Universidad del Valle de Guatemala
Departamento de Ciencias de la Computación
CC3074 - Minería de Datos
Catedrático: Gabriel Brolo Tobar
Auxiliar: Ángel Leonel Higueros Cifuentes

Proyecto 2

Optimal Binary Search Tree

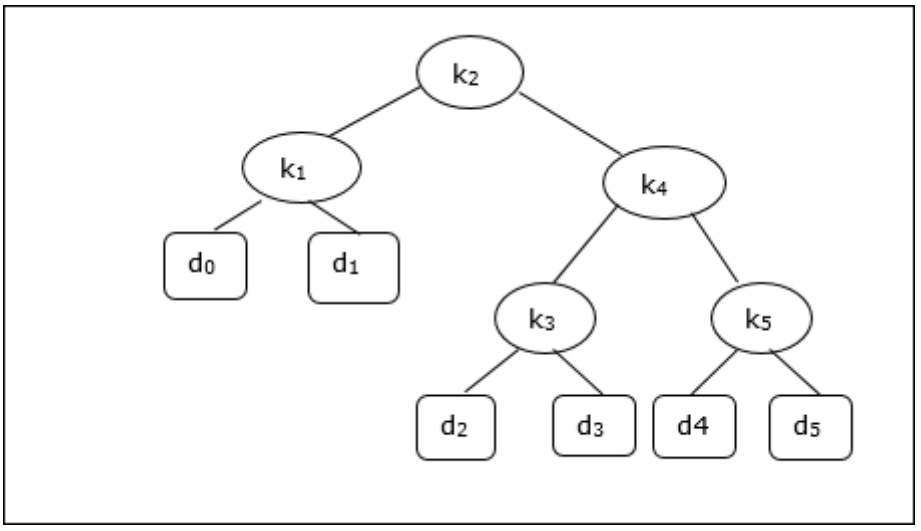
Daniel Armando Valdez Reyes - 21240
Mario Antonio Guerra Morales - 21008

Guatemala, 1 de abril de 2024

- **Problema:**

  El problema de la "Optimal Binary Search Tree" (Árbol de Búsqueda Binaria Óptimo) se basa en encontrar una estructura de árbol binario de búsqueda que minimice el costo esperado de búsqueda de elementos. Este problema es común en la optimización de estructuras de datos para la búsqueda eficiente de elementos en conjuntos ordenados.

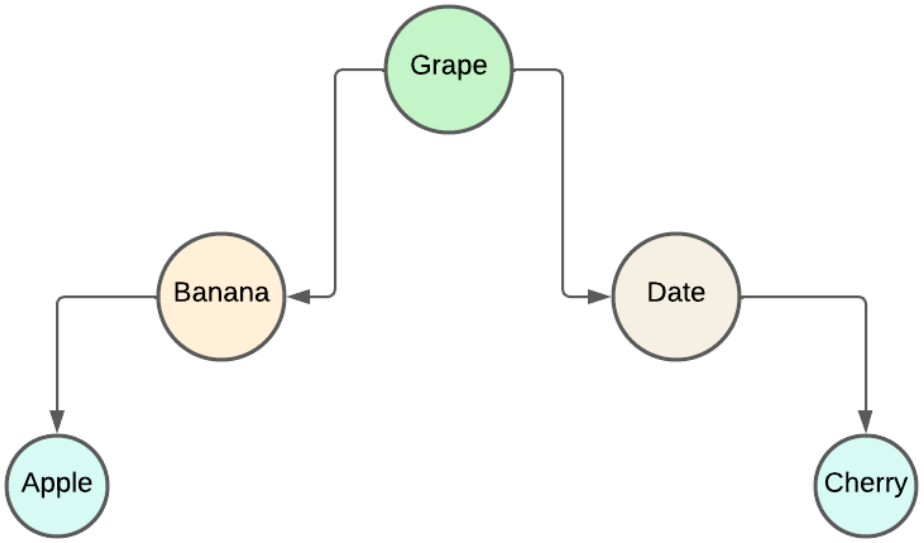
  Imaginemos que tenemos un conjunto de claves (por ejemplo, palabras en un diccionario) junto con sus probabilidades de búsqueda. Queremos organizar estas claves en un árbol binario de búsqueda de manera que las claves más buscadas estén más cerca de la raíz del árbol, lo que minimiza el tiempo de búsqueda esperado.



  El costo esperado de búsqueda en un árbol binario de búsqueda está determinado por la profundidad de cada clave en el árbol. Cuanto menor sea la profundidad de una clave, menor será el costo de búsqueda. El objetivo del problema es encontrar la disposición óptima de las claves en el árbol binario de búsqueda de manera que el costo total esperado de búsqueda sea mínimo.

  - Para ilustrar el problema de la Optimal Binary Search Tree supongamos que tenemos las siguientes claves (palabras) junto con sus probabilidades de búsqueda:
    - "apple": 0.15
    - "banana": 0.10
    - "cherry": 0.05
    - "date": 0.10
    - "grape": 0.20

Queremos construir un árbol binario de búsqueda óptimo para estas claves de modo que minimice el costo esperado de búsqueda. El costo esperado de búsqueda se define como la suma del producto de la profundidad de cada clave en el árbol y su probabilidad de búsqueda.



En este árbol, "grape" es la raíz, "banana" y "date" son sus hijos izquierdo y derecho respectivamente, y "apple" y "cherry" son los hijos izquierdo y derecho de "banana" y "date" respectivamente.

El costo esperado de búsqueda para este árbol sería:

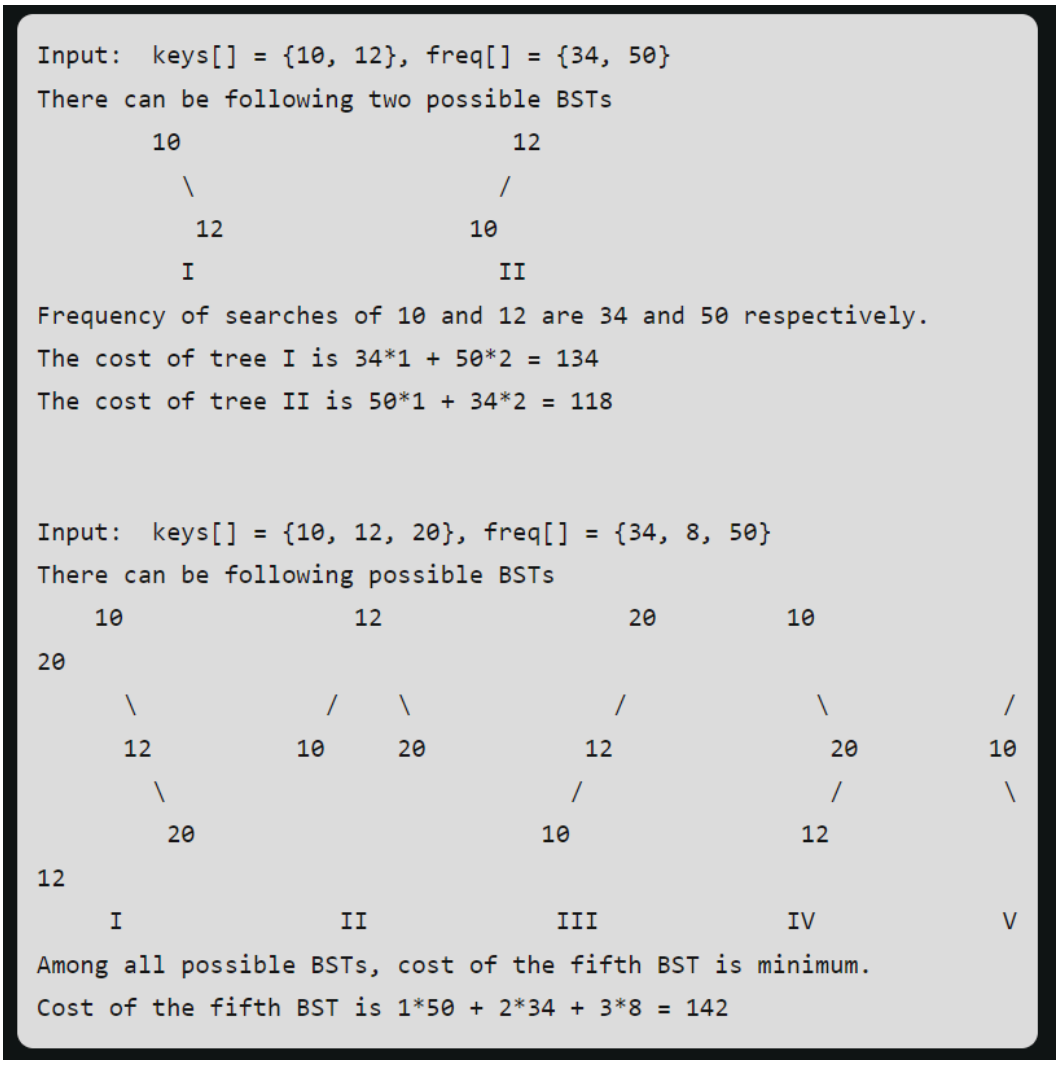
- "grape": Profundidad 1 * Probabilidad 0.20 = 0.20
- "banana": Profundidad 2 * Probabilidad 0.10 = 0.20
- "date": Profundidad 2 * Probabilidad 0.10 = 0.20
- "apple": Profundidad 3 * Probabilidad 0.15 = 0.45
- "cherry": Profundidad 3 * Probabilidad 0.05 = 0.15

Por lo tanto, el costo total esperado de búsqueda para este árbol sería 0.20 + 0.20 + 0.20 + 0.45 + 0.15 = 1.20.

El problema es encontrar la disposición óptima de las claves en el árbol de manera que este costo total esperado de búsqueda sea mínimo. Esto implica elegir qué clave debe ser la raíz del árbol, qué claves deben estar a la izquierda y a la derecha de cada nodo, y así sucesivamente, de modo que se minimice el costo total esperado de

búsqueda. Este es el desafío que aborda el algoritmo de Optimal Binary Search Tree, utilizando programación dinámica para encontrar la solución óptima de manera eficiente.

- Otros ejemplos:



- **Algoritmos de Solución:**

  El algoritmo en cuestión para determinar la solución de este problema es el siguiente:

```
3 usages  new *
def OptimalBST(dataList, frequencyList):
    if len(dataList) == 0:
        return None
    if len(dataList) == 1:
        return Node(dataList[0])
    else:
        max_frequency = max(frequencyList)
        max_indices = [i for i, freq in enumerate(frequencyList) if freq == max_frequency]

        if len(max_indices) % 2 == 0:
            # Si hay un número par de índices máximos, devuelve el índice en la mitad del rango
            mid = max_indices[len(max_indices) // 2 - 1]
        else:
            # Si hay un número impar de índices máximos, devuelve el índice central
            mid = max_indices[len(max_indices) // 2]

        root = Node(dataList[mid])
        root.left = OptimalBST(dataList[:mid], frequencyList[:mid])
        root.right = OptimalBST(dataList[mid + 1:], frequencyList[mid + 1:])
        return root
```

  En esta función, se utilizan ambos conceptos, tanto Divide and Conquer como también Programación Dinámica.

  La aplicación de Programación Dinámica en este algoritmo consiste en obtener el índice óptimo de la lista, en el cual el programa definirá para poder realizar las particiones para hallar el árbol en base a este índice. Esto se hace por medio de los términos de frecuencia que se muestran en cada uno de los índices de los datos para hacer el árbol de búsqueda.

  Mientras que, la aplicabilidad de Divide and Conquer se basa principalmente en estas líneas de código:

  max_frequency = max(frequencyList)
  max_indices = [i for i, freq in enumerate(frequencyList) if freq == max_frequency]

  Estas indican que, al obtenerse los índices óptimos anteriormente por medio de programación dinámica y la frecuencia de estos índices, se harán las particiones por la mitad en ese índice para encontrar el árbol de búsqueda óptimo para la entrada que se envía a la función. Dividiéndolo en problemas más pequeños y, de esta manera, encontrando la solución óptima

de cada uno de estos subproblemas por medio de la recurrencia estipulada en la definición de root como el nodo que representa la mitad del árbol, o en este caso, el índice óptimo y sus hijos izquierdo y derecho.

Esta función para obtener el árbol binario de búsqueda óptimo (o OBST), consta también de ciertas condiciones. Primero, para que sí sea considerado el árbol de búsqueda como óptimo, debe de encontrar el índice que más se ajuste a esa descripción. Utilizando la programación dinámica para esto, ya que según lo aprendido, debemos buscar el índice que posea la mayor frecuencia posible, es decir, necesitamos que se maximice el valor de este para obtener un mejor resultado. Por lo cual, si no se implementa este paradigma, es muy probable que el algoritmo no sea capaz de brindarnos una solución óptima para este problema, convirtiéndose en un algoritmo para árboles de búsqueda binarios únicamente, más no aquel que encuentre la optimización del árbol.

Mientras que, otra de las condiciones de este algoritmo es la utilización de Divide and Conquer para dividir el problema en subproblemas más pequeños. Esta implementación es indispensable para poder realizar el árbol de búsqueda, ya que para este se requiere de dividir en nodos hijos respecto al nodo padre que representa la raíz del árbol. Por otro lado, se emplea para que sea posible encontrar la solución óptima en el problema, ya que, al incluirse el índice óptimo para realizar la partición del árbol, se realizan las denominadas subestructuras óptimas, ya que se indica que si una solución óptima se compone de soluciones óptimas en los subproblemas ya definidos.

- **Análisis Teórico:**

  Al ser un único programa, se realizará únicamente un procedimiento de análisis y tasa de crecimiento solamente para este algoritmo.

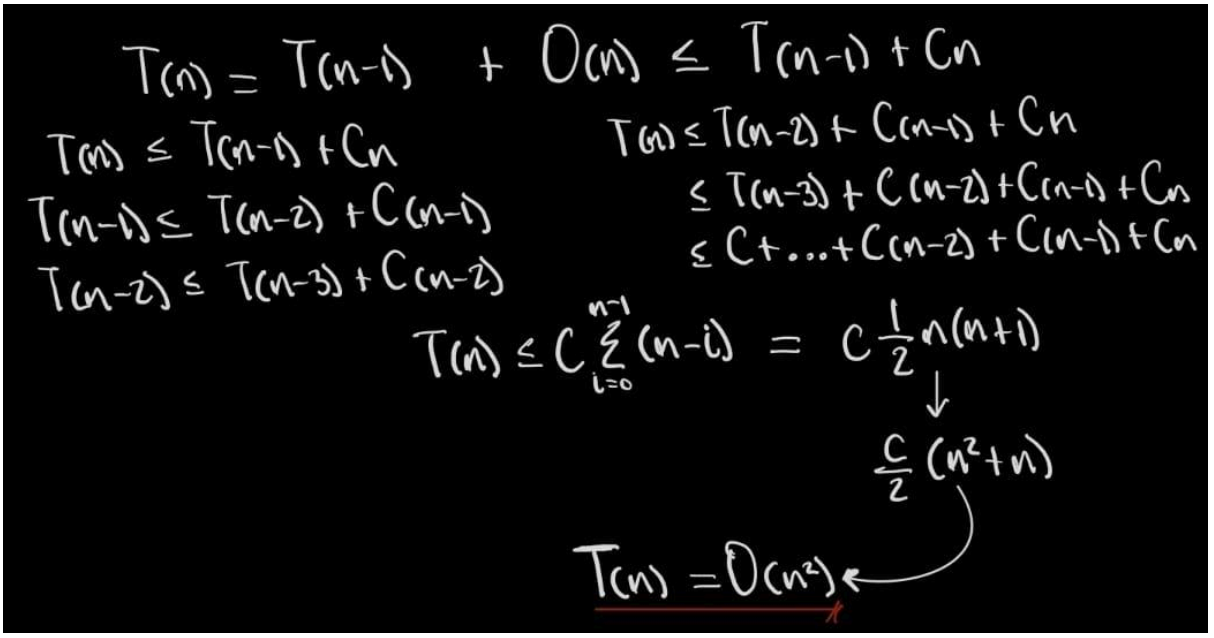
  Considerando que, la programación dinámica en este algoritmo se utiliza para maximizar las frecuencias y de esta forma, incluirla en los índices máximos para encontrar la mitad del arreglo dataList. Se puede inferir que el tiempo de ejecución que tiene la parte de programación dinámica es de O(n).

  Mientras que, por parte de Divide and Conquer, se encuentra recurrencia un total de dos veces, ya que empleamos esta para encontrar los nodos hijos, o izquierdo y derecho del índice en la mitad del arreglo. Esto significa que esta parte tendría como una relación de recurrencia de 2, en el mejor caso, implicando un tiempo de ejecución que dependerá del tamaño del arreglo y de la recurrencia dentro del algoritmo. Por parte del peor de los casos, sería que el

arreglo en donde no está ordenado, por lo que tiene que buscar ordenarlo para que pueda realizar el procedimiento correcto para encontrar el árbol de búsqueda binaria óptimo.

El procedimiento de este algoritmo es el siguiente:

El peor caso se da cuando el índice máximo se encuentra cerca de cualquiera de los extremos del arreglo, por lo que el programa debe ir ordenando el arreglo poco a poco hasta llegar al índice máximo. Esto nos quiere decir que el tiempo de ejecución de este ordenamiento sería de n-1, sumado a la parte de programación dinámica, se considera que el tiempo de ejecución está definido por $T(n) = T(n-1) + O(n)$. Obteniendo así este procedimiento:



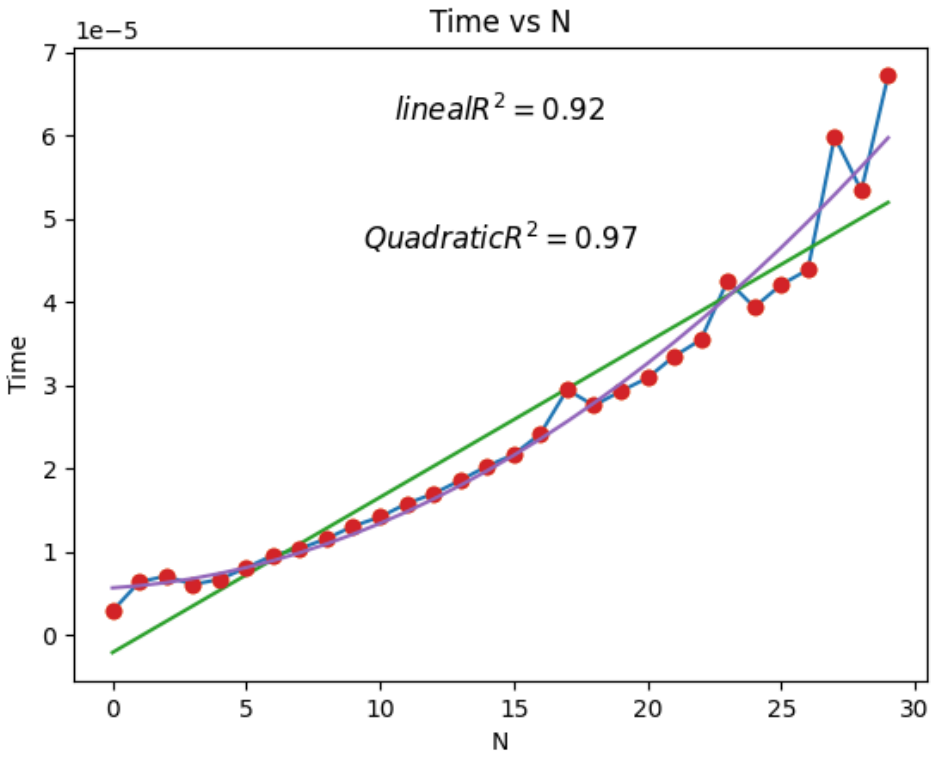
Esto debido a que, como se mencionó antes, se hacen las recurrencias necesarias a los lados izquierdo y derecho del índice que está definido en esa línea de código, se define como una expresión cuadrática. Tomando en cuenta que por cada elemento n en la lista irá recorriendo uno a uno hasta llegar al elemento que se considera el índice con mayor valor máximo. Esto puede definirse como una serie en la cual, el tiempo de ejecución será este: $T(n) \leq C \sum_{i=0}^{n-1} (n - i)$. Al observar la serie, se considera que posee la fórmula $\frac{1}{2}n(n + 1)$, esto multiplicado por la constante C. Al hacer las multiplicaciones correspondientes, se obtiene la forma $\frac{C}{2}(n^2 + n)$. Al hacer esto, se considera que el tiempo de ejecución para el worst case de este algoritmo es $O(n^2)$, ya que es la variable con mayor grado en la ecuación.

- **Análisis Empírico:**

  Para el análisis empírico, se utilizó un conjunto de datos almacenados en el archivo "data2.csv", que consta de dos columnas: "test Array", que contiene los listados de prueba utilizados para la creación del árbol de búsqueda, y "test Frequency", que indica la frecuencia con la que se busca cada elemento normalmente. Se prepararon en total 30 conjuntos de prueba con diferentes tamaños, los cuales representaron la variable N en este caso.

  Se registró el tiempo que tarda el algoritmo en ejecutarse con cada conjunto de prueba y se realizó su respectiva representación gráfica. A partir de esta gráfica, se llevó a cabo tanto una regresión lineal como una regresión cuadrática para verificar su ajuste.

  Los resultados mostraron un coeficiente de determinación (R^2) de 0.92 para la regresión lineal y de 0.97 para la regresión cuadrática. Basándonos en estos datos, se concluyó que el algoritmo cumple con el tiempo de ejecución teórico estimado, es decir, O(n^2).



- **Link del repositorio:**
  https://github.com/Danval-003/AlgorithmOptimalSearchBinaryTree

Referencias.

- Jain, S. (2023, July 10). Optimal Binary Search Tree | DP-24. GeeksforGeeks. https://www.geeksforgeeks.org/optimal-binary-search-tree-dp-24/