Programación III

Ricardo Wehbe

UADE

13 de septiembre de 2021

Programa

Grafos. Un souvenir

Grafos. Un souvenir

- Recordemos que un *grafo* es un par (V, A) en el que V es un conjunto de *vértices* y $A \subseteq V \times V$ es un conjunto de *aristas*.

Grafos. Un souvenir

- Recordemos que un *grafo* es un par (V, A) en el que V es un conjunto de *vértices* y $A \subseteq V \times V$ es un conjunto de *aristas*.
- Un grafo puede ser *dirigido* o *no dirigido*. En el último caso, tenemos la equivalencia $(x, y) \equiv (y, x)$ para todo $(x, y) \in A$.

Algunas definiciones sobre grafos

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.
- Un grafo (V, A) es *fuertemente conexo* si para cualquier par de nodos $x, y \in V$, $(x, y) \in A$.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.
- Un grafo (V, A) es *fuertemente conexo* si para cualquier par de nodos $x, y \in V$, $(x, y) \in A$.
- Se pueden asociar distancias o costos a las aristas. Para ello, basta modificar ligeramente la definición de grafo.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.
- Un grafo (V, A) es *fuertemente conexo* si para cualquier par de nodos $x, y \in V$, $(x, y) \in A$.
- Se pueden asociar distancias o costos a las aristas. Para ello, basta modificar ligeramente la definición de grafo.
- En un grafo (V, A) con costos, tenemos $A \subseteq \mathbb{N} \times V \times V$.

Caminos mínimos en un grafo. El algoritmo de Dijkstra

Caminos mínimos en un grafo. El algoritmo de Dijkstra

- El problema es el siguiente: dado un grafo dirigido con costos (V, A) y un vértice $x \in V$, encontrar el camino de mínimo costo desde x a cualquier otro vértice de V.

Caminos mínimos en un grafo. El algoritmo de Dijkstra

- El problema es el siguiente: dado un grafo dirigido con costos (V, A) y un vértice $x \in V$, encontrar el camino de mínimo costo desde x a cualquier otro vértice de V.
- El resultado del proceso es un grafo con los mismos vértices pero que sólo tiene aristas desde el vértice x hacia los demás con el costo encontrado.

Caminos mínimos en un grafo. El algoritmo de Dijkstra

- El problema es el siguiente: dado un grafo dirigido con costos (V, A) y un vértice $x \in V$, encontrar el camino de mínimo costo desde x a cualquier otro vértice de V.
- El resultado del proceso es un grafo con los mismos vértices pero que sólo tiene aristas desde el vértice x hacia los demás con el costo encontrado.
- El algoritmo de Dijkstra es un algoritmo *greedy* que resuelve este problema.

El algoritmo de Dijkstra. Estrategia

El algoritmo de Dijkstra. Estrategia

- El conjunto de candidatos es el conjunto de vértices del grafo sin el vértice de origen.

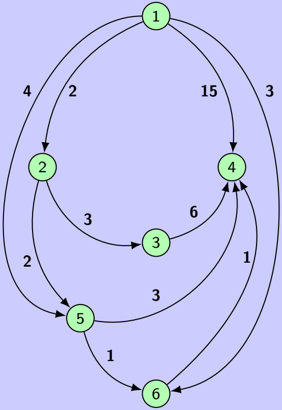
El algoritmo de Dijkstra. Estrategia

- El conjunto de candidatos es el conjunto de vértices del grafo sin el vértice de origen.
- En cada iteración, el método de selección elige el candidato que tenga el camino de mínimo costo desde el origen.

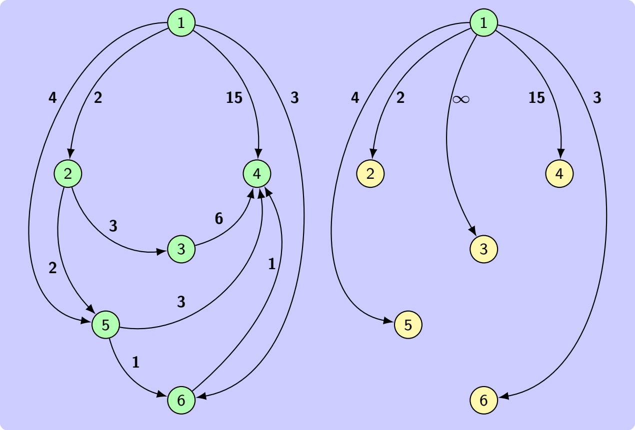
El algoritmo de Dijkstra. Estrategia

- El conjunto de candidatos es el conjunto de vértices del grafo sin el vértice de origen.
- En cada iteración, el método de selección elige el candidato que tenga el camino de mínimo costo desde el origen.
- La función factibilidad compara el costo del camino directo con el costo del camino que pasa por el vértice seleccionado, en caso de que este camino exista.

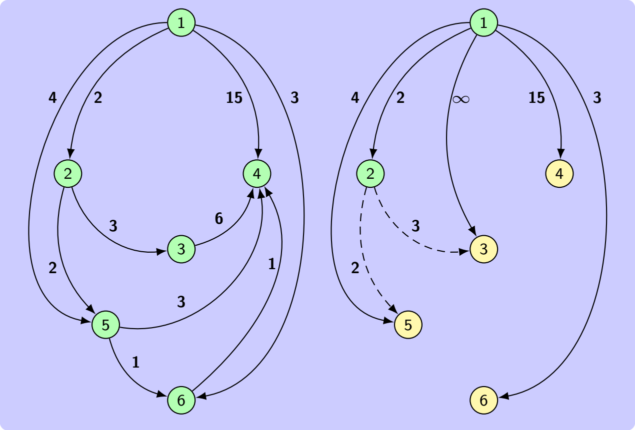
El algoritmo de Dijkstra. Un ejemplo



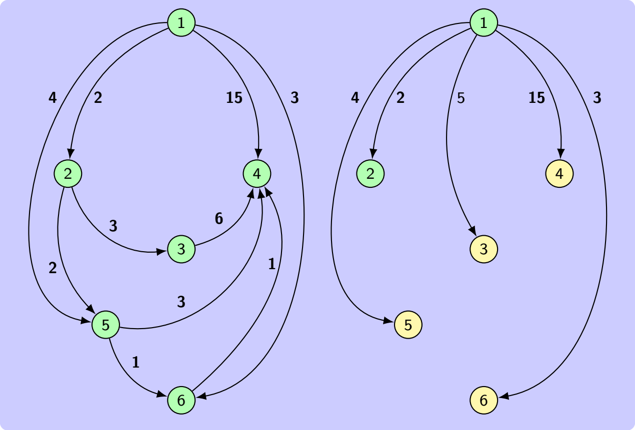
El algoritmo de Dijkstra. Un ejemplo



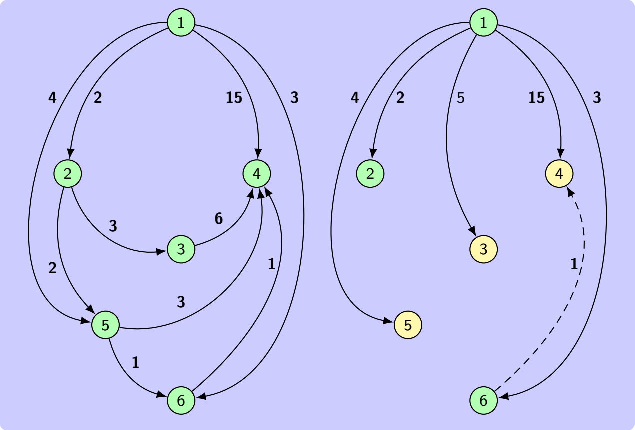
El algoritmo de Dijkstra. Un ejemplo



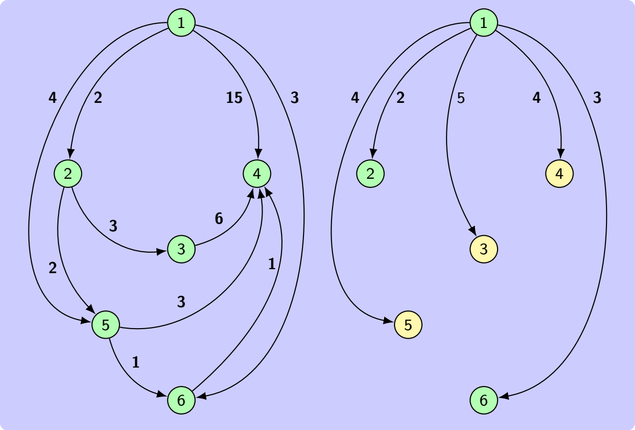
El algoritmo de Dijkstra. Un ejemplo



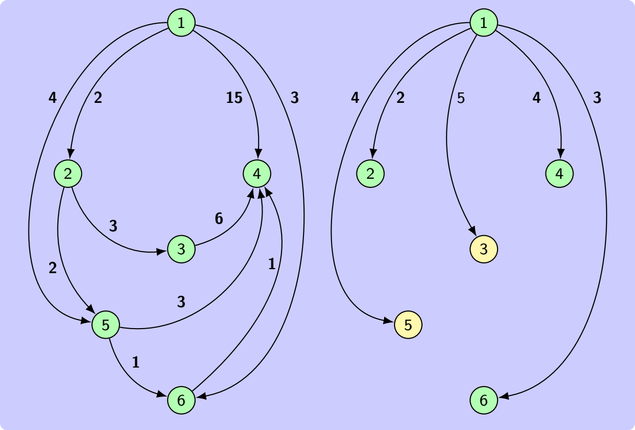
El algoritmo de Dijkstra. Un ejemplo



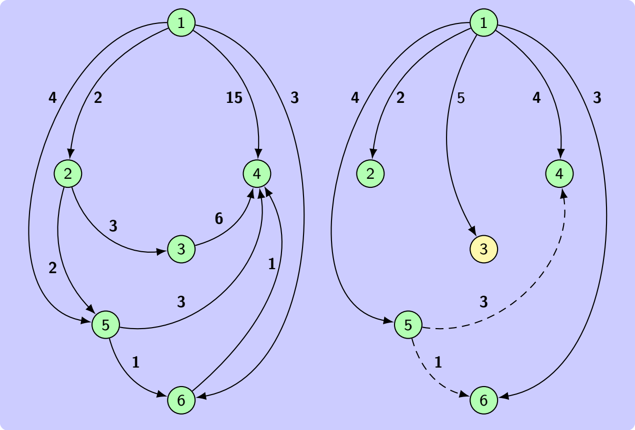
El algoritmo de Dijkstra. Un ejemplo



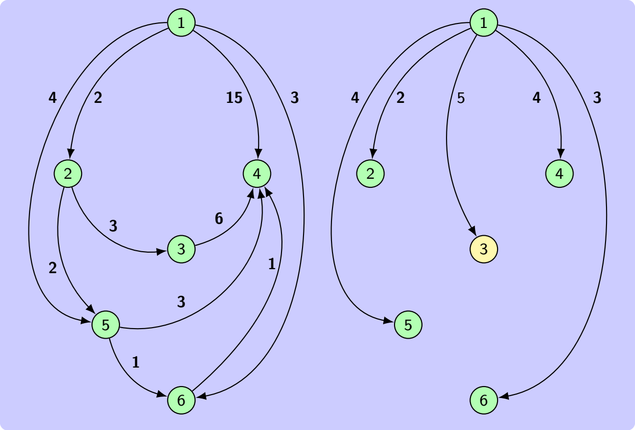
El algoritmo de Dijkstra. Un ejemplo



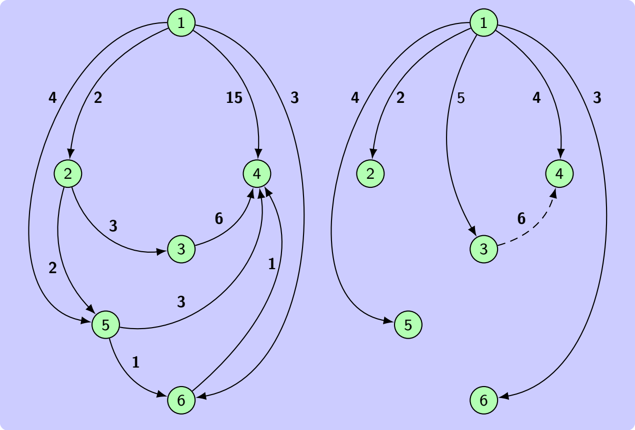
El algoritmo de Dijkstra. Un ejemplo



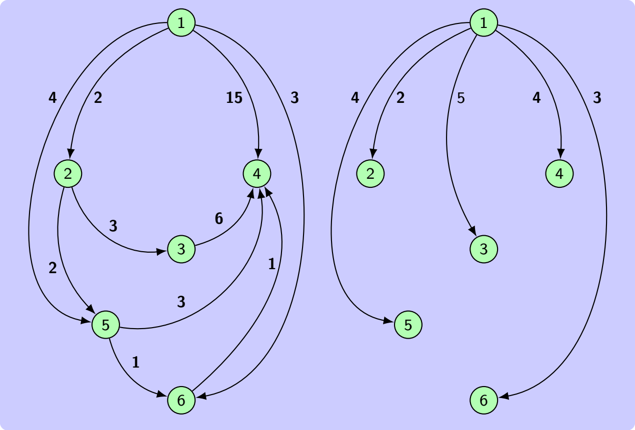
El algoritmo de Dijkstra. Un ejemplo



El algoritmo de Dijkstra. Un ejemplo



El algoritmo de Dijkstra. Un ejemplo



Árbol de recubrimiento mínimo (*minimum spanning tree*)

Árbol de recubrimiento mínimo (*minimum spanning tree*)

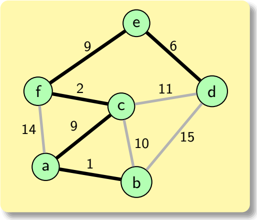
- Dado un grafo no dirigido y conectado con costos $G = (V, A)$, un *árbol de recubrimiento mínimo* para G es un grafo $G' = (V, A')$ con $A' \subseteq A$ que tiene estructura de árbol y que tiene costo mínimo.

Árbol de recubrimiento mínimo (*minimum spanning tree*)

- Dado un grafo no dirigido y conectado con costos $G = (V, A)$, un *árbol de recubrimiento mínimo* para G es un grafo $G' = (V, A')$ con $A' \subseteq A$ que tiene estructura de árbol y que tiene costo mínimo.
- Pueden existir múltiples árboles de recubrimiento mínimo para un mismo grafo.

Árbol de recubrimiento mínimo (*minimum spanning tree*)

- Dado un grafo no dirigido y conectado con costos $G = (V, A)$, un *árbol de recubrimiento mínimo* para G es un grafo $G' = (V, A')$ con $A' \subseteq A$ que tiene estructura de árbol y que tiene costo mínimo.
- Pueden existir múltiples árboles de recubrimiento mínimo para un mismo grafo.
- Por ejemplo:



Algoritmos *greedy* para árboles de recubrimiento mínimos

Algoritmos *greedy* para árboles de recubrimiento mínimos

- Existen dos algoritmos *greedy* para obtener un árbol de recubrimiento mínimo: el algoritmo de Prim y el de Kruskal.

Algoritmos *greedy* para árboles de recubrimiento mínimos

- Existen dos algoritmos *greedy* para obtener un árbol de recubrimiento mínimo: el algoritmo de Prim y el de Kruskal.
- Si el árbol de recubrimiento es único, ambos algoritmos arrojan el mismo resultado; si hay más de uno, es posible que los resultados difieran por la diferente estrategia que sigue cada uno de los algoritmos.

Algoritmo de Prim: estrategia

Algoritmo de Prim: estrategia

- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.

Algoritmo de Prim: estrategia

- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.
- La función de selección elige entre los candidatos al que tiene una arista de costo mínimo al conjunto de los vértices ya elegidos.

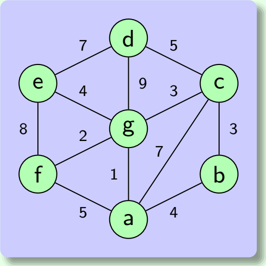
Algoritmo de Prim: estrategia

- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.
- La función de selección elige entre los candidatos al que tiene una arista de costo mínimo al conjunto de los vértices ya elegidos.
- Dependiendo de la implementación, la función de factibilidad puede decidir no utilizar la arista encontrada.

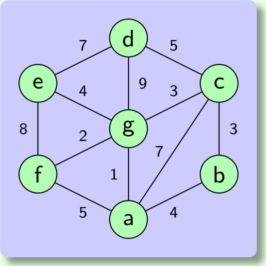
Algoritmo de Prim: estrategia

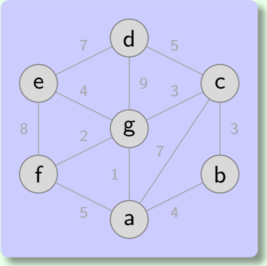
- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.
- La función de selección elige entre los candidatos al que tiene una arista de costo mínimo al conjunto de los vértices ya elegidos.
- Dependiendo de la implementación, la función de factibilidad puede decidir no utilizar la arista encontrada.
- La función solución verifica que todos los vértices se hayan utilizado.

El algoritmo de Prim. Un ejemplo



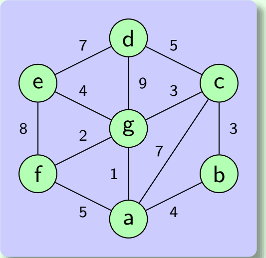
El algoritmo de Prim. Un ejemplo

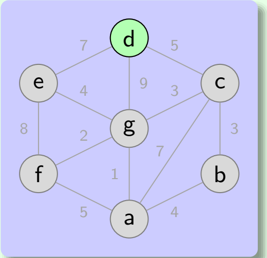




visitados	candidatos
$\emptyset$	$\{a, b, c, d, e, f, g\}$

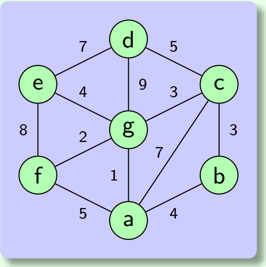
El algoritmo de Prim. Un ejemplo

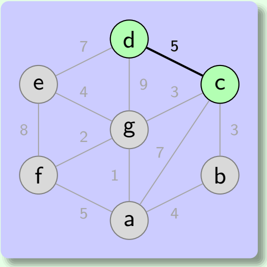




visitados	candidatos
$\{d\}$	$\{a,b,c,e,f,g\}$

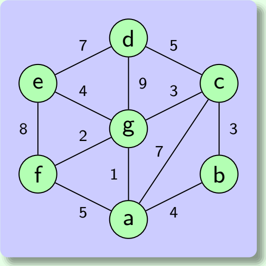
El algoritmo de Prim. Un ejemplo

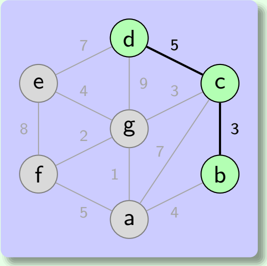




visitados	**candidatos**
$\{c,d\}$	$\{a,b,e,f,g\}$

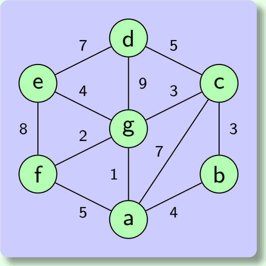
El algoritmo de Prim. Un ejemplo

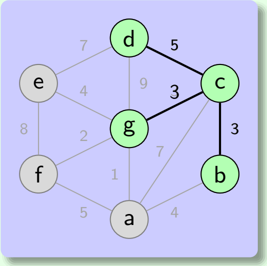




visitados	candidatos
$\{b,c,d\}$	$\{a,e,f,g\}$

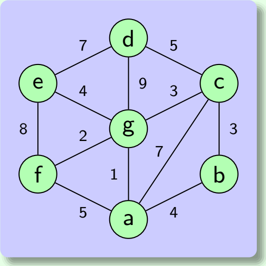
El algoritmo de Prim. Un ejemplo

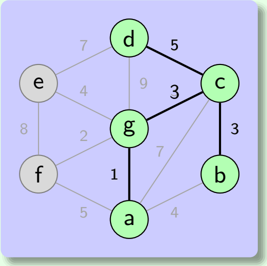




visitados	candidatos
$\{b, c, d, g\}$	$\{a, e, f\}$

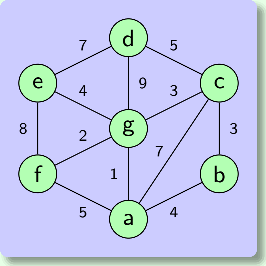
El algoritmo de Prim. Un ejemplo

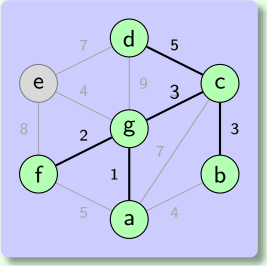




visitados	**candidatos**
$\{a,b,c,d,g\}$	$\{e,f\}$

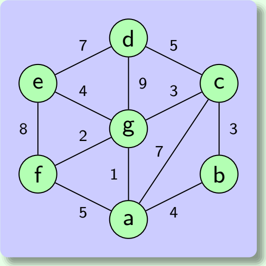
El algoritmo de Prim. Un ejemplo

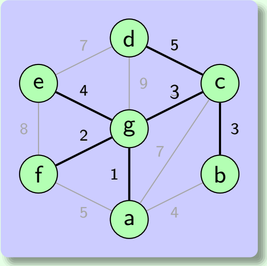




visitados	candidatos
$\{a,b,c,d,f,g\}$	$\{e\}$

El algoritmo de Prim. Un ejemplo





visitados	candidatos
$\{a,b,c,d,e,f,g\}$	$\{\ \}$

Algoritmo de Kruskal: estrategia

Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.

Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.

Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.
- La función de selección elige la arista de menos peso entre las disponibles.

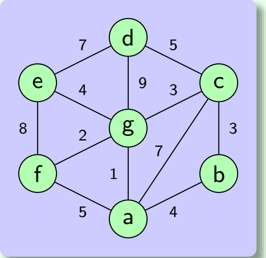
Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.
- La función de selección elige la arista de menos peso entre las disponibles.
- La función de factibilidad verifica que la arista seleccionada tenga sus vértices en diferentes árboles.

Algoritmo de Kruskal: estrategia

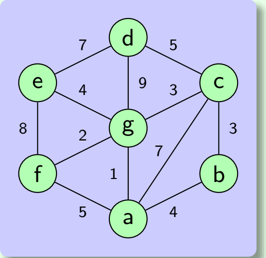
- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.
- La función de selección elige la arista de menos peso entre las disponibles.
- La función de factibilidad verifica que la arista seleccionada tenga sus vértices en diferentes árboles.
- La función solución verifica que haya quedado un único árbol.

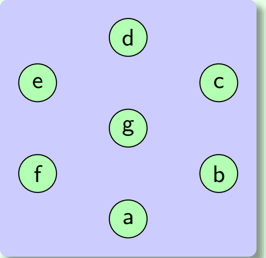
El algoritmo de Kruskal. Un ejemplo



El algoritmo de Kruskal. Un ejemplo

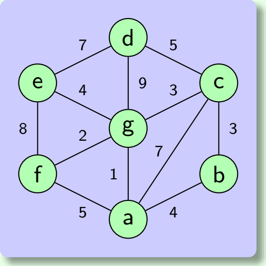
(a,g):1
(f,g):2
(b,c):3
(c,g):3
(a,b):4
(e,g):4
(a,f):5
(c,d):5
(a,c):7
(d,e):7
(e,f):8
(d,g):9

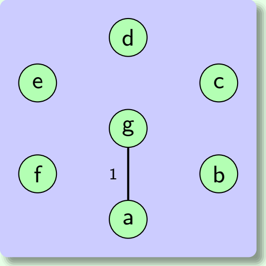




El algoritmo de Kruskal. Un ejemplo

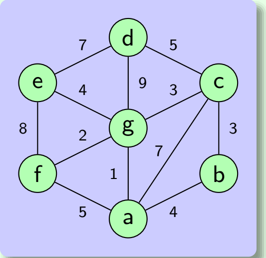
- ✓ (a,g):1
- (f,g):2
- (b,c):3
- (c,g):3
- (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

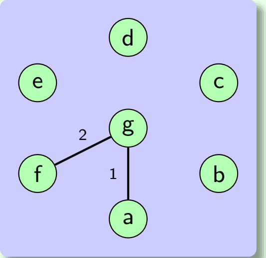




El algoritmo de Kruskal. Un ejemplo

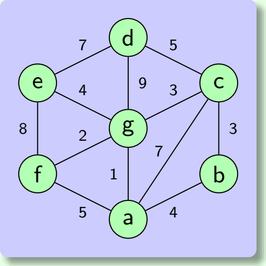
- ✓ (a,g):1
- ✓ (f,g):2
- (b,c):3
- (c,g):3
- (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

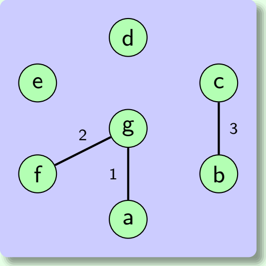




El algoritmo de Kruskal. Un ejemplo

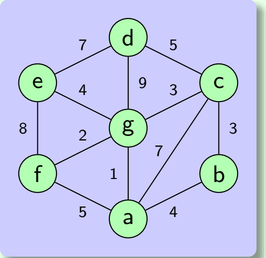
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- (c,g):3
- (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

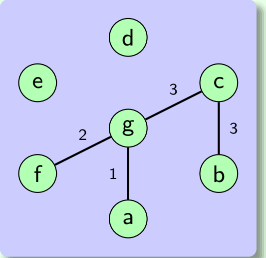




El algoritmo de Kruskal. Un ejemplo

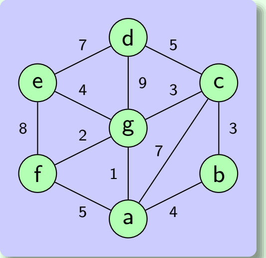
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

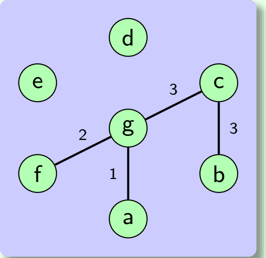




El algoritmo de Kruskal. Un ejemplo

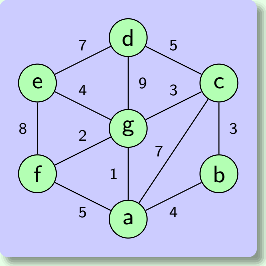
✓ (a,g):1
✓ (f,g):2
✓ (b,c):3
✓ (c,g):3
✗ (a,b):4
(e,g):4
(a,f):5
(c,d):5
(a,c):7
(d,e):7
(e,f):8
(d,g):9

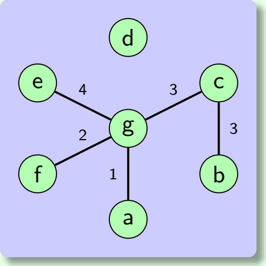




El algoritmo de Kruskal. Un ejemplo

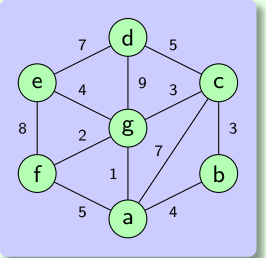
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- ✗ (a,b):4
- ✓ (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

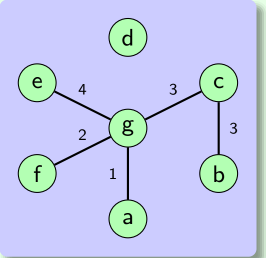




El algoritmo de Kruskal. Un ejemplo

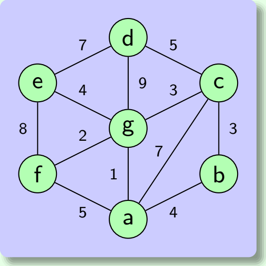
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- ✗ (a,b):4
- ✓ (e,g):4
- ✗ (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

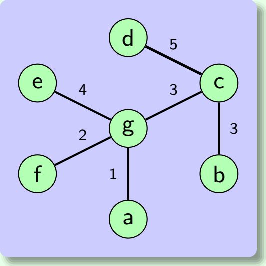




El algoritmo de Kruskal. Un ejemplo

- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- ✗ (a,b):4
- ✓ (e,g):4
- ✗ (a,f):5
- ✓ (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9





Prim vs. Kruskal

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.
- Un grafo conectado oscila entre $n-1$ aristas (si es un árbol) y $n(n-1)/2$ aristas (si se trata de un grafo fuertemente conectado).

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.
- Un grafo conectado oscila entre $n-1$ aristas (si es un árbol) y $n(n-1)/2$ aristas (si se trata de un grafo fuertemente conectado).
- Tenemos entonces dos casos límite para Kruskal: $\mathcal{O}(n \log n)$ (baja conectividad) y $\mathcal{O}(n^2 \log n)$ (alta conectividad).

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.
- Un grafo conectado oscila entre $n-1$ aristas (si es un árbol) y $n(n-1)/2$ aristas (si se trata de un grafo fuertemente conectado).
- Tenemos entonces dos casos límite para Kruskal: $\mathcal{O}(n \log n)$ (baja conectividad) y $\mathcal{O}(n^2 \log n)$ (alta conectividad).
- Dependiendo del tipo de grafo, uno de los dos algoritmos será entonces más adecuado.

Those who cannot remember the past are condemned to repeat it.
(Aquellos que no pueden recordar el pasado estÃąn condenados a repetirlo.)

George Santayana, *The Faces of Human Progress*

Introducción a la programación dinámica

Introducción a la programación dinámica

- La técnica de *programación dinámica* se basa en la resolución de problemas a través de su descomposición en subproblemas más pequeños del mismo tipo.

Introducción a la programación dinámica

- La técnica de *programación dinámica* se basa en la resolución de problemas a través de su descomposición en subproblemas más pequeños del mismo tipo.
- ¿Pero no es esto lo que ya hemos hecho con *divide & conquer*?

Introducción a la programación dinámica

- La técnica de *programación dinámica* se basa en la resolución de problemas a través de su descomposición en subproblemas más pequeños del mismo tipo.
- ¿Pero no es esto lo que ya hemos hecho con *divide & conquer*?
- En esencia, sí. La diferencia es que cuando los subproblemas no son independientes, la eficiencia de *divide & conquer* puede no ser aceptable.

Introducción a la programación dinámica

- La técnica de *programación dinámica* se basa en la resolución de problemas a través de su descomposición en subproblemas más pequeños del mismo tipo.
- ¿Pero no es esto lo que ya hemos hecho con *divide & conquer*?
- En esencia, sí. La diferencia es que cuando los subproblemas no son independientes, la eficiencia de *divide & conquer* puede no ser aceptable.
- En el caso de la programación dinámica, los subproblemas no son independientes. Hay subproblemas comunes, cuya resolución se almacena para su reutilización.

Algunas consideraciones sobre programación dinámica

Algunas consideraciones sobre programación dinámica

- Como en el caso de *divide & conquer*, esta técnica se aplica a problemas cuya solución óptima es una combinación de soluciones óptimas a subproblemas.

Algunas consideraciones sobre programación dinámica

- Como en el caso de *divide & conquer*, esta técnica se aplica a problemas cuya solución óptima es una combinación de soluciones óptimas a subproblemas.
- Esto se llama a veces *principio de optimalidad*. Si este principio se aplica al problema que queremos resolver, la programación dinámica o *divide & conquer* son dos buenos candidatos.

Algunas consideraciones sobre programación dinámica

- Como en el caso de *divide & conquer*, esta técnica se aplica a problemas cuya solución óptima es una combinación de soluciones óptimas a subproblemas.
- Esto se llama a veces *principio de optimalidad*. Si este principio se aplica al problema que queremos resolver, la programación dinámica o *divide & conquer* son dos buenos candidatos.
- Si los subproblemas tienen elementos en común, la programación dinámica es la técnica más prometedora.

Algunas consideraciones sobre programación dinámica

- Como en el caso de *divide & conquer*, esta técnica se aplica a problemas cuya solución óptima es una combinación de soluciones óptimas a subproblemas.
- Esto se llama a veces *principio de optimalidad*. Si este principio se aplica al problema que queremos resolver, la programación dinámica o *divide & conquer* son dos buenos candidatos.
- Si los subproblemas tienen elementos en común, la programación dinámica es la técnica más prometedora.
- Esta técnica se aplica a menudo a problemas de optimización, aunque esto no es excluyente.

1. Repaso de la clase anterior
   - Caminos más cortos en grafos: Dijkstra
   - Árboles de recubrimiento mínimo
   - El algoritmo de Prim
   - El algoritmo de Kruskal
2. Introducción a la programación dinámica
   - Nueva visita a Fibonacci
   - Nueva visita al problema del cambio
   - Nueva visita al problema de la mochila
3. Ejercicios propuestos

Nueva visita a Fibonacci

Nueva visita a Fibonacci

- Recordemos que la función de Fibonacci, que es una función $F : \mathbb{N}_0 \to \mathbb{N}_0$, se define como sigue:

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n > 1 \end{cases}$$

Nueva visita a Fibonacci

- Recordemos que la función de Fibonacci, que es una función $F : \mathbb{N}_0 \to \mathbb{N}_0$, se define como sigue:

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n > 1 \end{cases}$$

- Una implementación directa (versión 1) de esto es la siguiente:

```
int Algoritmo Fib (int n) {
    if (n ≤ 1) {                          // casos base
        return n;
    } else {
        return Fib(n − 1)+Fib(n − 2);
    }
}
```

La función de Fibonacci, versión 1. Complejidad

La función de Fibonacci, versión 1. Complejidad

- Aquí tenemos un caso de recurrencia por substracción.

La función de Fibonacci, versión 1. Complejidad

- Aquí tenemos un caso de recurrencia por substracción.
- Tenemos $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en $\Theta(n^k a^{n/b}) = \Theta(2^n)$.

La función de Fibonacci, versión 1. Complejidad

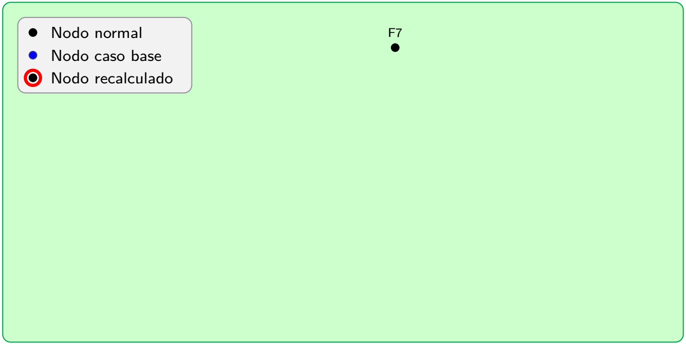
- Aquí tenemos un caso de recurrencia por substracción.
- Tenemos $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en $\Theta(n^k a^{n/b}) = \Theta(2^n)$.
- ¿Por qué tanto?

La función de Fibonacci, versión 1. Complejidad

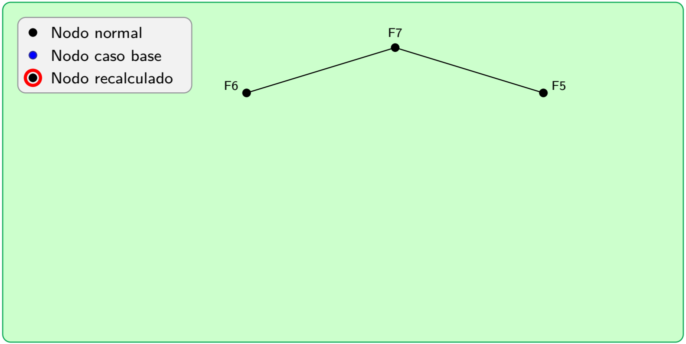
- Aquí tenemos un caso de recurrencia por substracción.
- Tenemos $a = 2$, $b = 1$ y $k = 0$. Por lo tanto, estamos en $\Theta(n^k a^{n/b}) = \Theta(2^n)$.
- ¿Por qué tanto?
- Hay demasiadas cosas que se calculan una y otra vez. Los subproblemas no son independientes.

Fibonacci. Cálculos redundantes

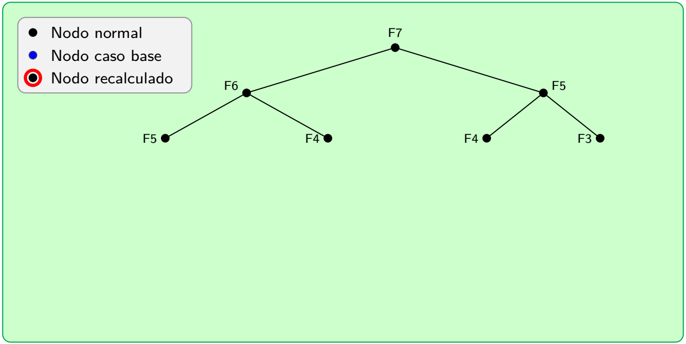
Fibonacci. Cálculos redundantes



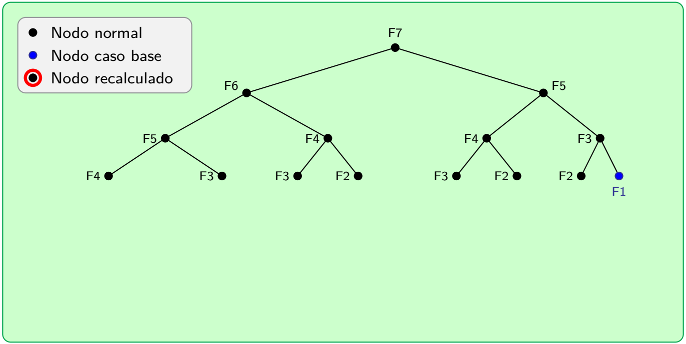
Fibonacci. Cálculos redundantes



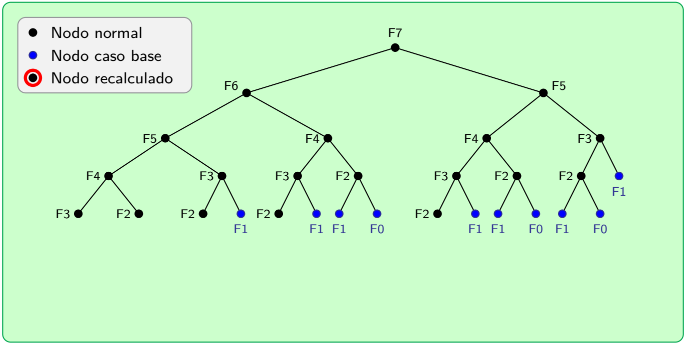
Fibonacci. Cálculos redundantes



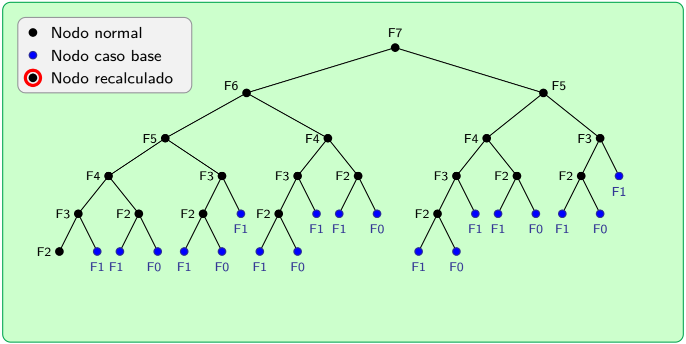
Fibonacci. Cálculos redundantes



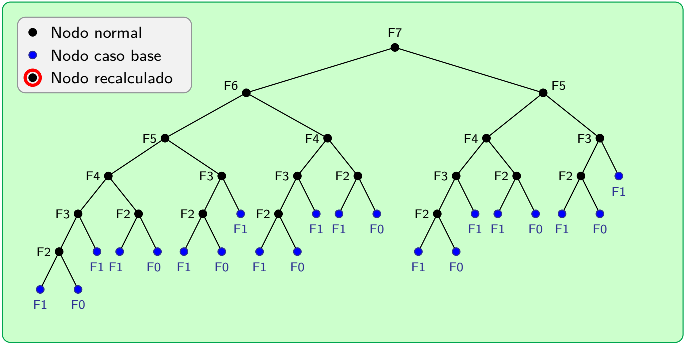
Fibonacci. Cálculos redundantes



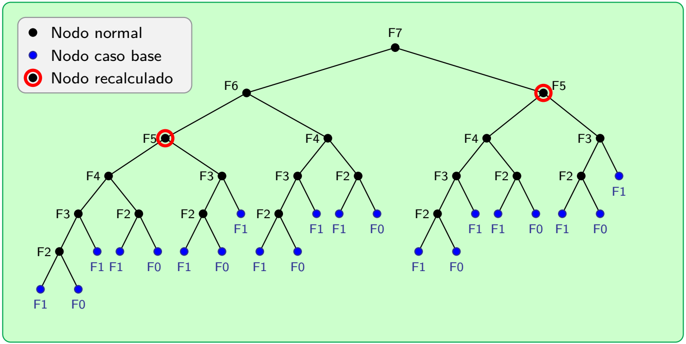
Fibonacci. Cálculos redundantes



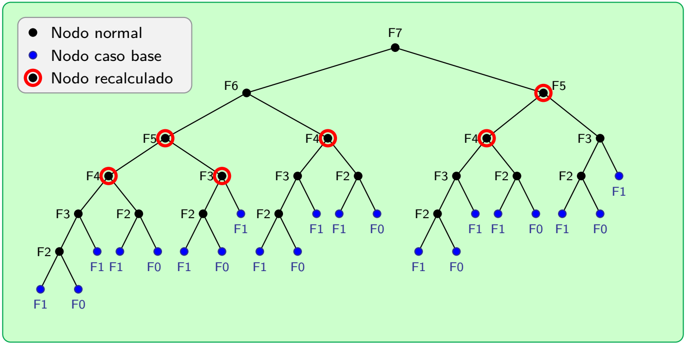
Fibonacci. Cálculos redundantes



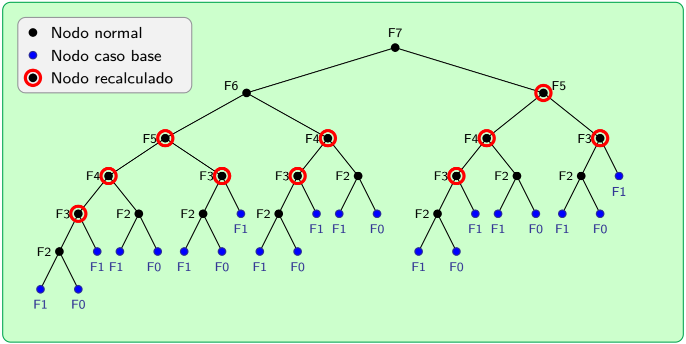
Fibonacci. Cálculos redundantes



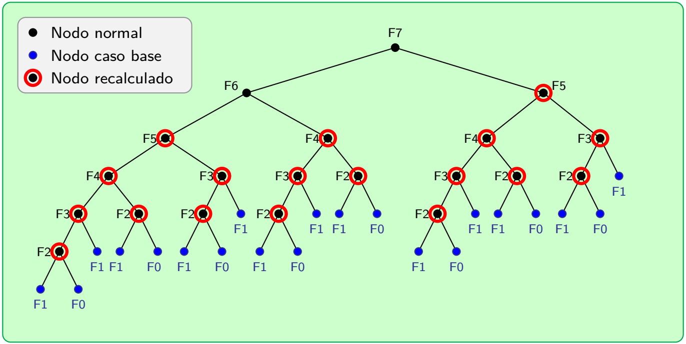
Fibonacci. Cálculos redundantes



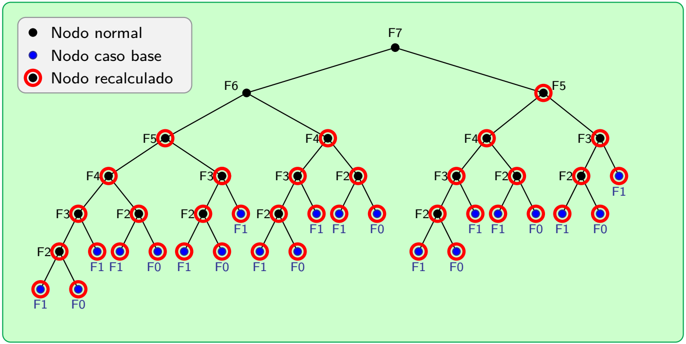
Fibonacci. Cálculos redundantes



Fibonacci. Cálculos redundantes



Fibonacci. Cálculos redundantes



Otra versión de Fibonacci, parte 1

Otra versión de Fibonacci, parte 1

- Según el paradigma de la programación dinámica, podríamos almacenar los valores calculados en una tabla para su ulterior utilización

Otra versión de Fibonacci, parte 1

- Según el paradigma de la programación dinámica, podríamos almacenar los valores calculados en una tabla para su ulterior utilización
- La tabla tendrá el aspecto siguiente:

$F(0)$	$F(1)$	$F(2)$	$\ldots$	$F(n)$

Otra versión de Fibonacci, parte 2

```
int Algoritmo Fib (int n) {
    table = initializeArray;
    if (n ≤ 1) {                              // casos base
        return n;
    } else {
        table[0] = 0;
        table[1] = 1;
        for (i = 2; i <= n; i++){
            table[i] = table[i − 1] + table[i − 2]
        }
        return table[n];
    }
}
```

Otra versión de Fibonacci, parte 2

```
int Algoritmo Fib (int n) {
    table = initializeArray;
    if (n ≤ 1) {                                  // casos base
        return n;
    } else {
        table[0] = 0;
        table[1] = 1;
        for (i = 2; i <= n; i++){
            table[i] = table[i − 1] + table[i − 2]
        }
        return table[n];
    }
}
```

$\Theta(n)$

Programación dinámica. Algunas consideraciones

Programación dinámica. Algunas consideraciones

- En general, una solución de programación dinámica tiene semejanzas con una solución *divide & conquer*: la solución se calcula en función de soluciones menores y se plantea recursivamente.

Programación dinámica. Algunas consideraciones

- En general, una solución de programación dinámica tiene semejanzas con una solución *divide & conquer*: la solución se calcula en función de soluciones menores y se plantea recursivamente.
- Se llena una tabla de soluciones de manera *bottom-up*

Programación dinámica. Algunas consideraciones

- En general, una solución de programación dinámica tiene semejanzas con una solución *divide & conquer*: la solución se calcula en función de soluciones menores y se plantea recursivamente.
- Se llena una tabla de soluciones de manera *bottom-up*
- El resultado se construye con la ayuda de la tabla.

Nueva visita al problema del cambio

Nueva visita al problema del cambio

- Tenemos n denominaciones y queremos pagar v con la mínima cantidad de monedas.

Nueva visita al problema del cambio

- Tenemos n denominaciones y queremos pagar v con la mínima cantidad de monedas.
- Soluciones parciales: pagar cantidades menores con la misma cantidad de denominaciones y con menos denominaciones.

Nueva visita al problema del cambio

- Tenemos n denominaciones y queremos pagar v con la mínima cantidad de monedas.
- Soluciones parciales: pagar cantidades menores con la misma cantidad de denominaciones y con menos denominaciones.
- Considere primero el siguiente ejemplo.

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1									
4									
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$. Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0								
4									
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1							
4									
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4									
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3					
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1				
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$. Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2			
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$. Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2	3		
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$. Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2	3	4	
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2	3	4	2
6									

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2	3	4	2
6	0	1	2	3	1	2			

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2	3	4	2
6	0	1	2	3	1	2	1		

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$. Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2	3	4	2
6	0	1	2	3	1	2	1	2	

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$. Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7	8
4	0	1	2	3	1	2	3	4	2
6	0	1	2	3	1	2	1	2	**2**

Para comenzar. Un ejemplo de juguete

Supongamos que tenemos $D = (6, 4, 1)$ y debemos pagar $V = 8$.
Descomponemos el problema en pagos de cifras menores ($j \leq V$) con un número menor de denominaciones ($i \leq |D|$.)

	0	1	2	3	4	5	6	7	8
1	**0**	1	2	3	4	5	6	7	8
4	**0**	1	2	3	**1**	2	3	4	**2**
6	0	1	2	3	1	2	1	2	**2**

La construcción de la tabla. Prolegómenos

La construcción de la tabla. Prolegómenos

- La solución para pagar $j \leq V$ con $i \leq n$ será el mínimo entre
  - pagar j con $i - 1$ denominaciones, y
  - pagar $j - d_i$ con i denominaciones más uno.

La construcción de la tabla. Prolegómenos

- La solución para pagar $j \leq V$ con $i \leq n$ será el mínimo entre
  - pagar j con $i - 1$ denominaciones, y
  - pagar $j - d_i$ con i denominaciones más uno.
- Es decir, si uso una moneda d_i debo sumar uno más la cantidad de monedas que necesito para pagar $j - d_i$ con i denominaciones; si no, considero simplemente la cantidad de monedas que necesito para pagar j con $i - 1$ denominaciones (no uso la moneda d_i.)

La construcción de la tabla. Prolegómenos

- La solución para pagar $j \leq V$ con $i \leq n$ será el mínimo entre
  - pagar j con $i - 1$ denominaciones, y
  - pagar $j - d_i$ con i denominaciones más uno.
- Es decir, si uso una moneda d_i debo sumar uno más la cantidad de monedas que necesito para pagar $j - d_i$ con i denominaciones; si no, considero simplemente la cantidad de monedas que necesito para pagar j con $i - 1$ denominaciones (no uso la moneda d_i.)
- Esto continúa ...

La construcción de la tabla

La construcción de la tabla

- Tendremos entonces una función $C[i, j]$ que nos da la cantidad mínima de monedas que se necesitan si tenemos i denominaciones para pagar j.

La construcción de la tabla

- Tendremos entonces una función $C[i,j]$ que nos da la cantidad mínima de monedas que se necesitan si tenemos i denominaciones para pagar j.

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ and } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{si } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ y } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

La construcción de la tabla

- Tendremos entonces una función $C[i,j]$ que nos da la cantidad mínima de monedas que se necesitan si tenemos i denominaciones para pagar j.

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ and } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{si } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ y } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

- Se construye entonces una tabla de n filas (una para cada denominación y $v + 1$ columnas (una para cada valor entre 0 y V.) En cada posición $C[i,j]$ de la tabla tendremos la cantidad mínima de monedas para pagar j con i denominaciones.

La construcción de la tabla

- Tendremos entonces una función $C[i,j]$ que nos da la cantidad mínima de monedas que se necesitan si tenemos i denominaciones para pagar j.

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ and } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{si } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ y } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

- Se construye entonces una tabla de n filas (una para cada denominación y $v + 1$ columnas (una para cada valor entre 0 y V.) En cada posición $C[i,j]$ de la tabla tendremos la cantidad mínima de monedas para pagar j con i denominaciones.
- La solución estará en la posición $C[n, V]$.

El algoritmo para el problema del cambio

```
Algoritmo Cambio (int[] D: array [], int V)
    C = initializeMatrix(D.length, V + 1)
    for (i = 1; i <= D.length; i++) {
        for (j = 0; j <= V;j++) {
            if (i == 1) {
                if (j == 0) {
                    C[i, j] = 0
                } else if (D[i] > j) {
                    C[i, j] = ∞
                } else {
                    C[i, j] = C[i, j − D[i]] + 1
                }
            } else {
                if (D[i] > j) {
                    C[i, j] = C[i − 1, j]
                } else {
                    C[i, j] = min(C[i − 1, j], C[i, j − D[i]] + 1)
                }
            }
        }
    }
    return C[D.length, V + 1, V]
```

El algoritmo para el problema del cambio

$\Theta(nV)$

```
Algoritmo Cambio (int[] D: array [], int V)
    C = initializeMatrix(D.length, V + 1)
    for (i = 1; i <= D.length; i++) {
        for (j = 0; j <= V; j++) {
            if (i == 1) {
                if (j == 0) {
                    C[i, j] = 0
                } else if (D[i] > j) {
                    C[i, j] = ∞
                } else {
                    C[i, j] = C[i, j − D[i]] + 1
                }
            } else {
                if (D[i] > j) {
                    C[i, j] = C[i − 1, j]
                } else {
                    C[i, j] = min(C[i − 1, j], C[i, j − D[i]] + 1)
                }
            }
        }
    }
    return C[D.length, V + 1, V]
```

$\Theta(V)$

$\Theta(n)$

Una comparación entre las soluciones

Una comparación entre las soluciones

- La solución *greedy* tiene una complejidad temporal de $\Theta(V)$.

Una comparación entre las soluciones

- La solución *greedy* tiene una complejidad temporal de $\Theta(V)$.
- La solución por programación dinámica tiene una complejidad temporal de $\Theta(n \cdot V)$.

Una comparación entre las soluciones

- La solución *greedy* tiene una complejidad temporal de $\Theta(V)$.
- La solución por programación dinámica tiene una complejidad temporal de $\Theta(n \cdot V)$.
- La solución *greedy* es por lo tanto más eficiente, pero no garantiza una solución óptima.

Una comparación entre las soluciones

- La solución *greedy* tiene una complejidad temporal de $\Theta(V)$.
- La solución por programación dinámica tiene una complejidad temporal de $\Theta(n \cdot V)$.
- La solución *greedy* es por lo tanto más eficiente, pero no garantiza una solución óptima.
- Observe que la solución por programación dinámica tiene también una complejidad espacial de $\Theta(n \cdot V)$.

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2									
4									
7									

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2	0								
4									
7									

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2	0	∞							
4									
7									

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2	0	∞	1	∞	2	∞	3	∞	4
4									
7									

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2	0	∞	1	∞	2	∞	3	∞	4
4	0	∞	1	∞					
7									

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2	0	∞	1	∞	2	∞	3	∞	4
4	0	∞	1	∞	1	∞	2	∞	2
7									

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2	0	∞	1	∞	2	∞	3	∞	4
4	0	∞	1	∞	1	∞	2	∞	2
7	0	∞	1	∞	1	∞	2		

¿Qué formula estamos usando?

$$C[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } j = 0 \\ +\infty & \text{si } i = 1 \text{ y } d_i > j \\ 1 + C[i, j - d_i] & \text{if } i = 1 \text{ y } d_i \leq j \\ C[i-1, j] & \text{si } i > 1 \text{ and } d_i > j \\ \min(1 + C[i, j - d_i], C[i-1, j]) & \text{sino} \end{cases}$$

	0	1	2	3	4	5	6	7	8
2	0	∞	1	∞	2	∞	3	∞	4
4	0	∞	1	∞	1	∞	2	∞	2
7	0	∞	1	∞	1	∞	2	1	2

1. Repaso de la clase anterior
   - Caminos más cortos en grafos: Dijkstra
   - Árboles de recubrimiento mínimo
   - El algoritmo de Prim
   - El algoritmo de Kruskal
2. Introducción a la programación dinámica
   - Nueva visita a Fibonacci
   - Nueva visita al problema del cambio
   - Nueva visita al problema de la mochila
3. Ejercicios propuestos

El problema de la mochila 0-1

El problema de la mochila 0-1

- Tenemos n objetos y una mochila. Para $i \in \{1, \ldots, n\}$, cada objeto $O[i]$ tiene un peso positivo p_i y un valor positivo v_i.

El problema de la mochila 0-1

- Tenemos n objetos y una mochila. Para $i \in \{1, \ldots, n\}$, cada objeto $O[i]$ tiene un peso positivo p_i y un valor positivo v_i.
- La mochila tiene una capacidad máxima de P. Si se le carga más peso, se desfonda.

El problema de la mochila 0-1

- Tenemos n objetos y una mochila. Para $i \in \{1, \ldots, n\}$, cada objeto $O[i]$ tiene un peso positivo p_i y un valor positivo v_i.
- La mochila tiene una capacidad máxima de P. Si se le carga más peso, se desfonda.
- Los objetos no pueden ser fraccionados. O se los incluye completos en la mochila o se los deja de lado.

El problema de la mochila 0-1

- Tenemos n objetos y una mochila. Para $i \in \{1, \ldots, n\}$, cada objeto $O[i]$ tiene un peso positivo p_i y un valor positivo v_i.
- La mochila tiene una capacidad máxima de P. Si se le carga más peso, se desfonda.
- Los objetos no pueden ser fraccionados. O se los incluye completos en la mochila o se los deja de lado.
- El objetivo es maximizar el valor cargado en la mochila respetando el límite de capacidad impuesto.

El problema de la mochila 0-1. Estrategia

El problema de la mochila 0-1. Estrategia

- El problema se descompone en problemas menores. Un problema menor es encontrar el valor máximo que podemos almacenar en una mochila de capacidad máxima $j \leq P$ considerando i objetos $1 \leq i \leq n$.

El problema de la mochila 0-1. Estrategia

- El problema se descompone en problemas menores. Un problema menor es encontrar el valor máximo que podemos almacenar en una mochila de capacidad máxima $j \leq P$ considerando i objetos $1 \leq i \leq n$.
- Construimos una matriz V de tamaño $V^{n \times P+1}$.

El problema de la mochila 0-1. Estrategia

- El problema se descompone en problemas menores. Un problema menor es encontrar el valor máximo que podemos almacenar en una mochila de capacidad máxima $j \leq P$ considerando i objetos $1 \leq i \leq n$.
- Construimos una matriz V de tamaño $V^{n \times P+1}$.
- El elemento $V[i,j]$ contendrá el valor máximo que podemos almacenar en una mochila de capacidad máxima j teniendo la opción de objetos $O[1], \ldots, O[i]$.

El problema de la mochila 0-1. Estrategia

- El problema se descompone en problemas menores. Un problema menor es encontrar el valor máximo que podemos almacenar en una mochila de capacidad máxima $j \leq P$ considerando i objetos $1 \leq i \leq n$.
- Construimos una matriz V de tamaño $V^{n \times P+1}$.
- El elemento $V[i, j]$ contendrá el valor máximo que podemos almacenar en una mochila de capacidad máxima j teniendo la opción de objetos $O[1], \ldots, O[i]$.
- Un nuevo objeto (una nueva fila) puede ser incluido o no; en el último caso, utilizamos la solución previa (la que se encuentra en la fila superior.)

El problema de la mochila 0-1. Estrategia

- El problema se descompone en problemas menores. Un problema menor es encontrar el valor máximo que podemos almacenar en una mochila de capacidad máxima $j \leq P$ considerando i objetos $1 \leq i \leq n$.
- Construimos una matriz V de tamaño $V^{n \times P+1}$.
- El elemento $V[i, j]$ contendrá el valor máximo que podemos almacenar en una mochila de capacidad máxima j teniendo la opción de objetos $O[1], \dots, O[i]$.
- Un nuevo objeto (una nueva fila) puede ser incluido o no; en el último caso, utilizamos la solución previa (la que se encuentra en la fila superior.)
- Si la solución parcial es óptima, la nueva solución, que elige la mejor de ambas alternativas, también lo es.

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3										
2										
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0							
2										
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3						
2										
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2										
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0								
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8							
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8					
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11				
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6										
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11				
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11	11	11		
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11	11	11	13	
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11	11	11	13	13
7										

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11	11	11	13	13
7	0	0	8	8	8	11	11			

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11	11	11	13	13
7	0	0	8	8	8	11	11	11		

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11	11	11	13	13
7	0	0	8	8	8	11	11	11	13	

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	0	0	0	3	3	3	3	3	3	3
2	0	0	8	8	8	11	11	11	11	11
6	0	0	8	8	8	11	11	11	13	13
7	0	0	8	8	8	11	11	11	13	**15**

El problema de la mochila 0-1. Un ejemplo

Supongamos que tenemos cuatro objetos de pesos $peso = (3, 2, 6, 7)$ con valores $valor = (3, 8, 5, 7)$. El peso máximo es $P = 9$. Descomponemos el problema como se explicó:

	0	1	2	3	4	5	6	7	8	9
3	**0**	0	0	3	3	3	3	3	3	3
2	0	0	**8**	8	8	11	11	11	11	11
6	0	0	**8**	8	8	11	11	11	13	13
7	0	0	8	8	8	11	11	11	13	**15**

La construcción de la tabla

La construcción de la tabla

- Luego del ejemplo, debería ser relativamente fácil construir la tabla. Recuerde que tenemos n objetos y que la carga máxima que puede llevar la mochila es P.

La construcción de la tabla

- Luego del ejemplo, debería ser relativamente fácil construir la tabla. Recuerde que tenemos n objetos y que la carga máxima que puede llevar la mochila es P.
- Recuerde también que, por conveniencia, el índice i se mueve entre 1 y n, mientras que el índice j se mueve entre 0 y P. Las fórmulas de recurrencia son:

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

El algoritmo *Mochila*

```
Algoritmo Mochila (object[] O: array [], int P)
    V = initializeMatrix(O.length, P + 1)
    for (i = 1; i <= O.length; i++) {
        for (j = 0; j <= P; j++) {
            if (i == 1) {
                if (O[i].peso > j) {
                    V[i, j] = 0
                } else {
                    V[i, j] = O[i].valor
                }
            } else {
                if (O[i].peso > j) {
                    V[i, j] = V[i − 1, j]
                } else {
                    V[i, j] = max(V[i − 1, j], V[i − 1, j − O[i].peso] + O[i].valor)
                }
            }
        }
    }
    return V[O.length, P]
```

El algoritmo *Mochila*

$\Theta(n \cdot P)$

$\Theta(n)$

$\Theta(P)$

```
Algoritmo Mochila (object[] O: array [], int P)
    V = initializeMatrix(O.length, P + 1)
    for (i = 1; i <= O.length; i++) {
        for (j = 0; j <= P; j++) {
            if (i == 1) {
                if (O[i].peso > j) {
                    V[i, j] = 0
                } else {
                    V[i, j] = O[i].valor
                }
            } else {
                if (O[i].peso > j) {
                    V[i, j] = V[i - 1, j]
                } else {
                    V[i, j] = max(V[i - 1, j], V[i - 1, j - O[i].peso] + O[i].valor)
                }
            }
        }
    }
    return V[O.length, P]
```

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)											
(6, 5)											
(4, 4)											
(5, 3)											

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1,j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1,j],\, V[i\text{-}1,j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0				
(6, 5)											
(4, 4)											
(5, 3)											

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0	8	8	8	8
(6, 5)											
(4, 4)											
(5, 3)											

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0	8	8	8	8
(6, 5)	0	0	0	0	0	0					
(4, 4)											
(5, 3)											

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0	8	8	8	8
(6, 5)	0	0	0	0	0	0	5	8	8	8	8
(4, 4)											
(5, 3)											

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0	8	8	8	8
(6, 5)	0	0	0	0	0	0	5	8	8	8	8
(4, 4)	0	0	0	0							
(5, 3)											

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0	8	8	8	8
(6, 5)	0	0	0	0	0	0	5	8	8	8	8
(4, 4)	0	0	0	0	4	4	5	8	8	8	9
(5, 3)											

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0	8	8	8	8
(6, 5)	0	0	0	0	0	0	5	8	8	8	8
(4, 4)	0	0	0	0	4	4	5	8	8	8	9
(5, 3)	0	0	0	0	4						

¿Qué formula estamos usando?

$$V[i,j] = \begin{cases} 0 & \text{si } i = 1 \text{ y } peso[i] > j \\ valor[i] & \text{si } i = 1 \text{ y } peso[i] \leq j \\ V[i\text{-}1, j] & \text{si } i > 1 \text{ y } peso[i] > j \\ \max(V[i\text{-}1, j], V[i\text{-}1, j\text{-}peso[i]] + valor[i]) & \text{si } i > 1 \text{ y } peso[i] \leq j \end{cases}$$

	0	1	2	3	4	5	6	7	8	9	10
(7, 8)	0	0	0	0	0	0	0	8	8	8	8
(6, 5)	0	0	0	0	0	0	5	8	8	8	8
(4, 4)	0	0	0	0	4	4	5	8	8	8	9
(5, 3)	0	0	0	0	4	4	5	8	8	8	9

Algunas observaciones

Problema del cambio

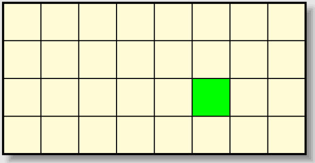
$C[i, j] = \texttt{min}(C[i\text{-}1, j], C[i, j\text{-}d[i]] + 1)$

Problema de la mochila

$V[i, j] = \texttt{max}(V[i\text{-}1, j], V[i\text{-}1, j\text{-}p[i]] + v[i])$

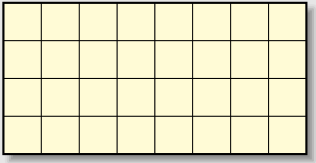
Algunas observaciones

Problema del cambio



$C[i,j] = \texttt{min}(C[i\text{-}1,j], C[i,j\text{-}d[i]] + 1)$

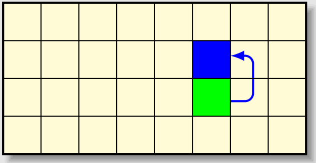
Problema de la mochila



$V[i,j] = \texttt{max}(V[i\text{-}1,j], V[i\text{-}1,j\text{-}p[i]] + v[i])$

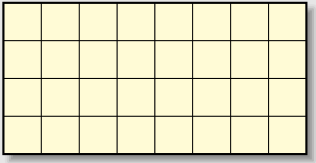
Algunas observaciones

Problema del cambio



$C[i,j] = \min(C[i\text{-}1,j],\ C[i,j\text{-}d[i]] + 1)$

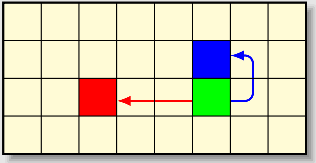
Problema de la mochila



$V[i,j] = \max(V[i\text{-}1,j],\ V[i\text{-}1,j\text{-}p[i]] + v[i])$

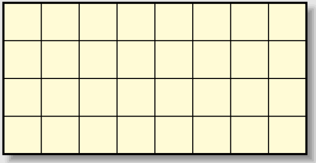
Algunas observaciones

Problema del cambio



$C[i,j] = \mathtt{min}(C[i\text{-}1,j], C[i,j\text{-}d[i]] + 1)$

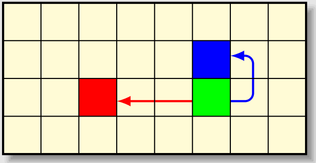
Problema de la mochila



$V[i,j] = \mathtt{max}(V[i\text{-}1,j], V[i\text{-}1,j\text{-}p[i]] + v[i])$

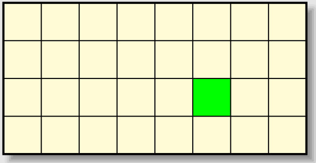
Algunas observaciones

Problema del cambio



$C[i, j] = \mathtt{min}(C[i\text{-}1, j], C[i, j\text{-}d[i]] + 1)$

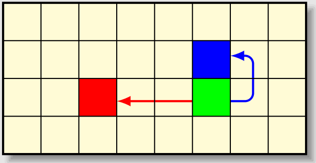
Problema de la mochila



$V[i, j] = \mathtt{max}(V[i\text{-}1, j], V[i\text{-}1, j\text{-}p[i]] + v[i])$

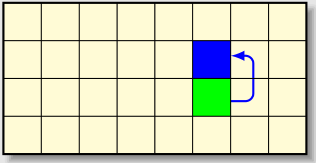
Algunas observaciones

Problema del cambio



$C[i,j] = \mathtt{min}(C[i\text{-}1,j],\ C[i,j\text{-}d[i]] + 1)$

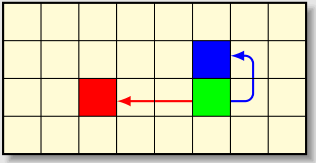
Problema de la mochila



$V[i,j] = \mathtt{max}(V[i\text{-}1,j],\ V[i\text{-}1,j\text{-}p[i]] + v[i])$

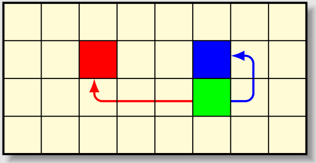
Algunas observaciones

Problema del cambio



$$C[i,j] = \mathtt{min}(C[i\text{-}1,j], C[i,j\text{-}d[i]] + 1)$$

Problema de la mochila



$$V[i,j] = \mathtt{max}(V[i\text{-}1,j], V[i\text{-}1,j\text{-}p[i]] + v[i])$$

Ejercicios propuestos 1

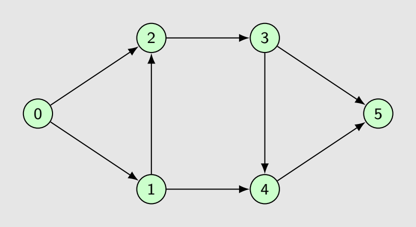
1. En el problema del cambio, complete el pseudo-código de manera que se pueda saber cuáles fueron las monedas usadas.
2. En el problema de la mochila 0-1, complete el pseudo-código de manera que se pueda saber cuáles fueron los objetos seleccionados.
3. **Navegación en el río Ctalamochita, de nuevo**. Usted planea navegar en canoa aguas abajo por el río Ctalamochita entre las ciudades de Morrison y Monte Leña. Hay n puestos de canoas a lo largo de este trayecto. Antes de comenzar su excursión, usted consigue para cada $1 \leq i < j \leq n$, el precio $f_{i,j}$ para alquilar una canoa desde el puesto i hasta el puesto j. Estos precios son arbitrarios. Por ejemplo, es posible que sea $f_{1,3} = 10$ y $f_{1,4} = 5$. Usted comienza en el puesto 1 y debe terminar en el puesto n (usando canoas alquiladas). El objetivo es minimizar el costo.

   Aplique un abordaje de programación dinámica para resolver este problema. Considere la posibilidad de construir una tabla en la que la posición i contenga el costo mınimopara llegar al puesto 1 desde el puesto 1.

Suggested Exercises 2

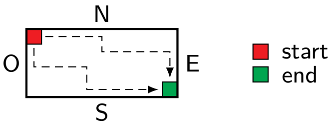
4. **Caminos más largos en grafos ordenados:** supongamos que tenemos un grafo dirigido $G = (V, E)$ con $V = \{v_1, \ldots, v_n\}$. Decimos que G es *ordenado* si satisface las siguientes condiciones:
   - Todas las aristas tienen la forma (v_i, v_j) con $i < j$.
   - Todos los vértices tienen aristas salientes excepto v_n.

   El objetivo es encontrar la longitud del camino más largo que va de v_1 a v_n. Se pide una estrategia de programación dinámica para resolver este problema.
   Por ejemplo, el siguiente grafo es ordenado. El camino más largo de 0 a 5 es (0-1-2-3-4-5), o sea tiene longitud 5.



Suggested Exercises 3

5. El paseo probabilístico. El párroco de la iglesia de San Hilario en Fraile Muerto es conocido por ser una persona de costumbres sistemáticas. Todos los días, luego de su inevitable siesta, se dirige desde su casa (situada en la esquina noroeste en el mapa del centro de la ciudad (posición $M[0,0]$) a su iglesia situada en el extremo opuesto, es decir la esquina sudeste del mapa (posición $M[m-1, n-1]$.) El mapa del centro de la ciudad de Fraile Muerto, como el de tantas otras ciudades de la Pampa, es una cuadrícula M de m filas por n columnas.



Su recorrido se dirige hacia el este o hacia el sur (nunca vuelve hacia el oeste ni hacia el norte.) La probabilidad de que en un punto dado se dirija hacia el este es del 40 %; la probabilidad de que se dirija hacia el sur es del 60 %. Por supuesto, una vez que llega al extremo sur (fila $m-1$), la probabilidad de tomar hacia el este es del 100 % y análogamente cuando llega al extremo este.
Describa una estrategia de programación dinámica para determinar la probabilidad de que el cura pase por un punto $M[y, y]$ dado con $x \in \{0, \ldots, m\text{-}1\}$, $y \in \{0, \ldots, n\text{-}1\}$.

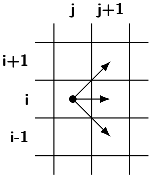
Suggested Exercises 4

6. **El campo de zanahorias**. Se tiene un campo de zanahorias representado por una matriz M, que es una cuadrícula $m \times n$, en la que cada posición $M[i,j]$ representa la cantidad de zanahorias que se pueden encontrar allí. Un conejo recorre el campo de izquierda a derecha. Comienza en la columna 0 y puede elegir cualquier fila desde 0 hasta m-1. Por supuesto, el conejo devorará todas las zanahorias que encuentre en cada posición por la que pase.

   Si el conejo está en la posición $M[i,j]$, tiene las siguientes posibilidades:

   - Arriba a la derecha: $M[i,j] \rightarrow M[i-1, j+1]$ (si $i > 0$ y $j <$ m-1.)
   - A la derecha: $M[i,j] \rightarrow M[i, j+1]$ (si $j <$ m-1.)
   - Abajo a la derecha: $M[i,j] \rightarrow M[i+1, j+1]$ (si $i <$ n-1 y $j <$ m-1.)

   Estas movidas se muestran abajo.



   Encuentre una estrategia de programación dinámica para que el conejo pueda maximizar la cantidad de zanahorias comidas.

Suggested Exercises 5

7. El triángulo de Tartaglia. O de Pascal. O de Yang Hui. El coeficiente binomial $C(n, k)$ da el coeficiente de la potencia k de x en el polinomio $(1 + x)^n$. Por ejemplo: $C(4, 3) = 4$ porque $(1 + x)^4 = 1 + 4x + 6x^2 + \underline{\mathbf{4}}x^3 + x^4$.
   Por ejemplo: $C(4, 3) = 4$ porque $(1 + x)^4 = 1 + 4x + 6x^2 + \underline{\mathbf{4}}x^3 + x^4$. Describa una estrategia de programación dinámica para encontrar $C(n, k)$ a partir de dos valores n y k dados. Observe que $C(n, k)$ puede ser computado a partir de resultados previos:

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$$