Backtracking

Algoritmos y Estructuras de Datos III

Santiago Cifuentes

Departamento de computación
FCEN – UBA

Agosto 2022

Técnicas algorítmicas

- Fuerza Bruta / Búsqueda exhaustiva
- *Backtracking*
- *Divide&Conquer*
- Algoritmos Golosos
- Programación Dinámica
- Heurísticas y algoritmos aproximados.

Técnicas algorítmicas

- Fuerza Bruta / Búsqueda exhaustiva
- *Backtracking*
- *Divide&Conquer*
- Algoritmos Golosos
- Programación Dinámica
- Heurísticas y algoritmos aproximados.

Técnicas algorítmicas

- Fuerza Bruta / Búsqueda exhaustiva
- *Backtracking*
- *Divide&Conquer*
- Algoritmos Golosos
- Programación Dinámica
- Heurísticas y algoritmos aproximados.

Fuerza Bruta / Búsqueda exhaustiva

- Para problemas de búsqueda en un conjunto S.

Fuerza Bruta / Búsqueda exhaustiva

- Para problemas de búsqueda en un conjunto S.
- Queremos hacer algo con los elementos que cumpla unan propiedad P.

Fuerza Bruta / Búsqueda exhaustiva

- Para problemas de búsqueda en un conjunto S.
- Queremos hacer algo con los elementos que cumpla unan propiedad P.
- La técnica más simple: recorremos todo S evaluando P en cada elemento.

Fuerza Bruta / Búsqueda exhaustiva

- Para problemas de búsqueda en un conjunto S.
- Queremos hacer algo con los elementos que cumpla unan propiedad P.
- La técnica más simple: recorremos todo S evaluando P en cada elemento.
- La complejidad en general será $\Omega(|S|)$.

Idea de Fuerza Bruta

```
for x ∈ S do:
    if P(x):
        procesar x
```

- Hay que definir quiénes son S, P y **procesar**.

Idea de Fuerza Bruta

```
for x ∈ S do:
    if P(x):
      procesar x
```

- Hay que definir quiénes son S, P y **procesar**.
- Ejemplo: S es el conjunto de tableros de ajedrez con 8 reinas, P verifica que no se ataquen entre ellas, y **procesar** lleva la cuenta de la cantidad de tableros.

Idea de Fuerza Bruta

for x $\in$ S **do**:
  if P(x):
    procesar x

- Hay que definir quiénes son S, P y **procesar**.
- Ejemplo: S es el conjunto de tableros de ajedrez con 8 reinas, P verifica que no se ataquen entre ellas, y **procesar** lleva la cuenta de la cantidad de tableros.
- ¿Cómo se genera S?

Backtracking

- Es una técnica útil para generar espacios de búsqueda "recursivos". En particular, lo hace mediante la extensión de **soluciones parciales**.

Backtracking

- Es una técnica útil para generar espacios de búsqueda "recursivos". En particular, lo hace mediante la extensión de **soluciones parciales**.
- La idea es definir este método de extensión, y mediante recursión generar *de forma ordenada* el espacio de soluciones S.

Backtracking

- Es una técnica útil para generar espacios de búsqueda "recursivos". En particular, lo hace mediante la extensión de **soluciones parciales**.
- La idea es definir este método de extensión, y mediante recursión generar *de forma ordenada* el espacio de soluciones S.
- La extensión muchas veces es una operación "local", y es fácil de definir e implementar.

Backtracking

```
algoritmo BT(a,k)
    si a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a' ∈ Sucesores(a,k)
            BT(a',k+1)
        fin para
    fin si
    retornar
```

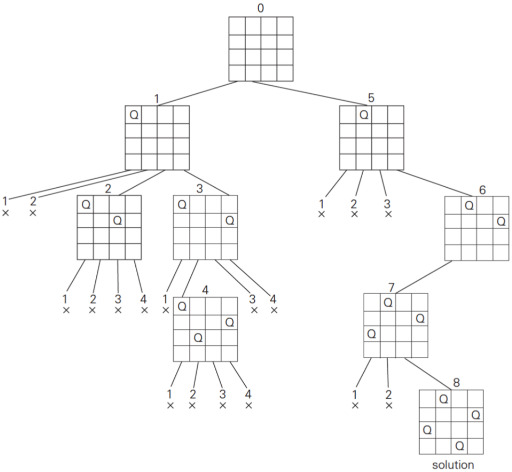
- La generación de S se redujo a implementar *Sucesores*.

Backtracking

```
algoritmo BT(a, k)
    si a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a' ∈ Sucesores(a, k)
            BT(a', k + 1)
        fin para
    fin si
    retornar
```

- La generación de S se redujo a implementar *Sucesores*.
- ¿Cómo es *Sucesores* para el caso del problema de las reinas?

Backtracking



Robado de TowardsDataScience

Ejercicio: CD

Enunciado

Tenemos un CD que soporta hasta P minutos de música, y dado un conjunto de N canciones de duración p_i (con $1 \leq i \leq m$, y $p_i \in \mathbb{N}$) queremos encontrar la mayor cantidad de minutos de música que podemos escuchar.

Ejercicio: CD

Enunciado

Tenemos un CD que soporta hasta P minutos de música, y dado un conjunto de N canciones de duración p_i (con $1 \leq i \leq m$, y $p_i \in \mathbb{N}$) queremos encontrar la mayor cantidad de minutos de música que podemos escuchar.

Con $P = 5$ y una lista de $N = 3$ canciones con duraciones [1, 4, 2] la solución es 5.

Ejemplo: CD

- ¿Podemos definir un espacio de búsqueda? ¿Qué queremos hacer con cada solución?

Ejemplo: CD

- ¿Podemos definir un espacio de búsqueda? ¿Qué queremos hacer con cada solución?
- Podemos considerar todos los subconjuntos, y quedarnos con el que maximice la suma de minutos de música sin exceder N (este es S) ¿Cuántos hay?

Ejemplo: CD

- ¿Podemos definir un espacio de búsqueda? ¿Qué queremos hacer con cada solución?
- Podemos considerar todos los subconjuntos, y quedarnos con el que maximice la suma de minutos de música sin exceder N (este es S) ¿Cuántos hay?
- Hay que considerar 2^N subconjuntos, y para cada uno calcular la suma.

Ejercicio: CD

- Generemos S recursivamente, usando backtracking.

Ejercicio: CD

- Generemos S recursivamente, usando backtracking.
- ¿Hay una forma recursiva de generar los subconjuntos de un conjunto?

Ejercicio: CD

- Generemos S recursivamente, usando backtracking.
- ¿Hay una forma recursiva de generar los subconjuntos de un conjunto?

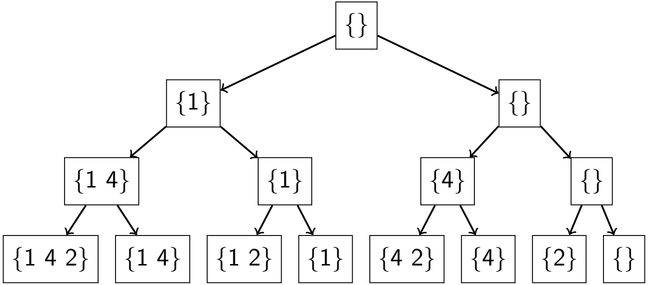
Ejercicio: CD

- Generemos S recursivamente, usando backtracking.
- ¿Hay una forma recursiva de generar los subconjuntos de un conjunto?

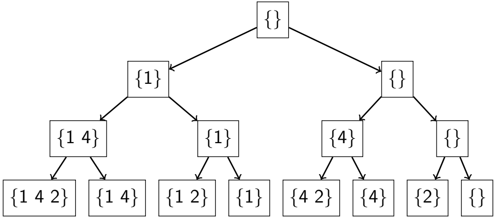
$$subsets(c : C) = c \times subsets(C) \cup subsets(C)$$

Ejercicio: CD

- Cada elemento puede o no estar en el subconjunto.

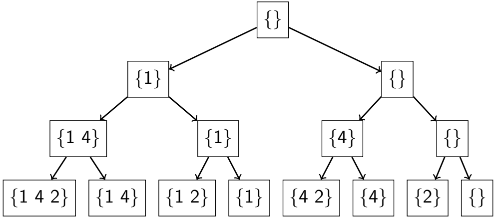


Ejercicio: CD



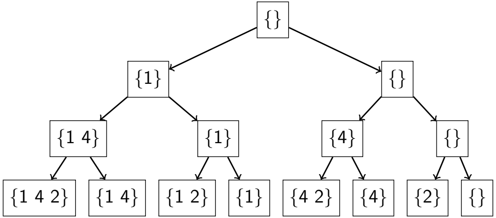
- ¿Qué representan las hojas?

Ejercicio: CD



- ¿Qué representan las hojas?
- ¿Qué representan los nodos del i-ésimo piso?

Ejercicio: CD



- ¿Qué representan las hojas?
- ¿Qué representan los nodos del i-ésimo piso?
- Cada nodo interno del nivel i representa un subconjunto de los primeros i elementos. Por ende para extender cada solución se agrega o no el elemento $i + 1$.

Pseudocódigo de CD

Algorithm $BT_{CD}(a, i)$ // a es una solución parcial

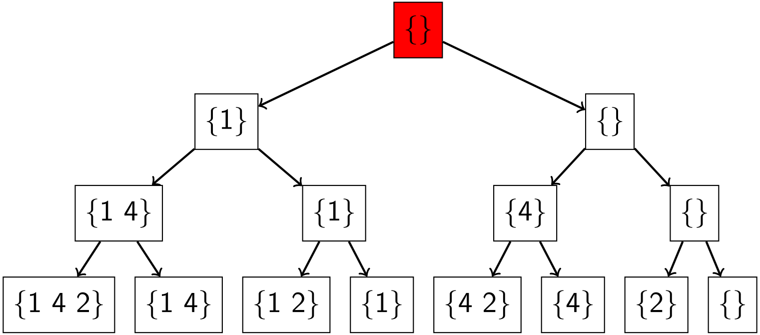
if $i = N$ **then**
    if $suma(a) \leq P$ & $suma(a) > mejorSuma$ **then**
        $mejorSuma \leftarrow suma(a)$
    end if
else
    $BT_{CD}(a \cup p_i, i + 1)$
    $BT_{CD}(a, i + 1)$
end if$=0$

- La respuesta es $BT_{CD}(\{\}, 0)$

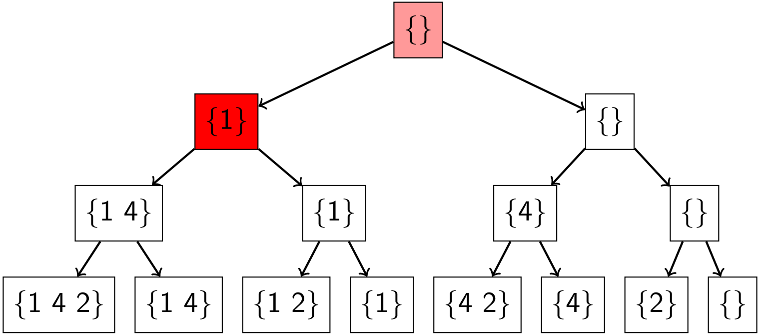
Pseudocódigo de Backtracking

```
algoritmo BT(a, k)
    si a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a' ∈ Sucesores(a, k)
            BT(a', k + 1)
        fin para
    fin si
    retornar
```

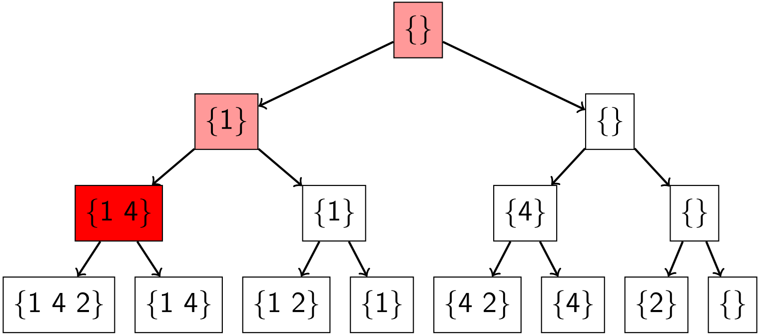
Recorrido en profundidad



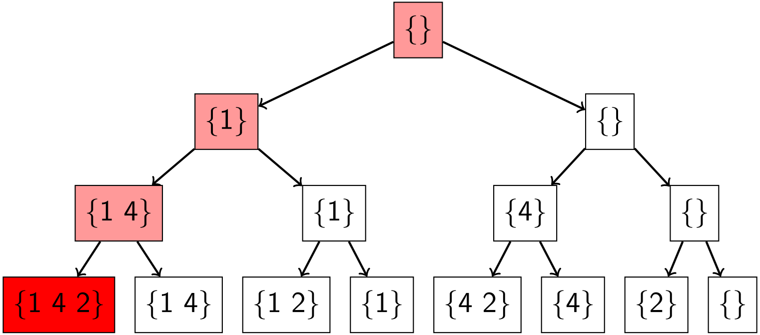
Recorrido en profundidad



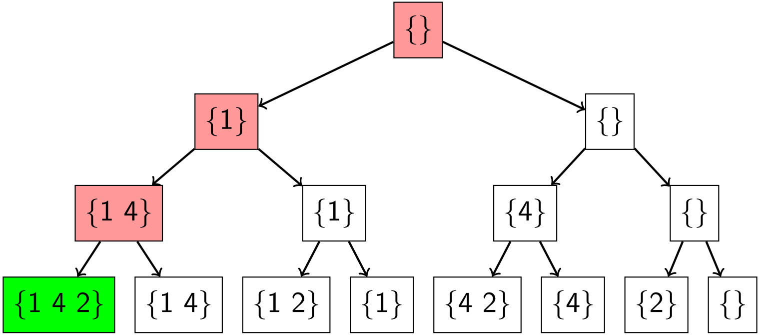
Recorrido en profundidad



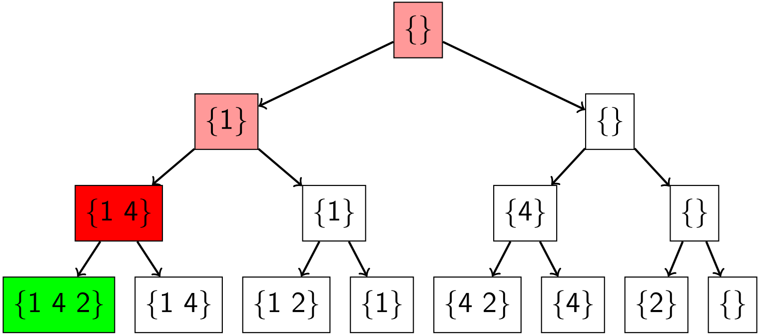
Recorrido en profundidad



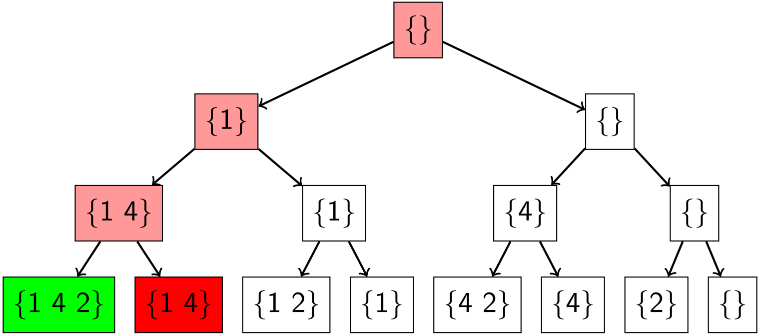
Recorrido en profundidad



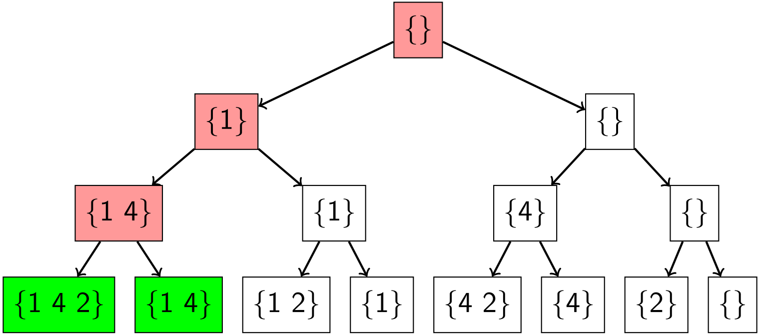
Recorrido en profundidad



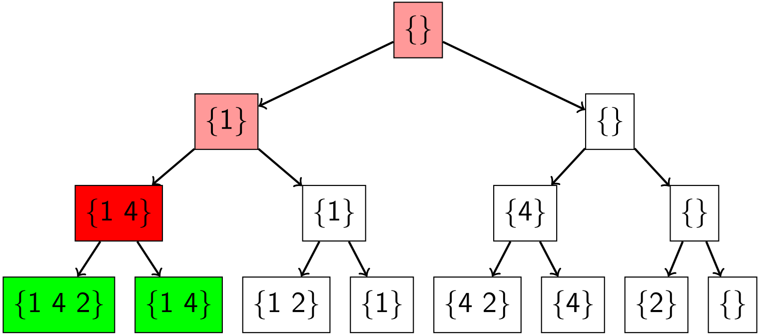
Recorrido en profundidad



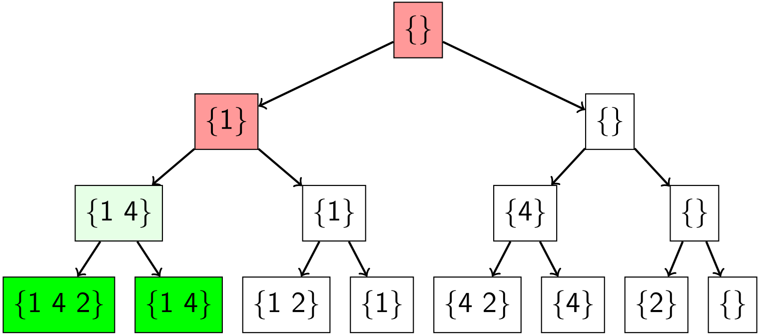
Recorrido en profundidad



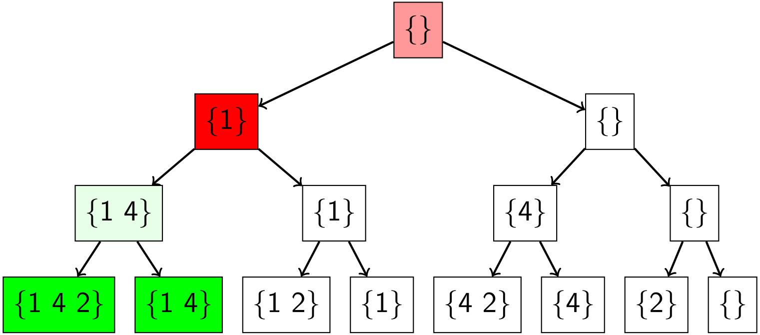
Recorrido en profundidad



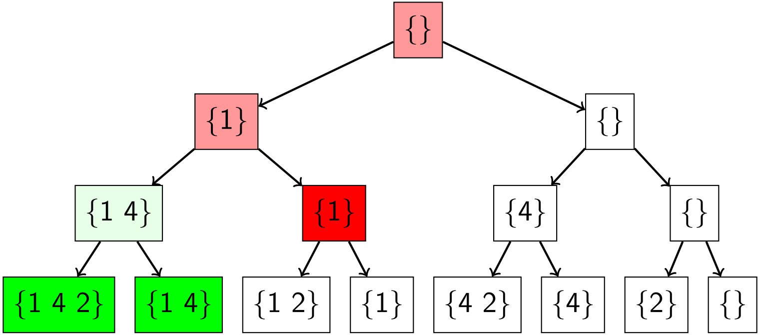
Recorrido en profundidad



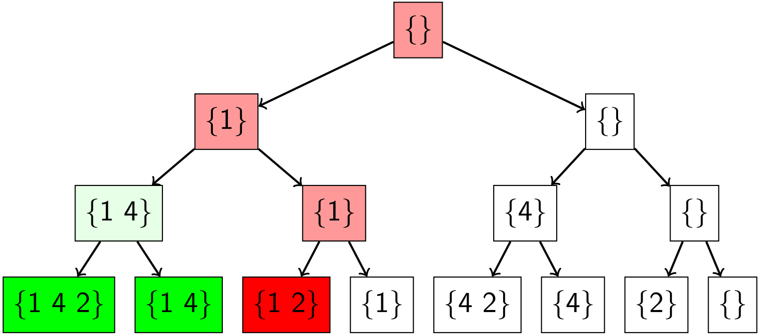
Recorrido en profundidad



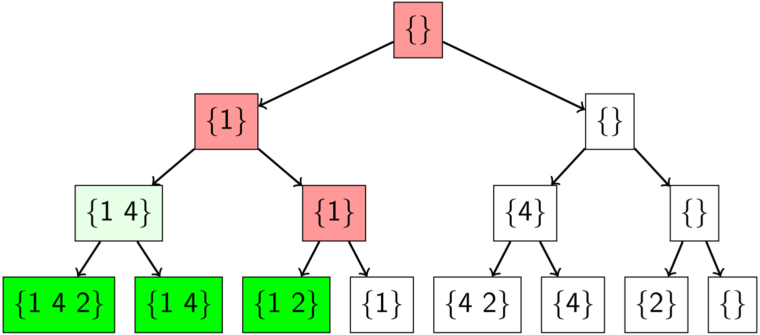
Recorrido en profundidad



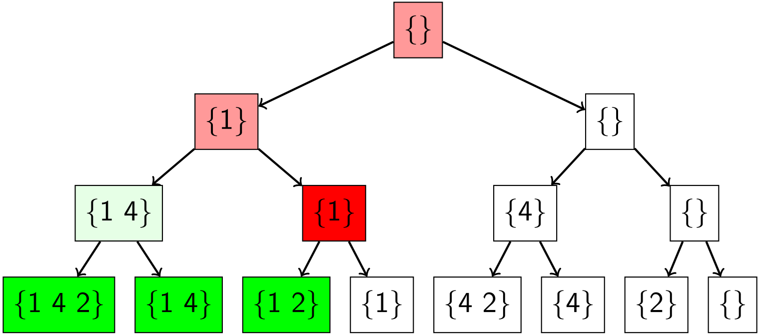
Recorrido en profundidad



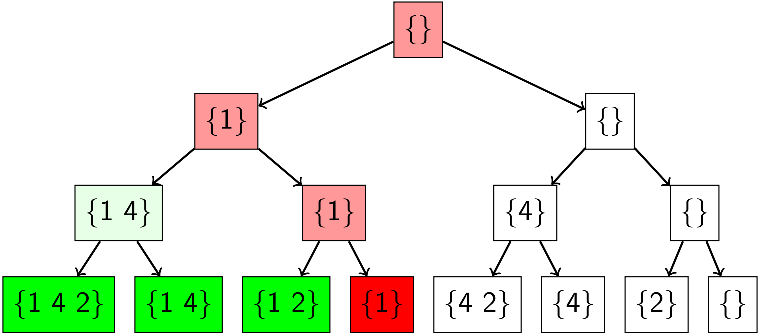
Recorrido en profundidad



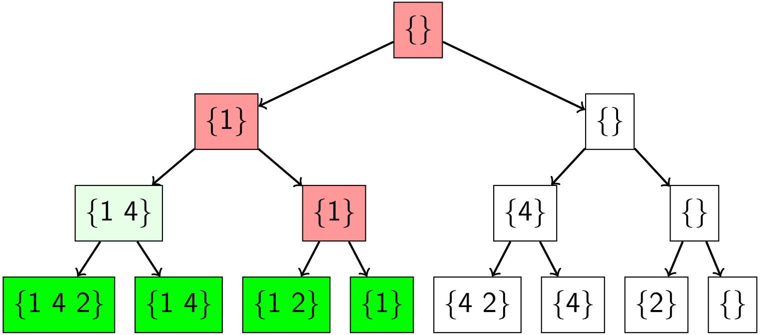
Recorrido en profundidad



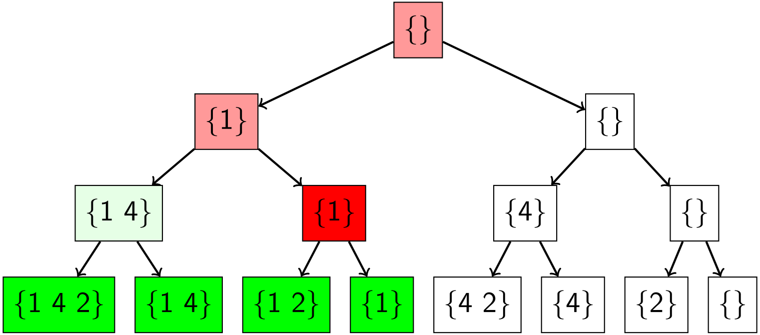
Recorrido en profundidad



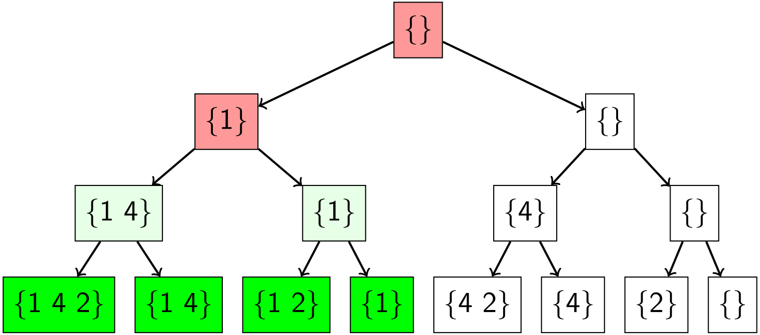
Recorrido en profundidad



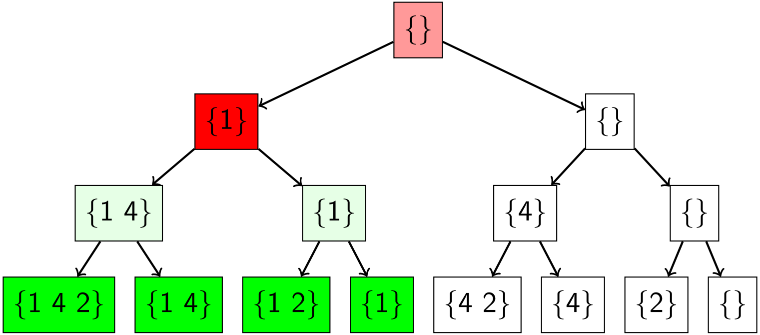
Recorrido en profundidad



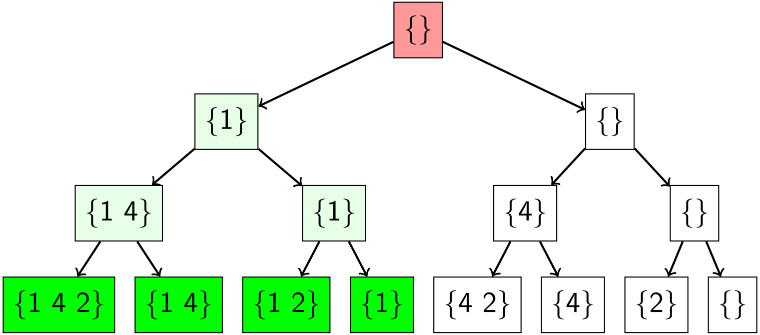
Recorrido en profundidad



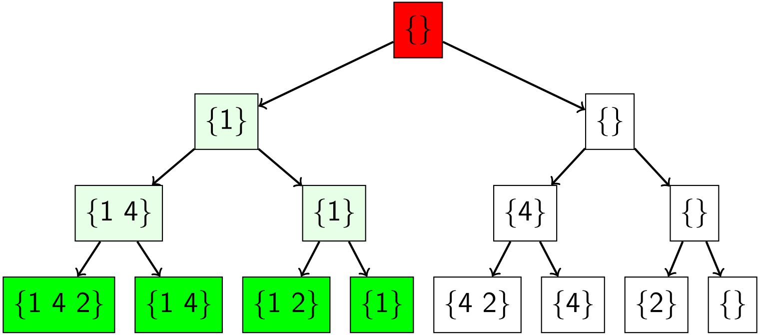
Recorrido en profundidad



Recorrido en profundidad



Recorrido en profundidad



Función recursiva de CD

El problema se puede pensar de otra forma: si quiero llegar a P, y agrego el primer elemento al CD ¿A cuánto quiero llegar con el resto de los temas?

Función recursiva de CD

El problema se puede pensar de otra forma: si quiero llegar a P, y agrego el primer elemento al CD ¿A cuánto quiero llegar con el resto de los temas? ¿Y si no lo agrego?

Función recursiva de CD

El problema se puede pensar de otra forma: si quiero llegar a P, y agrego el primer elemento al CD ¿A cuánto quiero llegar con el resto de los temas? ¿Y si no lo agrego? Si no me quedan temas (N=0), ¿Qué valores de P son válidos?

Función recursiva de CD

El problema se puede pensar de otra forma: si quiero llegar a P, y agrego el primer elemento al CD ¿A cuánto quiero llegar con el resto de los temas? ¿Y si no lo agrego? Si no me quedan temas (N=0), ¿Qué valores de P son válidos?

$$CD(i,k) = \begin{cases} -\infty & \text{si } i = N \text{ y } k < 0 \\ 0 & \text{si } i = N \text{ y } k \geq 0 \\ \max(CD(i+1,k), CD(i+1,k-p_i)+p_i) & \text{cc} \end{cases}$$

'La máxima cantidad de música que puedo obtener sin exceder k minutos empleando las canciones desde i hacia delante'

Función recursiva de CD

- Plantear funciones de esta pinta nos va a ser muy útil cuando hagamos programación dinámica.

Función recursiva de CD

- Plantear funciones de esta pinta nos va a ser muy útil cuando hagamos programación dinámica.
- Intuitivamente, el árbol de recursión de esta función es el mismo que el de los subconjuntos.

Función recursiva de CD

- Plantear funciones de esta pinta nos va a ser muy útil cuando hagamos programación dinámica.
- Intuitivamente, el árbol de recursión de esta función es el mismo que el de los subconjuntos.
- Sin embargo, en esta formulación queda claro que no importan qué elementos se fueron eligiendo, sino la suma de los pesos.

Podas

- Aprovechando la estructura del árbol podemos ‘podar‘ ramas que no nos lleven a soluciones útiles.

Podas

- Aprovechando la estructura del árbol podemos ‘podar‘ ramas que no nos lleven a soluciones útiles.
- Hay que tener en cuenta el *overhead* que genera computarlas.

Podas

- Aprovechando la estructura del árbol podemos ‘podar‘ ramas que no nos lleven a soluciones útiles.
- Hay que tener en cuenta el *overhead* que genera computarlas.
- Pueden (o no) recortar significativamente el espacio de búsqueda.

Podas para CD

- ¿Qué podas podemos usar en CD?

Podas para CD

- ¿Qué podas podemos usar en CD?
- En CD podemos dejar de avanzar si la solución parcial ya superó N (**factibilidad)**.

Podas para CD

- ¿Qué podas podemos usar en CD?
- En CD podemos dejar de avanzar si la solución parcial ya superó N (**factibilidad)**.
- También podemos ver si poniendo todas las canciones restantes no nos excedemos de k entonces. Si ese es el caso, la mejor solución desde donde estamos es agregar todo (**optimalidad**).

Podas para CD

$$CD(i,k) = \begin{cases} -\infty & \text{si } k < 0 \\ sumaRestante(i) & \text{si } sumaRestante(i) <= k \\ \max(CD(i+1,k), CD(i+1,k-p_i)+p_i) & \text{cc} \end{cases}$$

Complejidad de la solución de CD

- ¿Cuántos nodos tiene el árbol que estamos recorriendo?

Complejidad de la solución de CD

- ¿Cuántos nodos tiene el árbol que estamos recorriendo?
- Tiene $\sum_{i=0}^{N} 2^i = O(2^N)$ nodos.

Complejidad de la solución de CD

- ¿Cuántos nodos tiene el árbol que estamos recorriendo?
- Tiene $\sum_{i=0}^{N} 2^i = O(2^N)$ nodos.
- ¿Cuántas operaciones hacemos en cada nodo?

Complejidad de la solución de CD

- ¿Cuántos nodos tiene el árbol que estamos recorriendo?
- Tiene $\sum_{i=0}^{N} 2^i = O(2^N)$ nodos.
- ¿Cuántas operaciones hacemos en cada nodo?
- En todos los nodos internos realizamos una cantidad constante de operaciones.

Complejidad de la solución de CD

- ¿Cuántos nodos tiene el árbol que estamos recorriendo?
- Tiene $\sum_{i=0}^{N} 2^i = O(2^N)$ nodos.
- ¿Cuántas operaciones hacemos en cada nodo?
- En todos los nodos internos realizamos una cantidad constante de operaciones.
- La complejidad final entonces es $O(2^N)$.

Ejercicio: *Prime ring*

Prime Ring

Dados N números naturales $p_0, \ldots, p_{N-1}$, con $1 < p_i < 10N$, queremos saber cuántas permutaciones j de ellos hay que cumplan que $p_{j_i} + p_{(j_{i+1} \mod n)}$ sea primo para todo $0 \leq i \leq n-1$

- Lo vamos a resolver con backtracking.

Ejercicio: *Prime ring*

Prime Ring

Dados N números naturales $p_0, \ldots, p_{N-1}$, con $1 < p_i < 10N$, queremos saber cuántas permutaciones j de ellos hay que cumplan que $p_{j_i} + p_{(j_{i+1} \bmod n)}$ sea primo para todo $0 \leq i \leq n-1$

- Lo vamos a resolver con backtracking.
- ¿Cuál es el espacio de búsqueda?

Ejercicio: *Prime ring*

Prime Ring

Dados N números naturales $p_0, \ldots, p_{N-1}$, con $1 < p_i < 10N$, queremos saber cuántas permutaciones j de ellos hay que cumplan que $p_{j_i} + p_{(j_{i+1} \mod n)}$ sea primo para todo $0 \leq i \leq n-1$

- Lo vamos a resolver con backtracking.
- ¿Cuál es el espacio de búsqueda?
- ¿Cuáles son las soluciones parciales?

Ejercicio: *Prime ring*

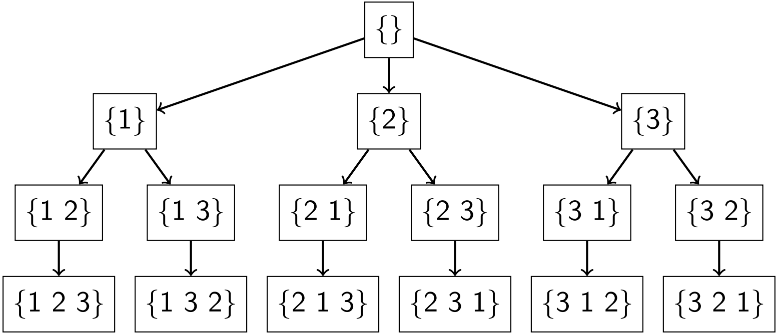
Prime Ring

Dados N números naturales $p_0, \ldots, p_{N-1}$, con $1 < p_i < 10N$, queremos saber cuántas permutaciones j de ellos hay que cumplan que $p_{j_i} + p_{(j_{i+1} \mod n)}$ sea primo para todo $0 \leq i \leq n-1$

- Lo vamos a resolver con backtracking.
- ¿Cuál es el espacio de búsqueda?
- ¿Cuáles son las soluciones parciales?
- ¿Cuál es la operación de extensión?

Árbol de *Prime ring*

- Árbol para $n = 3$, si $p = [1, 2, 3]$.



Árbol de *Prime ring*

- Cada nodo interno del piso I es una permutación de un subconjunto de i números.

Árbol de *Prime ring*

- Cada nodo interno del piso l es una permutación de un subconjunto de i números.
- En las hojas verificamos que se cumpla la condición de primalidad.

Prime ring

La función que hay que implementar entonces es:

$$primeRing(I) = \begin{cases} esValida(I) & \text{si } |I| = N \\ \sum_{p_i \notin I} primeRing(I \oplus p_i) & \text{cc} \end{cases}$$

'La cantidad de permutaciones que extienden a I, usan todos los elementos de p y generan un anillo de primos'

Prime ring

La función que hay que implementar entonces es:

$$primeRing(I) = \begin{cases} esValida(I) & \text{si } |I| = N \\ \sum_{p_i \notin I} primeRing(I \oplus p_i) & \text{cc} \end{cases}$$

'La cantidad de permutaciones que extienden a I, usan todos los elementos de p y generan un anillo de primos'

La solución al problema es $primeRing(\{\})$

Podas

- ¿Podas?
- Podríamos verificar la condición de primalidad durante la selección de sucesores.
- El árbol cambia: ahora las soluciones parciales son las permutaciones de los subconjuntos que cumplen la condición de primalidad.
- ¿Hay que verificar algo en las hojas?
- En las hojas solo tenemos que verificar que cierre bien el anillo.

Podas

La función queda entonces como

$$primeRing(l) = \begin{cases} esPrimo(ultimo(l) + primero(l)) & \text{si } |l| = N \\ \displaystyle\sum_{\substack{p_i \notin l \\ esPrimo(p_i + ultimo(l))}} primeRing(l \oplus p_i) & \text{cc} \end{cases}$$

Podas

La función queda entonces como

$$primeRing(l) = \begin{cases} esPrimo(ultimo(l) + primero(l)) & \text{si } |l| = N \\ \displaystyle\sum_{\substack{p_i \notin l \\ esPrimo(p_i + ultimo(l))}} primeRing(l \oplus p_i) & \text{cc} \end{cases}$$

Más podas

- Hay algunas podas interesantes que se pueden usar debido a que este es un problema de conteo.

Más podas

- Hay algunas podas interesantes que se pueden usar debido a que este es un problema de conteo.
- Podemos explotar simetrías: dada una permutación válida, se pueden obtener otras moviendo los elementos a la derecha x unidades.

Más podas

- Hay algunas podas interesantes que se pueden usar debido a que este es un problema de conteo.
- Podemos explotar simetrías: dada una permutación válida, se pueden obtener otras moviendo los elementos a la derecha x unidades.
- Podemos suponer fijo el primer elemento, y multiplicar por N la cantidad de permutaciones con ese elemento primero.

Más podas

- Hay algunas podas interesantes que se pueden usar debido a que este es un problema de conteo.
- Podemos explotar simetrías: dada una permutación válida, se pueden obtener otras moviendo los elementos a la derecha x unidades.
- Podemos suponer fijo el primer elemento, y multiplicar por N la cantidad de permutaciones con ese elemento primero.
- ¿Y qué pasa respecto a la paridad de los números continuos?

Más podas

- Hay algunas podas interesantes que se pueden usar debido a que este es un problema de conteo.
- Podemos explotar simetrías: dada una permutación válida, se pueden obtener otras moviendo los elementos a la derecha x unidades.
- Podemos suponer fijo el primer elemento, y multiplicar por N la cantidad de permutaciones con ese elemento primero.
- ¿Y qué pasa respecto a la paridad de los números continuos?
- Tiene que haber tantos pares como impares (y, por lo tanto, la cantidad de números debe ser par). Aparte, pares e impares están "interlineados" en las permutaciones válidas.

Más podas

La función queda entonces como

$$primeRing(I) = \begin{cases} esPrimo(ultimo(I) + primero(I)) & \text{si } |I| = N \\ \displaystyle\sum_{\substack{i \notin I \\ esPrimo(i+ultimo(I))}} primeRing(I \oplus i) & \text{cc} \end{cases}$$

La función no cambia, pero ahora sabemos que la solución se puede escribir como $N * primeRing([p_0])$ (y antes verificamos la cantidad de números pares e impares).

Complejidad de la solución de *Prime Ring*

- ¿Cuántos nodos tiene el árbol de recursión?

[1] Esto se puede hacer en $O(n)$, ver `https://cp-algorithms.com/algebra/prime-sieve-linear.html`

Complejidad de la solución de *Prime Ring*

- ¿Cuántos nodos tiene el árbol de recursión?
- El árbol tiene $O(n-1!)$ nodos (en la práctica tienen que demostrar un caso similar).

[1]Esto se puede hacer en $O(n)$, ver `https://cp-algorithms.com/algebra/prime-sieve-linear.html`

Complejidad de la solución de *Prime Ring*

- ¿Cuántos nodos tiene el árbol de recursión?
- El árbol tiene $O(n-1!)$ nodos (en la práctica tienen que demostrar un caso similar).
- En cada nodo hacemos $O(n)$ operaciones, y en particular $O(n)$ llamados a *esPrimo*.

[1] Esto se puede hacer en $O(n)$, ver `https://cp-algorithms.com/algebra/prime-sieve-linear.html`

Complejidad de la solución de *Prime Ring*

- ¿Cuántos nodos tiene el árbol de recursión?
- El árbol tiene $O(n - 1!)$ nodos (en la práctica tienen que demostrar un caso similar).
- En cada nodo hacemos $O(n)$ operaciones, y en particular $O(n)$ llamados a *esPrimo*.
- Si *esPrimo* es $O(1)$ (podemos precalcular la criba de Heratóstenes hasta $20n$ en $O(n \log \log n)$[1]), la complejidad final es $O(n \log \log n + (n-1)!\ n) = O(n!)$.

[1] Esto se puede hacer en $O(n)$, ver `https://cp-algorithms.com/algebra/prime-sieve-linear.html`

Detalles adicionales

- En realidad la complejidad es menor, ya que en cada paso hay a lo sumo $\frac{n}{2}$ opciones por la paridad.

Detalles adicionales

- En realidad la complejidad es menor, ya que en cada paso hay a lo sumo $\frac{n}{2}$ opciones por la paridad.
- Aparte, queda por explotar la simetría que surge de invertir las soluciones.

Sudoku

Enunciado

Dado un tablero de Sudoku de $N \times N$ con algunas casillas ocupadas hay que decidir si se puede completar de forma que sea el resultado final sea un tablero válido.

Sudoku

Enunciado

Dado un tablero de Sudoku de $N \times N$ con algunas casillas ocupadas hay que decidir si se puede completar de forma que sea el resultado final sea un tablero válido.

- ¿Cuáles son las soluciones parciales?

Sudoku

Enunciado

Dado un tablero de Sudoku de $N \times N$ con algunas casillas ocupadas hay que decidir si se puede completar de forma que sea el resultado final sea un tablero válido.

- ¿Cuáles son las soluciones parciales?
- ¿Cuál es la función de extensión?

Sudoku

Enunciado

Dado un tablero de Sudoku de $N \times N$ con algunas casillas ocupadas hay que decidir si se puede completar de forma que sea el resultado final sea un tablero válido.

- ¿Cuáles son las soluciones parciales?
- ¿Cuál es la función de extensión?
- ¿Qué verificamos en las hojas?

Sudoku

$$sudoku(T,(i,j)) = \begin{cases} esValido(T) & \text{si } i = N \\ sudoku(T, sig(i,j)) & \text{si } T[i][j] \neq 0 \\ \bigvee\limits_{1 \leq k \leq N} sudoku(T \oplus ((i,j) \to k), sig(i,j)) & \text{cc} \end{cases}$$

Sudoku

$$sudoku(T,(i,j)) = \begin{cases} esValido(T) & \text{si } i = N \\ sudoku(T, sig(i,j)) & \text{si } T[i][j] \neq 0 \\ \bigvee\limits_{1 \leq k \leq N} sudoku(T \oplus ((i,j) \to k), sig(i,j)) & \text{cc} \end{cases}$$

La solución es $sudoku(T, 0, 0)$

Sudoku

- ¿Cuántos nodos tiene el árbol?

Sudoku

- ¿Cuántos nodos tiene el árbol?
- ¿Cuántas operaciones hacemos en los nodos internos?

Sudoku

- ¿Cuántos nodos tiene el árbol?
- ¿Cuántas operaciones hacemos en los nodos internos?
- ¿Y en las hojas?

Sudoku

- ¿Cuántos nodos tiene el árbol?
- ¿Cuántas operaciones hacemos en los nodos internos?
- ¿Y en las hojas?
- La complejidad final se puede acotar por $O(n^{n^2} n^2)$

Podas

- ¿Qué podas podemos implementar?

Podas

- ¿Qué podas podemos implementar?
- No pongamos números que ya están prohibidos. Para eso revisamos las filas, columnas y subcuadrados en cada paso.

Podas

- ¿Qué podas podemos implementar?
- No pongamos números que ya están prohibidos. Para eso revisamos las filas, columnas y subcuadrados en cada paso.
- ¿Hace falta ir en orden¿

Podas

- ¿Qué podas podemos implementar?
- No pongamos números que ya están prohibidos. Para eso revisamos las filas, columnas y subcuadrados en cada paso.
- ¿Hace falta ir en orden¿
- No, usemos siempre la posición mas condicionada.

Sudoku

$$\text{sudoku(T)} = \begin{cases} esValido(T) & \text{si } mas\_cond(T) = \perp \\ \bigvee_{k \in cand(mas\_cond(T))} sudoku(T \oplus (max\_cond(T) \to k)) & \text{cc} \end{cases}$$

Sudoku

$$\text{sudoku(T)} = \begin{cases} esValido(T) & \text{si } mas\_cond(T) = \perp \\ \displaystyle\bigvee_{k \in cand(mas\_cond(T))} sudoku(T \oplus (max\_cond(T) \to k)) & \text{cc} \end{cases}$$

- La solución es *sudoku*(T).

Sudoku

$$\text{sudoku(T)} = \begin{cases} esValido(T) & \text{si } mas\_cond(T) = \perp \\ \bigvee\limits_{k \in cand(mas\_cond(T))} sudoku(T \oplus (max\_cond(T) \to k)) & \text{cc} \end{cases}$$

- La solución es $sudoku(T)$.
- ¿La complejidad cambia?

Dobra

Enunciado

Tenemos una cadena I de N caracteres que son letras en mayúscula o bien comodines _. Queremos ver cuántas cadenas podemos formar si reemplazamos los comodines por mayúsculas, teniendo en cuenta que:

- No queremos que haya 3 vocales ni 3 consonantes seguidas.
- Tiene que haber una L en la palabra.

- ¿Un posible espacio de búsqueda? ¿Soluciones parciales? ¿Extensión?
- ¿Qué verificamos en las hojas?
- ¿Cuántas opciones tenemos en cada _?

Dobra

$$dobra(i, l) = \begin{cases} verificar(l) & \text{si } i = n \\ dobra(i+1, l) & \text{si } l[i] \neq \_ \\ \sum_{c \in MAYUS} dobra(i+1, l \oplus (i \to c)) & cc \end{cases}$$

Dobra

$$dobra(i, l) = \begin{cases} verificar(l) & \text{si } i = n \\ dobra(i+1, l) & \text{si } l[i] \neq \_ \\ \sum_{c \in MAYUS} dobra(i+1, l \oplus (i \to c)) & cc \end{cases}$$

- ¿Complejidad?

La solución es $dobra(0, l)$

Dobra

$$dobra(i, l) = \begin{cases} verificar(l) & \text{si } i = n \\ dobra(i+1, l) & \text{si } l[i] \neq \_ \\ \sum_{c \in MAYUS} dobra(i+1, l \oplus (i \to c)) & cc \end{cases}$$

- ¿Complejidad?
- El árbol tiene una cantidad de nodos acotable por $O(26^N)$. En las hojas hacemos $O(N)$ operaciones.

La solución es $dobra(0, l)$

Podas

- ¿Qué podas podemos hacer?

Podas

- ¿Qué podas podemos hacer?
- Vamos verificando si los reemplazos que hacemos de los comodines son válidos _.

Podas

- ¿Qué podas podemos hacer?
- Vamos verificando si los reemplazos que hacemos de los comodines son válidos _.
- Por otro lado, ¿Importa *cuál vocal* / *consonante* usamos?

Podas

- ¿Qué podas podemos hacer?
- Vamos verificando si los reemplazos que hacemos de los comodines son válidos _.
- Por otro lado, ¿Importa *cuál vocal* / *consonante* usamos?
- Qué vocal se usa es irrelevante. Solo importa el hecho de que usamos una vocal, y entonces podemos usar una vocal cualquiera y multiplicar por 5.

Podas

- ¿Qué podas podemos hacer?
- Vamos verificando si los reemplazos que hacemos de los comodines son válidos _.
- Por otro lado, ¿Importa *cuál vocal / consonante* usamos?
- Qué vocal se usa es irrelevante. Solo importa el hecho de que usamos una vocal, y entonces podemos usar una vocal cualquiera y multiplicar por 5.
- ¿Podemos hacer lo mismo para las consonantes?

Podas

- ¿Qué podas podemos hacer?
- Vamos verificando si los reemplazos que hacemos de los comodines son válidos _.
- Por otro lado, ¿Importa *cuál vocal* / *consonante* usamos?
- Qué vocal se usa es irrelevante. Solo importa el hecho de que usamos una vocal, y entonces podemos usar una vocal cualquiera y multiplicar por 5.
- ¿Podemos hacer lo mismo para las consonantes?
- Hay que controlar si usamos o no una L.

Dobra

Los casos de la función recursiva $dobra(i, l, tiene\_L)$ quedan en

- Si $i == N$:

Dobra

Los casos de la función recursiva $dobra(i, l, tiene\_L)$ quedan en

- Si $i == N$: Devolvemos $tiene\_L$.
- Si $l[i] \neq \_$:

Dobra

Los casos de la función recursiva $dobra(i, l, tiene\_L)$ quedan en

- Si $i == N$: Devolvemos $tiene\_L$.
- Si $l[i] \neq \_$:verificamos que esté bien, y en caso afirmativo seguimos con $dobra(i + 1, l, tiene\_L \vee l[i] == L)$.
- Si $l[i] = \_$, no puede ir una consonante, pero si una vocal:

Dobra

Los casos de la función recursiva $dobra(i, l, tiene\_L)$ quedan en

- Si $i == N$: Devolvemos $tiene\_L$.
- Si $l[i] \neq \_$:verificamos que esté bien, y en caso afirmativo seguimos con $dobra(i+1, l, tiene\_L \vee l[i] == L)$.
- Si $l[i] = \_$, no puede ir una consonante, pero si una vocal: hacemos recursión con $5 * dobra(i+1, l \oplus (i \rightarrow A), tiene\_L)$
- Si $l[i] = \_$, no puede ir una vocal, pero si una consonante:

Dobra

Los casos de la función recursiva $dobra(i, l, tiene\_L)$ quedan en

- Si $i == N$: Devolvemos $tiene\_L$.
- Si $l[i] \neq \_$:verificamos que esté bien, y en caso afirmativo seguimos con $dobra(i+1, l, tiene\_L \vee l[i] == L)$.
- Si $l[i] = \_$, no puede ir una consonante, pero si una vocal: hacemos recursión con $5 * dobra(i+1, l \oplus (i \to A), tiene\_L)$
- Si $l[i] = \_$, no puede ir una vocal, pero si una consonante: hacemos dos recursiones, devolviendo $20 * dobra(i+1, l \oplus (i \to B), tiene\_L) + dobra(i+1, l \oplus (i \to L), true)$.
- Si $l[i] = \_$ y podemos tanto vocal como consonante:

Dobra

Los casos de la función recursiva $dobra(i, l, tiene\_L)$ quedan en

- Si $i == N$: Devolvemos $tiene\_L$.
- Si $l[i] \neq \_$:verificamos que esté bien, y en caso afirmativo seguimos con $dobra(i + 1, l, tiene\_L \vee l[i] == L)$.
- Si $l[i] = \_$, no puede ir una consonante, pero si una vocal: hacemos recursión con $5 * dobra(i + 1, l \oplus (i \to A), tiene\_L)$
- Si $l[i] = \_$, no puede ir una vocal, pero si una consonante: hacemos dos recursiones, devolviendo $20 * dobra(i + 1, l \oplus (i \to B), tiene\_L) + dobra(i + 1, l \oplus (i \to L), true)$.
- Si $l[i] = \_$ y podemos tanto vocal como consonante: sumamos los casos anteriores.
- Caso contrario:

Dobra

Los casos de la función recursiva $dobra(i, l, tiene\_L)$ quedan en

- Si $i == N$: Devolvemos $tiene\_L$.
- Si $l[i] \neq \_$:verificamos que esté bien, y en caso afirmativo seguimos con $dobra(i+1, l, tiene\_L \vee l[i] == L)$.
- Si $l[i] = \_$, no puede ir una consonante, pero si una vocal: hacemos recursión con $5 * dobra(i+1, l \oplus (i \rightarrow A), tiene\_L)$
- Si $l[i] = \_$, no puede ir una vocal, pero si una consonante: hacemos dos recursiones, devolviendo $20 * dobra(i+1, l \oplus (i \rightarrow B), tiene\_L) + dobra(i+1, l \oplus (i \rightarrow L), true)$.
- Si $l[i] = \_$ y podemos tanto vocal como consonante: sumamos los casos anteriores.
- Caso contrario: devolvemos 0;

Dobra

- ¿Cuál es la complejidad de esta nueva solución?

Dobra

- ¿Cuál es la complejidad de esta nueva solución?
- Hay a lo sumo 3^n nodos, y hacemos $O(1)$ operaciones en cada paso.

Dobra

- ¿Cuál es la complejidad de esta nueva solución?
- Hay a lo sumo 3^n nodos, y hacemos $O(1)$ operaciones en cada paso.
- Complejidad final: $O(3^n)$.

Dobra

- ¿Cuál es la complejidad de esta nueva solución?
- Hay a lo sumo 3^n nodos, y hacemos $O(1)$ operaciones en cada paso.
- Complejidad final: $O(3^n)$.
- ¿Hace falta arrastrar el l en la recursión?

Dobra

- ¿Cuál es la complejidad de esta nueva solución?
- Hay a lo sumo 3^n nodos, y hacemos $O(1)$ operaciones en cada paso.
- Complejidad final: $O(3^n)$.
- ¿Hace falta arrastrar el l en la recursión?
- No hace falta, alcanza con contar con los últimos dos caracteres. Podemos definir $dobra(i, ant, ant\_ant, tiene\_L)$.

Dobra

- ¿Cuál es la complejidad de esta nueva solución?
- Hay a lo sumo 3^n nodos, y hacemos $O(1)$ operaciones en cada paso.
- Complejidad final: $O(3^n)$.
- ¿Hace falta arrastrar el l en la recursión?
- No hace falta, alcanza con contar con los últimos dos caracteres. Podemos definir $dobra(i, ant, ant\_ant, tiene\_L)$.
- Gracias a esta formulación, es posible resolver el problema en $O(n)$ con programación dinámica.