Algo Design 2

Cours 2

Sort, Recursivity and Linked Lists

Algorithm Complexity

- **Goal : mesure the inherent efficiency**
  - As a function of the data input size
  - Compute the number of elementary operations
  - Asymptotic measure
  - Worst-case/Average-case
  - Time / Space complexity

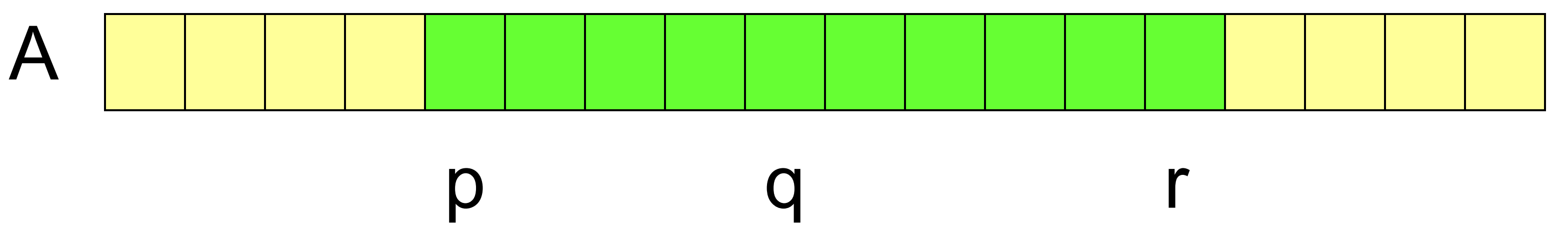
Sorting Problem

Algorithms	Worst-case	Average
Insertion Buble Sort	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$\Theta(n^2)$	$\Theta(n.\log(n))$
Merge Sort	$\Theta(n.\log(n))$	$\Theta(n.\log(n))$

Merge Sort

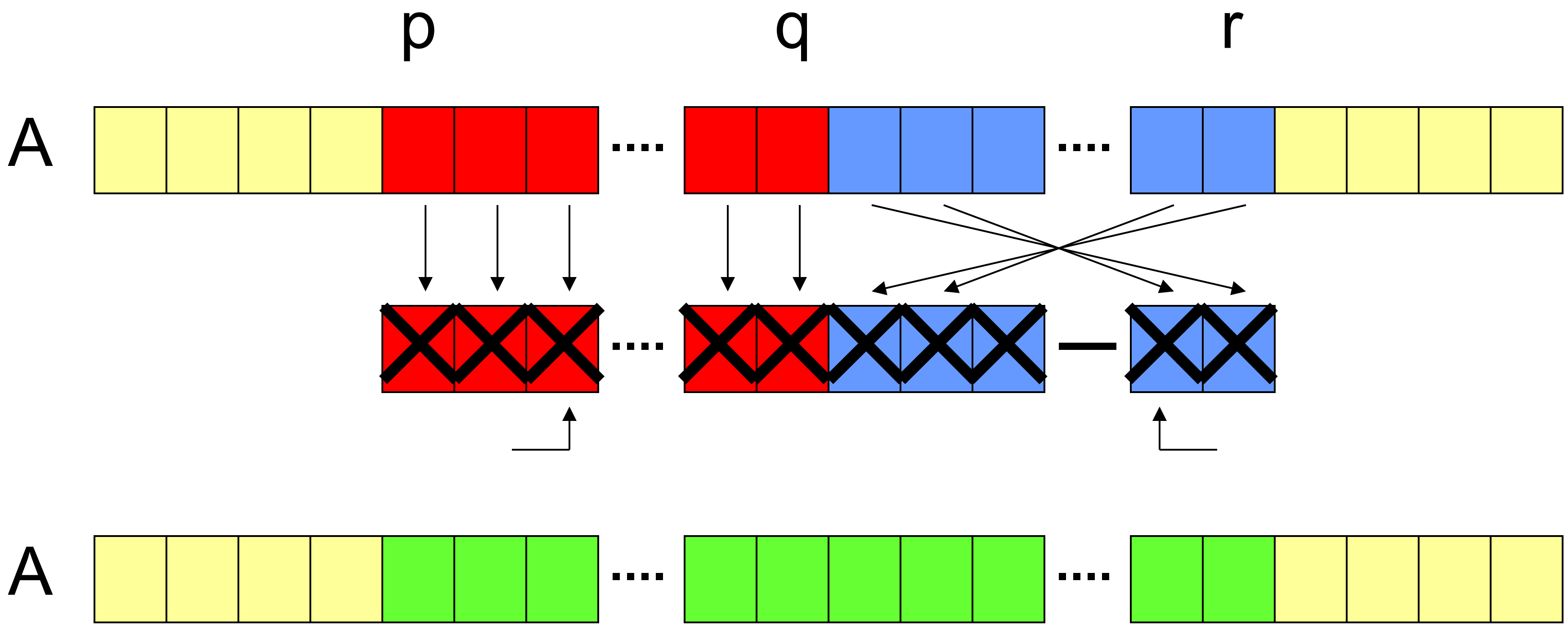
- Sort an array A between the indices p & r :

```
• if p<r then
        q=(p+r)/2
        mergesort(A,p,q)
        mergesort(A,q+1,r)
        merge(A,p,q,r)
```



Merge Sort

- Merge of two sorted arrays:



- Complexity ≈ number of elements to merge

Merge Sort

2	8	12	1	7	16	15	5	3	4	9	14	10	13	11	6	
2		2		2		2		2		2		2		2		
2	8	1	12	7	16	5	15	3	4	9	14	10	13	6	11	2×2^3
4				4				4				4				
1	2	8	12	5	7	15	16	3	4	9	14	6	10	11	13	$2^2 \times 2^2$
8								8								
1	2	5	7	8	12	15	16	3	4	6	9	10	11	13	14	$2^3 \times 2$
16																
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	$2^4 \times 1$
																4×2^4

$$C_n = \Theta(\log(n) \times n)$$

Quicksort

- Recursive Sort based on partition
- **`if p<r then`**

```
        q=partition(A,p,r)
        quicksort(A,p,q-1)
        quicksort(A,q+1,r)
```

- Worst-case complexity: $C_n = \Theta(n^2)$
- Average-case complexity: $\mathbf{C_n = \Theta(n.log(n))}$

Sort : can we do better ?

- If we do not make any additional assumption on the data
- Representation of the algorithm with a *decision tree*
- **Any sorting algorithm needs about *n.log(n)* comparisons**

Sort : can we do better ?

- *Decision tree :*

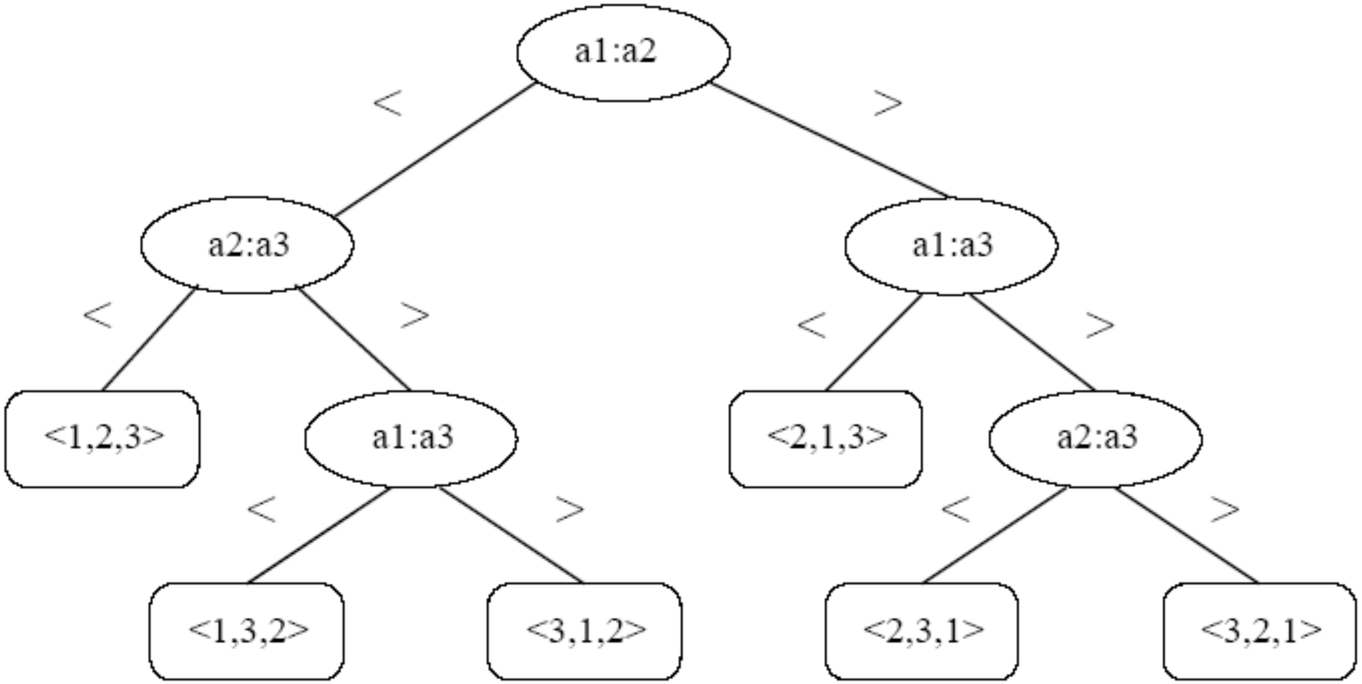


FIG. 2.3 – *Arbre de décision d'un algorithme de tri*

Sort : can we do better ?

- Number of « leaves » ≥ number of permutations with ***n*** elements = ***n*!**

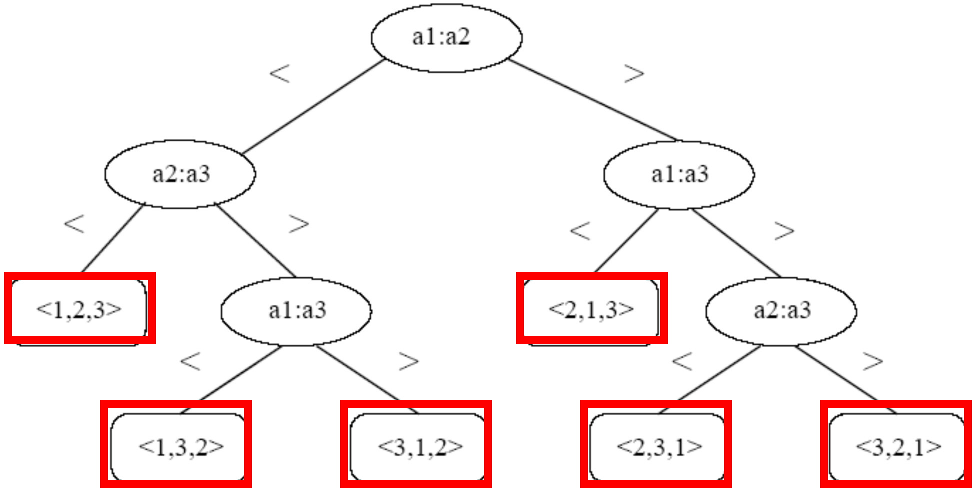


FIG. 2.3 – *Arbre de décision d'un algorithme de tri*

Sort : can we do better ?

- Number of comparisons to sort = length of the branch
- Height Tree = maximum number of comparisons

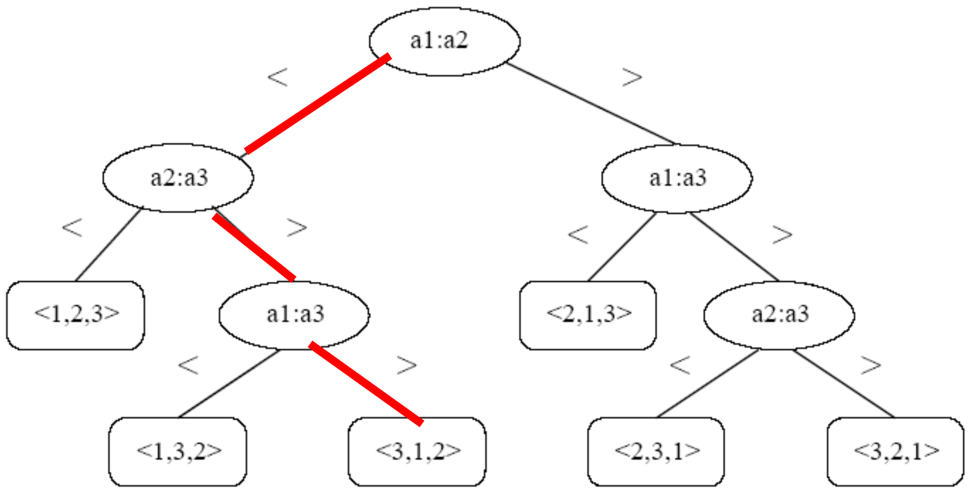


FIG. 2.3 – *Arbre de décision d'un algorithme de tri*

Sort : can we do better ?

- Height Tree ***h***

  Maximum number of leaves = 2^h

- $2^h \geq$ Number of leaves $\geq$ ***n*!**

$$h \geq \log n! > \log\left(\left(\frac{n}{e}\right)^n\right) = n\log n - n\log e$$

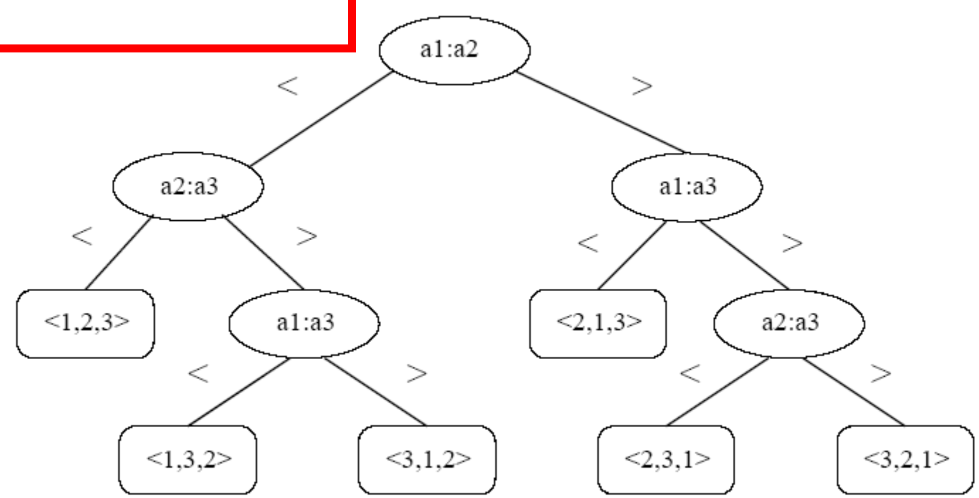
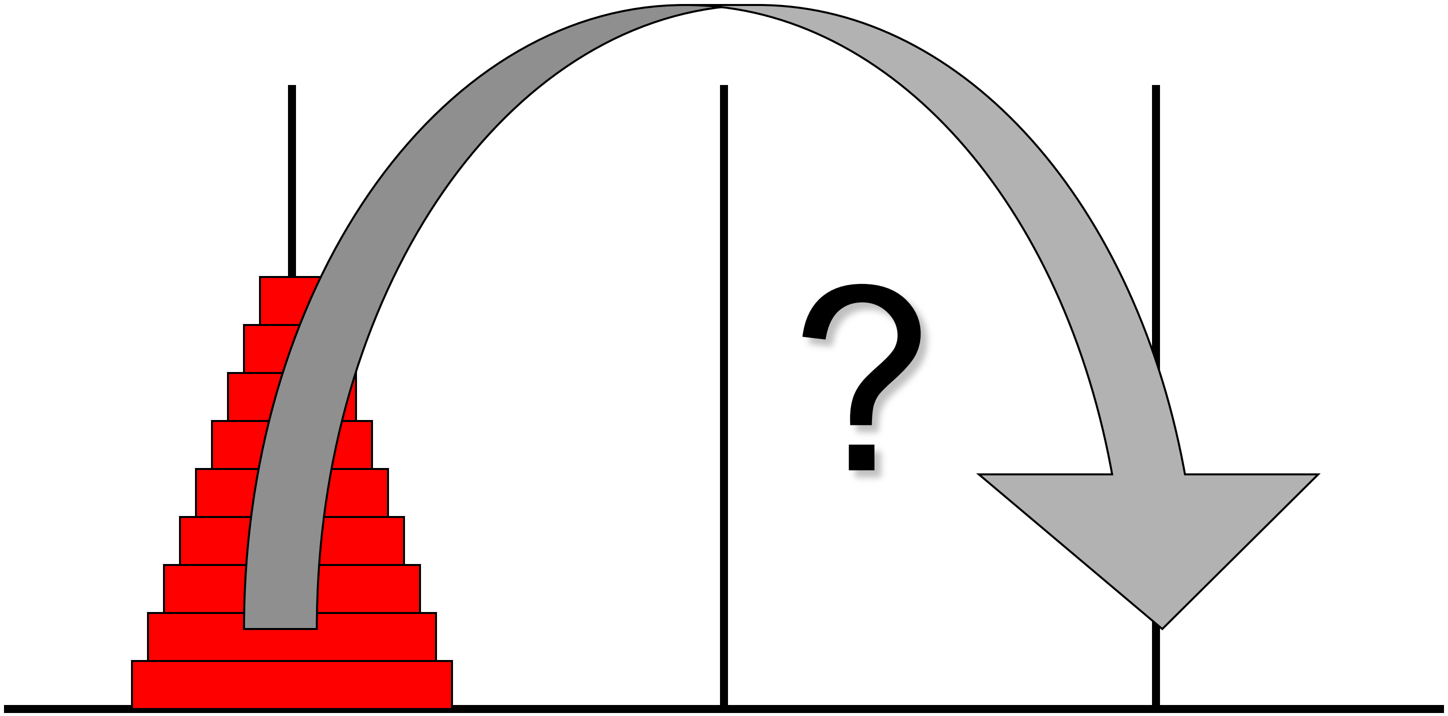


FIG. 2.3 – *Arbre de décision d'un algorithme de tri*

Sort : can we do better?

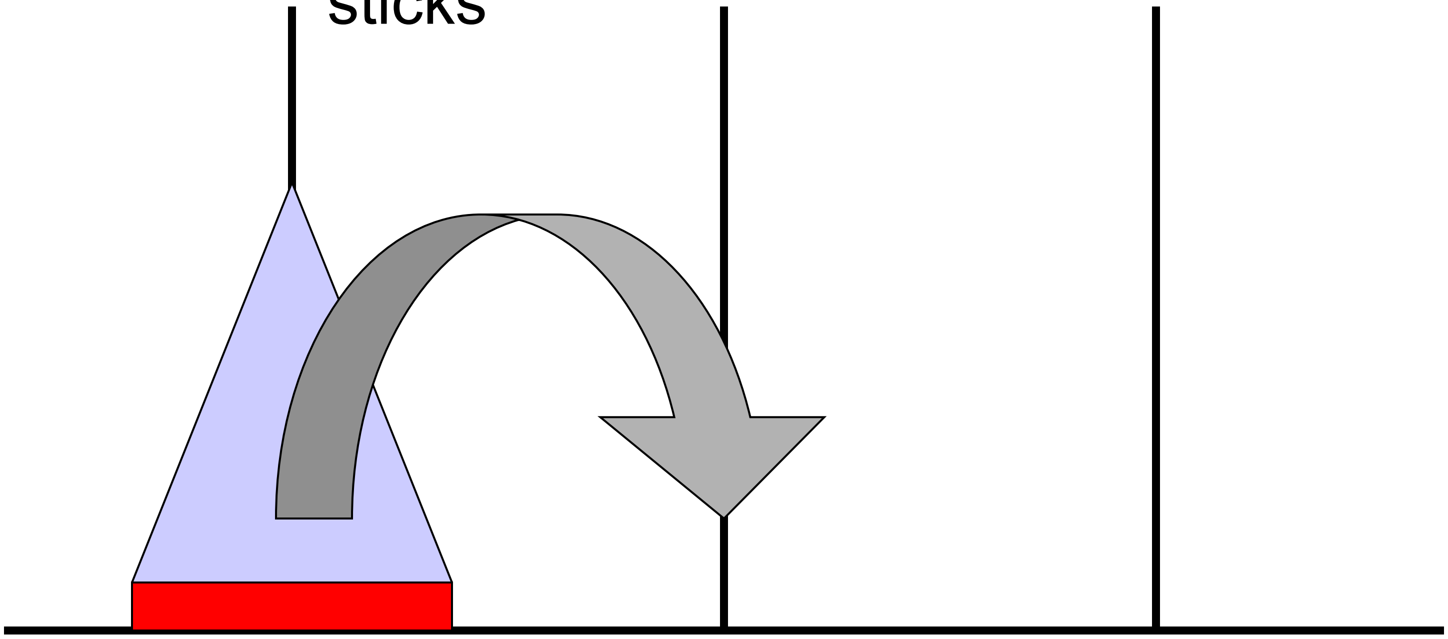
- If we do not make any additional assumption on the data, **any sorting algorithm needs at least about *n.log(n)* comparisons**
- And if we do additional assumption, …?

Hanoï Towers



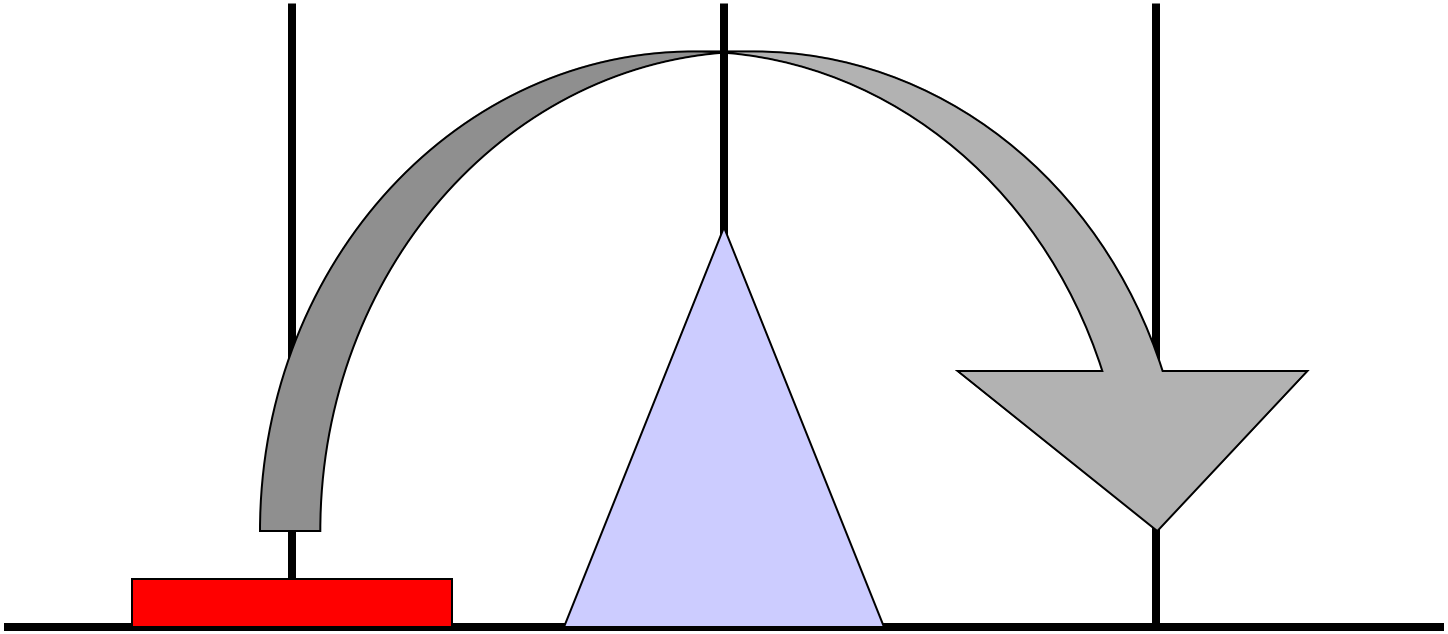
Hanoï Towers

Recursive move of **n-1** sticks

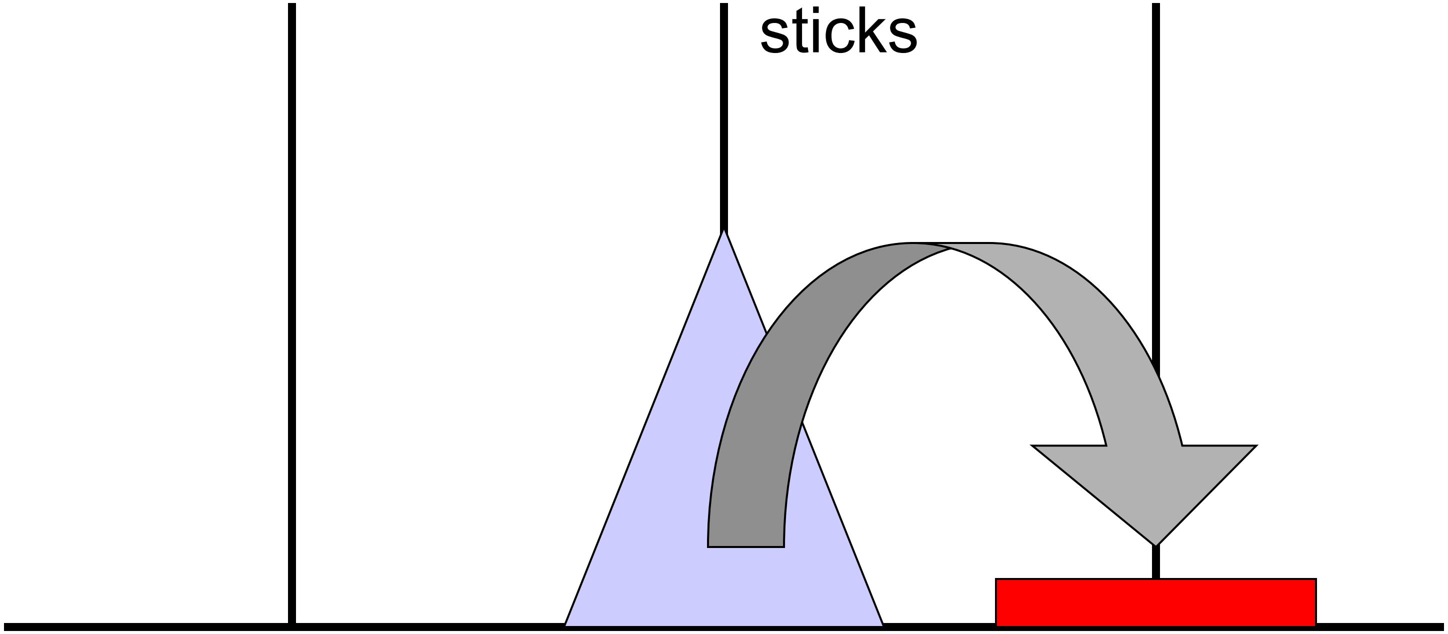


Hanoï Towers

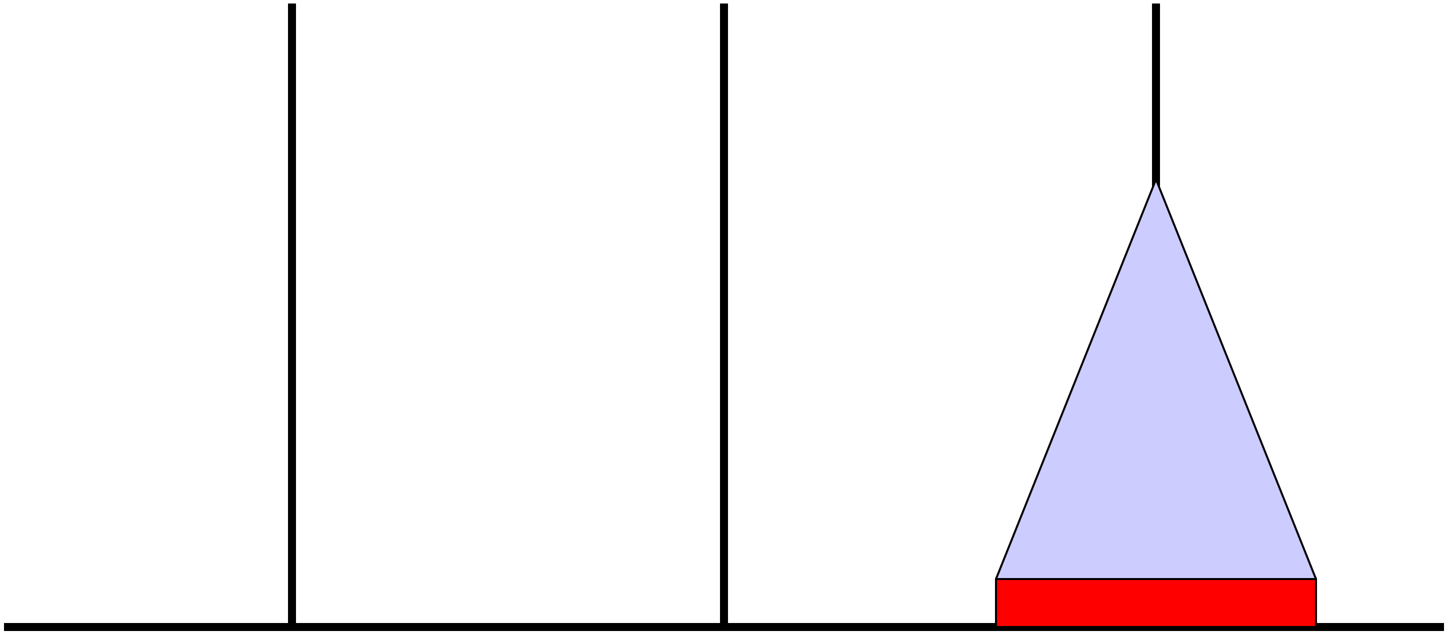
Move largest one



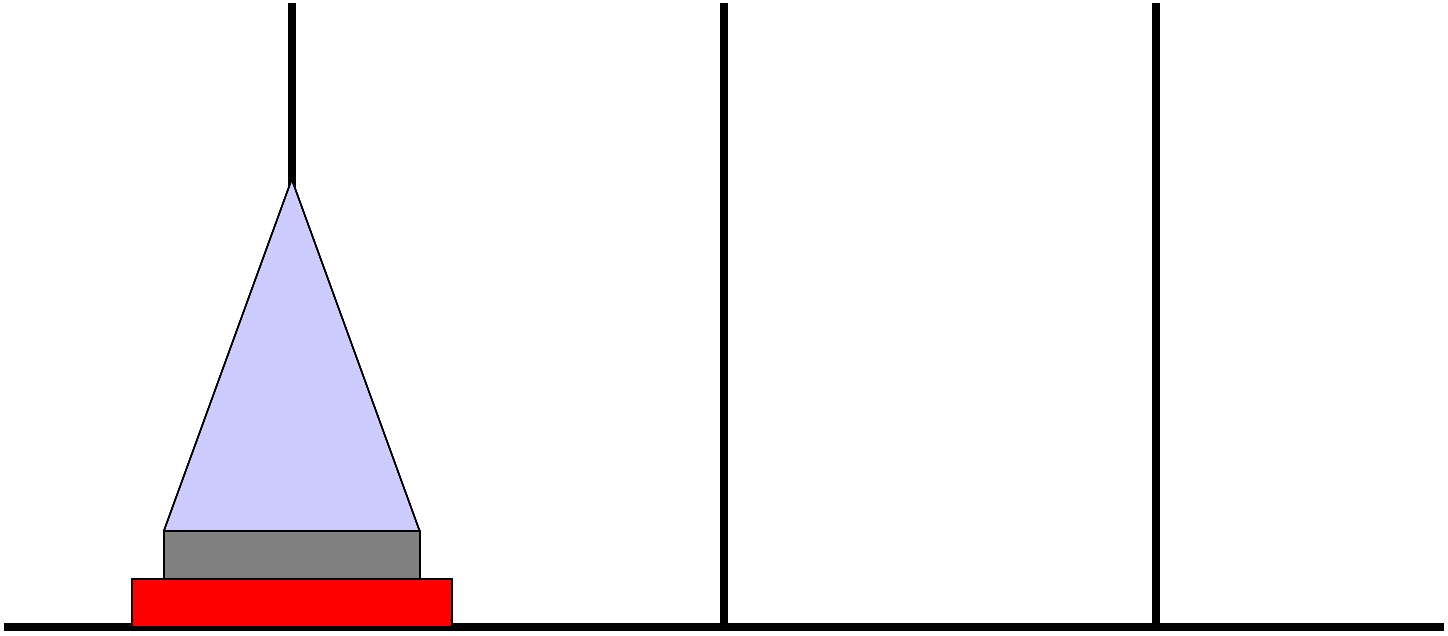
Hanoï Towers



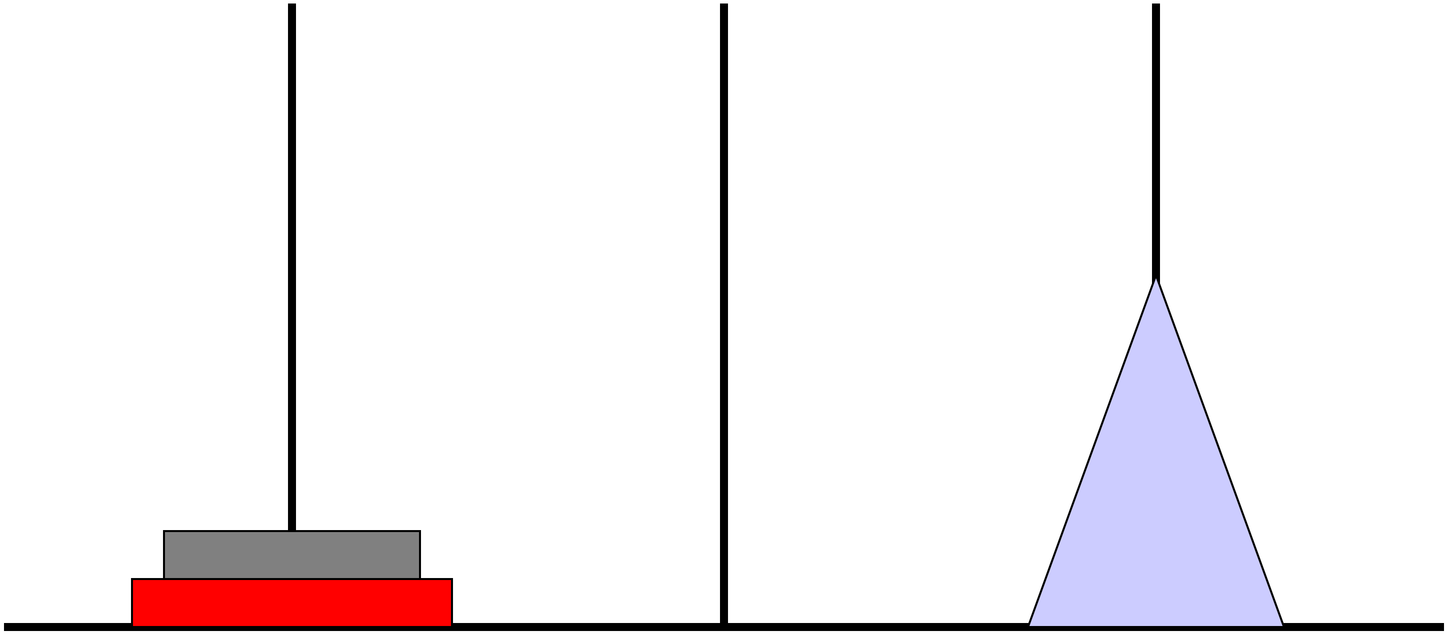
Hanoï Towers



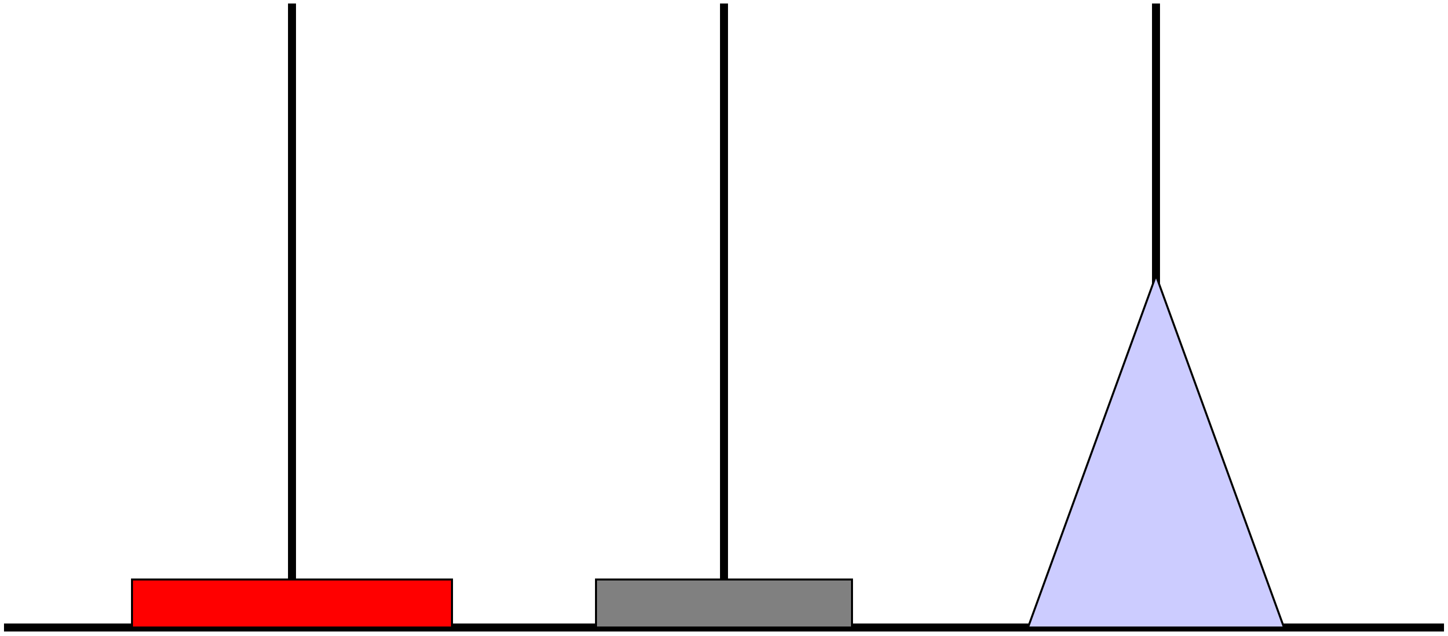
Hanoï Towers



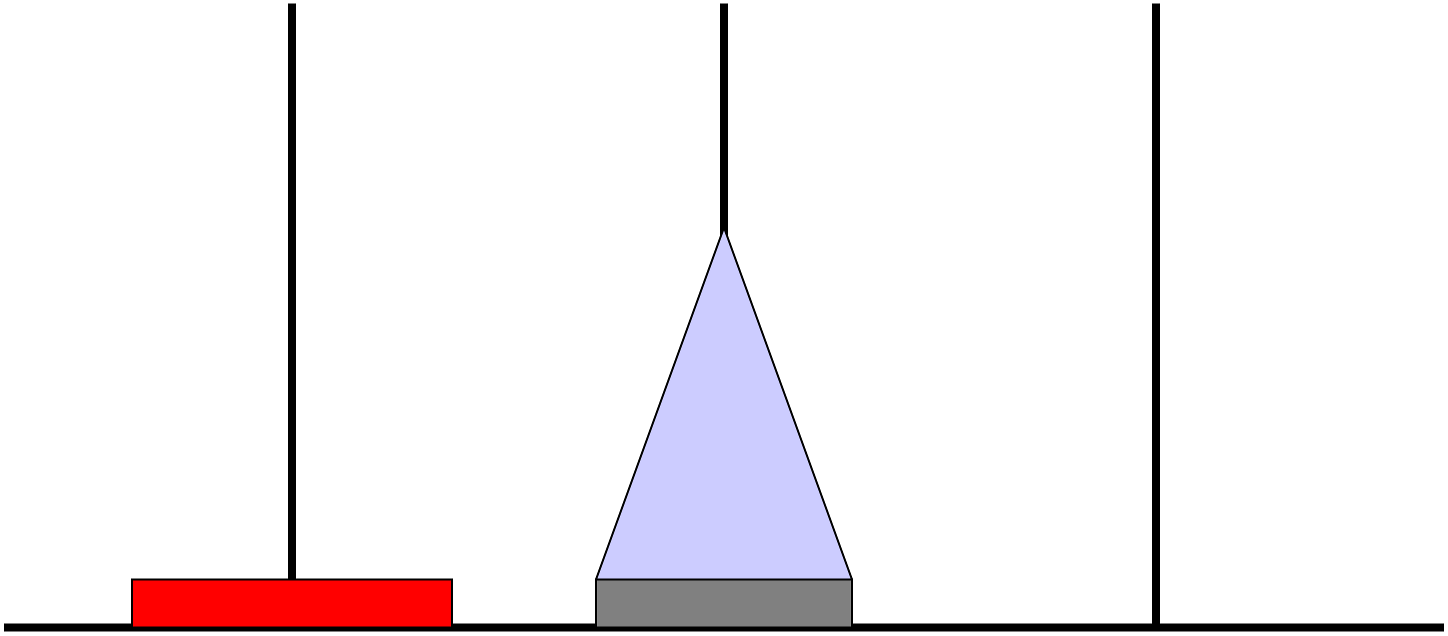
Hanoï Towers



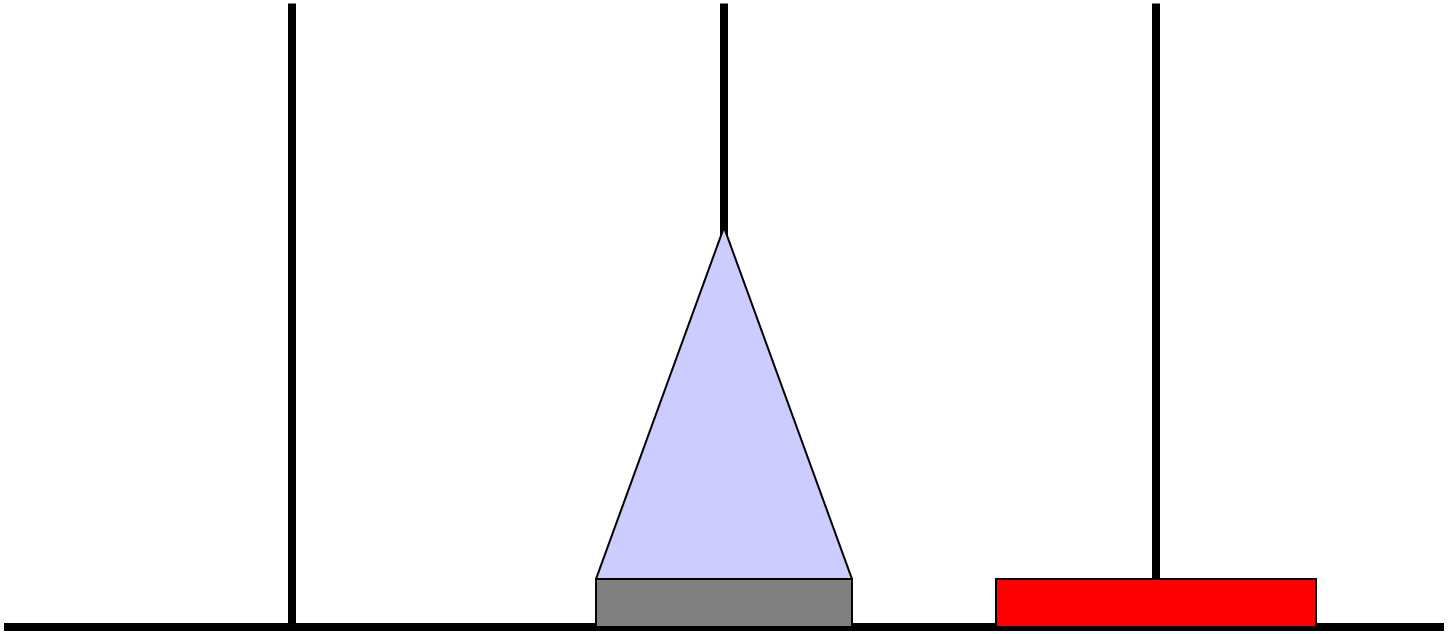
Hanoï Towers



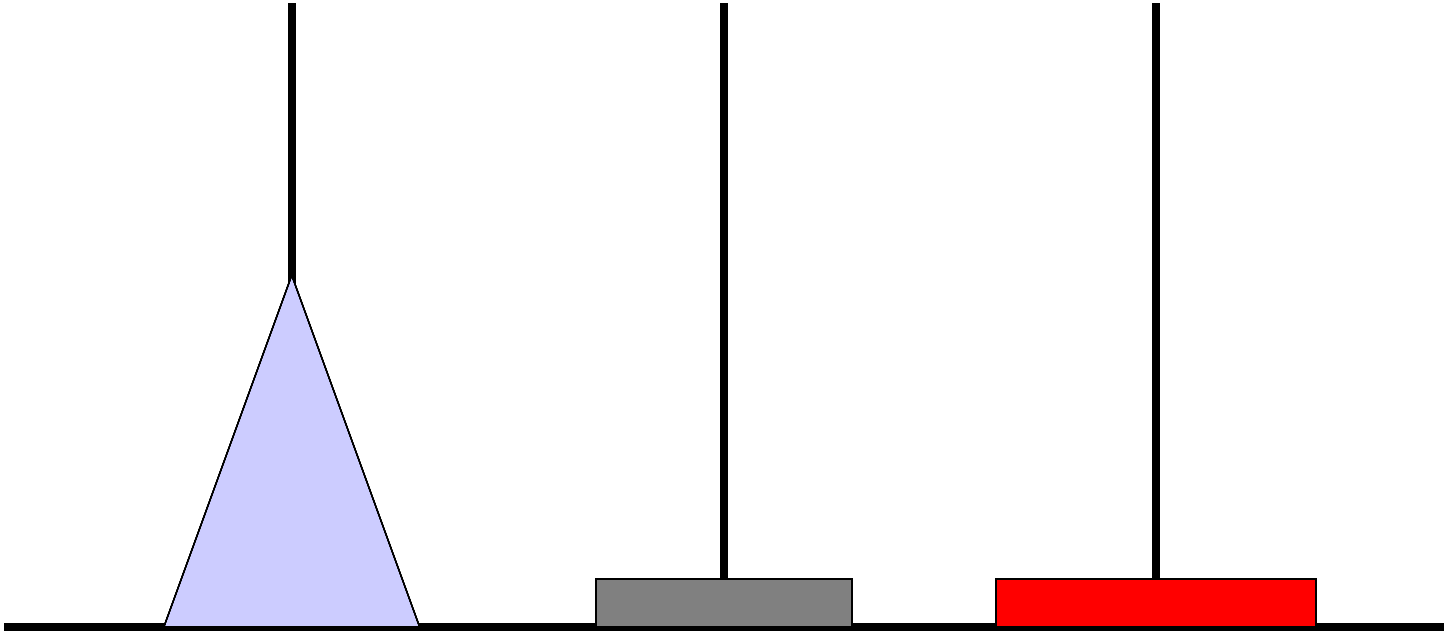
Hanoï Towers



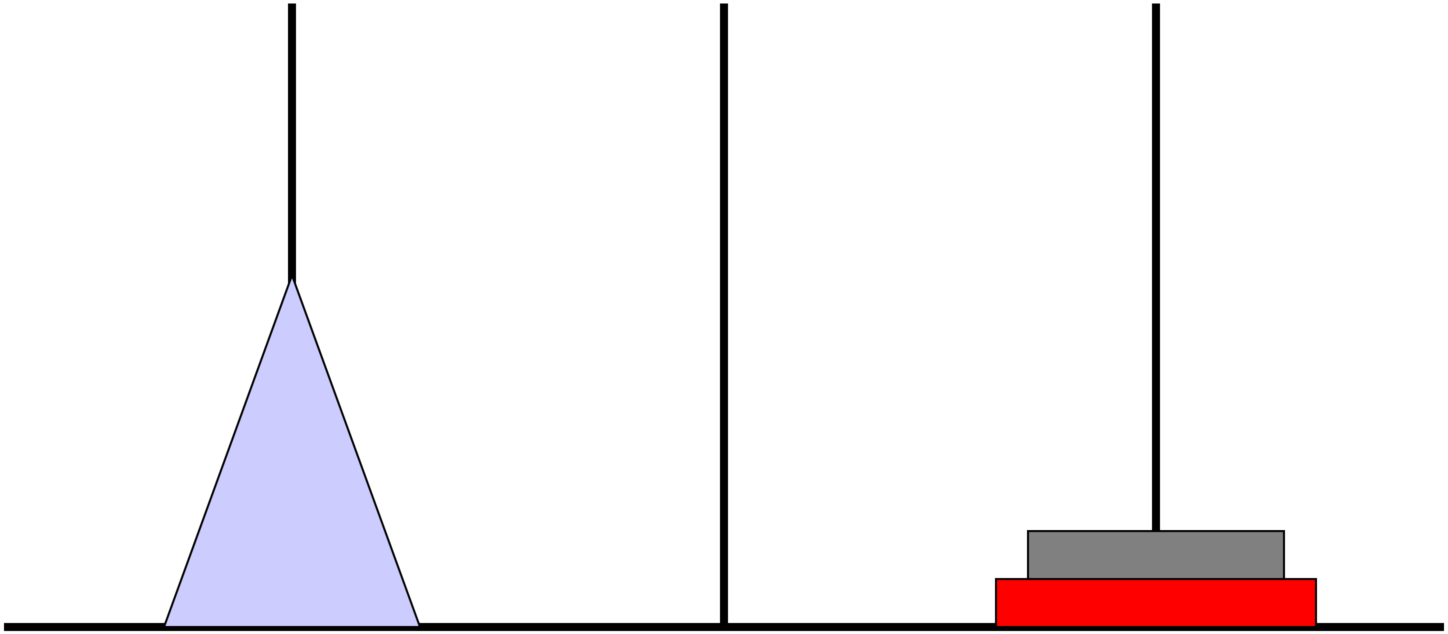
Hanoï Towers



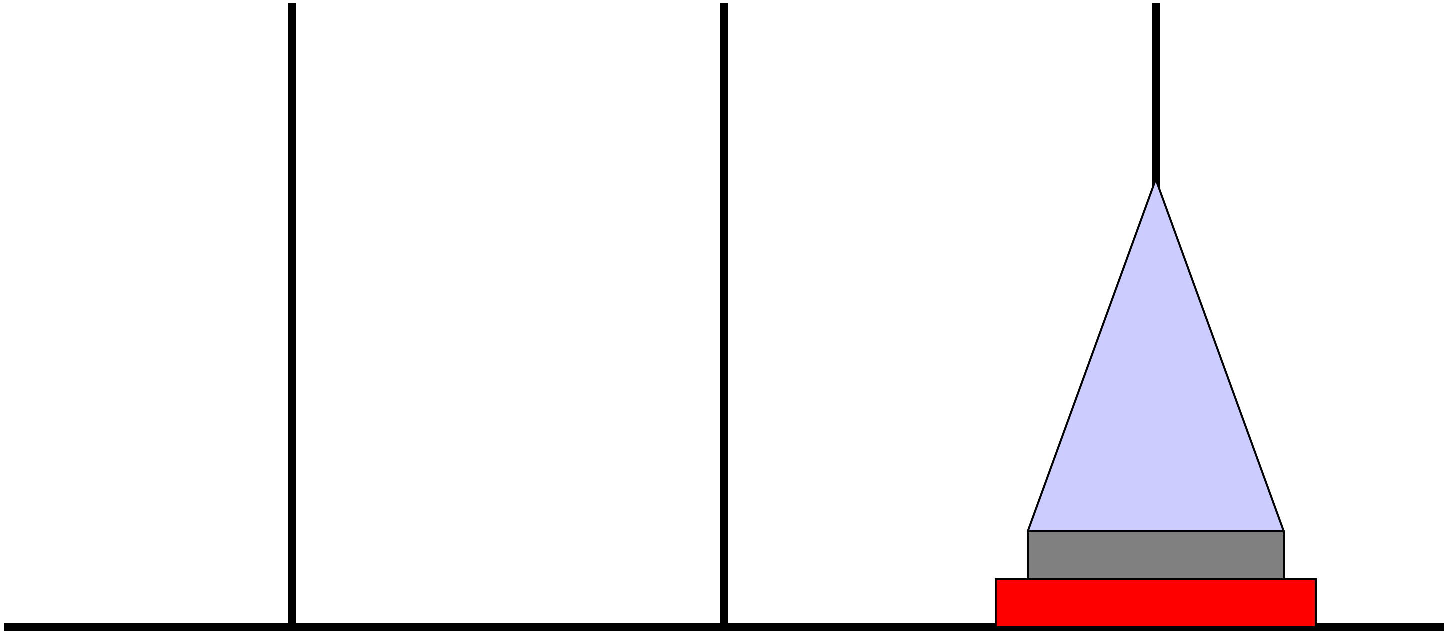
Hanoï Towers



Hanoï Towers



Hanoï Towers



Hanoï Towers

- Recursive solution

```
void Hanoi(int n, int i, int j)
{ int intermediaire=6-(i+j);
  if (n>0)
    { Hanoi(n - 1, i, intermediaire);
      printf("Mouvement du piquet %d
                vers le piquet %d\n",i,j);
      Hanoi(n - 1, intermediaire, j);
    }
}
```

Hanoï : solution

- **`Mouvement du piquet 1 vers le piquet 3`**
  `Mouvement du piquet 1 vers le piquet 2`
  `Mouvement du piquet 3 vers le piquet 2`
  `Mouvement du piquet 1 vers le piquet 3`
  `Mouvement du piquet 2 vers le piquet 1`
  `Mouvement du piquet 2 vers le piquet 3`
  `Mouvement du piquet 1 vers le piquet 3`

Hanoï : complexity

- Number of moves :
  - $C_0=0$
  - $C_n=1+2.C_{n-1}$
- $C_n = 2^n-1 = \Theta(\mathbf{2^n})$

  The exponential complexity is inherent to the problem
- Space complexity : $\Theta(n)$

Data structures

- Array
- Lists
- Variants of lists
  - stack
  - queue
  - Circular list
  - Double-linked list
- Advanced data structures: trees, graphs,...

Data Structures

- **Static Array**
  - Simple, efficient, limited

```
int t[10];
t[5]=17;

t[10]=4;

int n=4;
int t[n];
```

Data Structure

- **Dynamic Array**
  - less simple, as efficient, less limited

```
int *t;
int n;
printf("Valeur de n ? ");
scanf("%d",&n);
t=(int *)malloc(sizeof(int)*n);
t[0]=17;
...
free(t);
```

List

- More complex to program
- Different efficiency than array
- We can do anything

- <u>Theoretical definition </u>:

list=Nil *and* **`Cons`**(*x*,**list**)

Lists : manipulation

- **`Cons`**(1,**`Cons`**(2,**`Cons`**(3,Nil)))

  noted [1,2,3]

- **`Cons`**(1,[2,3,4]) = [1,2,3,4]
- **`Head`** ([1,2,3,4]) = 1
- **`Tail`** ([1,2,3,4]) = [2,3,4]

C Implementation

- ```
  struct cell
  {
     int val;
     struct cell *next;
  };
  typedef struct cell *List;
  ```
- Graphical Representation

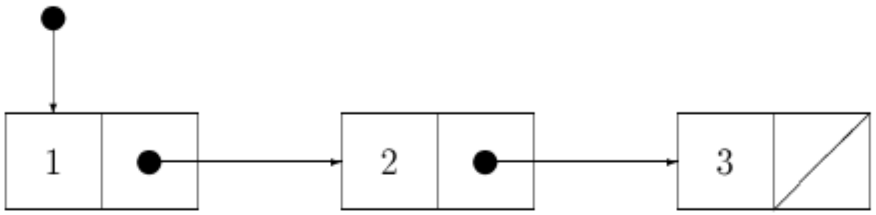


FIG. 3.3 – *Représentation graphique de la liste chaînée* [1, 2, 3]
  ```

C Implementation

- ```
  List cons(int v, List L)
   {List nouv;
    nouv=(List) malloc(sizeof(struct cell));
    nouv->val=v;  nouv->next=L;
    return nouv;}
  ```
- ```
  int head(List L)
   {if (L==NULL)
      {printf("head(NULL)\n"); exit(-1);}
    return L->val;}
  ```
- ```
  Liste tail(List L)
   {if (L==NULL)
      {printf( "tail(NULL)\n"); exit(-1);}
    return L->next;}
  ```
  ```

Iterative *vs* fonctionnal

```
• void print_List1(List L) //     (ITERATIVE)
  { List P=L;
    while (P!=NULL)
      {printf("%d ",P->val);
       P=P->next;}
    printf("\n");
  }

• void print_List2(List L) //     (FONCTIONNAL)
  { if (L==NULL) printf("\n");
    else
      {printf("%d ",head(L));
       print_Liste2(tail(L));}
  }
```

Searching an element

- ```
  int element1(int v, List L)
  { List P=L;
    while (P!=NULL)
      {if (P->val==v) return 1;
       P=P->next;}
    return 0;}
  ```
- ```
  int element2(int v, List L)
  { if (L==NULL) return 0;
    return ((v==head(L))||element2(v,tail(L)));
  }
  ```
- Complexity : $\Theta(n)$

Insertion at kth rank

- ```
  Liste add1(int v, int k, List L)
  {
    if (k==0) return cons(v,L);
    return
     cons(head(L),add1(v,k-1,tail(L)));
  }
  ```
- What is going on ?
  ```

Insertion at kth rank

- ```
  void add2(int v, int k, List *L)
  { int i;
    List P,nouv;
    if (k==0) *L=cons(v,*L);
    else
      {if (k>length(*L))
       {printf("Add impossible\n"); exit(-1);}
       P=*L;
       for(i=0;i<k-1;i++)
         P=P->next;
       nouv=(List) malloc(sizeof(int));
       nouv->val=v;
       nouv->next=P->next;
       P->next=nouv;}
  }
  ```
  ```

Deletion of a list

- ```
  void freelist(List L)
  {
    if (L!=NULL)
      {
        freelist(L->next);
        free(L);
      }
  }
  ```
  ```

Inversion of a list

- ```
  Liste renverse1(List L)
  { if (L==NULL)
        return L;
    else
        return
  concat(renverse1(tail(L)),cons(head(L),NULL));
  }
  ```
- ```
  Liste renverse2(List L, List Acc)
  { if (L==NULL)
        return Acc;
    else
        return renverse2(tail(L),cons(head(L),Acc));
  }
  ```

Variants

- Double-linked lists
- Circular Lists
- Stack (LIFO)
- Queue (FIFO)