Guía para aprender algoritmos: resumen del curso

En esta clase de lectura encuentras un resumen de todo lo que acabas de aprender en este curso. De esta forma, vas a reforzar los conocimientos que adquiriste.

¿Qué son las estructuras de datos y algoritmos?

Las estructuras de datos se utilizan para guardar y organizar información. Dependiendo de las necesidades del proyecto se elige un tipo diferente de estructura.

Ninguna estructura de datos es perfecta, cada una tiene ventajas y desventajas dependiendo el caso de uso.

Por un lado, las **estructuras de datos lineales** son en las que los elementos están ordenados de forma secuencial, un elemento después de otro, como por ejemplo los arrays, las pilas, las colas y las listas enlazadas. Todas estas serán explicadas a detalle en el Curso Avanzado de Algoritmos y Estructuras de Datos: Patrones de Arreglos y Strings y el Curso Avanzado de Algoritmos: Estructuras de Datos Lineales.

Por otra parte, las **estructuras de datos no lineales** son en las que la información no está organizada secuencialmente, sino que se estructura de forma jerárquica donde los elementos están conectados entre sí. Este tipo de estructura será detallado en el Curso Avanzado de Algoritmos: Grafos y Árboles.

Un algoritmo es un conjunto de instrucciones que resuelven un problema. Toma un conjunto de entradas y produce la salida deseada.

¿Por qué importan las estructuras de datos y algoritmos?

Aprender sobre estructuras de datos, algoritmos y sus patrones enseña a diseñar soluciones que prioricen los atributos de calidad más importantes para el proyecto que estemos realizando, pues para un problema pueden haber muchas posibles soluciones, pero si por ejemplo lo que nos importa es el tiempo de ejecución, al programar una solución debemos entender cómo podemos hacer que esta sea realmente eficiente así se sacrifiquen otros aspectos.

Estas habilidades nos ayudan a crear una **estructura mental**, una forma de resolver problemas… que nos sirve sin importar qué tecnologías estemos utilizando.

Es por eso que en empresas de tecnología grandes como Google, Microsoft o Amazon las entrevistas evalúan cómo resolver un problema utilizando estructuras de datos y algoritmos. Las soluciones que esta clase de compañías necesitan deben ser escalables, ya que millones de usuarios hacen uso de sus productos y servicios a diario.

El tiempo y la memoria utilizada importan mucho y hacen la diferencia no solo en el dinero que les cuesta correr un programa o almacenar información, sino también en **la experiencia del usuario final**.

Estructuras de datos y algoritmos a estudiar

Estructuras	Conceptos y algoritmos
Arreglos	Notación Big O
Strings	Algoritmos de ordenamiento
Listas encadenadas	Algoritmos de búsqueda
Tablas de Hash y Conjuntos de Hash	DFS y BFS
Pilas	Manipulación de bits
Colas	Recursión
Árboles	Programación dinámica
Grafos	Dos Apuntadores
Tries	Apuntador rápido y lento
Ventana Deslizante	

¿Cómo es (comúnmente) una entrevista con problemas de programación?

La entrevista de programación es una entrevista técnica orientada a la resolución de problemas que se utiliza para evaluar las habilidades de un candidato.

Aunque cada compañía tiene procesos diferentes en la entrevista de programación de muchas medianas y empresas grandes de tecnología como Google, Microsoft, Apple y Amazon, entre otras, se hará mucho énfasis en comprobar tu comprensión de conceptos como las estructuras de datos, los algoritmos y el diseño de sistemas.

Normalmente, el entrevistador te hará una serie de preguntas técnicas. Pueden solicitarte responder a estas preguntas escribiendo código a través de un editor colaborativo en tiempo real o incluso en un tablero, depende de si es presencial o en línea.

En los primeros minutos el entrevistador se presentará contigo y te hará preguntas sobre quién eres, tu experiencia, tus proyectos, etc. Este es tu momento para presentarte y dejar una impresión duradera en el entrevistador, así que no subestimes la importancia de este paso.

Luego el entrevistador te dará el problema ambigüo. Y tu tarea es solucionarlo de forma eficiente.

¿Cómo resolver un problema de programación?

Los programadores estamos todo el tiempo resolviendo problemas o encontrando formas de mejorar soluciones ya existentes. Es por eso que muchas entrevistas de programación se enfocan en evaluar nuestra habilidad para resolver problemas.

El proceso que explicaré posteriormente es útil para **destacar en este tipo de entrevistas**. Aunque más allá de esto, es una aproximación para crear soluciones que vayan más allá de la aplicación de conceptos técnicos para arreglar algo que falla o crear algo nuevo; soluciones que nacen desde la **comprensión profunda de una problemática** y cómo esta es moldeada por los diferentes actores que la conforman.

Comprensión del problema

Muchos candidatos se apresuran a programar tan pronto como leen el problema sin asegurarse de que realmente lo entienden. En vez de hacer eso, después de escuchar el problema, respira profundamente y antes de empezar a averiguar cómo resolverlo, **comienza con aclarar el problema**. Sin pensar en la implementación, aclara con el entrevistador lo que debe hacer esta función con ejemplos reales.

¡Haz preguntas!

Utiliza todos los recursos a tu disposición para asegurarte que tus expectativas están alineadas con las esperadas, que tu entendimiento del problema es el correcto. Puedes intentar explicar el problema con tus propias palabras o mostrando con ejemplos reales de entradas cuál sería su salida esperada. Incluso puedes intentar diagramar el problema.

Tienes muchas maneras de conocer a detalle un problema. Encuentra qué método te sirve más a ti. Asegúrate de tener claridad del "qué" tienes que solucionar antes de pensar en cómo hacerlo.

Es importante hacer preguntas relacionadas con las entradas y salidas.

Por ejemplo, si es un número:

- ¿Puede este ser negativo?
- ¿Pueden haber entradas extremas, números muy altos o bajos?

Si es una lista de números:

- ¿Pueden haber duplicados? ¿Están ordenados?

Si es una matriz:

- ¿Qué pasa si está vacía?

Si es un string:

- ¿Recibo mayúsculas y minúsculas?
- ¿Puedo recibir caracteres especiales?

También puedes preguntar cosas como:

- ¿Qué atributos de calidad son los más importantes a priorizar al plantear una solución?
- ¿Qué tan grande es la entrada?
- ¿Cuál es el rango de valores que debería esperar?
- ¿Cada cuanto se utilizará la función?
- ¿En qué casos se utilizará la función?

Trata de entender los casos de uso normales de la función, pero también de identificar los posibles casos extraños y compártelos con el entrevistador.

Diseña tu algoritmo

Una vez tengas claro el problema, empieza a diseñar una solución y **no te apresures a implementarla aún**.

Esta primera aproximación no tiene que ser la más eficiente. Puedes iniciar describiendo cómo resolverlo con fuerza bruta y desde ahí empezar a optimizar la solución. Es muy valioso si describes varias formas de solucionar el mismo problema, compartes las ventajas y desventajas de cada una y tomas decisiones sobre cual implementar.

Por ejemplo, si identificas un algoritmo con mejor complejidad espacial y otro con mejor complejidad temporal, comparte ambos, qué decides sacrificar y justifica cuál decides implementar.

Es natural a veces no saber por dónde empezar o cómo resolver algo. A mí me ha servido empezar por descomponer cada problema en problemas más pequeños y sencillos, luego hablar de cómo resolvería cada uno de forma lenta y sencilla y finalmente pensar cómo podría optimizarse con algún algoritmo, estructura, patrón o tecnología.

Explica o diagrama tu solución, discútela con el entrevistador, dale espacios en los que pueda ir dando retroalimentación, escucha sus comentarios, construye con ella o él, demuestra tus habilidades de trabajo en equipo.

Incluso si te sientes perdido, comunícalo, probablemente te darán pistas o ayudarán a clarificar las dudas. **No tengas miedo de ser sincero**, el entrevistador no quiere que falles, quiere conocerte y saber qué tanto encajas con el perfil que buscan.

Una vez hayas llegado a una solución que consideras apropiada u óptima, es hora de iniciar a programarla. A veces es útil preguntar al entrevistador si el considera que ya es momento de implementarla o si algo no le quedó claro de tu solución. Es muy importante interactuar con el entrevistador, saber si sigue conectado con lo que estás hablando, mantener su atención y sobre todo estar seguro de que ha entendido tu solución y todo lo que pensaste para llegar a ella.

Después de programar: prueba tu código

En resumen, inicia entendiendo el problema, haz preguntas, diseña una solución, optimiza, toma decisiones entre varias posibles soluciones, implementa… y al final no olvides probar tu código.

Notación Big O

Es un lenguaje y métrica para describir la cantidad de recursos utilizados al correr de un algoritmo. Por ejemplo, se puede medir la velocidad de un algoritmo y como este tiempo aumenta al incrementar la cantidad de datos procesados. En este tipo de notación se define la complejidad de un algoritmo con el peor caso posible.

Para definir la complejidad escribimos O y entre paréntesis la función que describe el aumento de recursos dependiendo la cantidad de datos recibidos como entrada.

A continuación puedes ver algunas de las funciones que definen la complejidad de un algoritmo.

En dónde las funciones de más a menos eficientes son:

- O(1)
- O(log n)
- O(n)
- O(n log n)

- O(n^2)
- O(n^3)
- O(n^k)
- O(n!)

Cuando en un algoritmo tenemos como complejidad una constante por una función, **el valor constante es despreciable**, por lo que no lo consideramos. Por ejemplo, si la complejidad de una solución es `O(45 * log n)`, la complejidad puede ser descrita simplemente como `O(log n)`.

Por otro lado, si un algoritmo tiene diferentes complejidades, aunque la complejidad sea la sumatoria de estas, realmente solo consideramos la mayor de ellas. Por ejemplo, si la complejidad de una solución es `O(n + log n + n2)`, podemos considerar la complejidad como `O(n2)`.

Algoritmos Greedy

En esta técnica se resuelve un problema seleccionando la mejor opción disponible en el momento y sin revertir decisiones anteriores. Estos algoritmos suelen ser fáciles de implementar, así no garanticen alcanzar la solución óptima global.

Este método es una buena alternativa a utilizar cuando se puede encontrar una solución óptima al problema, eligiendo la mejor opción en cada paso sin reconsiderar los pasos anteriores una vez elegidos, así la solución global óptima del problema está definida por la solución óptima de sus subproblemas.

Algunos algoritmos conocidos solucionados así son:

- Ordenamiento por selección
- El problema de la mochila
- Árbol de expansión mínima
- El problema del camino más corto de una sola fuente
- Prim
- Kruskal
- Dijkstra
- Ford-Fulkerson

Dividir y Conquistar

Esta técnica permite reducir la complejidad al resolver problemas extensos al dividir un problema difícil en subproblemas más pequeños que son llamados recursivamente hasta llegar al estado deseado. Después de tener la solución para cada problema es necesario tener una función que combine apropiadamente los resultados.

Este método debe utilizarse cuando los mismos subproblemas no se evalúan muchas veces, pues en caso contrario es mejor utilizar la Programación Dinámica o la Memorización (conceptos encontrados en cursos más adelante).

Un ejemplo de esta estrategia aplicada es esto es el algoritmo de búsqueda binaria.

Recursos Útiles

Para resolver problemas de programación:

- Leetcode

- HackerRank
- CodeChef

Para visualizar estructuras de datos y algoritmos:

- Visualgo
- University of San Francisco

Para practicar entrevistas con otras personas

- Pramp
- interviewing .io

Libros:

- Cracking the Coding Interview, Gayle Laakmann McDowell
- Grokking Algorithms, Aditya Bhargava
- Clean Code, Robert Martin
- Introduction to Algorithms, Thomas H. Cormen

Próximos cursos de estructuras de datos y algoritmos avanzados: