udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA Y CIENCIAS

Tutoría para Examen de Grado

Sesión 2: Diseño y Análisis de Algoritmos

Docente: Mauricio Hidalgo Barrientos

Temario de la sesión

- Recordar algunos conceptos introductorios para el diseño y análisis de Algoritmos
- Recordar la técnica de “Divide and Conquer” y “Decrease and Conquer”
- Recordar el concepto de “Greedy” y revisar algunos algoritmos codiciosos
- Recordar otras técnicas algorítmicas como “Programación Dinámica” y “Backtracking recursivo”

Conceptos Introductorios

Definiciones iniciales

El *Diseño y Análisis de Algoritmos* es el manejo de técnicas y algoritmos que permiten analizar, modelar, evaluar y resolver distintos problemas no triviales que están al alcance de los computadores.

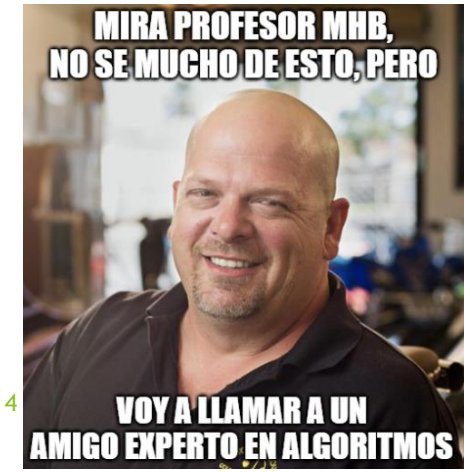
Tiempo de Ejecución (de un algoritmo) - Formalización

El tiempo de ejecución de un algoritmo es la cantidad de operaciones elementales que dicho algoritmo ejecuta, como función del tamaño de su entrada. Luego:

- Si n es el tamaño de la entrada de un algoritmo,
- entonces $T(n)$ denotará su tiempo de ejecución.

Algoritmo 1: BuscarMáximo(A[1..n])

```
1 m ← 1
2 for i ← 2 to n do
3 |   if A[m] < A[i] then
4 |   |   m ← i
5 |   end
6 end
7 return m
```



Definiciones iniciales

Si analizamos este Algoritmo, podemos notar que:

- Ejecuta $n - 1$ ciclos ***for***.
- Por cada uno de esos ciclos, se comparan dos posiciones del arreglo (la que contiene el máximo actual, y la que contiene el i-ésimo elemento del arreglo, siendo i el índice del ciclo).
- Si denotamos con c la cantidad de operaciones ejecutadas en cada ciclo, y
- teniendo en cuenta que además el algoritmo ejecuta dos asignaciones antes del ciclo for, y una operación ***return*** al finalizar el ciclo

Podemos decir que **SIEMPRE** el tiempo de ejecución del algoritmo es:

Algoritmo 1: BuscarMáximo(A[1..n])

```
1 m ← 1
2 for i ← 2 to n do
3 |   if A[m] < A[i] then
4 |   |   m ← i
5 |   end
6 end
7 return m
```

$$T(n) = c \cdot (n-1) + 3 = O(n)$$

Definiciones iniciales

IMPORTANTE

A menudo, en lugar de contar todas las operaciones elementales que ejecuta un algoritmo, tal como lo hicimos en el ejemplo, sólo contaremos cierto tipo de operaciones que, por alguna razón, sean más importantes que otras (ej.: Costo computacional de ejecutar la instrucción).

Entiendo el caso anterior, pero “Gran Barba”, al revisar el Cormen y recordando al profesor MHB, hay algoritmos que para algunos inputs pueden tener tiempos de ejecución diferente.

¡Efectivamente Rick!
Existen algoritmos que pueden tener distintos tiempos de ejecución para inputs diferentes

Análisis de mejor, peor y caso promedio

Al analizar un algoritmo, es posible que descubramos que el mismo no ejecuta la misma cantidad de operaciones para todas las entradas, sino que más bien el tiempo de ejecución depende de cuál caso se esté procesando. Por ejemplo, considere el siguiente algoritmo:

Algoritmo 2: BUSCARDESORDENADO(A[1..n], x)

```
1 for i ← 1 to n do
2 |   if x = A[i] then
3 |   |   return i
4 |   end
5 end
6 return n + 1                          ▷ Indica que x no está en A
```

¿Por qué no siempre se ejecuta la misma cantidad de ciclos FOR?

Análisis de mejor, peor y caso promedio

Tiempo de Ejecución de Mejor Caso

Sea P un problema abstracto cuyo conjunto de instancias es I y sea A un algoritmo que resuelve el problema P. Luego:

- Para una instancia $x \in I$, $sea\ T(x)$ el tiempo de ejecución del algoritmo A para la instancia x.
- Entonces, el tiempo de ejecución de mejor caso para A sobre una instancia de tamaño n se define como:

$$t_{\mathcal{A}}(n) \equiv \min_{\substack{x \in \mathcal{I} \\ |x|=n}} \{T_{\mathcal{A}}(x)\}$$

En la práctica, el análisis de mejor caso de un algoritmo consiste en calcular el tiempo de ejecución considerando la(s) instancia(s) que produce(n) la ejecución de la menor cantidad posible de operaciones.

Análisis de mejor, peor y caso promedio

Tiempo de Ejecución de Peor Caso

Sea P un problema abstracto cuyo conjunto de instancias es I y sea A un algoritmo que resuelve el problema P. Luego:

- Para una instancia $x \in I$, $sea\ T(x)$ el tiempo de ejecución del algoritmo A para la instancia x.
- Entonces, el tiempo de ejecución del peor caso para A sobre una instancia de tamaño n se define como:

$$T_A(n) \equiv \max_{\substack{x \in I \\ |x|=n}} \{T_A(x)\}$$

El análisis de peor caso de un algoritmo consiste en calcular el tiempo de ejecución considerando la(s) instancia(s) (o entrada(s)) que produce(n) la ejecución de la mayor cantidad posible de operaciones elementales.

Análisis de mejor, peor y caso promedio

Análisis de Caso Promedio

En general, el análisis de peor caso es adecuado y da suficiente información respecto al comportamiento de un algoritmo, pero:

- ¿Es un análisis justo?
- ¿El peor (o mejor) caso siempre ocurre?
- ¿Qué pasaría si todas las instancias tuvieran la misma probabilidad de ocurrir?

Sea P un problema abstracto cuyo conjunto de instancias es I y sea A un algoritmo que resuelve el problema P. Luego:

- Para una instancia $x \in I$, $sea\ T(x)$ el tiempo de ejecución del algoritmo A para la instancia x.
- Entonces, el tiempo de ejecución del caso promedio para A sobre una instancia de tamaño n se define como:

$$\overline{T}_{\mathcal{A}}(n) \equiv \frac{1}{|\mathcal{I}_{\mathcal{P},n}|} \sum_{x \in \mathcal{I}_{\mathcal{P},n}} T_{\mathcal{A}}(x)$$

Análisis de mejor, peor y caso promedio

Volvamos al algoritmo anterior:

Algoritmo 2: BuscarDesordenado(A[1..n], x)

```
1 for i ← 1 to n do
2 |   if x = A[i] then
3 |   |   return i
4 |   end
5 end
6 return n + 1                          ▷ Indica que x no está en A
```

- Se puede apreciar que buscar el elemento que está en la posición i del arreglo cuesta i comparaciones y que,
- Tenemos n casos posibles de búsqueda exitosa y todos tienen "la misma probabilidad".

Entonces

El costo promedio de búsqueda exitosa se puede calcular mediante:

$$\overline{T}(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Regla del Límite

Dadas dos funciones $f(n)$ y $g(n)$ asociadas a algoritmos diferentes que abordan un mismo problema, es normal que queramos saber si una de dichas funciones crece más rápido que la otra. Una manera de averiguar esto es mediante la Regla del Límite.

Si consideramos la siguiente expresión:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = k$$

Podemos decir que:

- Si $k \in \mathbb{R}^+$ es una constante (distinta de cero), significa que ambas funciones son equivalentes en complejidad.
- Si $k = \infty$ significa que $f(n)$ tiene un orden de crecimiento mayor que $g(n)$
- Si $k = 0$ significa que $g(n)$ tiene un orden de crecimiento mayor que $f(n)$

Ecuación de Recurrencia

Una ecuación de recurrencia es una relación que define una secuencia recursiva; cada término de la secuencia es definido como una función de términos anteriores. Por ejemplo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

Ecuaciones de Primer Orden

Son aquellas en que el valor de T(n) depende, principalmente, del valor de T(n-1) – sin que intervengan otros valores de T(n-x). Luego, el caso general para ecuaciones de primer orden está dado por:

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

Siendo a > 0 y b_n una función lineal de n.

$$\Rightarrow T(n) = a^n \cdot T(0) + \sum_{1 \leq j \leq n}^{1 \leq j \leq n} b_j \cdot a^{n-j}$$

Ecuación de Recurrencia

Ecuaciones lineales con coeficientes constantes

Son aquellas en que el valor de T(n) depende, de varios valores de T(n-x). Además, estas ecuaciones son de la forma:

$$f_n = Af_{n-1} + Bf_{n-2} + Cf_{n-3} + \cdots$$

Donde A, B y C son constantes. Además, se sabe que estas ecuaciones **siempre** tienen soluciones de la forma:

$$f_n = \lambda^n$$

Para la sucesión de Fibonacci tendríamos que:

$$\lambda^n = \lambda^{n-1} + \lambda^{n-2}$$
$$\lambda^2 - \lambda - 1 = 0$$
$$\Rightarrow \lambda = \frac{1 \pm \sqrt{5}}{2}$$

$$\begin{cases} \phi = \dfrac{1+\sqrt{5}}{2} \approx 1.618 \\ \hat{\phi} = \dfrac{1-\sqrt{5}}{2} \approx -0.618 \end{cases}$$

$$f_n = A \cdot \phi^n + B \cdot \hat{\phi}^n$$

Ecuación de Recurrencia

Por lo que si consideramos los casos base (n = 0 y n = 1), tenemos que:

$$f_0 = 0 \Rightarrow B = -A \Rightarrow f_n = A \cdot \left(\phi^n - \hat{\phi}^n\right)$$

$$f_1 = 1 \Rightarrow A \cdot \left(\phi - \hat{\phi}\right) = 1 \Rightarrow A = \frac{1}{\phi - \hat{\phi}} = \frac{1}{\sqrt{5}}$$

$$f_n = \frac{1}{\sqrt{5}} \cdot \left(\phi^n - \hat{\phi}^n\right)$$

Teorema Maestro

Es un “grupo de fórmulas” que permiten encontrar la complejidad de algoritmos recursivos del tipo “Divide and Conquer” que cumplen con un conjunto de reglas específicas en tres casos fundamentales para su “función de trabajo”. La forma general de las ecuaciones que se pueden resolver es:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Donde $a \geq 1$, $n > b > 1$ y $f(n)$ es una función polinómica continua positiva

Casos de Resolución

- Considerando que $f(n) \in O(n^c) \quad \forall \quad c < \log_b a$ se tendrá que $T(n) = O(n^{\log_b a})$
- Considerando que $f(n) \in \Omega(n^c) \quad \forall \quad c > \log_b a \ \wedge \ af\left(\frac{n}{b}\right) \leq kf(n); k < 1 \wedge n \to \infty$ se tendrá que $T(n) = \Theta\,(n^c)$
- Considerando que $f(n) \in \Theta\left(n^c log^k n\right) \quad \forall \quad c = \log_b a$ se tendrá que $T(n) = \Theta\,(n^c log^{k+1} n)$

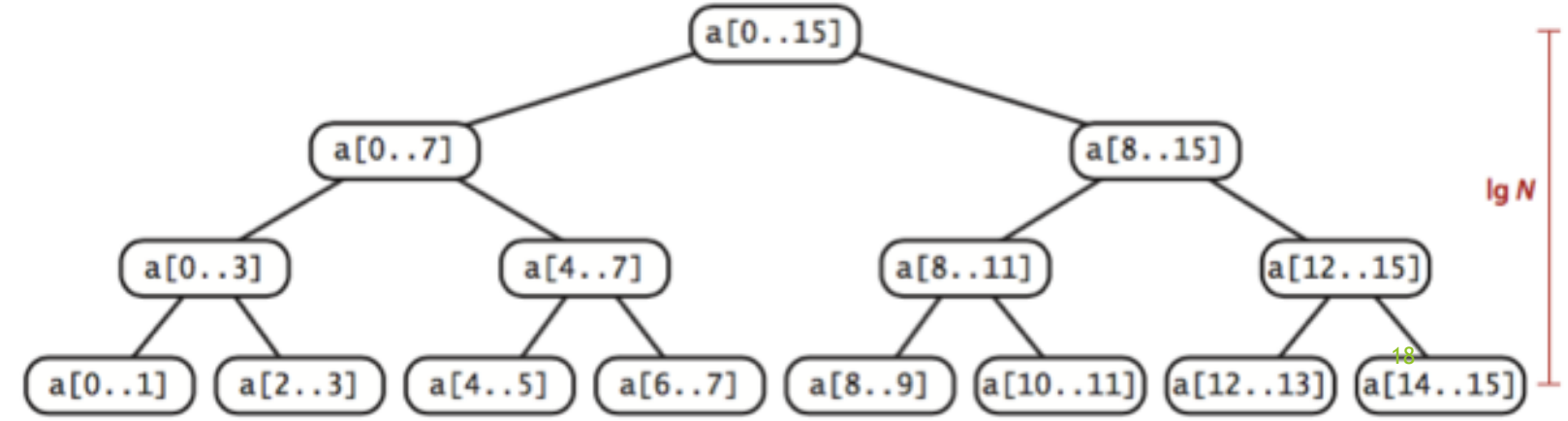
Divide and Conquer
Decrease and Conquer

Divide and Conquer

Es uno de los paradigmas de resolución de problemas más importantes de las Ciencias de la Computación donde el método aplicado consiste en solucionar un problema dividiendo el mismo, de manera recursiva, en dos o más subproblemas similares.

¿Cómo funciona?

- Dividir en subproblemas (caso recursivo)
- Resolver los subproblemas al caso más pequeño (caso base recursión)
- Al combinar las soluciones (return)



Divide and Conquer

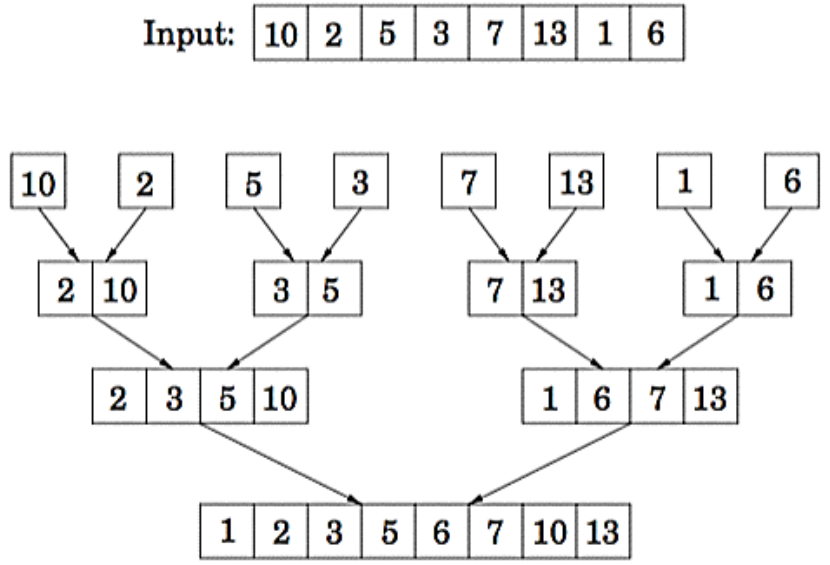
Ejemplo: Merge Sort

- **Divide** → El problema (arreglo) se separa en dos partes. Tiempo constante D(n) = Θ(1).
- **Conquer** → Resolver recursivamente en dos subproblemas de tamaño n/2 cada uno. → 2T(n/2).
- **Merge** → Combinar las dos mitades ordenadas, en un tercer arreglo ordenado. Función de trabajo con n combinaciones en promedio → C(n) = Θ(n)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

$$\boldsymbol{T(n) = cn(1 + \log_2 n) \rightarrow \Theta(n \log_2 n)}$$



Decrease and Conquer

Aunque algunos consideran que es un caso particular de Divide and Conquer, es un enfoque distinto ya que:

- Se disminuye el tamaño del problema en cada paso
- Resuelve el problema disminuido
- No hay combinación (como en Divide and Conquer)

Idea de la técnica

Explotar la relación entre la solución al problema más pequeño y el problema original: La solución del problema más pequeño es usada para construir la solución al problema original.

Enfoques Decrease and Conquer

Top down

Se comienza con el problema original, el que se va haciendo más pequeño a cada paso. Generalmente lleva a una solución recursiva, que en lo normal conviene implementar iterativamente (que suele ser más eficiente en la práctica).

Bottom up

Se resuelve el problema comenzando por soluciones muy pequeñas a partes del problema original. En cada paso se incremente el tamaño de la solución. Es un enfoque incremental, en donde la solución se construye paso a paso.

Formas de decrecer

Decrecer por una constante

En cada paso, el algoritmo hace decrecer el tamaño del problema en un valor constante. Por ejemplo, calcular a^n cuando n es un entero no negativo, usando el siguiente algoritmo:

$$\boldsymbol{a^n = a^{n-1} \cdot a}$$

Decrecer por factor constante

En cada paso, el algoritmo hace decrecer el tamaño del problema en un factor constante, por ejemplo, a la mitad del tamaño del problema original, o a un tercio, etc. Por ejemplo, calcular a^n cuando n es un entero no negativo, usando el siguiente algoritmo:

$$\boldsymbol{a^n = (a^{n/2})^2}$$

Aquí el tamaño del problema original disminuye a la mitad en cada paso

Formas de decrecer

Ejemplo:

El Algoritmo para $a^n = (a^{n/2})^2$ puede ser resuelto aplicando recursivamente las fórmulas:

$$\boldsymbol{Para\ n\ par\ \rightarrow\ a^n = (a^{n/2})^2\ si\ n > 0\ y\ a^0 = 1}$$
$$\boldsymbol{Para\ n\ impar\ \rightarrow\ a^n = (a^{(n-1)/2})^2 a}$$

Codifíquelo, compruebe y analice su complejidad utilizando el Teorema Maestro

Decrecer por factor variable

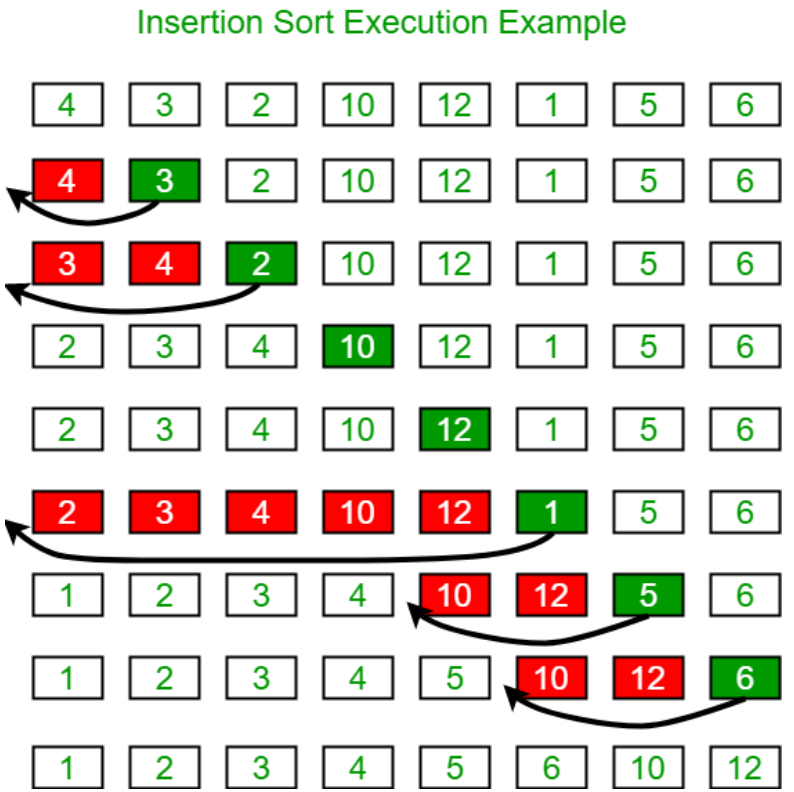
En cada paso, el algoritmo hace decrecer el tamaño del problema en un factor variable, es decir, no necesariamente decrece por el mismo factor en cada paso. Por ejemplo, en un paso se podría decrecer el problema a la mitad, mientras que en otro paso posterior se podría decrecer un tercio del tamaño actual.

Formas de decrecer

Ejemplo: Insertion Sort

Decrecer y conquistar de forma bottom up ya que se ordena incrementalmente en cada paso.

```
1 i ← 2
2 while i ⩽ n do
3 |   j ← i
4 |   while j > 1 and A[j − 1] > A[j] do
5 |   |   SWAP(A[j], A[j − 1])
6 |   |   j ← j − 1
7 |   end
8 |   i ← i + 1
9 end
```



Formas de decrecer

Ejemplo: Búsqueda Binaria

Decrecer y conquistar de forma top down ya que se genera, cada vez, un arreglo “más pequeño” (no se busca en el sub arreglo sino hasta el “caso base”).

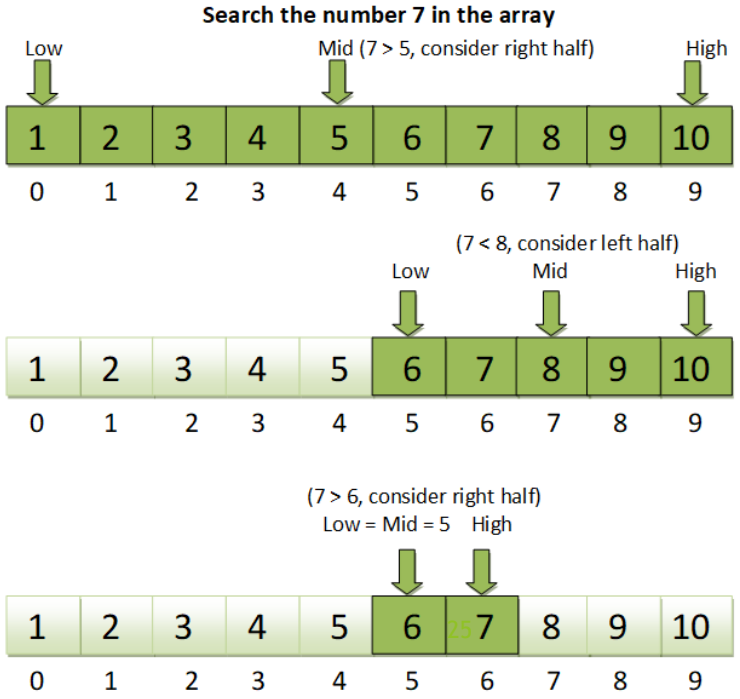
```
# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:
        mid = l + (r - l) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can
        # only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)
    else:
        # Element is not present in the array
        return -1
```



Algoritmos Codiciosos (Greedy)

Algoritmos codiosos

Es aquel algoritmo que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima.

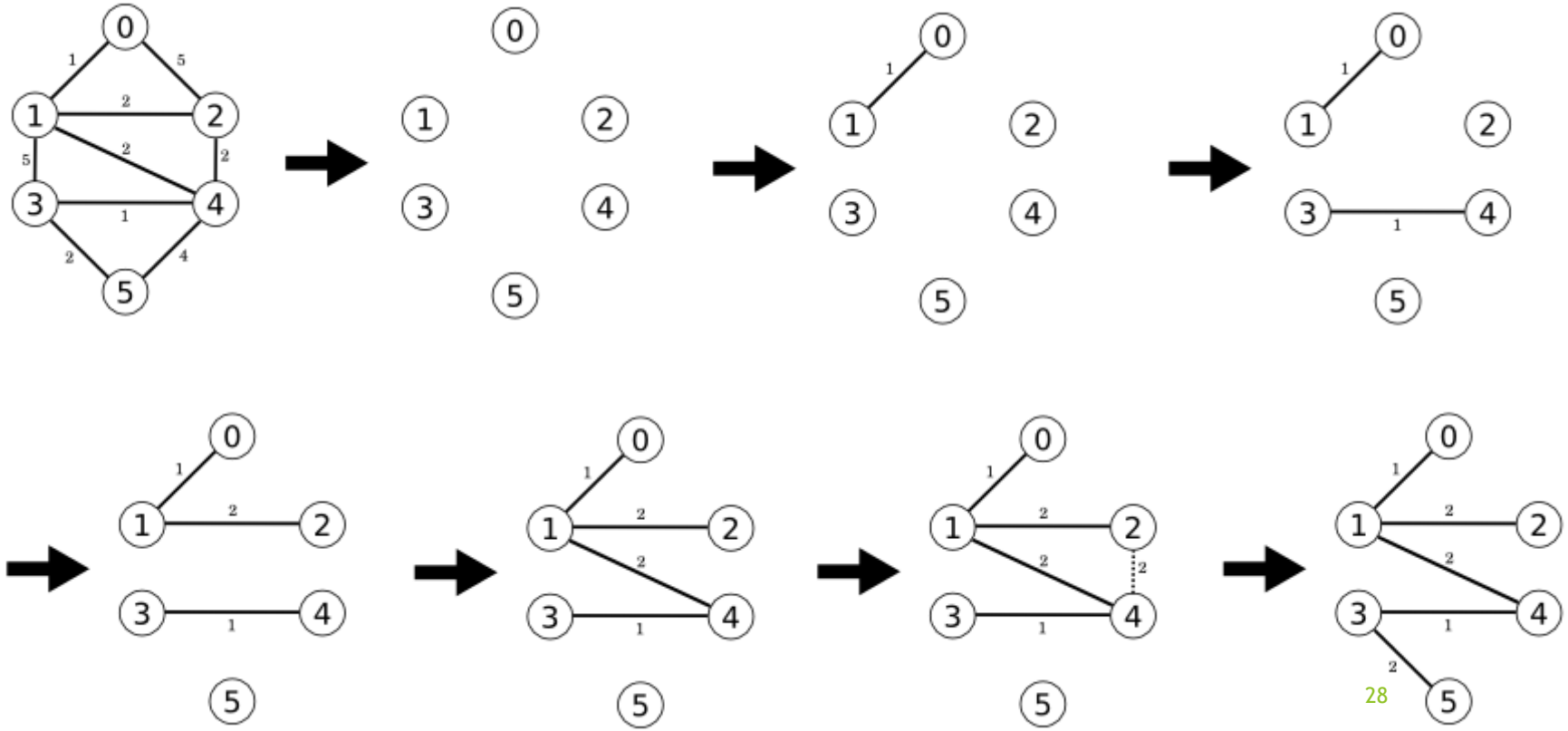
Ejemplo clásico

“Eres el capitán de un barco pirata y tu tripulación debe votar cómo se dividirá un botín de monedas de oro. Si menos de la mitad de los piratas está de acuerdo contigo, debes morir. ¿Cómo recomendarías dividir el oro, de forma que mantengas la mayor parte del tesoro que te sea posible, pero sobrevivas?”

¡El objetivo es maximizar tu ganancia!

Algoritmo de Kruskal

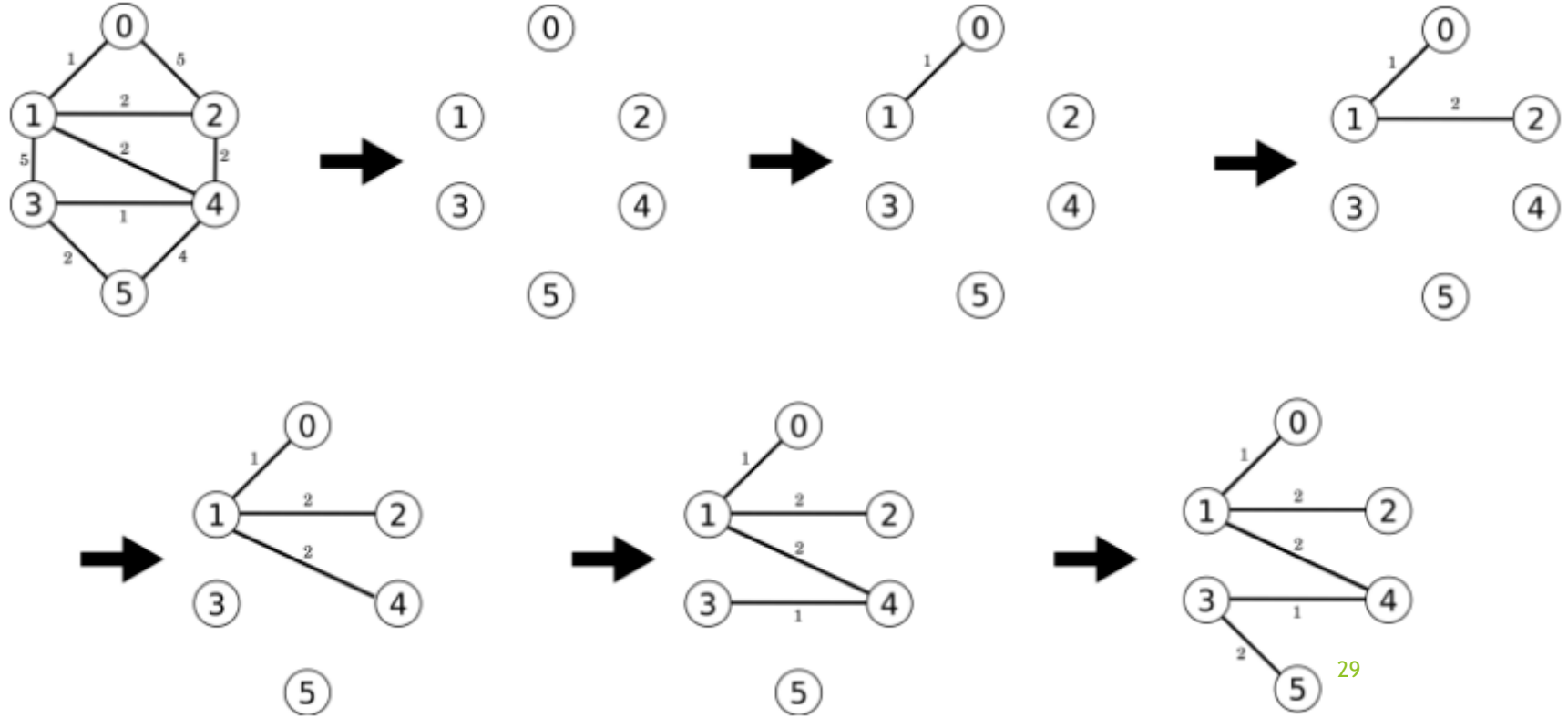
Es un algoritmo para encontrar el árbol de cobertura mínimo en un grafo conexo y ponderado donde dicho árbol incluye todos los vértices y el peso total de las aristas es el mínimo posible. Esto lo hace al optimizar las decisiones de corto plazo basado en el peso de las aristas.



Fuente ejemplo: *http://arantxa.ii.uam.es/~aa/practicas/P1/enunciadop1.html*

Algoritmo de Prim

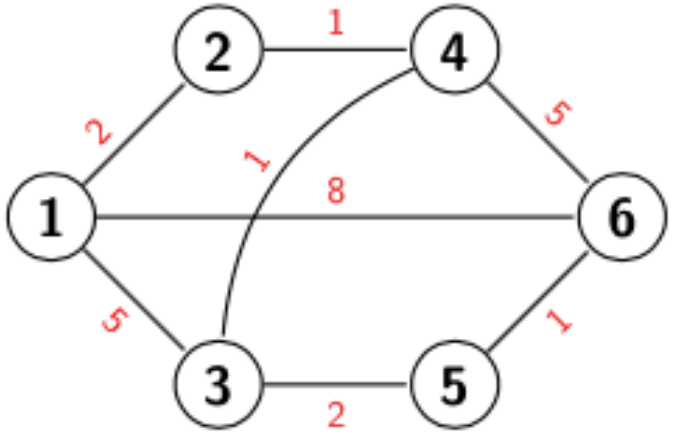
Es un algoritmo para encontrar el árbol de cobertura mínimo en un grafo conexo y ponderado donde dicho árbol incluye todos los vértices y el peso total de las aristas es el mínimo posible. Esto lo hace al optimizar las decisiones basado en el vértice “pivote” y el peso de la arista asociada al mismo y/o si es que existe un “camino mínimo” de un vértice A a un vértice B pasando con un vértice C que el directo de A hacia B.

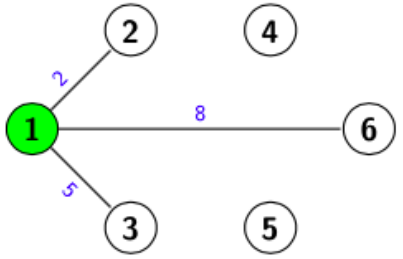


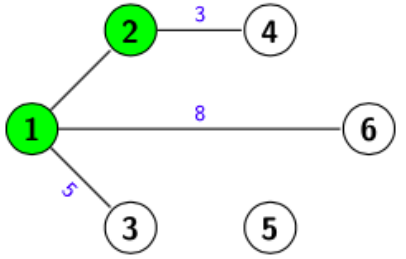
Fuente ejemplo: *http://arantxa.ii.uam.es/~aa/practicas/P1/enunciadop1.html*

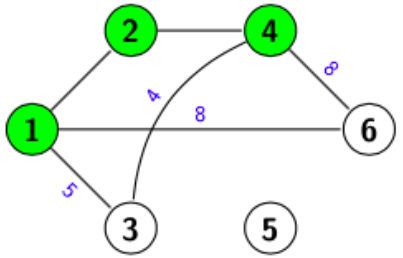
Algoritmo de Dijkstra

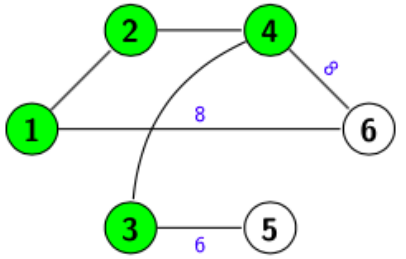
Es un algoritmo para encontrar la ruta más corta – de menor peso – en un grafo desde un vértice A hasta un vértice B pasando por “n” vértices intermedios, donde n > 0, y cada arista tiene un “peso” asociado p > 0. Las aristas pueden ser “dirigidas” (digrafo).

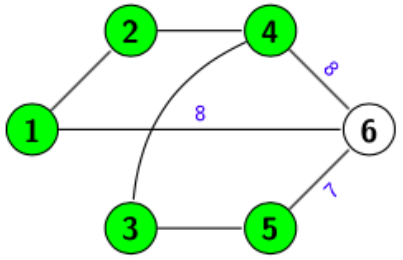


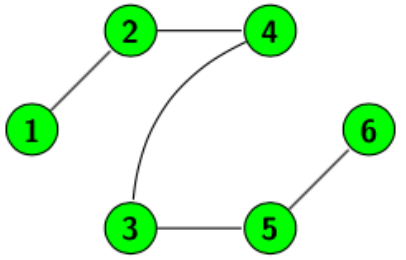








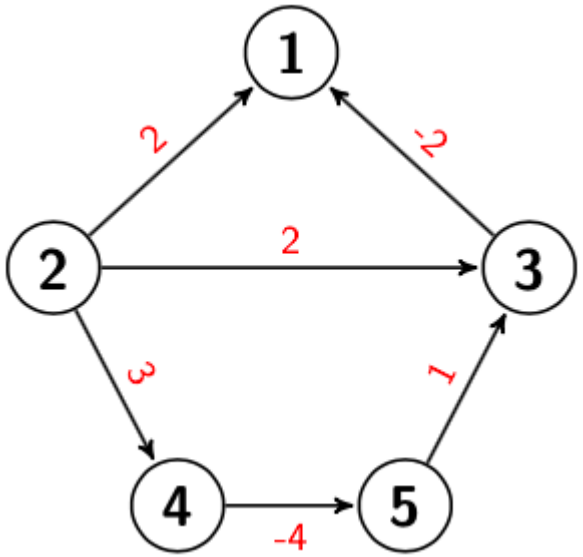




Fuente ejemplo: *https://aprende.olimpiada-informatica.org/algoritmia-dijkstra-bellman-ford-floyd-warshall*

Algoritmo de Bellman-Ford

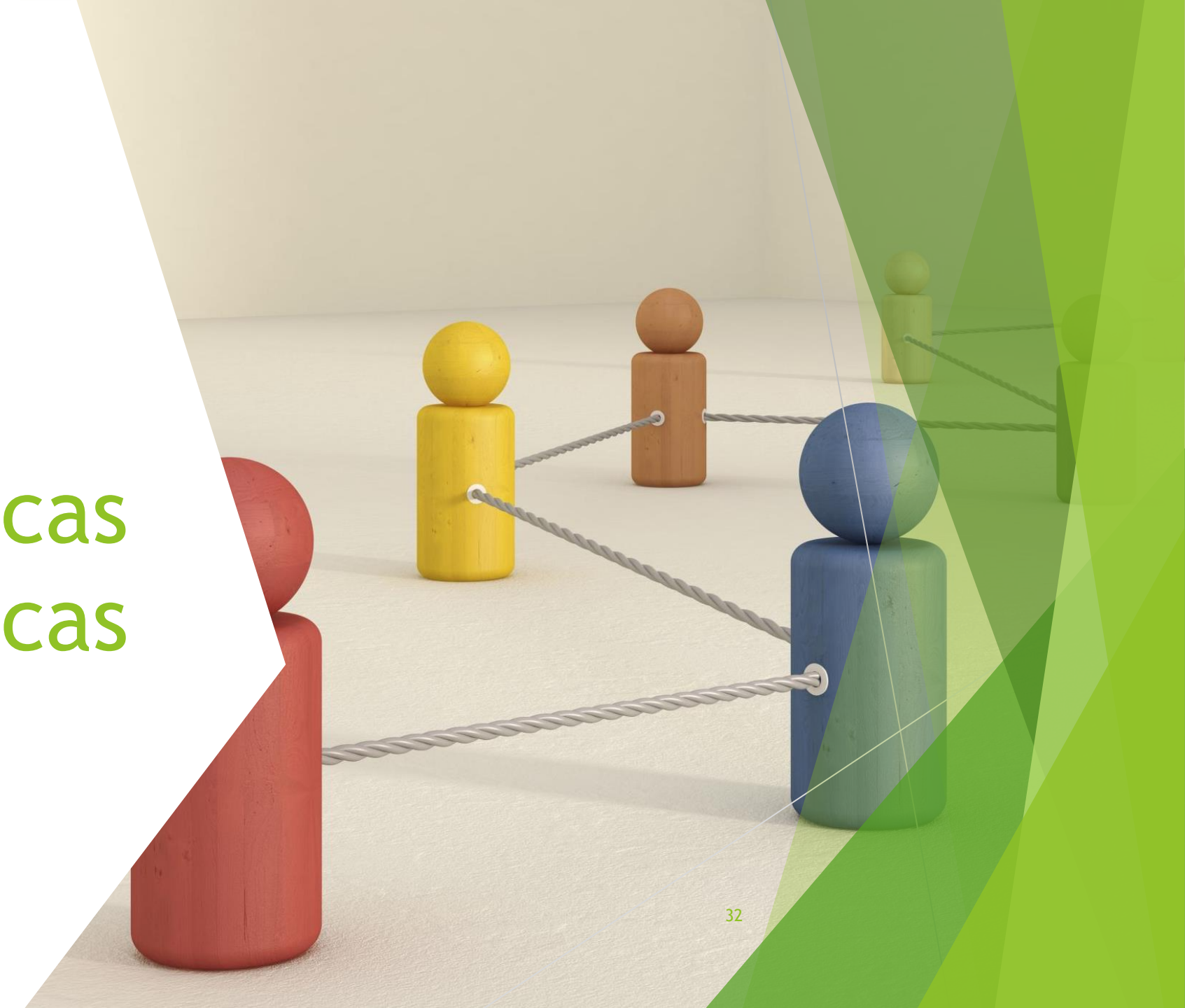
Es un algoritmo para encontrar la ruta más corta – de menor peso – en un grafo dirigido desde un vértice A hasta un vértice B pasando por “n” vértices intermedios, donde n > 0, y cada arista tiene un “peso” asociado p.



iteraciones	arco	distancia1	distancia2	distancia3	distancia4	distancia5
0		inf	0	inf	inf	inf
1	2 --> 1	2	0	inf	inf	inf
	2 --> 3	2	0	2	inf	inf
	2 --> 4	2	0	2	3	inf
	3 --> 1	0	0	2	3	inf
	5 --> 3	0	0	2	3	inf
	4 --> 5	0	0	2	3	-1
2	2 --> 1	0	0	2	3	-1
	2 --> 3	0	0	2	3	-1
	2 --> 4	0	0	2	3	-1
	3 --> 1	0	0	2	3	-1
	5 --> 3	0	0	0	3	-1
	4 --> 5	0	0	0	3	-1
3	2 --> 1	0	0	0	3	-1
	2 --> 3	0	0	0	3	-1
	2 --> 4	0	0	0	3	-1
	3 --> 1	-2	0	0	3	-1
	5 --> 3	-2	0	0	3	-1
	4 --> 5	-2	0	0	3	-1
4	2 --> 1	-2	0	0	3	-1
	2 --> 3	-2	0	0	3	-1
	2 --> 4	-2	0	0	3	-1
	3 --> 1	-2	0	0	3	-1
	5 --> 3	-2	0	0	3	-1
	4 --> 5	-2	0	0	3	-1

Fuente ejemplo: *https://aprende.olimpiada-informatica.org/algoritmia-dijkstra-bellman-ford-floyd-warshall*

Otras técnicas Algorítmicas



Programación Dinámica

Es una técnica que consiste en resolver los subproblemas una sola vez, guardando sus soluciones (por ejemplo, en un vector) para una futura utilización.

Algunos objetivos de la Programación Dinámica

- Revisar el desarrollo de la solución para conocer los puntos intermedios.
- Determinar el máximo rendimiento posible de obtener entre una combinación de opciones (funciones) para un valor total X.
- Determinar el máximo de un ítem A basado en disminuir el ítem B.

```
def FibDinamico(N):
    if(N<=1):
        return 1
    else:
        F1 = 1
        F2 = 1
        for i in range(2,N):
            suma = F1 + F2
            F2 = F1
            F1 = suma

        return suma
```

Programación Dinámica

Ejemplo: Coeficientes Binomiales

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{si } 0 < k < n, \qquad \binom{n}{0} = \binom{n}{n} = 1$$

```
int CoefBinRec(int n, int k)
{
    if(k==0 or n==k)
    {
        return 1;
    }
    else
    {
        return CoefBinRec(n-1,k-1)+CoefBinRec(n-1,k);
    }
}
```

¿Cuál es el tiempo de ejecución de este algoritmo?

¿Existe una opción mejor?

Programación Dinámica

Si consideramos el triángulo de Pascal, podemos definir que el término **C(n,k)** está dado por una relación directa entre los valores asociados al **n-1** y los **k** y **k-1**, de la forma:

```
int CoefBinDin(int n, int k)
{
    if(k==0 or n==k) {  return 1; }
    else
    {
        //Generamos una matriz "llena de 1" en su parte inferior izquierda - Conservaremos solo las diagonales y la columna 0
        int pascal[n+1][n+1];
        for(int i=0;i<=n;i++)
        {
            for(int j=0;j<=i;j++)
            {
                pascal[i][j]=1;
            }
        }
        //Nos preocuparemos del triángulo entre la diagonal y la columna 0
        for(int i=1;i<=n;i++)
        {
            for(int j=2;j<=n;j++)
            {
                if(i<j)
                {
                    //Está invertido dado que vamos por la parte inferior izquierda de la matriz pascal
                    pascal[j][i]=pascal[j-1][i-1]+pascal[j-1][i];
                }
            }
        }
        return pascal[n][k];
    }
}
```

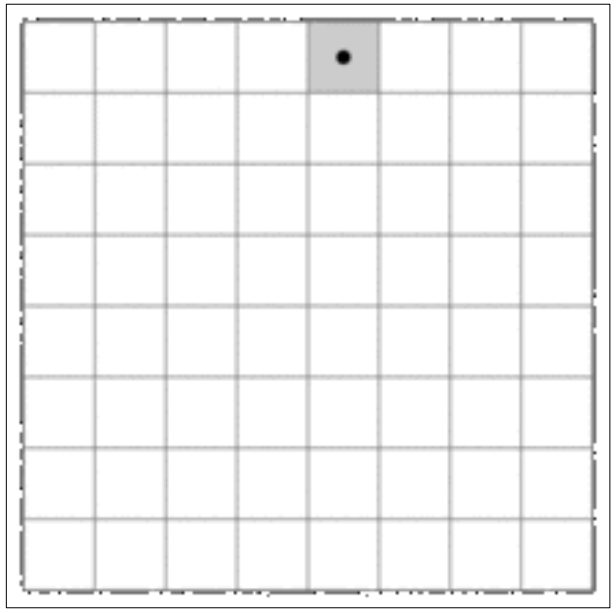
	0	1	2	3	…	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
…	…	…	…	…	…		
…	…	…	…	…	…	…	
n-1						C(n-1,k-1) +	C(n-1,k)
n							C(n,k)

Backtracking Recursivo

El backtracking es un enfoque recursivo que explora todas las opciones posibles para encontrar una solución a un problema. Funciona dividiendo el problema en subproblemas más pequeños, realizando pruebas y retrocediendo (backtracking) cuando se encuentra una solución inválida.

IMPORTANTE

El hecho de tener “condiciones” para continuar la iteración (adicionales al caso base) es que ahorra “espacio” y “tiempo” ya que no continúa si las soluciones serán peores que la actualmente encontrada.

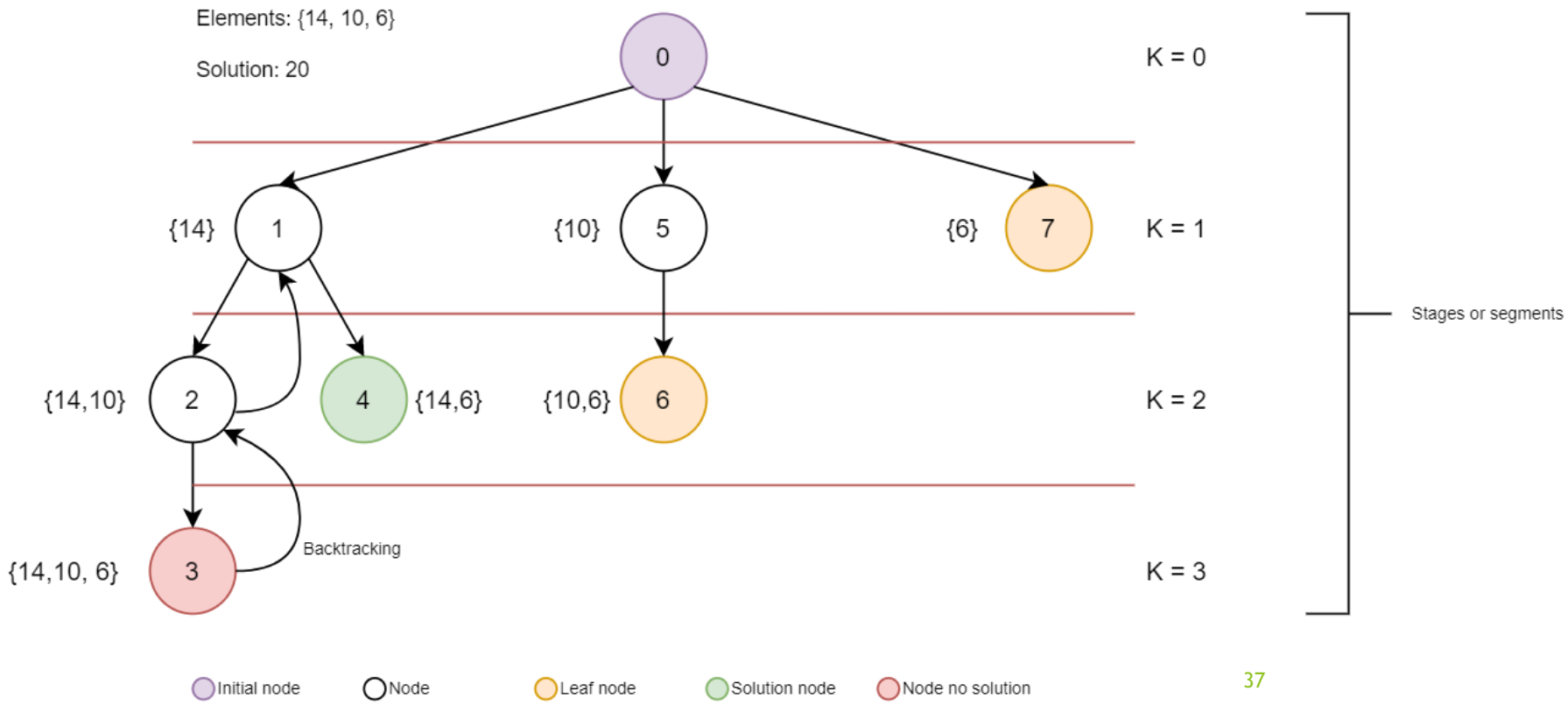


5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Backtracking Recursivo

Ejemplo

Dado un conjunto de números enteros {14, 10, 6} encontrar si existe algún subconjunto cuya suma sea igual a 20.



Fuente ejemplo: *https://docs.jjpeleato.com/algoritmia/backtracking*

Cierre de la sesión