Estructuras de datos sobre árboles (+Treap)

Matías Hunicken[1]

[1]FaMAF
Universidad Nacional de Córdoba

Training Camp 2018

Contenidos

Contenidos

Queries sobre árboles

Problema motivación

Dado un árbol con pesos en los nodos, con $n \leq 2 \cdot 10^5$, ejecutar los siguientes dos tipos de consulta (hasta $2 \cdot 10^5$ consultas):

- `1 x v`: Cambiar el peso del nodo x por el valor v.
- `2 x y`: Preguntar por el máximo valor de los nodos en el camino entre los nodos x e y.

- O, en otras palabras: Tener una forma de hacer segment tree "sobre árboles".

Casos particulares del problema

- Si tenemos un árbol que es “una cadena”: podemos usar segment tree normal directamente.
- Si tenemos que el árbol es balanceado (es decir, la mayor profundidad del árbol es $O(logn)$), entonces es trivial: para hacer el update actualizamos el valor, y podemos hacer la query recorriendo el camino desde cada nodo hacia arriba, hasta que coincidan.

.

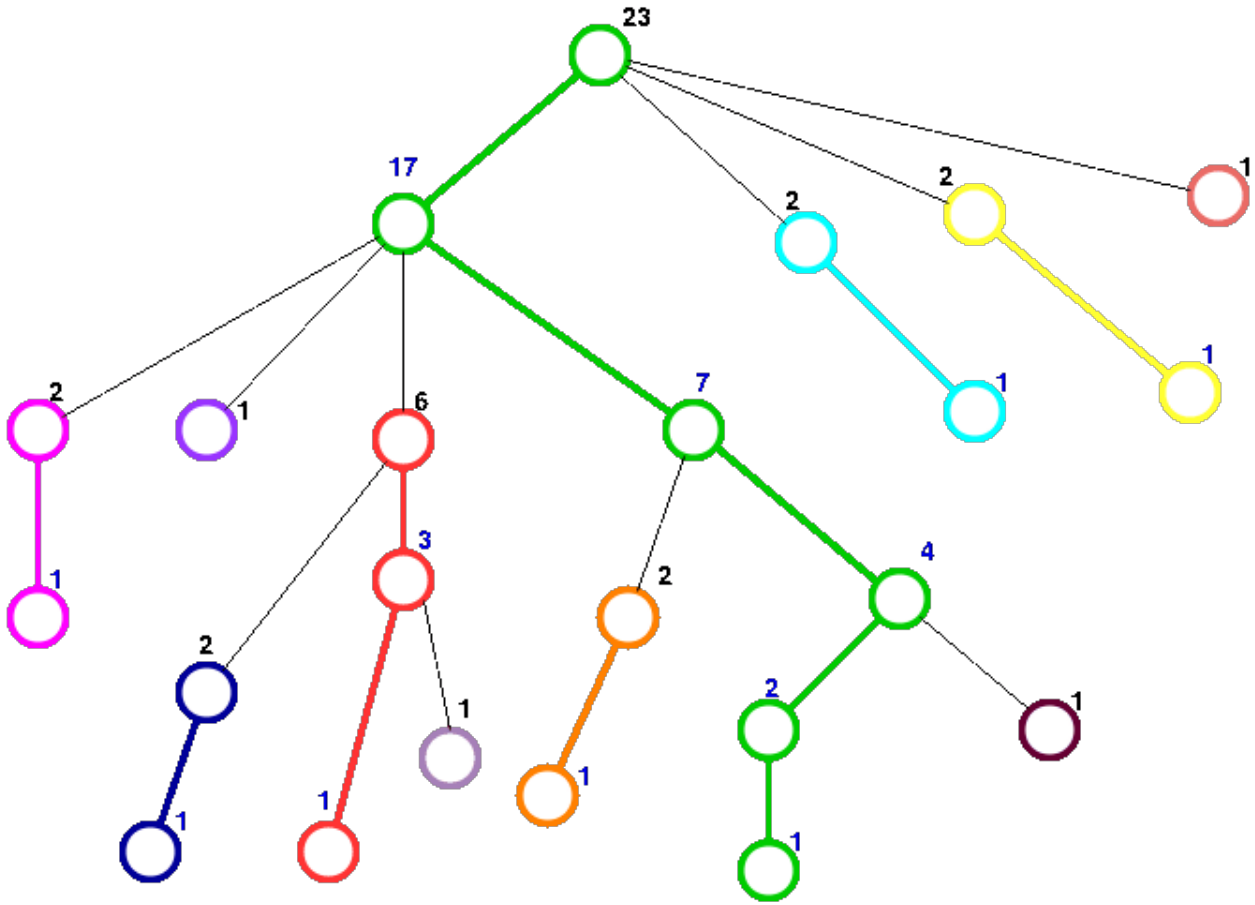
- Se pueden combinar ambas ideas para resolver las queries eficientemente para cualquier árbol.
- La técnica se llama “Heavy Light Decomposition” (o HLD).

Heavy Light Decomposition

La idea de HLD es como sigue:

- Elegimos un nodo que sea la raíz del árbol.
- Particionamos las aristas en dos conjuntos: Las aristas “heavy” y las aristas “light” (de ahí el nombre):
  - Para cada nodo que tenga hijos, elegimos la arista que vaya hacia abajo y que lleve al sub-árbol de mayor tamaño (desempatando arbitrariamente). A éstas las llamamos **heavy**.
  - Las demás aristas son **light**.
- Propiedad clave: Para cada nodo, si tomamos el camino de ese nodo hacia la raíz, entonces hay $O(log(n))$ aristas heavy (porque cuando subo por una heavy se me duplica (como mínimo) el tamaño del subárbol).

Heavy Light Decomposition (ejemplo)



- Los nodos del árbol quedan particionados en caminos de aristas heavy.

Heavy Light Decomposition (cont.)

- Tenemos entonces una partición del grafo en “cadenas”, tal que al moverme hacia la raíz cambio de cadena $O(log(n))$ veces.
- Para resolver el problema original, podemos usar un segment tree para cada cadena, que guarde los valores de los nodos correspondientes.
- Para hacer el update, es simplemente actualizar el segment tree en la posición correspondiente al nodo. Queda $O(log(n))$
- Para hacer la query, podemos subir desde cada uno de los nodos hasta el LCA (“Lowest Common Ancestor”). En cada una de las cadenas hacemos query del segment tree en el intervalo correspondiente. Si llegamos a la parte de arriba de la cadena pasamos a la de arriba. Queda $O(log^2(n))$.

Heavy-Light - Implementación

Consejos para la implementación:

- Queda más simple usando un sólo segment tree, en lugar de uno separado por cada cadena. Cada cadena debe ser un segmento continuo en el segment tree y se debe guardar para cada nodo su posición.
- Se debe guardar para cada cadena el nodo más alto (se necesita para la query).
- Conviene adicionalmente guardar la profundidad de cada nodo (distancia a la raíz). Guardando esto (más el nodo más alto de cada cadena) alcanza para calcular el LCA usando HLD.

Heavy-Light - Pesos en las aristas

- Otro problema similar es lo mismo, pero los pesos en lugar de estar en los nodos están en las aristas.
- Este problema puede reducirse fácil a el caso de pesos en los nodos.
- Para cada arista ponemos su costo en el nodo “de más abajo”. En el nodo raíz ponemos cualquier valor.
- Lo único que hay que cambiar es ignorar el LCA cuando se hace la query.

Heavy-Light en DFS

Problema - Tree and Queries

Dado un árbol de hasta 10^5 nodos, donde todos los nodos tienen un color, responder m (hasta 10^5) queries de la forma: $(v, k) \rightarrow$ "cantidad de colores c tales que el subárbol del vértice v tiene al menos k nodos de color c".

¿Les suena?

- Este problema sale en $O((m+n) \cdot \sqrt{n})$ con Mo's algorithm sobre el orden dfs de los nodos.
- También se puede resolver en $O(m + n \cdot log(n))$ con una idea similar a HLD.

Heavy-Light en DFS - cont.

- Necesito una “mini estructura de datos que soporte tres queries”:
  - Agregar un color.
  - Quitar un color
  - Contar cuantos colores hay que ocurran al menos k veces.
- Es fácil de implementar todas en $O(1)$ con dos arreglos:
  - Uno que guarde la cantidad de veces que aparece cada color.
  - Otro que guarde la respuesta a la consulta para cada k.

Heavy-Light en DFS - cont.

Primer approach ($O(m + n^2)$):

- Calculo las respuestas offline: para cada nodo guardo un vector de todas las queries que corresponden a ese nodo.
- Hago un DFS para calcular la respuesta a cada query:
  - Primero recursiono para todos los hijos.
  - Luego agrego a la estructura los colores de los nodos del subárbol.
  - Respondo las queries del nodo en el que estoy parado.
  - Elimino de la estructura los colores de los nodos del subárbol.

Heavy-Light en DFS - cont.

El approach anterior es ineficiente, pero se puede mejorar:

- El paso de eliminar de la estructura los colores de los nodos del subárbol, sólo lo realizo cuando el nodo no corresponde al subárbol más grande del padre (es decir, no borro cuando me moví por una arista heavy).
- Cuando recursiono para calcular la respuesta para los hijos, empiezo por los que no son el subárbol más grande, y al último recursiono para el más grande.
- Cuando agrego a la estructura los colores del subárbol, no tengo que hacerlo para los nodos del subárbol más grande, porque este no borró la estructura cuando recursioné.
- La complejidad queda $O(m + n \cdot log(n))$ porque cada nodo lo agrego y lo borro una vez adicional por cada vez que me muevo por una arista light hasta llegar a la raíz, y hay $O(log(n))$ de éstas.

Heavy-Light en DFS - conclusión

- La idea anterior también se conoce con el nombre “dsu on tree”.
- Aplica a muchos problemas donde hay que responder queries sobre subárboles.

Contenidos

Centroide

Centroide - definición

Dado un árbol, un **centroide** del árbol es un nodo tal que si lo sacamos del árbol las componentes conexas que quedan tienen a lo sumo $\lfloor \frac{n}{2} \rfloor$ nodos.

- Todo árbol tiene centroide: Puede tener uno sólo, o dos conectados por una arista.
- Es fácil encontrar un centroide fijando un nodo como raíz y precomputando los tamaños de los subárboles.

Centroid decomposition

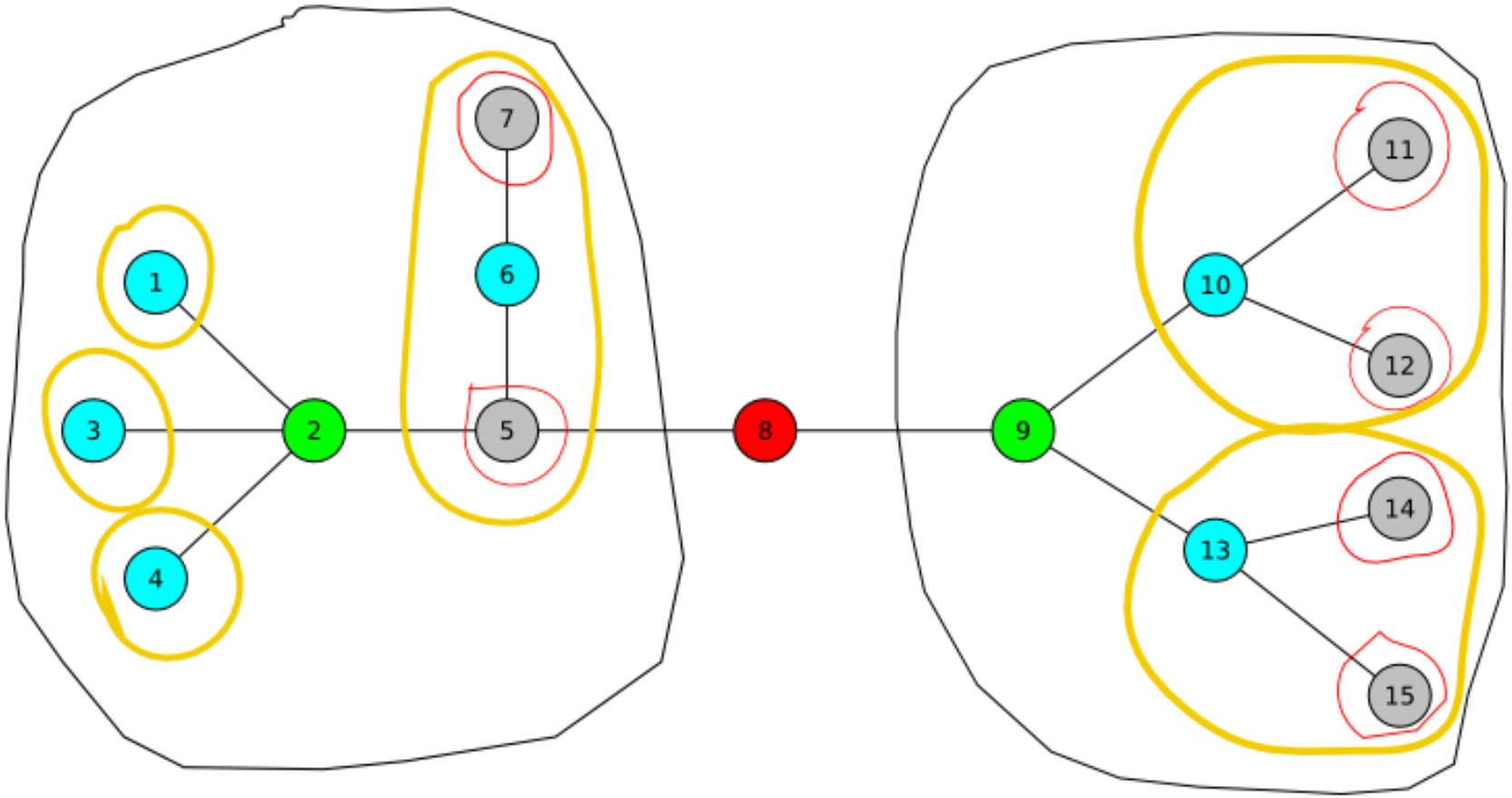
La descomposición en centroides es una técnica general que tiene esta forma:

- Encontrar el centroide del árbol en $O(n)$.
- Hacer "algo" en el árbol (donde "algo" depende del problema).
- Borrar el centroide del árbol.
- Recursionar para los subárboles.

.

- Si "algo" es $O(n)$, entonces la complejidad de todo es $O(n \cdot log(n))$ porque:
  - En cada nivel de la recursión hago $O(n)$ operaciones.
  - La profundidad de la recursión es $O(log(n))$ porque en cada llamada el tamaño de los subárboles se reduce a la mitad en el peor caso.

Centroid decomposition - Ejemplo



Centroid decomposition - Problema ejemplo

Problema

Dado un árbol de hasta 10^5 nodos, responder m (hasta 10^5) queries de la forma: $(v, d) \rightarrow$ "cantidad de vértices a distancia $\leq d$ desde el vértice v".

- La solución straight-forward tiene complejidad $O(m \cdot n)$.
- Podemos hacerlo mejor usando descomposición en centroides.

Centroid decomposition - Problema ejemplo (cont.)

- Respondemos las queries offline. Mantenemos un arreglo con la respuesta a cada query, inicializado en 0.
- Realizamos descomposición en centroides: En cada paso calcularemos para cada query correspondiente a un nodo del árbol, la respuesta si consideramos **únicamente los caminos que pasan por el centroide del árbol** (llamémoslo *c*).
- Si después de eliminar *c*, recursionamos para los subárboles y vamos acumulando la respuesta, entonces al final tenemos el resultado para todas las queries.

Centroid decomposition - Problema ejemplo (cont.)

- Armamos para cada subárbol que queda después de borrar c, un arreglo que tenga las distancias de los nodos del subárbol a c (ordenados).
- Armamos un arreglo similar, pero para todos los nodos del árbol.
- Para responder las queries con $v = c$, podemos hacer binary sobre el arreglo de todo el árbol.
- Para cada query (v, d) con v en el árbol, $v \neq c$, si t es la distancia de c a v, entonces el valor que queremos es "cantidad de elementos $\leq d - t$ en el arreglo de todo el arbol" $-$ "cantidad de elementos $\leq d - t$ en el arreglo del subarbol de v"

.

- Complejidad: $O((n+m) \cdot log^2(n))$ (cada query la evalúo $O(log(n))$ veces y la complejidad de cada vez es $O(log(n))$ (binary search)).

Centroid decomposition como D&C - Conclusión

- La mayoría de los problemas que salen con centroid decomposition (e.g. el que vimos) se pueden ver como un *divide and conquer* sobre árboles.
- El caso típico de problema que sale así son los que dicen “dado un árbol, contar los caminos que cumplen tal propiedad” o “hacer tal suma para todos los caminos”.
- Hay otro tipo de problemas que salen con centroid decomposition, en los que hay que guardar el árbol de la descomposición (la raíz es el centroide de todo el árbol, sus hijos son los centroides de los subárboles y así sucesivamente).

Centroid decomposition - Ejemplo 2

Problema - QTREE5

Dado un árbol con $n \leq 10^5$, donde los vértices inicialmente están pintados de negro, ejecutar los siguientes dos tipos de consulta (hasta 10^5 consultas):

- `1 x`: Cambiar el color del nodo x (de negro a blanco o de blanco a negro).
- `2 x`: Preguntar por la menor distancia desde el nodo x a algún nodo blanco (si x es blanco la respuesta es 0).

- Solución: en pizarrón.

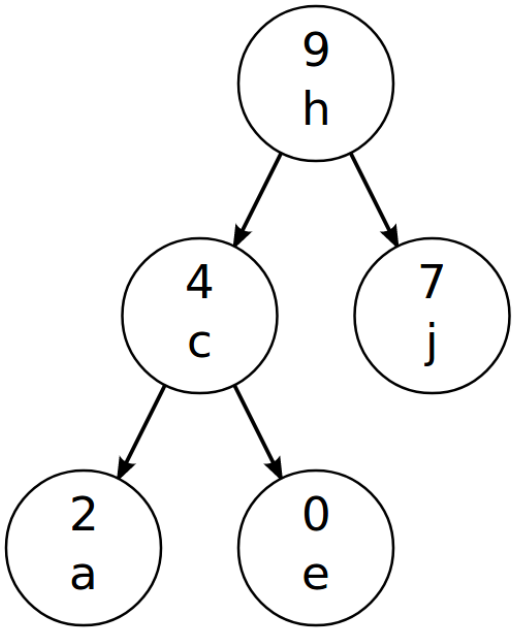
Contenidos

Treap

Definición

Un treap es un árbol binario en el que cada nodo contiene dos valores: una prioridad y una clave, y que cumple las siguientes propiedades:

- Si miramos sólo las prioridades, el árbol es un heap binario.
- Si miramos sólo las claves, el árboles un árbol binario de búsqueda.



En el ejemplo:

- El valor de arriba es la prioridad (ordena como heap).
- El valor de abajo es la clave (ordena como binary search tree).

Propiedades de treap

Propiedades locas de treap:

- Dado un conjunto de pares (*prioridad*, *valor*), hay uno y sólo un treap que se puede formar con esos pares.
- Si las prioridades son valores aleatorios, entonces la forma del árbol tiene distribución uniforme entre todos los árboles binarios de n nodos.
- Por la propiedad anterior, la profundidad del árbol es $O(log(n))$ en valor esperado.
- Por esto, si a las prioridades las asignamos aleatoriamente (lo cuál casi siempre es el caso), se suele llamar también “árbol binario de búsqueda randomizado”.

Operaciones de treap

Treap soporta las siguientes dos operaciones en $O(log(n))$:

- **split(t,k)**: Divide el treap t en dos árboles, donde en el primero quedan todas las keys $< k$ y en el segundo las keys $\geq k$.
- **merge(t1,t2)**: Une los treaps t1 y t2. Requiere como precondición que todas las keys de t1 sean menores que las de t2.

.

- Con estas dos operaciones se pueden implementar en $O(log(n))$ las otras operaciones típicas de árbol binario de búsqueda (inserción, borrado).
- Las operaciones se pueden implementar fácilmente de forma recursiva. El split es igual a como se haría en cualquier BST, el merge tiene en cuenta cuál ubicar como raíz dependiendo de la prioridad.
- Ejemplos: en pizarrón.

Treap - cont.

Además de la clave, se puede guardar más información en los nodos del treap. Por ejemplo:

- Un valor (de este modo funciona como un map).
- La cantidad de descendientes en el subárbol (de este modo se pueden hacer en $O(log(n))$ consultas como “dar la k-ésima key”).
- La suma de todas las claves del sub-árbol. Combinando con lo anterior se pueden hacer consultas como “dar la suma de las primeras k claves”.

En cualquiera de estos casos se puede mantener esta información fácilmente si actualizamos los valores de un nodo cada vez que lo modificamos.

Treap con clave implícita (o “cartesian tree”)

Además de como árbol binario de búsqueda, el treap también puede verse como un arreglo que soporta dos operaciones (también en $O(log(n))$):

- **split(t,k):** Dividir el arreglo en dos partes: los primeros k elementos y los últimos $n - k$.
- **merge(t1,t2):** Formar un nuevo arreglo que sea la concatenación de t1 y t2.

.

- Para implementar esto, en lugar de usar keys explícitas, usamos la key implícita que consiste en la posición en el *inorder traversal* (podemos calcularla fácilmente manteniendo la cantidad de descendientes para cada nodo).
- Además, guardamos para cada nodo un valor de algún tipo cualquiera, que representa el valor en la posición respectiva del arreglo.

Treap con clave implícita - cont.

Al igual que en el caso con clave explícita, podemos guardar información extra en los nodos.

- Por ejemplo: podemos guardar en cada nodo el máximo de todos los valores de los descendientes.
- Notar que en este caso tendriamos una especie de segment tree con operaciones adicionales: dividir un arreglo en dos partes (split) y concatenar dos arreglos (merge).

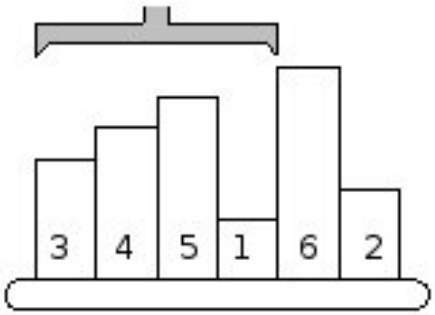
Treap con clave implícita - operaciones lazy

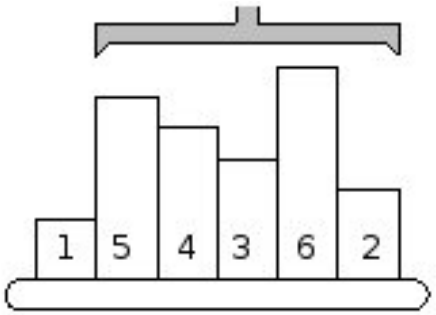
- Notemos que también podemos tener operaciones lazy (como en segment tree), si mantenemos un valor que se propagará a los hijos en caso de modificar el nodo.
- Podemos soportar también esta operación lazy: Dar vuelta el arreglo (reverse). En este caso es mantener un booleano que dice si tenemos que dar vuelta el arreglo. Cuando propagamos, swapeamos los punteros de los dos hijos y actualizamos el booleano para los mismos (negamos el valor que ya tenían).

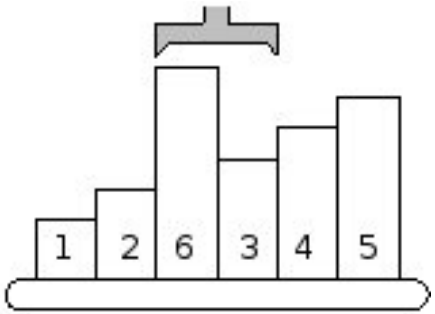
Treap - problema ejemplo

Problema - Robotic sort

Dada una permutación de hasta 10^5 elementos, tenemos un robot que la ordena de la siguiente manera: En el paso *i*-ésimo, posiciona el elemento *i* en la posición *i* dando vuelta el intervalo desde la posición *i* hasta la posición dónde está el valor *i*. Para cada *i* entre 1 y *n* decir donde se encuentra el valor *i* en la *i*-ésima iteración.







- Solución: en pizarrón.

¿Preguntas?