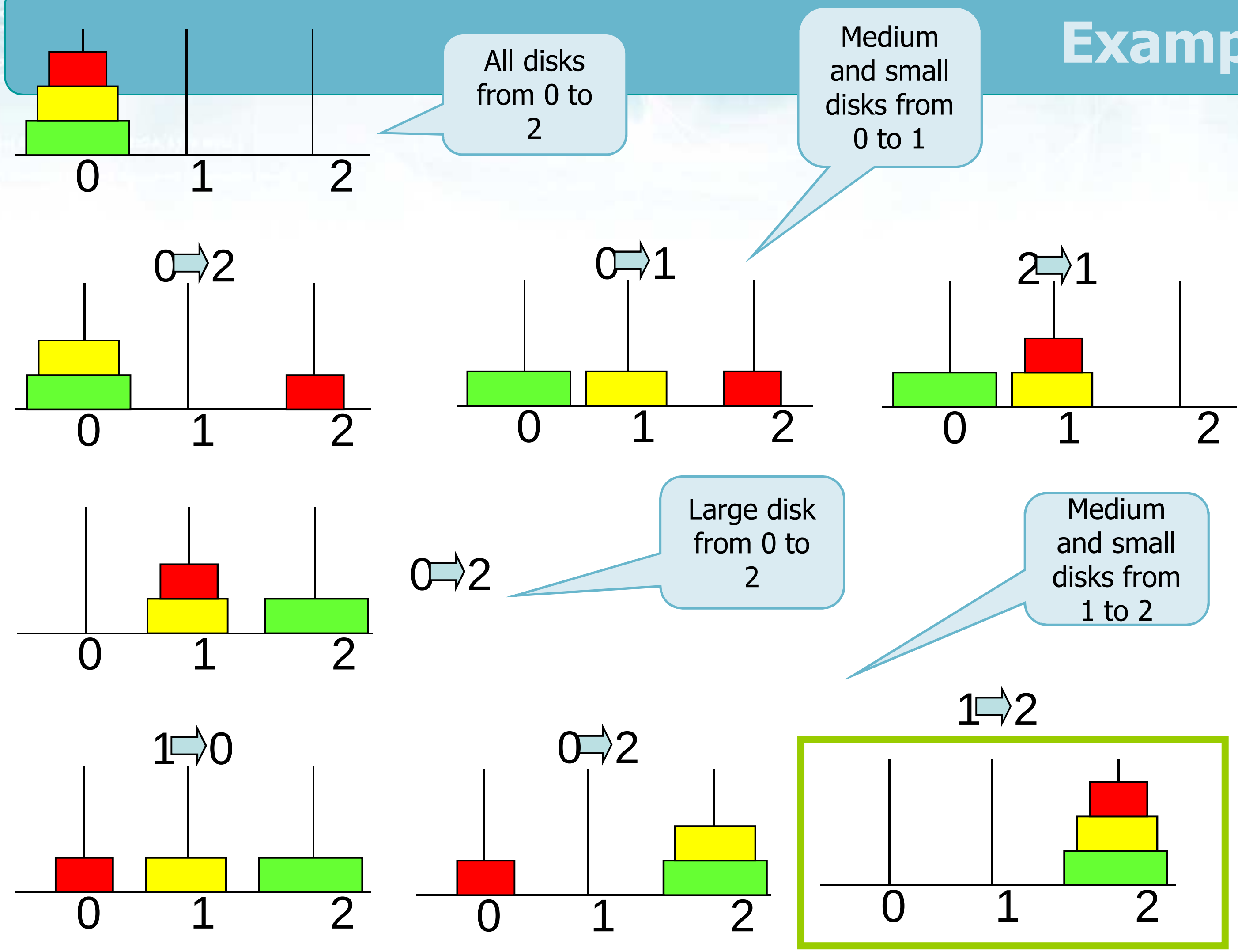
- Rules
  - Access only to the top disk
  - On each disk overalp only smaller disks
- Generalization
  - Work with n disks and k pegs



Solution

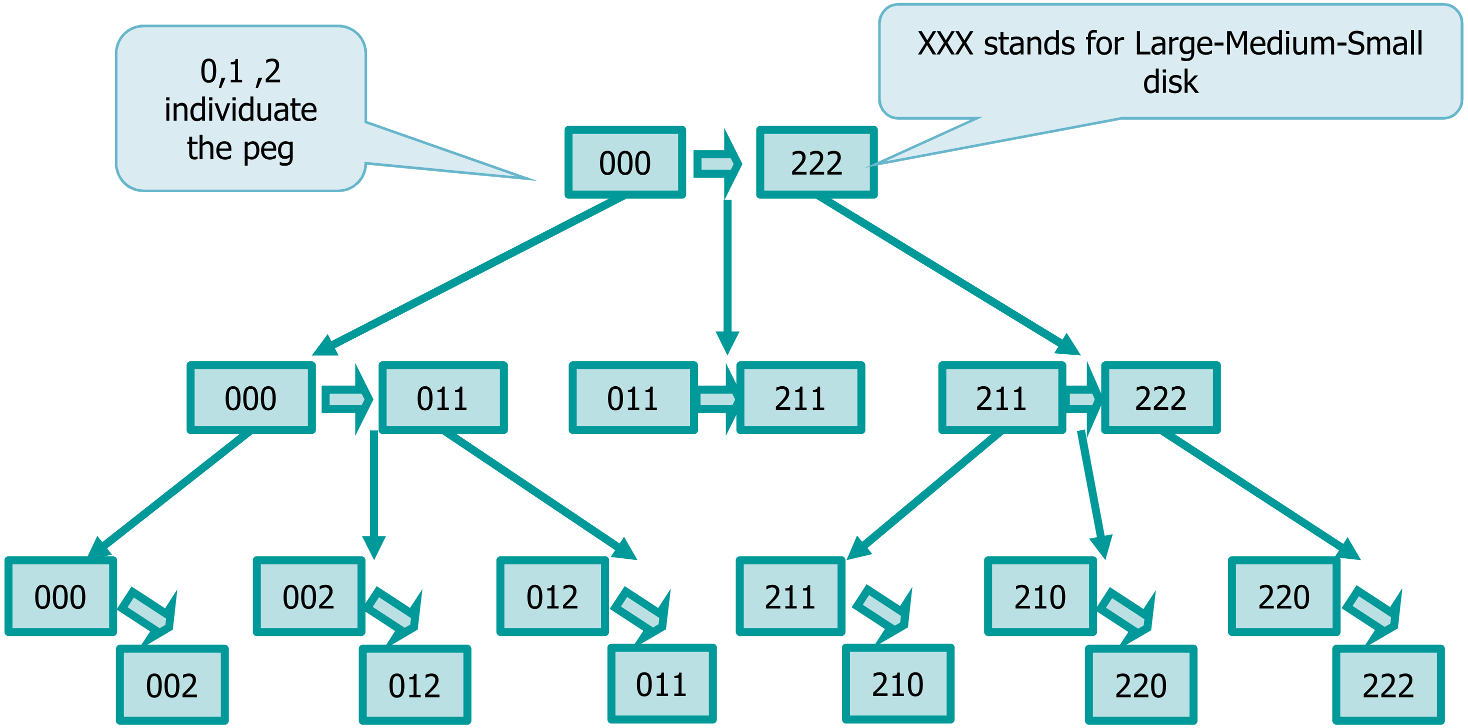
- Divide and Conquer strategy
  - Initial problem
    - Move n disks from 0 to 2
  - Reduction to subproblems
    - Move n-1 disks from 0 to 1, 2 temporary storage
    - Move last disk from 0 to 2
    - Move n-1 disks from 1 to 2, 0 temporary storage
  - Termination condition
    - Move just 1 disk

Example


0
1
2
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
0⇨2
1⇨2

Recursion tree

- The previous divide and conquer strategy generates the following recursion tree



Solution

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

  aux = 3 - (src + dest);

  if (n == 1) {
    printf("src %d -> dest %d \n", src, dest);
    return;
  }

  hanoi (n-1, src, aux);
  printf("src %d -> dest %d \n", src, dest);
  hanoi (n-1, aux, dest);

  return;
}
```

Termination condition

Divide

Recursion

Elementary solution

Recursion

Divide

Complexity Analysis

- Decrease and conquer problem with
  - Number of subproblems
    - $a = 2$
  - Reduction value
    - $k_i = 1$
- Divide
  - Consider n-1 disks
  - $D(n) = \Theta(1)$

```
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;
}
```

Complexity Analysis

- Solve
  - Solve 2 subproblems whose size is n-1 each
  - $T(n) = 2 \cdot T(n-1)$
- Termination
  - Move 1 disk
  - $T(1) = \Theta(1)$
- Combine
  - No action
  - $C(n) = \Theta(1)$

```
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;
}
```

Complexity Analysis

- Recurrence equation
  - $T(n) = D(n) + \sum_{i=0}^{a-1} T(N - ki) + C(n)$
- That is
  - $T(n) = 2 \cdot T(n-1) + 1$ $\quad n > 1$
  - $T(1) = 1$ $\quad n = 1$

As for the fibonacci sequence ...

```
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;
}
```

- Timec complexity
  - $T(n) = O(2^n)$