

Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación



# IIC2115 – Programación como herramienta para la Ingeniería

## Capítulo 2 - Parte 2: Técnicas y Algoritmos

**Profesores:** Hans Löbel  
Francisco Garrido

## ¿Por qué revisamos técnicas de programación?

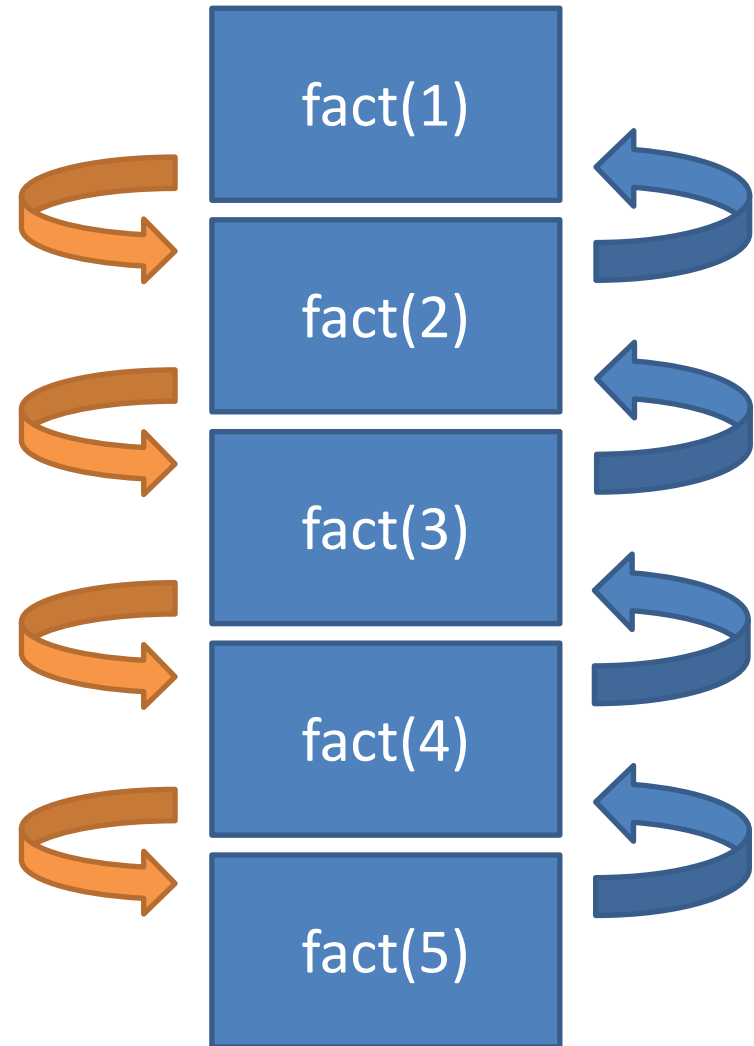
- Para resolver muchos problemas reales, no basta con usar estructuras de datos para obtener una solución eficiente.
- Generalmente, se requieren técnicas o algoritmos complejos, que utilicen estas estructuras de maneras no triviales.
- La clave para aprender a usarlas correctamente, es tratar de resolver una gran cantidad de problemas de distinto tipo con ellas.
- Dicho en otras palabras, no sirve memorizar un par de ejemplos dónde cada técnica puede ser aplicada.

## Recursión: difícil pero práctica

- Se basa en que las funciones se llamen a si mismas, cada vez con instancias levemente distintas del problema.
- Muy práctica, pero es difícil de conceptualizar para muchos problemas y requiere práctica en su uso.
- Conceptualmente, se parece a un stack: en vez de almacenar datos, almacena llamados a funciones. Luego, estas se van ejecutando en orden LIFO.

## Recursión: difícil pero práctica

```
def factorial_recursivo(n):  
    if n == 1:  
        return 1  
    return n*factorial_recursivo(n-1)  
factorial_recursivo(5)
```



## Recursión: otra forma de verla en tres pasos

```
def factorial_recursivo(n):  
    if n == 1:  
        return 1  
    return n*factorial_recursivo(n-1)  
  
factorial_recursivo(5)
```

---

## Recursión: otra forma de verla en tres pasos

```
def factorial_recursivo(n):  
    if n == 1:  
        return 1  
    return n*factorial_recursivo(n-1)  
  
factorial_recursivo(5)
```

---

```
def funcion_que_resuelve_el_problema(parametros):  
    # resolver el problema de alguna forma  
    # como sea, no importa
```

## Recursión: otra forma de verla en tres pasos

```
def factorial_recursivo(n):  
    if n == 1:  
        return 1  
    return n*factorial_recursivo(n-1)  
  
factorial_recursivo(5)
```

---

```
def funcion_que_resuelve_el_problema(parametros, n):  
    # resolver el problema de alguna forma  
    # como sea, no importa  
  
def funcion_que_dada_solucion_para_n_menos_uno_resuelve_para_n(parametros, n, solucion_n_menos_uno):  
    # tomo la solución parcial y la transformo  
  
def instancia_es_trivial_por_tamaño(n)  
    # retorna True si la solución es calculable trivialmente  
    # retorna False si no lo es  
  
def solucion_caso_base(parametros, n)  
    # retorna la solución al problema cuando el tamaño  
    # permite resolverlo directamente  
  
def funcion_mas_humilde(parametros, n)  
    if instancia_es_trivial_por_tamaño(n):  
        return solucion_caso_base(parametros, n)  
    else:  
        solucion_n_menos_uno = funcion_que_resuelve_el_problema(parametros, n - 1)  
        return funcion_que_dada_solucion_para_n_menos_uno_resuelve_para_n(parametros, n, solucion_n_menos_uno)
```

## Recursión: otra forma de verla en tres pasos

```
def factorial_recursivo(n):  
    if n == 1:  
        return 1  
    return n*factorial_recursivo(n-1)  
  
factorial_recursivo(5)
```




```
def funcion_que_dada_solucion_para_n_menos_uno_resuelve_para_n(parametros, n, solucion_n_menos_uno):  
    # tomo la solución parcial y la transformo  
  
def instancia_es_trivial_por_tamaño(n)  
    # retorna True si la solución es calculable trivialmente  
    # retorna False si no lo es  
  
def solucion_caso_base(parametros, n)  
    # retorna la solución al problema cuando el tamaño  
    # permite resolverlo directamente  
  
def funcion_mas_humilde(parametros, n)  
    if instancia_es_trivial_por_tamaño(n):  
        return solucion_caso_base(parametros, n)  
    else:  
        solucion_n_menos_uno = funcion_que_resuelve_el_problema(parametros, n - 1)  
        return funcion_que_dada_solucion_para_n_menos_uno_resuelve_para_n(parametros, n, solucion_n_menos_uno)
```



## Recursión: otra forma de verla en tres pasos

```
def factorial_recursivo(n):  
    if n == 1:  
        return 1  
    return n*factorial_recursivo(n-1)  
  
factorial_recursivo(5)
```

```
d ...nci ..._an ...s ...e ...n ...oh ...a(n ...):  
...f ...l ...n ...e ...d ...a ...a ...  
...s ...r ...f ...  
  
def funcion_que_dada_solucion_para_n_menos_uno_resuelve_para_n(parametros, n, solucion_n_menos_uno):  
    # tomo la solución parcial y la transformo  
  
def instancia_es_trivial_por_tamaño(n)  
    # retorna True si la solución es calculable trivialmente  
    # retorna False si no lo es  
  
def solucion_caso_base(parametros, n)  
    # retorna la solución al problema cuando el tamaño  
    # permite resolverlo directamente  
  
def funcion_mas_humilde(parametros, n)  
    if instancia_es_trivial_por_tamaño(n):  
        return solucion_caso_base(parametros, n)  
    else:  
        solucion_n_menos_uno = funcion_mas_humilde(parametros, n - 1)  
        return funcion_que_dada_solucion_para_n_menos_uno_resuelve_para_n(parametros, n, solucion_n_menos_uno)
```



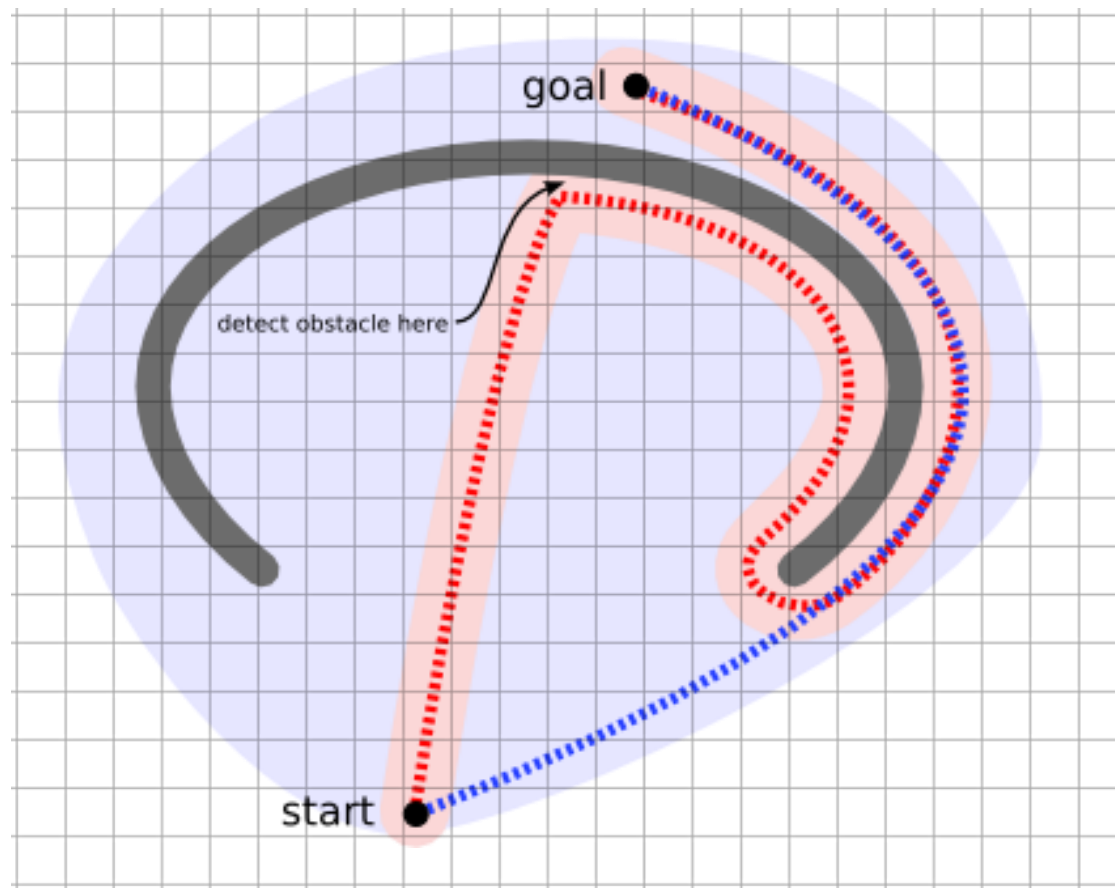
## Backtracking: búsqueda con recursión eficiente

- Permite evitar recorrido de todo el árbol de recursión para encontrar una solución.
- Se almacena el último estado válido antes de la recursión, y se recupera si el nuevo estado no lo es.
- También puede entenderse como un stack, pero en este caso, al llevar registro del estado del problema, podemos evitar agregar más cosas al tope de este.

## Backtracking: búsqueda eficiente con recursión

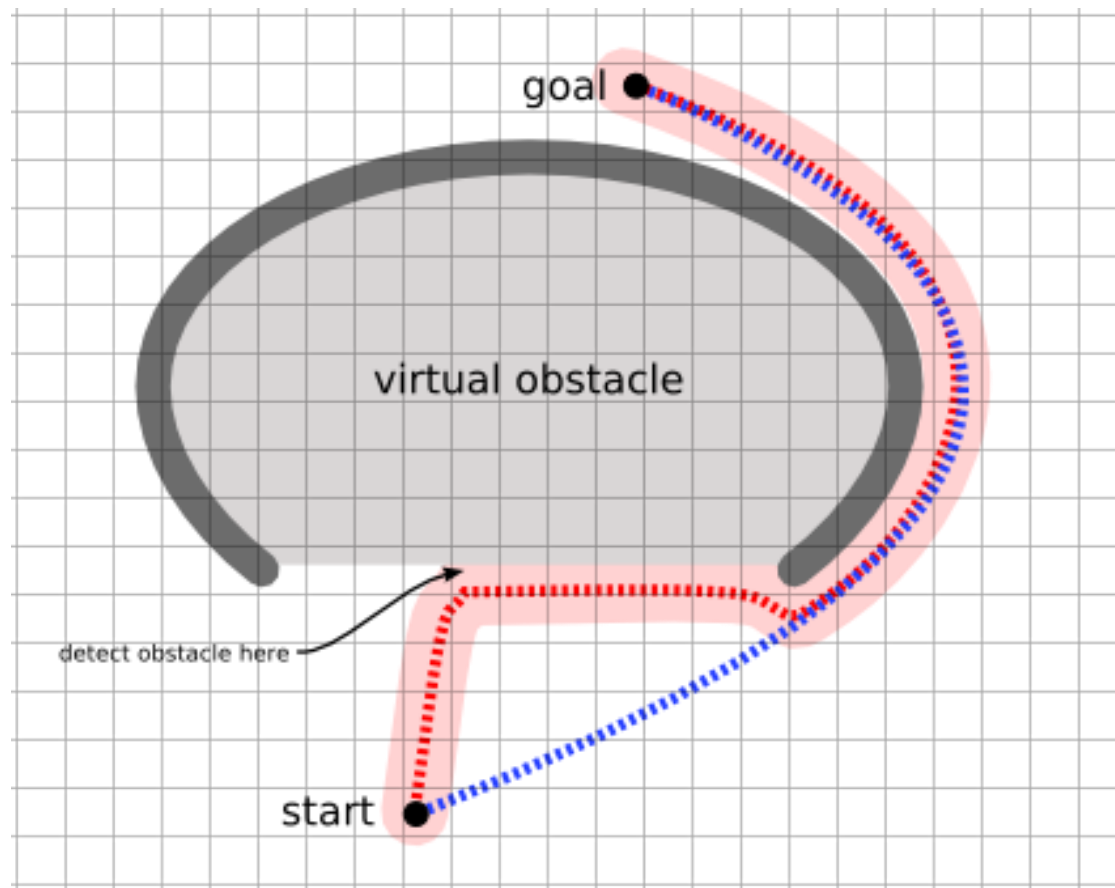
```
def es_estado_valido(estado):  
    # revisa si el estado es valido  
  
def es_estado_solucion(estado):  
    # revisa si es posible entregar una solución de manera trivial  
  
def actualizar_estado(estado, movida):  
    # actualiza el estado del mundo en base a la movida  
  
def deshacer_ultima_movida(estado, movida):  
    # deshace la última movida y vuelve al estado previo  
  
def resolver(estado):  
    if not es_estado_valido(estado):  
        return False  
    if es_estado_solucion(estado):  
        return True  
    else:  
        for movida in movidas:  
            actualizar_estado(estado, movida)  
            if resolver(estado):  
                return True  
            else:  
                deshacer_ultima_movida(estado, movida) # backtracking  
        return False
```

A\*: búsqueda de ruta óptima altamente eficiente



$$\text{costo\_total}(\text{nodo}) = \text{costo}(\text{nodo}) + \text{heurística}(\text{nodo})$$

A\*: búsqueda de ruta óptima altamente eficiente

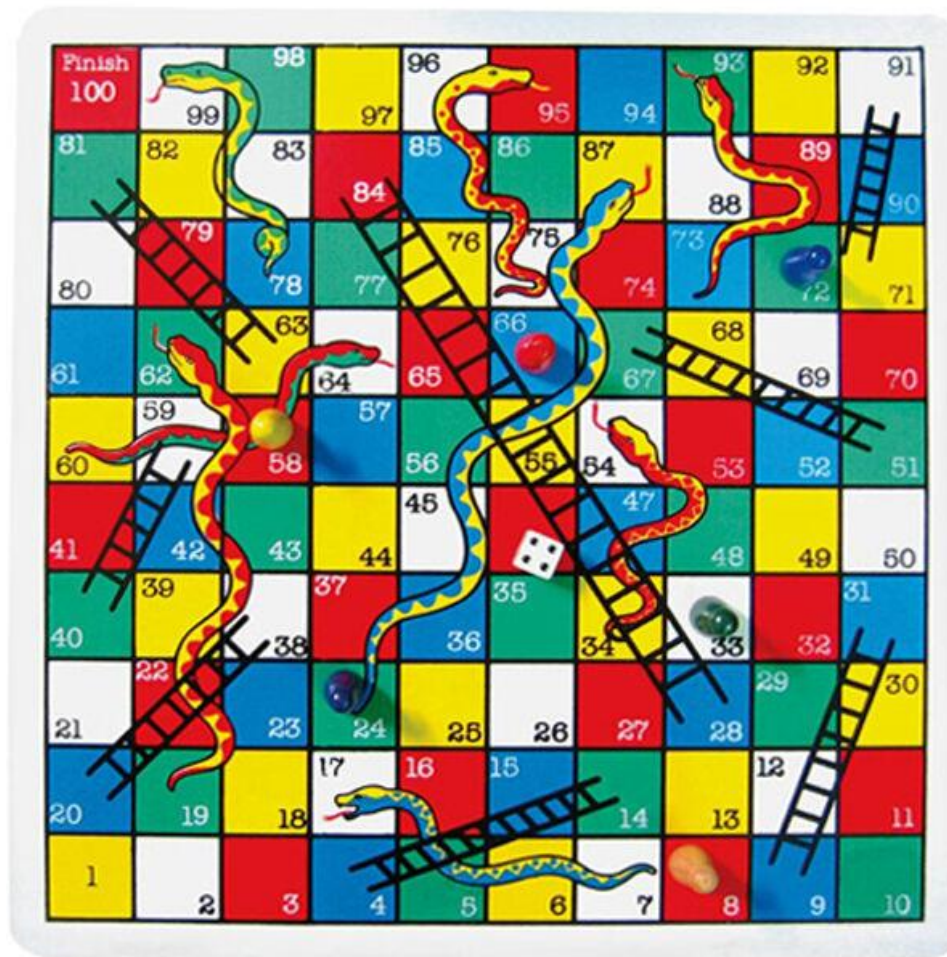


$$\text{costo\_total}(\text{nodo}) = \text{costo}(\text{nodo}) + \text{heurística}(\text{nodo})$$

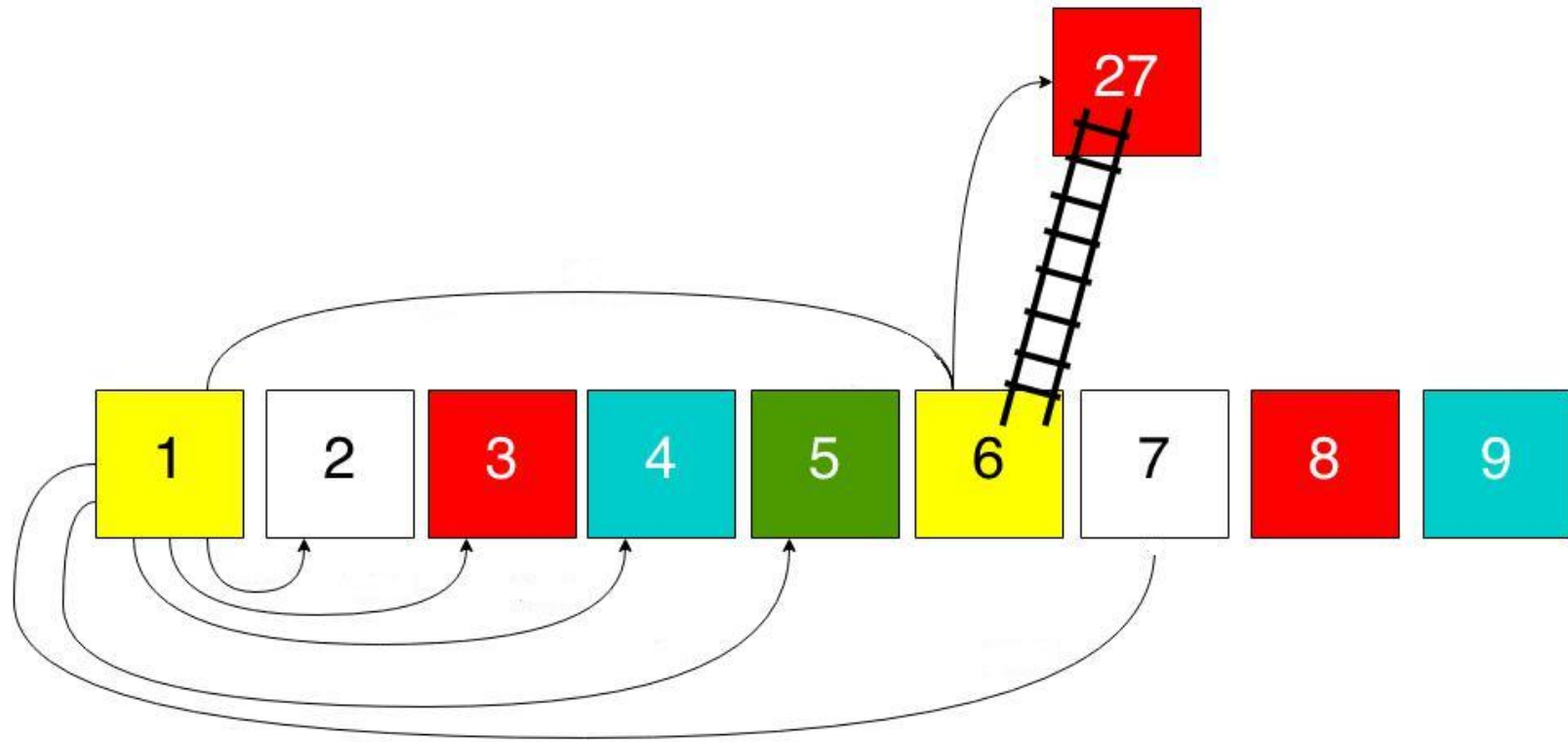
## A\*: búsqueda de ruta óptima altamente eficiente

- A diferencia de otros algoritmos (BFS, DFS, Dijkstra), A\* utiliza una heurística que le permite disminuir la cantidad de nodos que recorre.
- Esta heurística (sub)estima el costo entre un nodo y la solución.
- En cada paso, A\* selecciona el nodo para el cual la suma entre el costo acumulado y la heurística sea menor.

Cada vez que hay que buscar algo, y se puede usar un grafo como estructura, hay que pensar en A\*

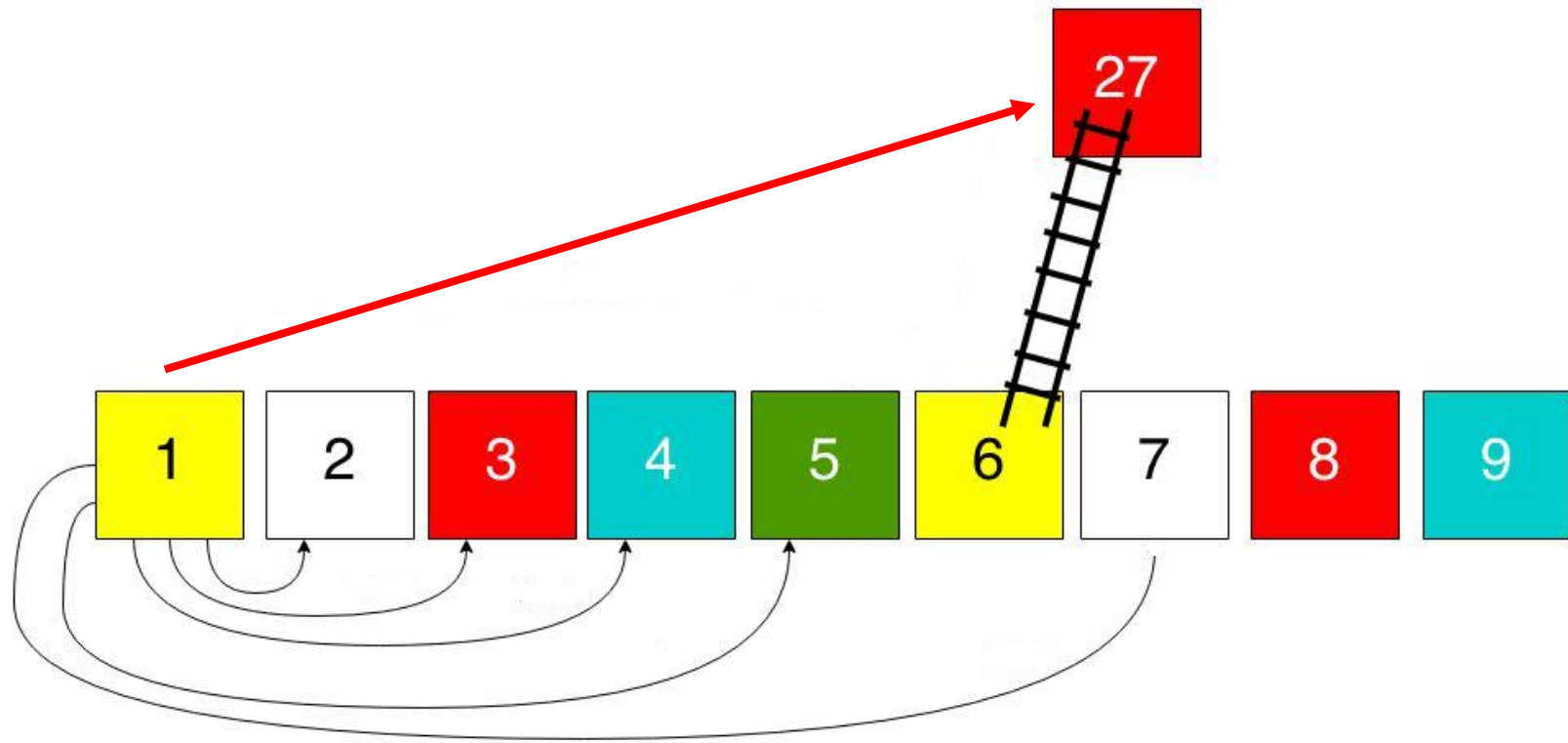


Cada vez que hay que buscar algo, y se puede usar un grafo como estructura, hay que pensar en A\*





Cada vez que hay que buscar algo, y se puede usar un grafo como estructura, hay que pensar en A\*



¿Heurística?

## Resolvamos un problema

Considere un conjunto de  $a$  alumnos ( $a \in \mathbb{N}^+$ ), donde cada uno tiene asociado una nota  $n$  ( $n \in \mathbb{N}^+$ ).

Escriba un programa que dado el conjunto de alumnos y sus notas, y un número  $k$  ( $k \in \mathbb{N}^+$ ), retorne  $k$  grupos de estudio disjuntos, tales que la suma de las notas parciales de los alumnos en cada grupo sea la misma.

¿Qué es lo primero que debemos hacer?

Siempre lo primero es aterrizar el problema

- Dadas **N** notas, se buscan **K** grupos que sumen lo mismo

Siempre lo primero es aterrizar el problema

- Dadas **5** notas [8, 6, 2, 8, 4], se buscan **2** grupos que sumen lo mismo.
  - La suma total es  $8 + 6 + 2 + 8 + 4 = 28$
  - Como son 2 grupos, cada uno debe sumar 14
  - Ahora buscamos números que sumen 14: (8+6) y (8+2+4)
  - Por lo que, dados los *input*  $N=[8, 6, 2, 8, 4]$  y  $K=2 \rightarrow [[6,8], [2,4,8]]$

**¿Qué más es posible de concluir?**

Luego, extraemos reglas más generales

si  $\text{sum}(N) \% K \neq 0 \rightarrow$  Nunca se puede

si  $|N| < K \rightarrow$  Nunca se puede

$\forall n \in N, n \leq \text{sum}(N)/K$

si  $n = \text{sum}(N)/K$ ,  $n$  forma un grupo solo

A continuación planteamos el problema como un algoritmo

1. Verificar las condiciones que ya definimos.
2. Si existen alumnos con nota igual a  $\frac{\text{sum}(N)}{K}$ :
  - a) Extraerlos de la lista
  - b) Disminuir el valor de K (grupos que faltan)
  - c) Agregar un grupo de un alumno
3. ¿Cómo iteramos?

(Casi) Finalmente, seguimos el algoritmo

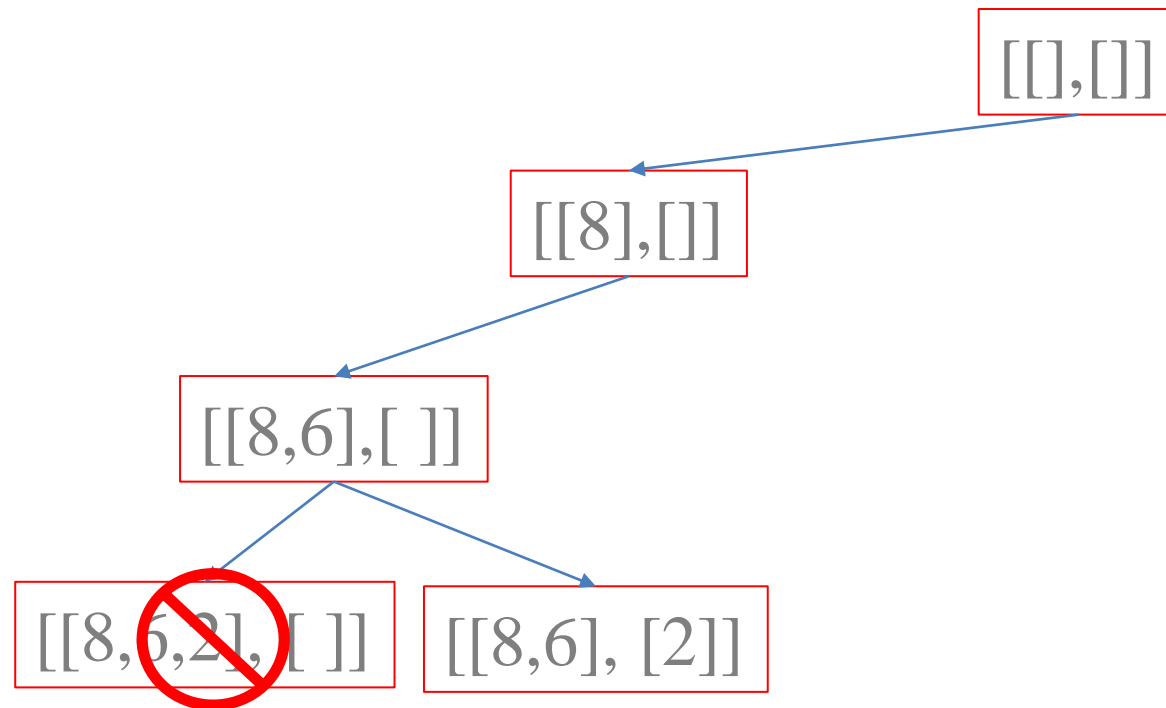
$n = [8, 6, 2, 8, 4]$		$\text{notas} = 5$
$k = 2$	$\rightarrow$	$\text{suma} = 28$
		$\text{suma\_por\_grupo} = 28/2 = 14$

- El número de notas (5) es mayor al valor de  $k$  (2)
- La suma total (28) es dividida perfectamente por  $k$  (2) (no deja resto)
- No hay elementos mayores que  $\text{suma\_por\_grupo}$  (14)



- Tenemos que formar dos grupos

[8,6,2,8,4]



¿Cuándo paro?

*¿Qué reglas podemos definir para terminar de iterar?*

- Si algún grupo  $>$  que el objetivo  $\rightarrow$  No hay solución en esa rama
- Si asigné todas las notas  $\rightarrow$  Encontramos solución

Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación



# IIC2115 – Programación como herramienta para la Ingeniería

## Capítulo 2 - Parte 2: Técnicas y Algoritmos

**Profesores:** Hans Löbel  
Francisco Garrido