

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



IIC2115 – Programación como herramienta para la Ingeniería

Consolidación Cap. 3: Técnicas de Programación

Profesores: Hans Löbel
Francisco Garrido

¿Por qué revisamos técnicas de programación?

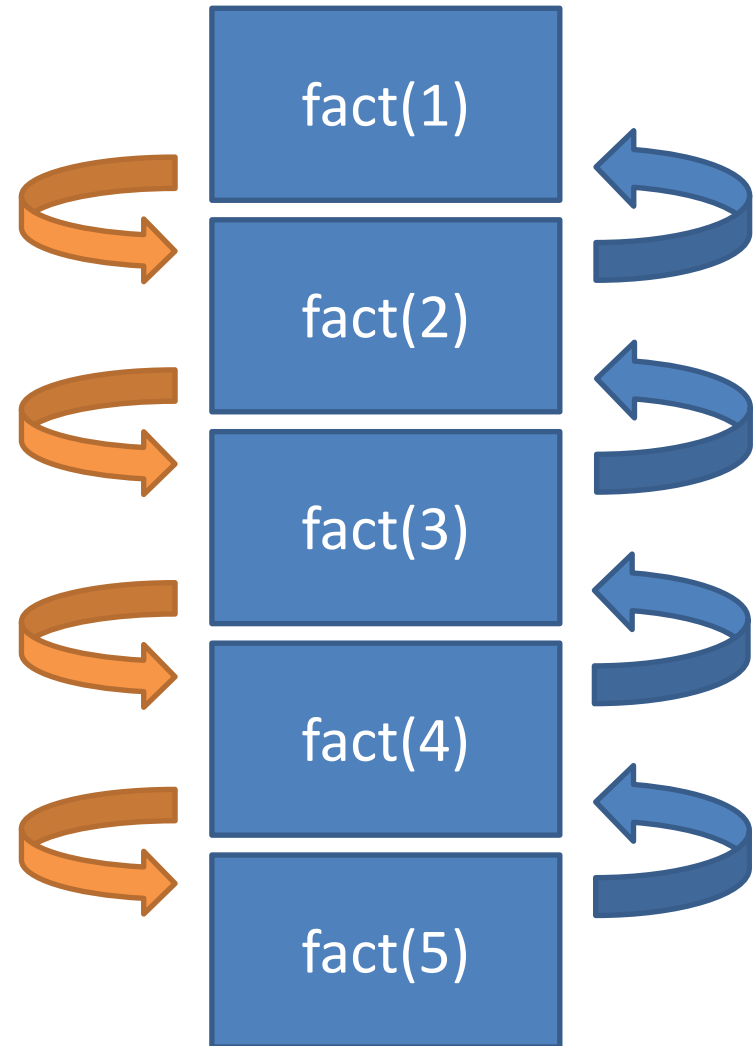
- Para resolver muchos problemas reales, no basta con usar estructuras de datos para obtener una solución eficiente.
- Generalmente, se requieren técnicas o algoritmos complejos, que utilicen estas estructuras de maneras no triviales.
- La clave para aprender a usarlas correctamente, es tratar de resolver 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 funcion_mas_humilde(parametros,n)  
    if instancia_es_trivial_por_tamaño:  
        return resultado  
    else:  
        resultado_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 funcion_mas_humilde(parametros, n)  
    if instancia_es_trivial_por_tamaño:  
        return resultado  
    else:  
        resultado_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 con recursión eficiente

```
def es_estado_valido(estado):  
    # revisa si el estado es valido  
  
def se_puede_resolver_trivialmente(estado):  
    # revisa si es posible entregar una solución de manera trivial  
  
def siguiente_estado(estado, movida):  
    # generar nuevo estado del mundo en base a la movida  
  
def resolver(estado):  
    if not es_estado_valido(estado):  
        return False  
    if se_puede_resolver_trivialmente(estado):  
        return True #o el estado, o una solución  
    else:  
        for movida in movidas:  
            if resolver(siguiente_estado(estado, movida)):  
                return True #o el estado, o una solución  
        return False
```

Dividir y conquistar: llamados progresivamente más simples

- Técnica naturalmente basada en recursión para hacer llamados recursivos a subproblemas incrementalmente más sencillos/pequeños.
- Esto continua hasta que el subproblema a resolver es trivial y puede ser resuelto sin división.
- Finalmente el algoritmo combina los resultados con el fin de generar la solución final. Este paso es muchas veces el más complicado.

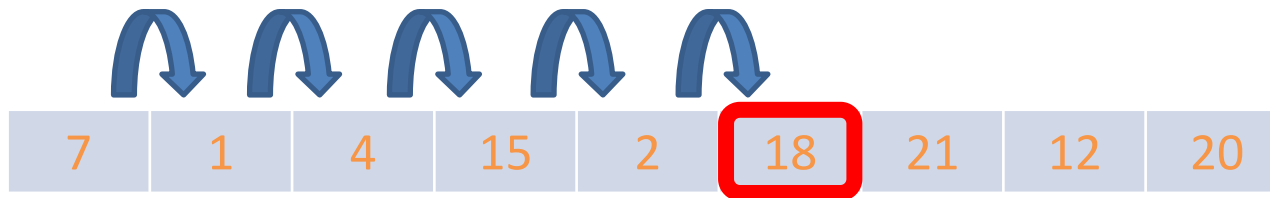
Dividir y conquistar: llamados progresivamente más simples

```
def combinar_soluciones_parciales(soluciones, n):  
    # tomo soluciones parciales y las combina  
  
def dividir_problema(parametros, n)  
    # divide el problema en subproblemas algo más fáciles/pequeños  
  
def resolver(parametros, n)  
    soluciones = []  
    if instancia_es_trivial_por_tamaño:  
        return resultado  
    else:  
        for sub_problema in dividir_problema(parametros, n):  
            soluciones.append(resolver(sub_problema.parametros, sub_problema.n))  
        return combinar_soluciones_parciales(soluciones, n)
```

Algoritmos de ordenamiento: útiles en múltiples dominios

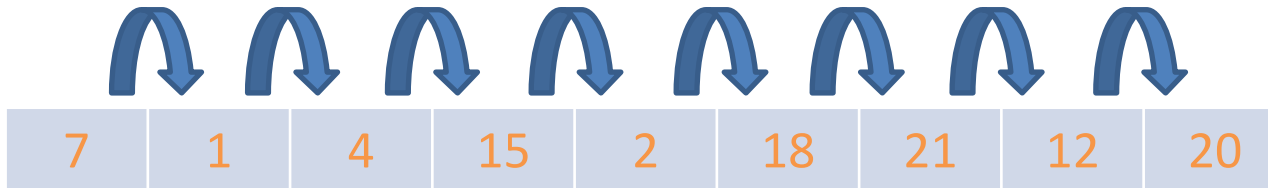
- Inicialmente pueden parecer de aplicación es limitada (nadie anda ordenando número enteros por la vida)...
-sin embargo, suelen usarse mucho como pasos intermedios para solucionar problemas.
- No porque un algoritmo tenga una menor complejidad que otro (notación \mathcal{O}), va a ser siempre mejor.
- Esto último depende, por ejemplo, del tamaño de la instancia y de cuál es el orden inicial de los elementos.

Acelerar las búsquedas es quizá el uso más común del ordenamiento



¿18?

Acelerar las búsquedas es quizá el uso más común del ordenamiento



¿3?

Acelerar las búsquedas es quizá el uso más común del ordenamiento

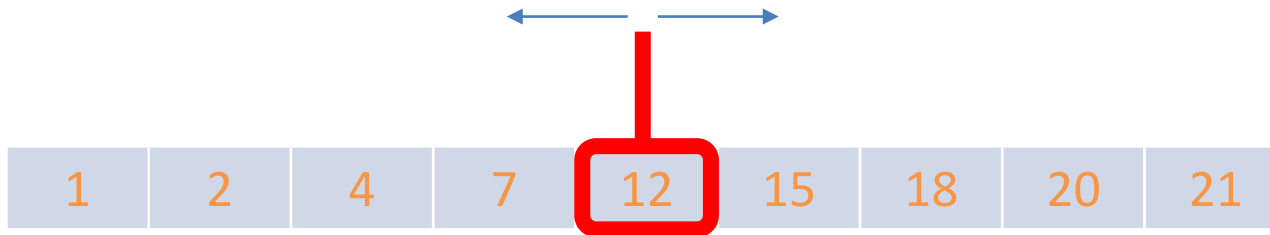
7	1	4	15	2	18	21	12	20
---	---	---	----	---	----	----	----	----



1	2	4	7	12	15	18	20	21
---	---	---	---	----	----	----	----	----

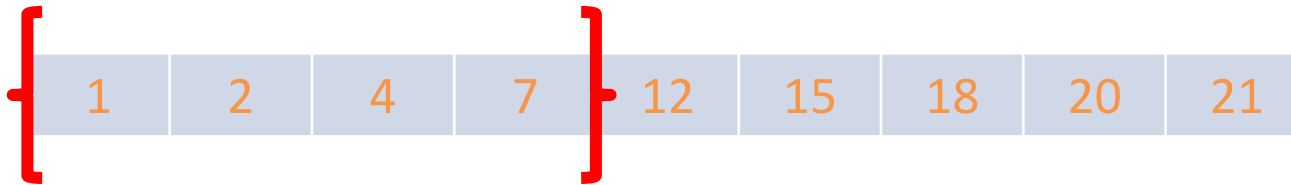
¿3?

Acelerar las búsquedas es quizá el uso más común del ordenamiento



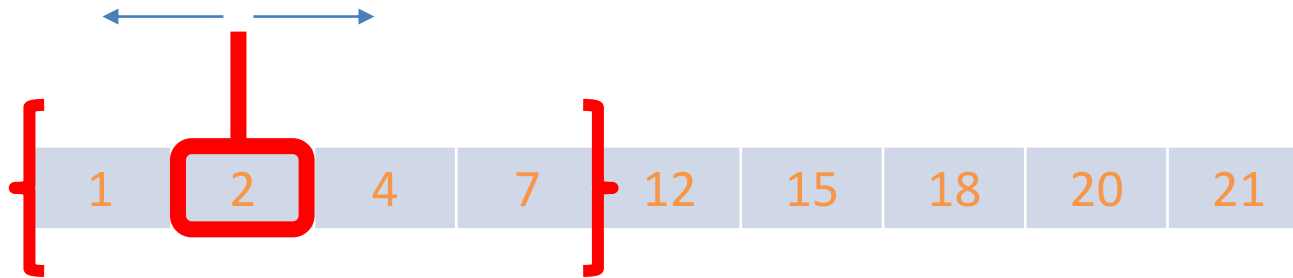
¿3?

Acelerar las búsquedas es quizá el uso más común del ordenamiento



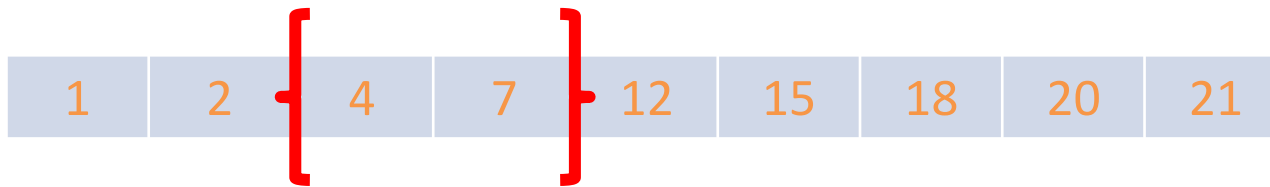
¿3?

Acelerar las búsquedas es quizá el uso más común del ordenamiento



¿3?

Acelerar las búsquedas es quizá el uso más común del ordenamiento

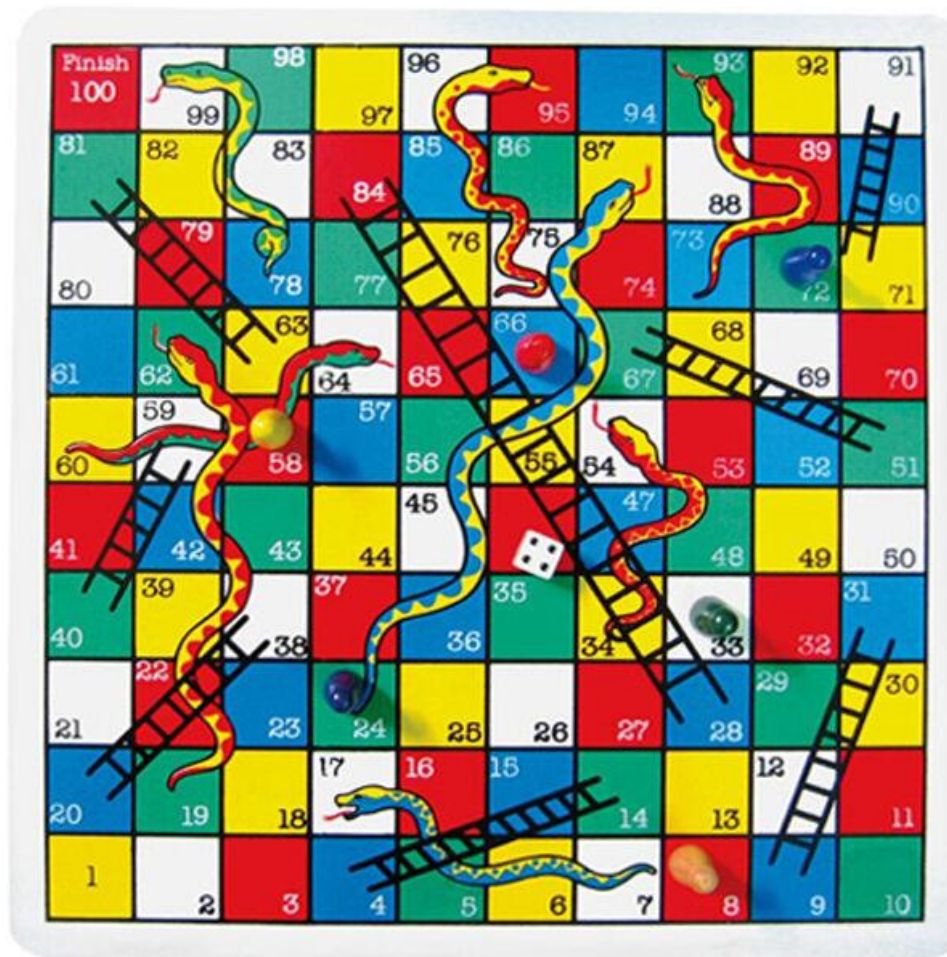


¿3?

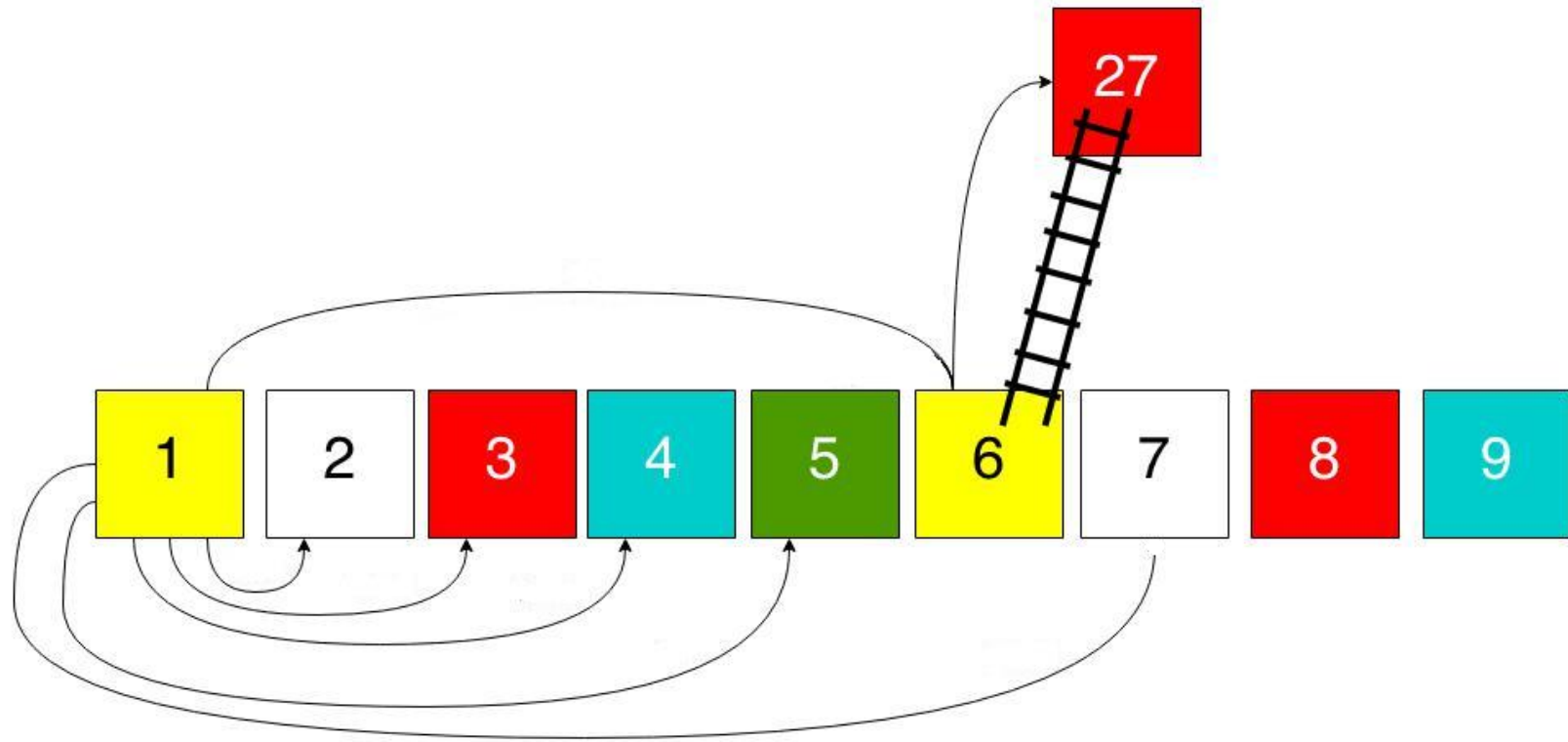
A*: búsqueda de ruta óptima altamente eficiente

- 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.

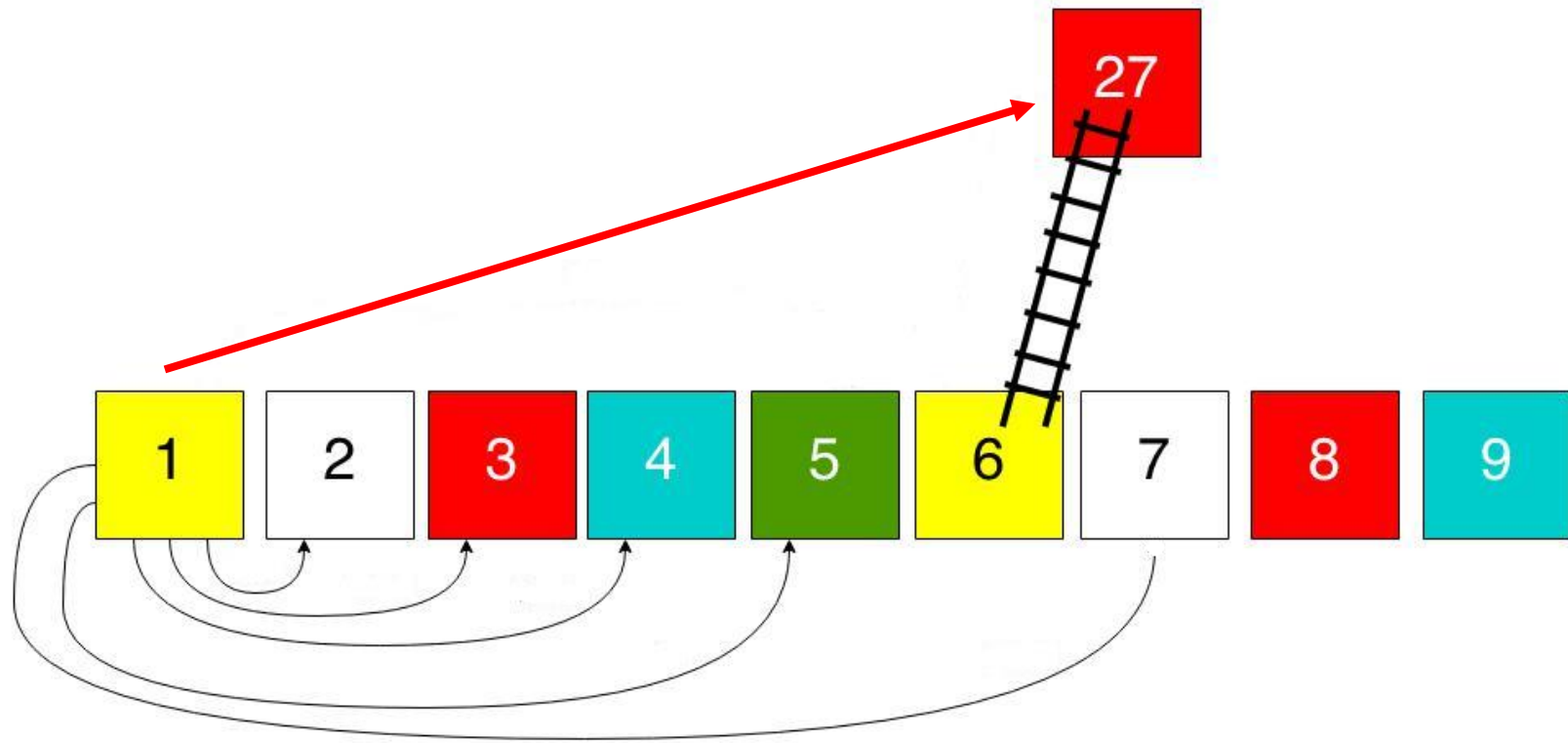
Cada vez que hay que buscar algo, y se puede usar un grafo como estructura, hay que pensar en A*



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?

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



IIC2115 – Programación como herramienta para la Ingeniería

Consolidación Cap. 3: Técnicas de Programación

Profesores: Hans Löbel
Francisco Garrido