Semana 3

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

08-2021

Backtracking
- Introducción
- Algunos ejemplos
  - 8 reinas
  - Permutaciones
  - Suma de subconjuntos
  - Prime Ring Problem

Ramificación y Poda
- Introducción
- Algunos ejemplos
  - Problema de asignación

Manejo de cadenas de caracteres

- Introducción
- Función Z
  - Implementación ingenua (fuerza bruta)
  - Implementación eficiente
- Problema de la coincidencia de un patrón
  - Coincidencia de un patrón (fuerza bruta)
  - Algoritmo KMP (Knuth-Morris-Pratt)

Introducción

- Existen problemas importantes para los cuales no se ha logrado encontrar un algoritmo eficiente para solucionarlos. Sin embargo, como son importantes, se tiene que buscar alguna forma de obtener su solución.
- En algunas ocasiones, la única herramienta con la que se cuenta es la búsqueda exhaustiva, esto es, buscar en todo el espacio de soluciones un elemento que cumpla con las condiciones para ser la solución.
- Cuando estos problemas son combinatorios, esto es, cuando su solución es una **permutación**, **combinación** o **subconjunto** de un conjunto de valores que pueden tomar las variables involucradas, el espacio de búsqueda es discreto

- La búsqueda exhaustiva podría funcionar para solucionar cualquier problema combinatorio, sin embargo, en algunos casos los espacios de solución son infinitos o muy grandes y este proceso se haría imposible de realizar en un tiempo adecuado debido a que el número posible de soluciones crece exponencialmente con el número de valores posibles de las variables.
- Backtracking es una mejora de la búsqueda exhaustiva. Va construyendo la solución paso a paso, en donde cada paso analiza los posibles valores que pueda tomar una variable. Utiliza una estructura de árbol cuya raíz es el inicio del problema y cada nivel está formado por los posibles valores que puede tomar una de las variables involucradas en el problema.

- Para no generar el árbol, utiliza el recorrido DFS.
- Al explorar un nodo, se verifica que la solución parcial hasta ese momento cumpla con las condiciones de una posible solución, si es así, este nodo es clasificado como "prometedor", por lo que se expande y se continúa el proceso de "primero en profundidad". Si, por el contrario, se encuentra que la solución parcial obtenida hasta ese momento no cumple con las condiciones para ser una solución al problema, ese nodo se clasifica como "no promisorio", por lo que se desecha, se regresa a su padre.

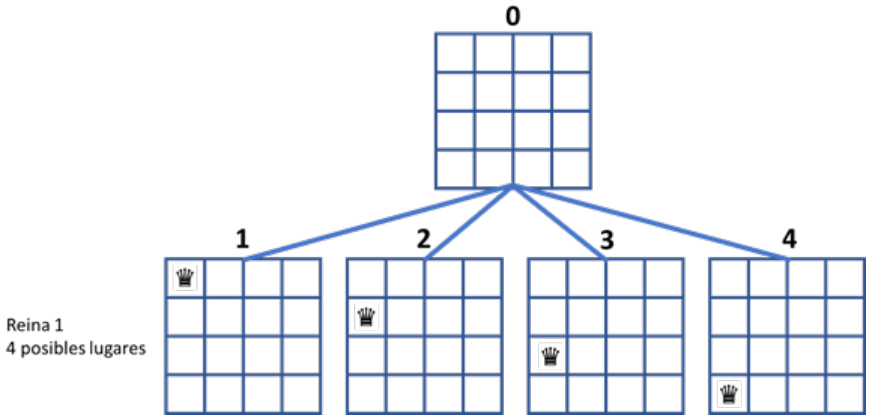
Forma general

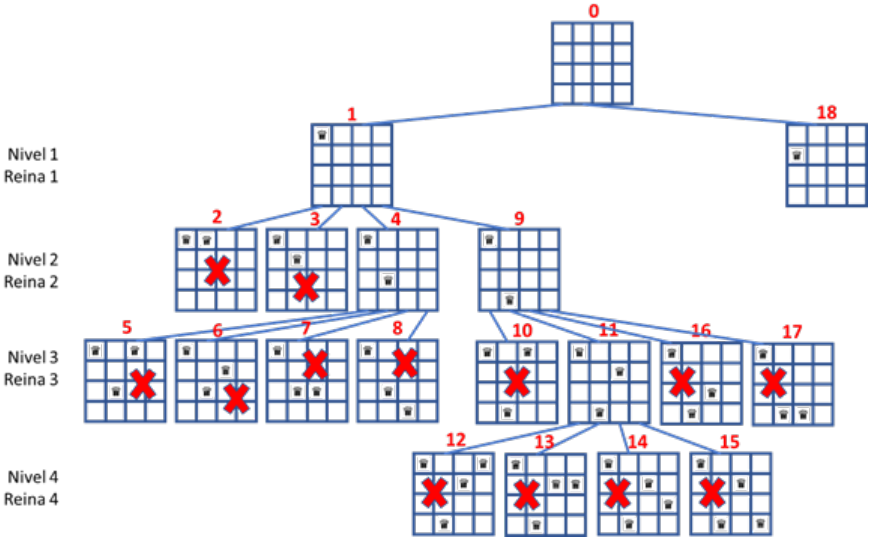
Procedure 1 BACKTRACKING

Input: *node*

```
if SOLUTION(node)) then
    print node
end if
if NOT(PROMISSORY(node))) then
    return
end if
for i ← 1 to CHILDS(node) do
    BACKTRACKING(i)
end for
```

En un tablero de ajedrez ($8X8$ casillas) se deben colocar 8 reinas sin que se ataquen. Recordar que una reina puede atacar vertical, horizontal o diagonalmente, recorriendo cuantas casillas requiera. El problema se puede generalizar a N reinas, es decir, en un tablero de NXN casillas se colocan n reinas sin que se ataquen


0
1
2
3
4
Reina 1
4 posibles lugares


0
1
18
Nivel 1
Reina 1
2
3
4
9
Nivel 2
Reina 2
5
6
7
8
10
11
16
17
Nivel 3
Reina 3
12
13
14
15
Nivel 4
Reina 4

Procedure 2 CAN_PLACE

Input: $rows : Array, col : Integer, ren : Integer$

```
for i ← 1 to col do
    if rows[i] = ren or abs(rows[i] − ren) = abs(i − c) then
        return false
    end if
end for
return true
```

Procedure 3 EIGHT_QUEENS

Input: $rows : Array, a : Integer, b : Integer, col : Integer$

```
if c = 8 and rows[b] = a then
  print rows
end if
for ren ← 1 to 8 do
  if CAN_PLACE(rows, ren, col) then
    rows[col] ← ren
    EIGHT_QUEENS(rows, a, b, col + 1)
  end if
end for
```

Permutaciones

Hallar todas las permutaciones de un número N. Por ejemplo, las permutaciones de 123 son: 123, 231, 321, 312, 132, 213, 123.

Procedure 4 PERMUTATION

Input: $S : String, pos : Integer$

```
if pos = 0 then
  print S
else
  for i ← 1 to pos do
    SWAP(S, i, pos)
    PERMUTATION(S, pos − 1)
    SWAP(S, i, pos)
  end for
end if
```

Suma de subconjuntos

Dado un conjunto S de números enteros positivos, encontrar los subconjuntos que sumen la cantidad C. Si no se encuentra algún subconjunto que cumpla, se debe responder con el conjunto vacío.

Procedure 5 SUBSET_SUM

Input: $A : Array, S : Set, acum : Integer, target : Integer, level : Integer$

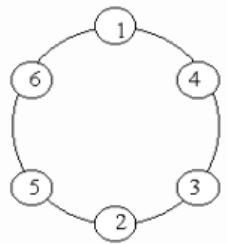
```
if acum = target then
  print solution
end if
if acum > target then
  return
end if
if level < A.length then
  SUBSET_SUM(A, S, acum, target, level + 1)
  S.INSERT(A[level])
  SUBSET_SUM(A, S, acum + A[level], target, level + 1)
end if
```

Prime Ring Problem

524 Prime Ring Problem

A ring is composed of n (even number) circles as shown in diagram. Put natural numbers $1, 2, \ldots, n$ into each circle separately, and the sum of numbers in two adjacent circles should be a prime.

Note: the number of first circle should always be 1.



Input

n $(0 < n \leq 16)$

Output

The output format is shown as sample below. Each row represents a series of circle numbers in the ring beginning from 1 clockwisely and anticlockwisely. The order of numbers must satisfy the above requirements.

You are to write a program that completes above process.

[1] https://onlinejudge.org/external/5/524.pdf

Sample Input

```
6
8
```

Sample Output

```
Case 1:
1 4 3 2 5 6
1 6 5 2 3 4

Case 2:
1 2 3 8 5 6 7 4
1 2 5 8 3 4 7 6
1 4 7 6 5 8 3 2
1 6 7 4 3 8 5 2
```

Introducción

- Ramificación y poda (Branch and Bound, B&B) es una técnica similar a Backtracking. La diferencia es que en el momento en que se encuentre que una solución parcial no puede ser la solución buscada, se detiene el proceso (poda). Esta técnica se utiliza para solucionar problemas de optimización.
- En el argot de los problemas de optimización,
  - Una solución factible es aquella que cumple con todas las restricciones que tenga el problema.
  - La solución óptima es la solución factible que maximiza o minimiza la función, dependiendo de lo que se busque.
  - La función que se quiere maximizar o minimizar se le llama función objetivo.

- Para hacer B&B es necesario que para cada nodo del árbol se pueda establecer un límite para el mejor valor de la función objetivo sobre cualquier solución que pueda ser obtenida.
- No hay una técnica general para establecer esta cota, es decir, es muy particular para cada problema. Sin embargo, una forma que normalmente resulta adecuada es la de relajar el problema original, es decir, quitar o cambiar algunas de las restricciones de tal forma que tengamos un problema más sencillo de resolver y podamos encontrar una solución adecuada para iniciar el proceso, en forma más rápida.

- En cada nodo, se compara el límite de dicho nodo con el mejor valor que se haya obtenido hasta ahora, el cual al inicio es ∞, si el problema es de minimización y $-\infty$, si el problema es de maximización.
- Si el límite no es mejor que la mejor solución obtenida hasta ahora, entonces el nodo no es promisorio y ahí se detiene la búsqueda con esa rama (se poda la rama), lo que significa que ninguna rama que parte de este nodo nos dará una mejor solución.
- También se puede podar una rama si la solución parcial en el nodo actual viola alguna de las restricciones establecida por el problema, es decir, si representa una solución no factible.

- Cuando se llega a una solución completa, se compara con el mejor valor encontrado hasta ahora. Si es mejor, se actualiza el mejor valor. Si no, la solución se desecha.
- A diferencia de Backtracking, donde se generaba un solo hijo del nodo a la vez, en B&B, se deben generar todos los hijos (ramificación) del nodo más prometedor, de entre todos los que se encuentran en la frontera del árbol, que no han sido podados.
- La frontera del árbol está formada por todos los nodos que no han sido expandidos, a los cuales se les llama nodos vivos.
- Mientras que Backtracking emplea "primero profundidad", B&B utiliza "primero el mejor".

Forma general

Procedure 6 BRANCH_AND_BOUND

Input: *frontier* : *Set* < *Node* >, *currentSolution* : *Node*, *currentBest* : *Number*

```
node ← BEST(frontier)
if NOT(PROMISSORY(node))) then
  return
end if
if SOLUTION(node)) and FEASIBLE(node) and LIMIT(node) IS BETTER
THAN currentBest then
  currentSolution ← node)
  currentBest ← LIMIT(node)
  return
end if
NEXT SLIDE
```

```
for i ← 1 to CHILDS(node) do
  ADD(i, frontier)
end for
if NOT(EMPYT(frontier)) then
  BRANCH_AND_BOUND(frontier)
end if
```

Dado un conjunto W trabajos, $W = w_1, w_2, ..., w_n$ y un conjunto de T de trabajadores, $T = t_1, t_2, ..., t_n$, y una matriz C de tamaño (nxn), donde las filas representan los n trabajadores y las columnas los n trabajos. Cada celda c_{ij} contiene el costo de asignar el trabajador i al trabajo j. Encontrar la mejor asignación de trabajos y trabajadores, es decir, el menor costo total, donde, cada trabajo sólo se puede ser asignado a un solo trabajador y a cada trabajador sólo se le puede asignar un trabajo.

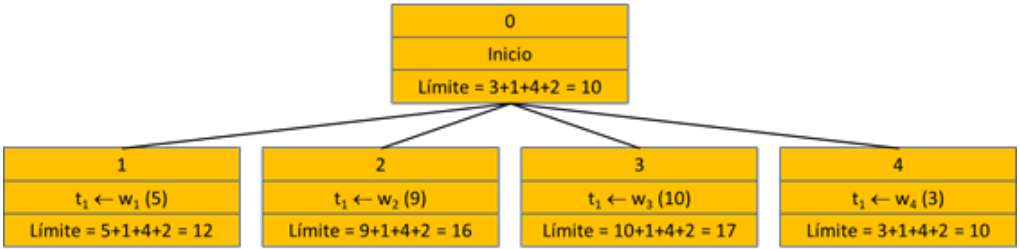
Por ejemplo, 4 trabajadores y 4 trabajos. Los costos de asignación están dados por la siguiente matriz:

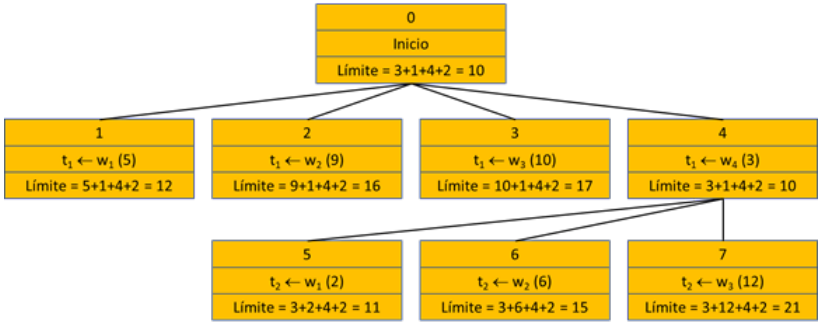
5	9	10	3
2	6	12	1
7	4	4	8
11	16	2	14

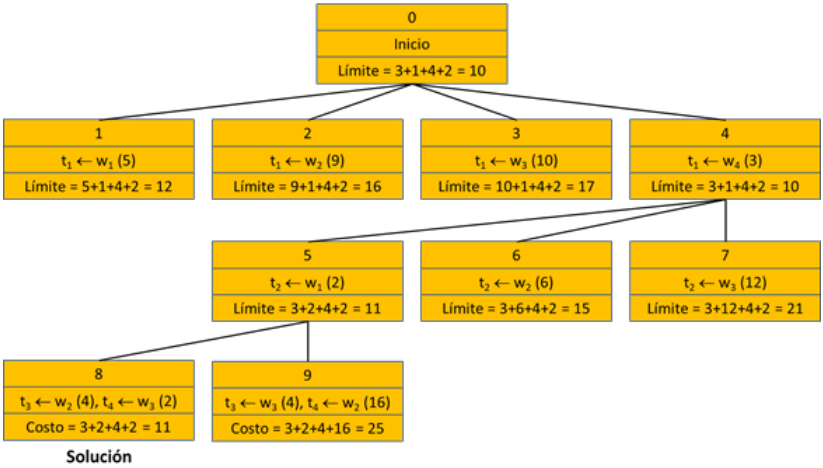
Si tomamos como punto de partida el menor costo de cada columna, obtenemos el siguiente nodo raíz:

0
Inicio
Límite = 3+1+4+2 = 10

IMPORTANTE: Este nodo es factible, se repiten asignaciones; sin embargo, nos sirve como punto de arranque.


0
Inicio
Límite = 3+1+4+2 = 10
1
$t_1 \leftarrow w_1$ (5)
Límite = 5+1+4+2 = 12
2
$t_1 \leftarrow w_2$ (9)
Límite = 9+1+4+2 = 16
3
$t_1 \leftarrow w_3$ (10)
Límite = 10+1+4+2 = 17
4
$t_1 \leftarrow w_4$ (3)
Límite = 3+1+4+2 = 10


0
Inicio
Límite = 3+1+4+2 = 10
1
$t_1 \leftarrow w_1$ (5)
Límite = 5+1+4+2 = 12
2
$t_1 \leftarrow w_2$ (9)
Límite = 9+1+4+2 = 16
3
$t_1 \leftarrow w_3$ (10)
Límite = 10+1+4+2 = 17
4
$t_1 \leftarrow w_4$ (3)
Límite = 3+1+4+2 = 10
5
$t_2 \leftarrow w_1$ (2)
Límite = 3+2+4+2 = 11
6
$t_2 \leftarrow w_2$ (6)
Límite = 3+6+4+2 = 15
7
$t_2 \leftarrow w_3$ (12)
Límite = 3+12+4+2 = 21


0
Inicio
Límite = 3+1+4+2 = 10
1
t1 ← w1 (5)
Límite = 5+1+4+2 = 12
2
t1 ← w2 (9)
Límite = 9+1+4+2 = 16
3
t1 ← w3 (10)
Límite = 10+1+4+2 = 17
4
t1 ← w4 (3)
Límite = 3+1+4+2 = 10
5
t2 ← w1 (2)
Límite = 3+2+4+2 = 11
6
t2 ← w2 (6)
Límite = 3+6+4+2 = 15
7
t2 ← w3 (12)
Límite = 3+12+4+2 = 21
8
t3 ← w2 (4), t4 ← w3 (2)
Costo = 3+2+4+2 = 11
Solución
9
t3 ← w3 (4), t4 ← w2 (16)
Costo = 3+2+4+16 = 25

Problema de asignación

Procedure 7 ASSIGMENT_PROBLEM

Input: C : Matrix[N][N]
frontier : Priority Queue<Node>,
currentSolution : Node,
currentBest : Number,
assigned : Array[N],
assigment : Array[N],
node : Node,
child : Node

```
currentBest ← ∞
currentSolution ← None
for i ← 1 to N do
  assigned[i] ← false
  minimum ← C[i][0]
  work ← 0
  for j ← 2 to N do
    if C[i][j] < minimum then
      minimum ← C[i][j]
      work ← j
    end if
  end for
  assigment[i] ← work
end for
NEXT SLIDE
```

```
node ← new Node(0, assigment, assigned, -1)
frontier.enqueue(node)
while not frontier.empty() do
  node ← frontier.dequeueMin()
  if node.limit < currentBest then
    NEXT_SLIDE
  end if
end while
```

```
if node.level < N then
  for i ← | to N do
    if not node.assigned[i] then
      child ← new Node(node.level + 1, node.assigment, node.assigned, i)
      frontier.push(child)
    end if
  end for
else
  currentBest ← node.limit
  currentSolution ← node
end if
```

Introducción

- El manejo de texto involucra una gran cantidad de información, por lo que es evidente la necesidad de automatizar ciertas tareas del manejo de texto. Funciones tales como búsqueda de un cadena, búsqueda y reemplazo de palabras y detección de errores ortográficos, entre otros.
- En años recientes, la biología molecular, al representar el genoma como una secuencia de caracteres llamados bases (A de adenina, C de citosina, G de guanina y T de timina), se ha enfrentado a la necesidad de realizar el tratamiento y análisis de grandes cantidad de cadenas de caracteres.

- Un cadena de caracteres (o string) es una secuencia ordenada de caracteres. El primer carácter en la cadena es el más izquierdo y ocupa la posición 1, el siguiente la posición 2 y así sucesivamente.
- La longitud de un cadena S, $|S|$, está dada por el número de caracteres que contiene.

- Una subcadena (o substring) de la cadena S es cualquier cadena de caracteres que se encuentran en S. La subcadena $S[i..j]$ es la cadena formada por los caracteres de S que están de la posición i a la j, inclusive.
  - Prefijo: dada una cadena S, un prefijo $S[1..i]$ del mismo es cualquier subcadena que inicia en la posición 1 y termina en la posición i.
  - Sufijo: dada una cadena S, un sufijo $S[i..n]$ del mismo es cualquier subcadena que inicia en la posición i y termina en el último elemento de S.

Función Z

Dada una cadena S de longitud n, la función Z es un arreglo de longitud n donde el i-ésimo elemento es igual al máximo número de caracteres empezando desde la posición I que coincide con los primeros caracteres de S. En otras palabras, $Z[i]$ es la longitud del prefijo común más largo entre S y el sufijo de S que empiezan en i, $S[i..n]$.

Veamos un ejemplo. Dada la cadena $S =$ "*aaabaaab*", para cualquier $i > 1$,

- Posición 2,

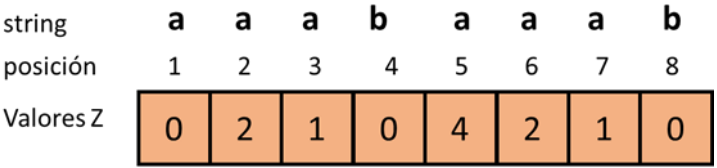
```
S    = aaabaaab
S[2..n] = aabaab
```

- Posición 3,

```
S    = aaabaaab
S[3..n] = abaaab
```

- Posición 4,

```
S    = aaabaaab
S[4..n] = baab
```


string
a a a b a a a b
posición
1 2 3 4 5 6 7 8
Valores Z
0 2 1 0 4 2 1 0

Función Z

Procedure 8 Z_FUNCTION

```
Input: S : String
  Z : Array[S.length]
  INIT(Z, 0)
  for i ← 1 to S.length do
    while i + Z[i] < n and S[Z[i]] == S[i + Z[i]] do
      Z[i] ← Z[i]
    end while
  end for
  return Z
```

Este algoritmo tiene una complejidad $O(n^2)$. Para obtener un algoritmo eficiente, calcularemos los valores de $Z[i]$ de 1 a (n - 1), pero al mismo tiempo, al calcular un nuevo valor, intentaremos hacer el mejor uso posible de los valores anteriormente calcular. Es decir, vamos a usar memorización. Y, bueno, recuerda que estamos buscando el máximo prefijo. ¿Qué técnica podríamos emplear para este algoritmo?

Implementación eficiente

- Llamemos "coincidencias de segmento" a las subcadenas que coinciden con el prefijo de S. Por ejemplo, el valor de $Z[i]$ es la longitud de la coincidencia de segmento que comienza en la posición i y que termina en la posición $i + Z[i] - 1$.
- Para hacer esto, mantendremos los índices $[l, r]$ del segmento coincidente más derecho. Es decir, entre todos los segmentos detectados mantendremos el que termina más al a derecha. En cierto modo, el índice r puede verse como el límite de lo que se ha registrado. Todo lo que está más allá aún no se conoce.

Entonces, si el índice actual, i, tenemos dos opciones:

- $i > r$: La posición está fuera de lo que ya hemos procesado. Entonces, no nos queda más utilizar la comparación carácter a carácter que vimos en el algoritmo ingenuo. Sin embargo, al final, si $Z[i] > 0$, tendremos que actualizar los índices del segmento más derecho, porque está garantizado que el nuevo $r = i + Z[i] - 1$ es menor que el r anterior.

- $i \leq r$: La posición está dentro del segmento de coincidencias actual $[l, r]$.
  - Entonces podemos usar los valores Z ya calculados para inicializar el valor de $Z[i]$ a algo (mejor que comenzar desde cero), tal vez incluso un número más grande.
  - Para ello, observamos que las observamos que las subcadenas $S[l..r]$ y $S[0..(r-l)]$ coinciden. Eso significa que como aproximación inicial para $Z[i]$ podemos tomar el valor ya calcular del segmento correspondiente $S[0..(r-l)]$, que es $Z[i-l]$.
  - Sin embargo, el valor $Z[i-l]$ podría ser demasiado grande: cuando se aplica a la posición i podría exceder el índice r. Esto no está permitido porque no sabemos nada sobre los caracteres a la derecha de r, ya que pueden diferir de los requeridos. Veamos un ejemplo de este punto. Si $S =$ "*aaaabaa*". Cuando llegamos a la último posición ($i = 6$), el segmento de coincidencia actual será $[5, 6]$,, para el cual el valor de la función Z es $Z[1] = 3$. Obviamente, no podemos inicializar $Z[6]$ en 3, sería completamente incorrecto. El valor máximo al que podríamos inicializarlo es 1, porque es el valor más graden que no nos lleva más allá del índice r. Por lo tanto, como una buena aproximación, podemos decir que $Z[i] = min(r - i + 1, Z[i-l])$. Ahora solo falta calcular las posiciones que se encuentran a la derecha de r, y para ello usamos el algoritmo previo.

Procedure 9 Z_FUNCTION_DP

```
Input: S : String
  Z : Array[S.length]
  left ← 0, right ← 0, INIT(Z, 0)
  for i ← 1 to S.length do
    if i ≤ right then
      Z[i] ← MIN(right - i + 1, Z[i - left])
    end if
    while i + Z[i] < n and S[Z[i]] == S[i + Z[i]] do
      Z[i] ← Z[i]
    end while
    if i + Z[i] - 1 > right then
      left ← i, right ← i + Z[i] - 1
    end if
  end for
  return Z
```

string	a	a	a	b	a	a	a	b
posición	1	2	3	4	5	6	7	8
Valores Z	0	2	1	0	4	2	1	0

Aplicaciones[2]

- Búsqueda de una subcadena (o patrón). (SPOILER: ¡Es el siguiente algoritmo!)
- Número de subcadenas distintas en una cadena.
- Comprensión de cadenas.

[2] https://bit.ly/3rDn1uZ

Problema de la coincidencia de un patrón

El problema de la coincidencia de un patrón en un texto (Pattern Matching Problem) también conocido como búsqueda de una subcadena en un texto (Substring Search), se define de la siguiente forma:

- Dada una cadena P, llamada patrón (pattern) y una cadena más grande T llamada el texto, encontrar la primer ocurrencia del patrón P en el texto T y regresar la posición del texto T donde inicial el patrón o -1 en caso de que el patrón no existe en el texto T.

Por ejemplo, si $P =$ "*aca*" y $T =$ "*bacacabcaca*", el patrón P se encuentra en T iniciando en las posiciones 2, 4 y 9. Observa que las ocurrencias pueden estar sobre puestas tal y como sucede en las ocurrencias que inicial en 2 y 4. Para este caso, la respuesta es 2, debido a que es la posición de la primera ocurrencia de P en T.

La forma más simple de resolver este problema es por fuerza bruta, el cual consiste en ir moviendo a la derecha (shifting) una ventana que contiene el patrón, llamada ventana de deslizamiento (sliding windows), casilla por casilla sobre la cadena T y, en cuanto se encuentre la coincidencia, se regresa el número de la casilla de inicio.

Coincidencia de un patrón

Procedure 10 PATTERN_MATCHING

```
Input: P, T : String
  for i ← 1 to T.length - P.length do
    if P = T[i..(i + P.length)] then
      return i
    end if
  end for
  return -1
```

La complejidad del algoritmo anterior es $O(nm)$. Sin embargo, existen varias formas de mejorar este algoritmo para hacerlo en tiempo lineal. Una de ellas es el uso de la función Z, calcula en tiempo lineal. Para lograrlo, basta con concatenar el patrón al inicio del texto, separados por un carácter que no esté en el texto. Normalmente se utiliza el signo de pesos ($).

- Por ejemplo, si $P =$ "*aca*" y $T =$ "*bacacabcaca*", haremos una nueva cadena T' formado con la concatenación de P y T, separados por un \$.
- Después de este proceso, las ocurrencias estarán indicadas en el arreglo de la función Z por las posiciones que tengan un valor de Z igual a la longitud del patrón y restándole la longitud del patrón más 1 (por el \$).

string	a	c	a	$	b	a	c	a	c	a	b	c
posición	1	2	3	4	5	6	7	8	9	10	11	12
Valores Z	0	0	1	0	0	3	0	3	0	1	0	0

Algoritmo KMP (Knuth-Morris-Pratt)

- Una mejor forma de resolver el problema de la coincidencia de un patrón es utilizar el algoritmo KPM, propuesto por Donal Knuth, Jame Morris y Vaughan Pratt en 1977.
- Recordemos que la forma ingenua resuelve el problema utilizando una ventana deslizante que se recorre un carácter a la vez. Lo que busca el algoritmo KPM es que, cuando se tenga una NO coincidencia, se recorran tantos caracteres como sea necesario siempre que se garantice que el patrón no estará ahí.

La idea básica es:

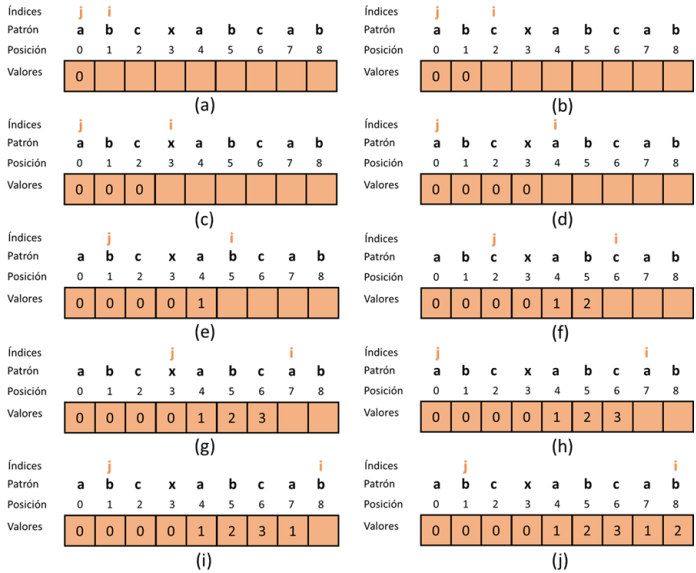
- Se van comparando los caracteres a partir de la posición i del texto con los del patrón.
- Cuando uno coincide, en la subcadena del patrón que se tiene antes la NO coincidencia, buscamos el máximo prefijo que al mimos tiempo sea sufijo de la subcadena.
- Si no existe, nos regresamos solo a la posición inicial de patrón.
- Si existe un prefijo que es sufijo, y la longitud de este prefijo es k, continuamos a partir de la posición $i + k$ del texto y $j + k$ del patrón.
- Este proceso se hace tan hacia atrás como sea necesario, pudiendo llegar a inicial el patrón a su posición inicial.

Por ejemplo, si parte del texto es $T =$ "*abcabx*...", y el patrón es $P =$ "*abcaby*", iniciamos comparando los caracteres uno a uno, hasta que llegamos a la posición 6. En esta posición falla porque tenemos una x en el texto y una y en el patrón. En el método ingenuo tendríamos que iniciar todo desde la posición 2 del texto y la 1 del patrón. Pero, podemos observar que en la subcadena anterior a la falla, es decir, en "*abcab*", existe el prefijo *ab*, con longitud 2, que al mismo tiempo es sufijo de la misma cadena (*abcab*), lo que nos dice qué es seguro que los 2 caracteres anteriores al fallo coinciden completamente con el inicio del patrón, por lo que las siguientes comparaciones se pueden hacer a partir de la 6 del texto y la posición 3 del patrón.

Para poder hacer todo lo anterior en tiempo lineal, es necesario hacer un pre-procesamiento del patrón. Este pre-procesamiento se hace usando un arreglo que inicial en la posición 0 (que sería la posición inicial de la cadena). Al final de este proceso, el arreglo contiene la longitud del máximo prefijo que es también sufijo en la subcadena $P[1..i]$.

Procedure 11 PREPROCESSING

```
Input: P : String
  V : Array[P.length]
  j ← 0, i ← 0, V[0] ← 0
  while i < P.length do
    if P[i] = P[j] then
      V[i] ← j + 1, i ← i + 1, j ← j + 1
    else
      if j = 0 then
        V[i] ← 0, i ← i + 1
      else
        j ← V[j - 1]
      end if
    end if
  end while
  return V
```


Índices
Patrón
Posición
Valores
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)

- Una vez preprocesado el patrón, el algoritmo es muy simple dado que el arreglo V no dice que, una vez hallado un fallo, a qué posición se debe regresar el patrón para seguir comparando con el texto.
- La complejidad del algoritmo resultante es lineal sobre la longitud del texto, $O(n)$.Y como la complejidad del preprocesamiento es lineal sobre la longitud del patrón, $O(m)$, la complejidad final del algoritmo KMP es lineal sobre los procesos, $O(n+m)$, mucho más eficiente que el $O(nm)$ del algoritmo ingenueo.

Procedure 12 KMP

```
Input: P, T : String
  V ← PREPROCESSING(P), i ← 0, j ← 0, pos ← 0
  while i < T.length do
    if T[i] = P[i] then
      i ← i + 1, j ← j + 1
      if j = P.length then
        return pos
      end if
    else
      if j = 0 then
        i ← i + 1, pos ← i
      else
        pos ← i - V[j - 1], j = V[j - 1]
      end if
    end if
  end while
  return -1
```