Greedy y Programación dinámica

Grupo de programación competitiva

Taller de Programación

Contenido

Contenido

Optimización

Definition

Un problema de optimización es aquel en el que se busca la mejor solución dentro de un conjunto de soluciones válidas

- Un problema de optimización usualmente pide *maximizar* o *minimizar* algún valor objetivo.

Optimización

Definition

Un problema de optimización es aquel en el que se busca la mejor solución dentro de un conjunto de soluciones válidas

- Un problema de optimización usualmente pide *maximizar* o *minimizar* algún valor objetivo.
- Por ejemplo, dado un conjunto de rutas posibles entre dos ciudades, hallar la ruta que minimice el tiempo de viaje.

Optimización

Definition

Un problema de optimización es aquel en el que se busca la mejor solución dentro de un conjunto de soluciones válidas

- Un problema de optimización usualmente pide *maximizar* o *minimizar* algún valor objetivo.
- Por ejemplo, dado un conjunto de rutas posibles entre dos ciudades, hallar la ruta que minimice el tiempo de viaje.
- En el ejemplo anterior, hay que notar que pueden existir múltiples rutas que válidas, pero no todas son óptimas.

Optimización

Definition

Un problema de optimización es aquel en el que se busca la mejor solución dentro de un conjunto de soluciones válidas

- Un problema de optimización usualmente pide *maximizar* o *minimizar* algún valor objetivo.
- Por ejemplo, dado un conjunto de rutas posibles entre dos ciudades, hallar la ruta que minimice el tiempo de viaje.
- En el ejemplo anterior, hay que notar que pueden existir múltiples rutas que válidas, pero no todas son óptimas.
- Así mismo, puede existir más de una ruta óptima, por tanto, más de una solución óptima. Estamos interesados en cualquier solución óptima.

Contenido

Algoritmos *Greedy*: voraces o golosos

- Se llama algoritmo voraz a aquel que escoge la mejor opción disponible en el momento.

Algoritmos *Greedy*: voraces o golosos

- Se llama algoritmo voraz a aquel que escoge la mejor opción disponible en el momento.
- Este tipo de algoritmos no observan opciones generales, sino que son muy focalizados en el momento.

Algoritmos *Greedy*: voraces o golosos

- Se llama algoritmo voraz a aquel que escoge la mejor opción disponible en el momento.
- Este tipo de algoritmos no observan opciones generales, sino que son muy focalizados en el momento.
- Usualmente son eficientes y procesan un dato a la vez.

Algoritmos *Greedy*: voraces o golosos

- Se llama algoritmo voraz a aquel que escoge la mejor opción disponible en el momento.
- Este tipo de algoritmos no observan opciones generales, sino que son muy focalizados en el momento.
- Usualmente son eficientes y procesan un dato a la vez.
- **No** todo problema de optimización puede resuelto aplicando un algoritmo voraz.

Algoritmos *Greedy*: voraces o golosos

- Se llama algoritmo voraz a aquel que escoge la mejor opción disponible en el momento.
- Este tipo de algoritmos no observan opciones generales, sino que son muy focalizados en el momento.
- Usualmente son eficientes y procesan un dato a la vez.
- **No** todo problema de optimización puede resuelto aplicando un algoritmo voraz.
- Dado un problema, se debe demostrar que un algoritmo voraz encuentra una solución óptima.

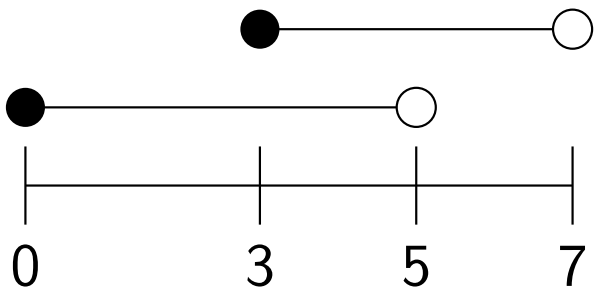
Interval scheduling

- Dado un conjunto de intervalos que definen peticiones, se requiere atender el conjunto máximo de intervalos de mayor tamaño, dado que los intervalos no se sobrepongan.

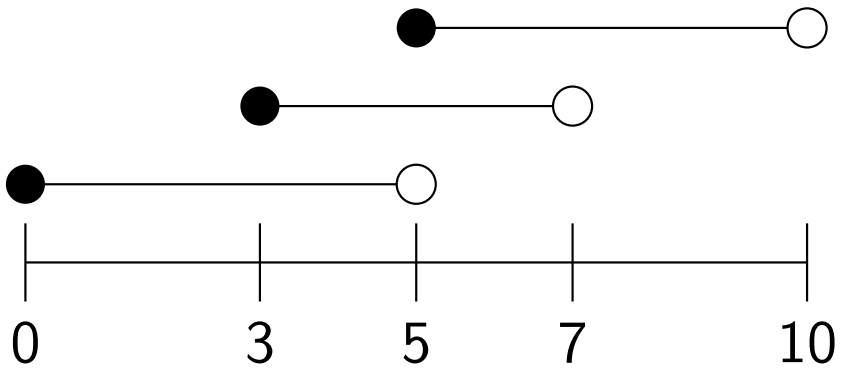
Interval scheduling

- Dado un conjunto de intervalos que definen peticiones, se requiere atender el conjunto máximo de intervalos de mayor tamaño, dado que los intervalos no se sobrepongan.
- Formalizando, sean n intervalos, tal que el tiempo de inicio del intervalo i-ésimo es $s(i)$ y el tiempo de finalización es $f(i)$. Sea S una solución, entonces $\forall j, k \in S$, vale que $f(j) \leq s(k)$ o $f(k) \leq s(j)$. Una solución óptima es aquella que maximiza $|S|$

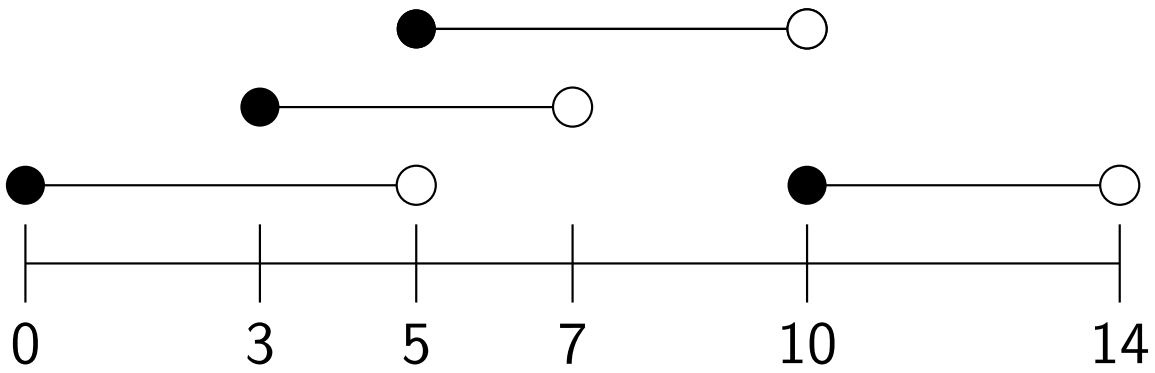
Interval scheduling



Interval scheduling



Interval scheduling



Algoritmo voraz

1. Seleccionar, bajo alguna estrategia, un intervalo i
2. Eliminar todos aquellos intervalos no compatibles con el intervalo i
3. Repetir hasta que todos los intervalos sean procesados

Algoritmo voraz

1. Seleccionar, bajo alguna estrategia, un intervalo i
2. Eliminar todos aquellos intervalos no compatibles con el intervalo i
3. Repetir hasta que todos los intervalos sean procesados

- Usualmente esta estructura básica se repite en varios problemas que admiten soluciones greedy

Algoritmo voraz

1. Seleccionar, bajo alguna estrategia, un intervalo i
2. Eliminar todos aquellos intervalos no compatibles con el intervalo i
3. Repetir hasta que todos los intervalos sean procesados

- Usualmente esta estructura básica se repite en varios problemas que admiten soluciones greedy
- ¿Qué estrategia se debe usar para seleccionar un intervalo?

Solución

- Seleccionar aquel intervalo que tenga el menor tiempo de finalización

input : $I = (s, f)$ es el conjunto de n intervalos
output: S conjunto máximo de intervalos

$Sort(I)$ *por* f ;
$S \leftarrow \{I[0]\}$;
$time \leftarrow I[0].f$;
for $i \in I \setminus \{I[0]\}$ **do**
    if $time \leq i.s$ **then**
        $S.append(i)$;
        $time \leftarrow i.f$
    end
end
return S

Problema de la mochila fraccionaria

- Dado un conjunto de elementos, cada uno con un valor v y un peso w, y dada un mochila de capacidad W, se requiere llenar la mochila con la mayor cantidad de elementos tal que el valor total de los objetos sea máximo sin sobrepasar la capacidad de la mochila. Note que los elementos pueden ser partidos en fracciones.

Problema de la mochila fraccionaria

- Dado un conjunto de elementos, cada uno con un valor v y un peso w, y dada un mochila de capacidad W, se requiere llenar la mochila con la mayor cantidad de elementos tal que el valor total de los objetos sea máximo sin sobrepasar la capacidad de la mochila. Note que los elementos pueden ser partidos en fracciones.
- Formalizando, sean n elementos $E = (v, w)$, se busca un subconjunto de elementos $S \subset E$ tal que $\sum_{s \in S} s.v$ es el máximo posible y $\sum_{s \in S} s.w \leq W$

Solución

- Seleccionar los elementos que tengan un mayor ratio v_i/w_i

Solución

input : $E = (v, w)$ es el conjunto de elementos y W el tamaño de la mochila

output: S conjunto de elementos en la mochila

$Sort(E)$ *por* v/w ;
$value \leftarrow 0$;
$c \leftarrow W$; `/* capacidad disponible en la mochila */`
for $e \in E$ **do**
  if $c < e.w$ **then**
    $value \leftarrow value + e.v/e.w * (W - c)$;
    break
  end
  $value \leftarrow value + e.v$;
  $c \leftarrow c - e.w$;
end

Contenido

Dynamic Programming (DP)

- Es una técnica de diseño de algoritmos usualmente utilizada para resolver problemas de optimización

Dynamic Programming (DP)

- Es una técnica de diseño de algoritmos usualmente utilizada para resolver problemas de optimización
- De manera análoga a la técnica de divide y vencerás, DP también divide el problema en subproblemas

Dynamic Programming (DP)

- Es una técnica de diseño de algoritmos usualmente utilizada para resolver problemas de optimización
- De manera análoga a la técnica de divide y vencerás, DP también divide el problema en subproblemas
- La diferencia es que los subproblemas se sobreponen, y la solución a cada subproblema es guardada en una tabla

Dynamic Programming (DP)

- Es una técnica de diseño de algoritmos usualmente utilizada para resolver problemas de optimización
- De manera análoga a la técnica de divide y vencerás, DP también divide el problema en subproblemas
- La diferencia es que los subproblemas se sobreponen, y la solución a cada subproblema es guardada en una tabla
- Usualmente la complejidad es, al menos, cuadrática

Dynamic Programming (DP)

- DP es utilizado para resolver problemas de optimización que satisfacen el *principio de optimalidad*: en una secuencia de decisiones óptimas, cada subsecuencia debe ser también óptima

Dynamic Programming (DP)

- DP es utilizado para resolver problemas de optimización que satisfacen el *principio de optimalidad*: en una secuencia de decisiones óptimas, cada subsecuencia debe ser también óptima
- Esto puede parecer obvio pero no siempre aplica

Dynamic Programming (DP)

- DP es utilizado para resolver problemas de optimización que satisfacen el *principio de optimalidad*: en una secuencia de decisiones óptimas, cada subsecuencia debe ser también óptima
- Esto puede parecer obvio pero no siempre aplica
- Alternativamente se puede definir para los problemas que aplica el principio de optimalidad: la solución óptima a cualquier instancia no trivial es una combinación de soluciones óptimas de algunas de sus subinstancias.

Dynamic Programming (DP)

- Primero se debe caracterizar la estructura de una solución óptima

Dynamic Programming (DP)

- Primero se debe caracterizar la estructura de una solución óptima
- Luego, se define recursivamente el valor de una solución óptima

Dynamic Programming (DP)

- Primero se debe caracterizar la estructura de una solución óptima
- Luego, se define recursivamente el valor de una solución óptima
- Computar el valor de la solución óptima, usualmente de "arriba hacia abajo" (recursión y memorización)

Dynamic Programming (DP)

- Primero se debe caracterizar la estructura de una solución óptima
- Luego, se define recursivamente el valor de una solución óptima
- Computar el valor de la solución óptima, usualmente de "arriba hacia abajo" (recursión y memorización)
- Construir la solución óptima de la información hallada

Mayor secuencia palindrómica

Definition

Un palíndrome es una secuencia de caracteres que se lee igual de izquierda a derecha como de derecha a izquierda. Por ejemplo: “radar”, “221122”, “oso”, “aerea”, “ala”

- En el problema es dada una secuencia de caracteres y se busca la subsecuencia más larga tal que sea un palíndromo

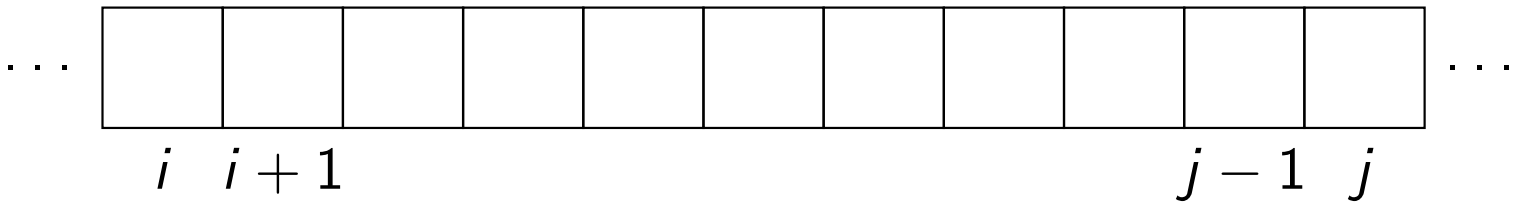
Mayor secuencia palindrómica

Definition

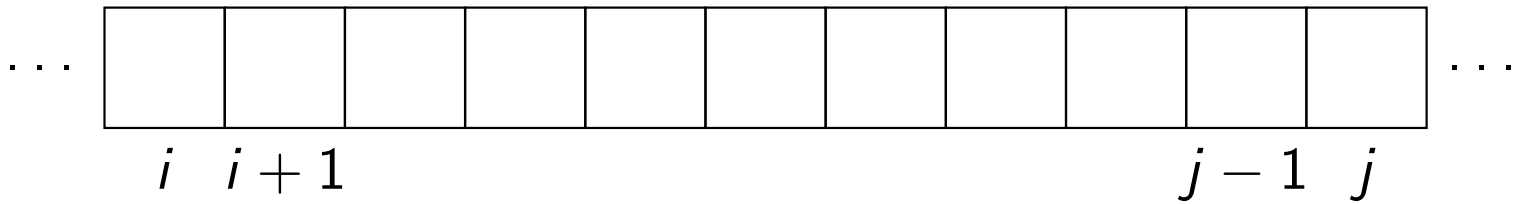
Un palíndrome es una secuencia de caracteres que se lee igual de izquierda a derecha como de derecha a izquierda. Por ejemplo: “radar”, “221122”, “oso”, “aerea”, “ala”

- En el problema es dada una secuencia de caracteres y se busca la subsecuencia más larga tal que sea un palíndromo
- Por ejemplo, dado “character”, las subsecuencias palindrómicas son “t”, “ara”, “carac”. La mayor de ellas es “carac”

Mayor secuencia palindrómica

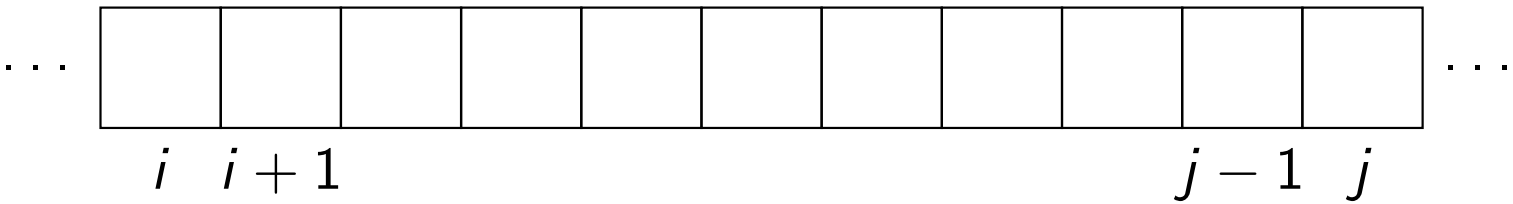


Mayor secuencia palindrómica



- $L(i,j) = \max\{L(i,j-1), L(i+1,j)\}$

Mayor secuencia palindrómica



- $L(i,j) = \max\{L(i,j-1), L(i+1,j)\}$
- Trivialmente, cualquier secuencia de tamaño 1 es un palíndromo, siendo este el caso base

LCP

input : S es la secuencia de caracteres en i y j indican los límites
output: l longitud de la mayor secuencia palindrómica

if $i == j$ **then**
  return 1
end
if $S[i] == S[j]$ **then**
  return $2 + LCP(i+1, j-1)$
end
else
  return $\max\{LCP(i, j-1), LCP(i+1, j)\}$
end

Complejidad

- $T(n) = 2T(n-1) = 2^{n-1}$

Complejidad

- $T(n) = 2T(n-1) = 2^{n-1}$
- Hay $\binom{n}{2}$ subproblemas distintos, uno para cada par (i, j) tal que $i < j$.

Complejidad

- $T(n) = 2T(n-1) = 2^{n-1}$
- Hay $\binom{n}{2}$ subproblemas distintos, uno para cada par (i, j) tal que $i < j$.
- Resolviendo cada problema una sola vez, $\Theta(n^2) \times \theta(1) = \Theta(n^2)$, dado que cada subproblema tenga los sub-subproblemas resueltos.

Memorización vs Iteración

- En general, memorización utiliza un diccionario (map) donde para cada para i, j se almacena el valor de L. En particular, para este ejemplo basta con un arreglo bidimensional.

Memorización vs Iteración

- En general, memorización utiliza un diccionario (map) donde para cada para i, j se almacena el valor de L. En particular, para este ejemplo basta con un arreglo bidimensional.
- La otra alternativa es resolver iterativamente, desde los problemas más pequeños a los más grandes.

Memorización vs Iteración

- En general, memorización utiliza un diccionario (map) donde para cada para i, j se almacena el valor de L. En particular, para este ejemplo basta con un arreglo bidimensional.
- La otra alternativa es resolver iterativamente, desde los problemas más pequeños a los más grandes.
- Teniendo la misma complejidad temporal pero diferente complejidad espacial.

Memorización vs Iteración

- En general, memorización utiliza un diccionario (map) donde para cada para i, j se almacena el valor de L. En particular, para este ejemplo basta con un arreglo bidimensional.
- La otra alternativa es resolver iterativamente, desde los problemas más pequeños a los más grandes.
- Teniendo la misma complejidad temporal pero diferente complejidad espacial.
- En algunos casos, se puede mejorar la cantidad de memoria necesaria manteniendo únicamente las partes necesarias de la matriz

Juego de la moneda

- Dadas n monedas con valores $v_1, v_2, ..., v_n$, siendo n par; se tiene un juego por turnos para 2 personas

Juego de la moneda

- Dadas n monedas con valores $v_1, v_2, ..., v_n$, siendo n par; se tiene un juego por turnos para 2 personas
- En cada turno, el jugador toma la primera o la última moneda, recibiendo el valor de dicha moneda

Juego de la moneda

- Dadas n monedas con valores $v_1, v_2, ..., v_n$, siendo n par; se tiene un juego por turnos para 2 personas
- En cada turno, el jugador toma la primera o la última moneda, recibiendo el valor de dicha moneda
- Gana el jugador que obtiene un valor mayor

Juego de la moneda

- Dadas n monedas con valores $v_1, v_2, ..., v_n$, siendo n par; se tiene un juego por turnos para 2 personas
- En cada turno, el jugador toma la primera o la última moneda, recibiendo el valor de dicha moneda
- Gana el jugador que obtiene un valor mayor
- ¿El jugador que inicia la partida podrá ganar siempre?

Juego de la moneda: ejemplos

- $4, 42, 39, 17, 25, 6$

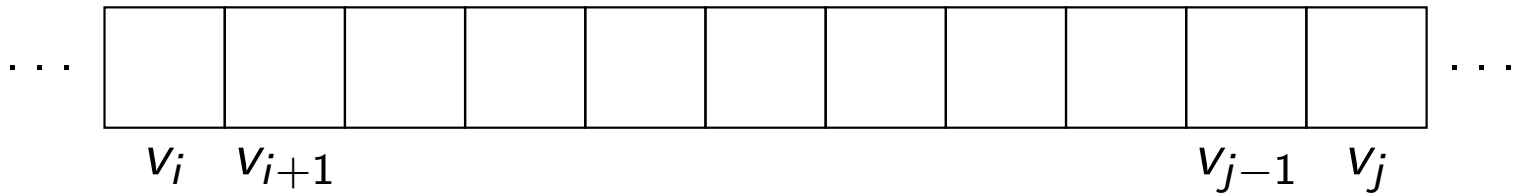
Juego de la moneda: ejemplos

- $4, 42, 39, 17, 25, 6$
- $8, 15, 3, 7$

Juego de la moneda: ejemplos

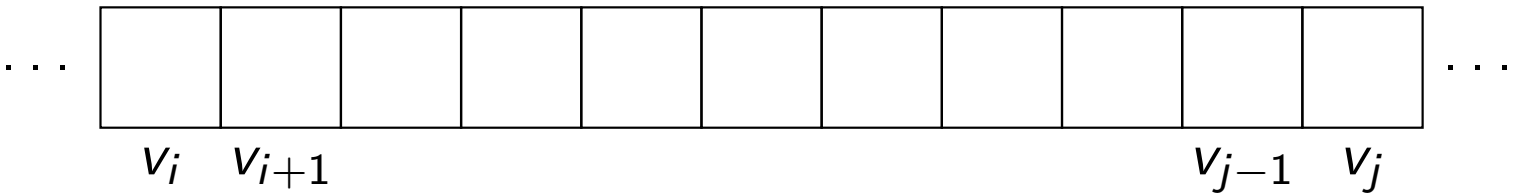
- $4, 42, 39, 17, 25, 6$
- $8, 15, 3, 7$
- ¿Una estrategia greedy funciona?

Juego de la moneda



- Sea $V(i, j)$ el valor máximo que podemos ganar en nuestro turno cuando quedan las monedas v_i hasta v_j

Juego de la moneda



- Sea $V(i, j)$ el valor máximo que podemos ganar en nuestro turno cuando quedan las monedas v_i hasta v_j
- Entonces, $V(i, j)$ tiene dos opciones: o seleccionamos v_i o seleccionamos v_j

Juego de la moneda

- El oponente tratará de hacer la mejor jugada posible

Juego de la moneda

- El oponente tratará de hacer la mejor jugada posible
- Por ello, luego de nosotros haber seleccionado una moneda, el oponente nos dejará con el menor valor posible de las monedas que quedan

Juego de la moneda

- El oponente tratará de hacer la mejor jugada posible
- Por ello, luego de nosotros haber seleccionado una moneda, el oponente nos dejará con el menor valor posible de las monedas que quedan
- Entonces, si nosotros seleccionamos v_i, el oponente seleccionará tal que lo que nos quede sea el mínimo entre (v_{i+2}, v_j) y (v_{i+1}, v_{j-1})

Juego de la moneda

- El oponente tratará de hacer la mejor jugada posible
- Por ello, luego de nosotros haber seleccionado una moneda, el oponente nos dejará con el menor valor posible de las monedas que quedan
- Entonces, si nosotros seleccionamos v_i, el oponente seleccionará tal que lo que nos quede sea el mínimo entre (v_{i+2}, v_j) y (v_{i+1}, v_{j-1})
- Analogamente, si nosotros seleccionamos v_j, el oponente seleccionará tal que lo que nos quede sea el mínimo entre (v_i, v_{j-2}) y (v_{i+1}, v_{j-1})

Complejidad

- $V(i,j) = \max\Big\{ v_i + \min\{V(i+1,j-1), V(i+2,j)\}, v_j + \min\{V(i,j-2), V(i+1,j-1)\} \Big\}$

Complejidad

- $V(i,j) = \max\Big\{ v_i + \min\{V(i+1,j-1), V(i+2,j)\}, v_j + \min\{V(i,j-2), V(i+1,j-1)\} \Big\}$
- Hay $\binom{n}{2}$ subproblemas distintos

Complejidad

- $V(i,j) = \max\Big\{ v_i + \min\{V(i+1,j-1), V(i+2,j)\}, v_j + \min\{V(i,j-2), V(i+1,j-1)\} \Big\}$
- Hay $\binom{n}{2}$ subproblemas distintos
- Resolviendo cada problema una sola vez, $\Theta(n^2) \times \theta(1) = \Theta(n^2)$, dado que cadasubproblema tenga los sub-subproblemas resueltos.

Jugada

input : V los valores de las n monedas, i y j los límites
output: Secuencia de monedas a elegir

if $i == j$ **then**
| **return** $V[i]$
end
$jugada1 \leftarrow \min(Jugada(i+2, j), Jugada(i+1, j-1)) + V[i]$;
$jugada2 \leftarrow \min(Jugada(i+1, j-1), Jugada(i, j-2)) + V[j]$;
return $\max\{jugada1, jugada2\}$

Top-down vs Bottom-up

- Ambas versiones son equivalentes, es decir, ambas solucionan el problema correctamente

Top-down vs Bottom-up

- Ambas versiones son equivalentes, es decir, ambas solucionan el problema correctamente
- En términos de complejidad temporal, son equivalentes

Top-down vs Bottom-up

- Ambas versiones son equivalentes, es decir, ambas solucionan el problema correctamente
- En términos de complejidad temporal, son equivalentes
- Sin embargo, difieren en términos de complejidad espacial

Top-down vs Bottom-up

- Ambas versiones son equivalentes, es decir, ambas solucionan el problema correctamente
- En términos de complejidad temporal, son equivalentes
- Sin embargo, difieren en términos de complejidad espacial
- Para que la solución sea eficiente, la forma top-down usualmente necesita de memorización