Resumen Final EDA 2020 Q2

Algoritmos para búsqueda de texto

Soundex

Algoritmo fonético, creado para el alfabeto inglés. Codifica de la misma forma a aquellas palabras que suenan igual. Encoding de longitud 4.

Complejidad temporal: *O(n)*.

Complejidad espacial: *O(1)*.

Metaphone

Algoritmo fonético. Es más exacto que Soundex porque "entiende" las reglas básicas de la pronunciación en inglés. El encoding genera símbolos de longitud arbitraria.

Q-Gram

Q-Grams (ó N-Grams) es un algoritmo que consiste en generar los pedazos que componen un string. La distancia entre 2 strings estará dada por la cantidad de componentes que tengan en común.
A diferencia de los demás métodos no tiene en cuenta la posición. *Para frases conviene Q-grams*. Por ejemplo: Metaphone y Phonemeta, y Big Data con Data big.

$$Q - Gram(str1, str2) = \frac{\#QGInCommon(str1, str2) * 2}{\#QG(str1) + \#QG(str2)}$$

Si hay Q-Gramas repetidos, éstos sólo se pueden usar una vez, por ejemplo si str1 tiene 2 Q-Gramas iguales y str1 tiene una vez ese mismo Q-Grama, se dice que sólo uno matchea.

Complejidad temporal: *(n+m)*.

Complejidad espacial: *(n+m+Q)*.

Levenshtein

Es una métrica de distancia. Devuelve el mínimo número de operaciones requeridas para transformar una cadena de caracteres en otra, sea mediante inserción, eliminación o sustitución.

Complejidad temporal: *O(m*n)*.

Complejidad espacial: *O(m*n)* si la implementación utiliza la matriz completa, si no *O(n)*.

ExactSearch

Fuerza Bruta o Naïve

Si hay mismatch el algoritmo hace backtracking en el query y en el target.
La tabla de Next tiene en cada posición i la longitud del borde propio más grande para el substring query desde 0 hasta i.

Peor caso: Que el query no se encuentre en el target.

Complejidad temporal: *O(n*m)*. Sea |target| = n y |source| = m

Complejidad espacial: *O(1).*

KMP

No hace backtracking en el target. Preprocesa el query antes de la búsqueda y construye una tabla de Next del mismo tamaño del query.

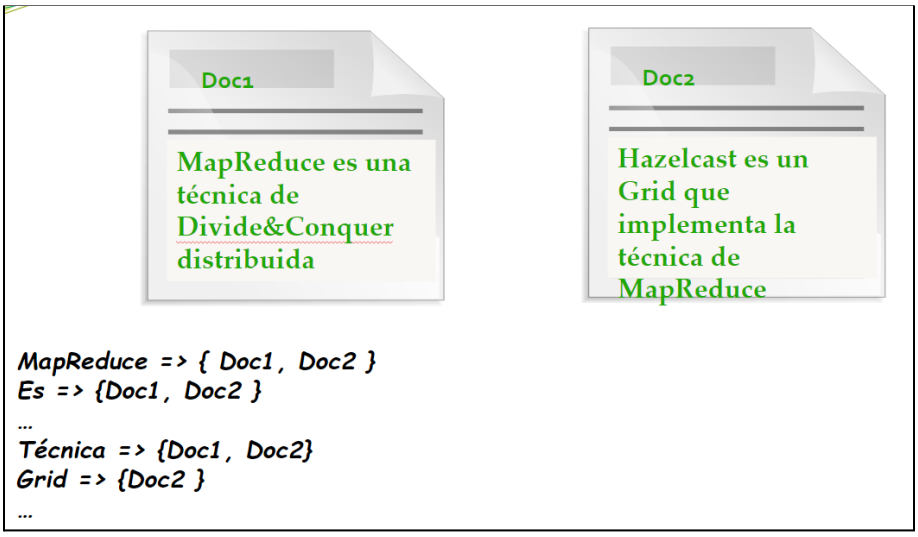
En la construcción de Next:

Complejidad temporal: O(m). Sea |target| = n y |source| = m

Complejidad espacial: O(m)

Lucene

Utilizado para buscar un query en un grupo de documentos. Genera índices de los susodichos para facilitar la búsqueda a través del mismo.
Indice: estructura que permite llegar rápidamente al dato buscado. Si se actualiza un documento de la colección debe actualizarse. Lucene usa un índice llamado Archivos Invertido: “conjunto de términos que dicen a qué documento pertenece”
Documento: se identifica por un número y es una colección de campos. Cada campo está compuesto por “nombre”, “tipo” y “valor”.



En cuanto al manejo de texto, hay 2 formas definir un campo de tipo texto indexable:

StringField

No se separa en tokens. Puede o no almacenarse. Si se almacena ocupa el doble de espacio: en el índice y en el almacenamiento.

TextField

Se separa en tokens. Si el texto proviene de un Stream no se almacena, en cambio si proviene de un String podría almacenarse. También ocuparía el doble de espacio.(Un stream ocuparía el triple). Este viene con diferentes clases para separar en tokens: *SimpleAnalyzer(), StandardAnalyzer(), WhitespaceAnalyzer() ,StopAnalyzer(), EnglishAnalyzer(), etc.*
Permite preguntar por diferentes tipos de Query:

- *TermQuery* acepta un único término.
- *PhraseQuery* buscar frases, palabras consecutivas
- *TermRangeQuery* espera buscar dentro un rango de términos que puede ser un intervalo [], (), [), (]. Se le indica extremo inferior, extremo superior y 2 booleanos para indicar el tipo de intervalo.
- *BooleanQuery*, etc.

Estructuras lineales

IndexService

IndexService es una interfaz que permite la búsqueda, inserción, remoción y contar ocurrencias en un arreglo.

Algoritmos de ordenamiento

Complejidades:

InsertionSort	$O(n^2)$
MergeSort	O(nlog(n))
QuickSort	$O(n^2)$
QuickSort (con búsqueda de pivote)	O(nlog(n))
BubbleSort	$O(n^2)$

Insertion Sort

Construye un arreglo en el que cada dato se inserta de manera ordenada.

Quick Sort

Ordena arreglos tomando un pivot y dejando del lado izquierdo los elementos más chicos y del derecho los más grandes.

Merge Sort

Ordena los arreglos dividiéndolos a la mitad y después ordenando las partes.
[Para visualizar el algoritmo merge sort.]

Evaluador de Expresiones (Stack)

Una expresión es una combinación de operadores y operandos.
Las expresiones se pueden clasificar según la notación que utilizan:

1. prefija: el operador se encuentra antes de los operandos sobre los que aplica.
2. infija: el operador se encuentra entre los operandos sobre los que aplica.
3. postfija: el operador se encuentra detrás de los operandos.

Elemento que está en el tope de la pila (previo)	Elemento que está siendo analizado (actual)						
	+	-	*	/	^	(	)
+	True	True	False	False	False	False	True
-	True	True	False	False	False	False	True
*	True	True	True	True	False	False	True
/	True	True	True	True	False	False	True
^	True	True	True	True	False	False	True
(	False	False	False	False	False	False	false

Listas

Una *Lista Lineal Simplemente Encadenada* es una estructura de datos compuesta por *0 o más nodos* que cumplen:

- Cada nodo (elemento) almacena 2 datos: Su información y una referencia al siguiente elemento.

Una *Lista Lineal Simplemente Encadenada con Header* es una estructura de datos compuesta por:

- Un elemento distinguido llamado header que tiene la referencia del primer elemento de la lista y además información global de la lista.
- Cada nodo/elemento (común) almacena 2 datos: Su información y la referencia al siguiente elemento.

Complejidad (sin header): Size y max de *O(N)*.

Complejidad (con header): Size y max de *O(1)*.

Grafos

BFS

Para visualizar BFS pero con un grafo predeterminado..

En BFS (Breadth-First Search) se recorre “de a niveles”. Se empieza marcando el nodo inicial y luego en cada paso se recorren y marcan los nodos no marcados que son adyacentes a un nodo marcado.

La complejidad de este algoritmo es O(N+E)

DFS

Para visualizar DFS pero con un grafo predeterminado.

En DFS (Depth-First Search) se recorre “por profundidad”. Se siguen los siguientes pasos, empezando por el nodo inicial:

Dado un nodo, se marca como visitado y elige uno de los vecinos no visitados al azar.

Nos paramos en ese nodo vecino y hacemos lo mismo que en el paso anterior.

Después de terminar de visitar un vecino en profundidad, visitamos el resto de ellos de igual forma.
La complejidad de este algoritmo es O(N+E)

Dijkstra

Para visualizar Dijkstra pero con un grafo predeterminado.
Los pasos del algoritmo de Dijkstra son:
Empezamos marcando todos los nodos con infinito excepto el nodo inicial. Este número representa el menor costo conocido de llegar hasta ese nodo.
En cada paso, miramos al nodo con menor costo que aún no haya sido visitado y actualizamos el costo de sus vecinos.
Este algoritmo funciona porque cuando nos paramos en un nodo sabemos que no es posible encontrar un camino más corto hacia él, ya que es el nodo con menor costo asociado. Si hubiera otro nodo con costo menor nos hubiéramos parado en ese otro.
La complejidad de este algoritmo es O((N+E)*log(N)) (el log(N viene del uso de una PriorityQueue).

Hashing

Open Addressing o Closed Hashing

Cada ranura puede tener null (está vacío o baja física). Aunque la ranura no esté vacía puede ser que el elemento no esté, ya que hay que manejar el concepto de bajas lógicas (además de las físicas). Es decir, una ranura representa 3 estados: tiene un elemento o no tiene un elemento (dado por baja lógica o bien por baja física).

- Rehasheo Lineal (linear probing). Si hay colisión en la ranura *i*, entonces intentar con la ranura *i+1*, y así siguiendo hasta encontrar que el elemento (se hace update) o encontrar un lugar vacío (baja física) y se inserta allí. Con esta técnica si hay lugar lo encuentra seguro.
- Resaheo Cuadrático (quadratic probing). El intervalo entre ranuras a usar, si hubiera colisión, será cuadrática. *Para verificar que una clave no se ingresa, con ver que en N intentos no encuentra lugar ya está (siendo N el tamaño de la tabla)* -Demo-.

El Borrado Físico se lo usa cuando la ranura que le sigue (la que se obtiene al aplicar *hash(i+1)*) está vacía (*null*).
El Borrado Lógico se lo usa en caso contrario, o sea cuando la ranura que le sigue está ocupada o bien borrada lógicamente.

Closed Addressing o Open Hashing o Chaining

Las colisiones se resuelven en una estructura auxiliar.
Cada ranura puede tener null, o bien una estructura auxiliar con las componentes que colisionaron en dicha ranura. Si la lista tiene un único componente, entonces no hubo colisión aún en esa ranura.
Como en un hashing no tienen orden los elementos, no se utiliza una lista ordenada simplemente encadenada, sino una lista.

Árboles

Árbol Binario de Expresiones

BST (Árbol Binario de Búsqueda)

Un Árbol Binario de Expresiones se usa para representar expresiones algebraicas. Los nodos internos representan operadores binarios y unarios. Las hojas representan los operandos, es decir, constantes y variables.
Según como se recorra el árbol (transversal) in-order, pre-order o post-order, se obtiene una expresión infija, prefija o postfija respectivamente y se obtiene la correspondiente expresión.

AVL

Para visualizar la inserción y borrado en un árbol AVL.
Un AVL es un BST donde en cada nodo la diferencia de alturas entre sus 2 subárboles es a los sumo 1.
En caso de desbalance, se utiliza el nodo más cercano al insertado:

- Si se inserta en la misma dirección (línea), rotación simple en el nodo desbalanceado.

- Si se inserta cambiando la dirección respecto al hijo (por ejemplo en el subárbol izquierdo del hijo derecho): rotación en el hijo y luego en el nodo.

Fibonacci

El Árbol de Fibonacci es el AVL que presenta el peor desbalance posible. Se define de la siguiente manera:

1. Fibonacci de orden 0 es el árbol nulo.
2. Fibonacci de orden 1 es un nodo.
3. Fibonacci de orden h es un árbol que tiene:
    a. Como hijo izquierdo un Fibonacci de orden $h - 1$.
    b. Como hijo derecho un Fibonacci de orden $h - 2$.

Sea un AVL de altura h con la menor cantidad de nodos posibles en esa altura. Este AVL de altura h tendrá la siguiente cantidad de nodos:

$1\,nodo + (cantNodos\ AVL\ altura\ h - 1) + (cantNodo\ AVL\ altura\ h - 2)$

$H = h$ Cant de Nodos = ?	$Fibo(h + 3) - 1$

Generador de números de Fibonacci (numberempire.com)

Así, sea un AVL con altura h, su cantidad de nodos es $fibo(h + 3) - 1$. Ese es el AVL que menos nodos tendrá. Otros AVL tendrán un número n de nodos, donde $n \geq fibo(h + 3) - 1$.

h es $O(log\ n)$, o sea, para un AVL con n nodos, la altura está acotada por $O(log\ n)$.

Hay diferentes algoritmos para garantizar que un árbol sea balanceado y esta propiedad sea invariante ante inserciones/borrados.

Las transformaciones que hay que hacer para mantenerlo apropiado llegan en general $O(log\ n)$

Árbol B (orden N)

Para visualizar la inserción y borrado en árboles B.

Si es de orden *N*, cada nodo puede tener de *N* hasta *2*N* claves y *M + 1* hijos (siendo *M* el número de claves).

Red Black Tree

Para visualizar la inserción y borrado en un red black tree.

Son árboles balanceados que siguen las siguientes reglas:

- Todo los nodos son **negros** o **rojos**.
- La raíz es **negra**, al igual que los nil o null.
- Si un nodo es **rojo**, no puede tener hijos **rojos**.
- Todo camino tienen que tener la misma cantidad de nodos **negros**.

Inserción

Hay cuatro casos posibles para la inserción:

- Si el nodo a insertar es la raíz → se pinta de **negro**.
- Si el tío del nodo a insertar es **rojo** → se re-colorea el padre, el abuelo y el tío.
- Si el tío del nodo a insertar es **negro** (forma un triangulo) → se rota el padre del nodo a insertar.
- Si el tío del nodo a insertar es **negro** (forma un linea) → se rota el abuelo del nodo a insertar y re-colorear padre y abuelo originales.

Complejidad temporal**:** *O(log(n))*.

Heurística

Búsqueda Exhaustiva

Para buscar todas las posibles soluciones explorando el espacio de soluciones en forma implícita.
¿Para qué sirve? Un problema que tiene muchas soluciones y las quiero todas.
Consideraciones: el grafo es implícito (no lo genero). Típicamente se
implementa con recursión (o sea Stack) aunque puede hacerse con una
Queue. Ej: DFS and BFS.
Idea para esta técnica (típicamente recursiva):

- Si el nodo no puede expandirse más (no hay más opciones a partir de él) → retornar/imprimir el resultado.

- Sino, por cada posibilidad para ese nodo de expandir un próximo nivel (ciclo for):
    - El nodo puede resolver un caso pendiente.
    - Explorar nuevos pendientes (soluciones quizás parciales).
    - El nodo puede deshacer/quitar el caso pendiente generado.

Si lo resuelvo con Búsqueda Exhaustiva el chequeo de las restricciones lo hago al final de la exploración de todas las posibles soluciones.
SE EXPLORA TODO EL ESPACIO DE SOLUCIONES POSIBLES.

Backtracking

Ante la presencia de restricciones, pueden aprovecharlas para no explorar todo el grafo de soluciones y PODAR aquellos nodos que no conducen a la solución.
Ahí en donde las técnicas de Backtracking entran en juego: no expande innecesariamente nodos que ya se saben (gracias a la restricción) no conducirán a la solución.
SE BUSCA EVITAR NO EXPANDIR MÁS UN NODO DEL GRAFO DE SOLUCIONES.

Divide & Conquer

Técnica que descompone un problema de tamaño *N* en problemas más pequeños que tengan solución. Finalmente, se debe proponer cómo construir la solución final a partir de las soluciones de los problemas menores.
Ej: *Mergesort, Quicksort, Búsqueda en un BST, Búsqueda en un arreglo ordenado*.

Greedy (Algoritmos Ávidos)

Técnica que busca en cada etapa un óptimo local con el objetivo de llegar al óptimo global (aunque de esta forma no siempre se consigue el óptimo global).
Ej: *algoritmo de Kruskal* para encontrar el mínimo árbol generador de un grafo.

Fuerza Bruta (con Stack o Queue)

Calcula y enumera todas las posibles soluciones. Si se agregan restricciones, ej:
se pide “la mejor”, recién en el final evalúa cual es la mejor. Es decir, en las hojas se evalúa si se satisface la restricción pedida.

Pero si hubiera una restricción que me permite “podar” el árbol en ramas que son innecesarias (las que no conducen a la solución) → mucho más eficiente!. Backtracking es una opción. Ante la presencia de restricciones, no exploran todas las posible soluciones, sino solo las prometedoras.
Ej: *N-Queens.*

Programación Dinámica

¿Y si no recálculo innecesariamente cosas que calculé previamente? Entonces se usa *Programación Dinámica*.
Permite almacenar valores que se calcularon previamente en soluciones anteriores para reusarlos en vez de re-calcularlos reiteradamente.
Ej: *Levenshtein, Dijkstra, Ackerman, Fibonacci.*

Branch & Bound

Técnica que ramifica y poda, pero tiene en cuenta un “costo” asociado a la solución. Busca la solución óptima.
Ej: *El problema del viajante*.