Ayudantía 11 - Algoritmos y Complejidad
Algoritmos aleatorios y aproximados

Randomized complexers

1. Algoritmos Deterministas

Todos los algorimos que hemos visto a lo largo del curso han sido deterministas. Tenian propiedades como un "buen" comportamiento en el peor caso y la entrega de soluciones exactas, invariables sin importar cuantas veces corrieramos el algoritmo.
Gran parte de la complejidad de los algoritmos (Algoritmos y complejidad) radica en la poca flexibilidad que tienen y en la dificultad de su diseño y sus *chamullisticas* demostraciones como las que la gente presenta en tareas y certamenes. Para terminar el curso veremos una(s) clase(s) de algoritmos que involucran azar.

2. Algoritmos Aleatorizados

Como el nombre lo dice, involucra azar como parte de sus operaciones. Un algoritmo aleatorizado generalmente tienen un buen comportamiento en el **caso promedio**, puede conseguir respuestas exactas con **alta probabilidad** aunque generalmente prefiere respuestas que estan **cerca** de la respuesta correcta. De este estilo hay muchos algoritmos muy faciles de entender pero con analisis muy densos y complejos.
Por ejemplo:

Las Vegas algorithms : Como `QuickSelect` y `QuickSort`(que tambien es D&C!), Siempre encuentran la respuesta correcta, pero se toman su tiempo.

Monte Carlo algorithms : Como `Karger`(para encontrar el corte minimo de un grafo) o las integraciones que vimos en ayudantias anteriores. Puede que no encuentren la solución correcta, pero la encuentran rapido.

Estructuras de Datos aleatorias : Como las `Hash Tables`, los `Treaps`(BT + Heaps) y los `De Bruijn graph`, tienen un exelente rendimiento con casos generales de uso.

3. Quicksort

Ya lo conoce bien, pero por si no se acuerda[1] :

- Secuencia de largo 0 esta ordenada

[1] https://www.youtube.com/watch?v=ywWBy6J5gz8 Quicksort con primer elemento como pivote (La herencia cultural de Hungria no son solo los bailes sino tambien los algoritmos(Aunque Quciksort lo invento un Ingles(Que sigue vivo)))

- Si hay que ordenar, primero elige un pivote, corre un paso de particionamiento para ponerlo en su lugar.
- Recursivamente aplica quicksort a la izquierda y derecha del puvote.

Como buen estudiante de informatica que ha usado quicksort muchas veces habrá notado que el comportamiento del algoritmo depende de la elección del pivote, aunque siempre de el resultado correcto.

3.1. Mejor Caso

Elige la **mediana** como pivote: Al ser la mediana, la ponemos de inmediato al centro del arreglo y nos quedan dos mitades iguales:

$$T(0) = \Theta(1)$$
$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n)$$

Por Master Theorem:

$$T(n) = \Theta\left(n \log n\right)$$

3.2. Peor Caso

Elige el **minimo** o el **maximo** como pivote: Al ir en una esquina, nos queda un solo arreglo de largo $n-1$:

$$T(0) = \Theta(1)$$
$$T(n) = T(n-1) + \Theta(n)$$

Recurrencias, ojalá generatrices:

$$T(n) = \Theta\left(n^2\right)$$

3.3. Randomized QuickSort

Elige pivotes **aleatorios**: Como estamos trabajando con probabilidades, no es tan simple escribir recurrencias y resolver generatrices, así que no lo haremos, el ramo llega hasta aqui, pero deberia llegar a que es $\Theta\left(n \log n\right)$ tambien!

4. Algoritmos Aproximados

Con la infinidad de divertidos problemas NP-hard para resolver, pero sin el tiempo para resolverlos, surge la idea de buscar soluciones *cercanas* a la óptima(**Garantias de cercania Probable**), pero en un tiempo acotado.
El diseño y analisis de los algoritmos aproximados involucra densas pruebas matematicas que certifican la calidad de las soluciones entregadas en el peor caso. Esto los **distingue** de las heuristicas como el recocido(annealing) o los algoritmos geneticos, en el sentido de que las heuristicas entregan soluciones razonables para algunos inputs, pero no tienen cotas superiores de tiempo

de ejecucion, ni dan señales de cuanto podrian tardar en encontrar buenas soluciones (TL;DR la inteligencia artificial es estupida, evite las metaheuristicas).

Un ejmplo simple de algoritmo aproximado es el que resuelve el Minimum Vertex Cover problem, donde el objetivo es encontrar el set mas pequeño de vertices de un grafo tal que cada arista contenga al menos uno de los vertices elegidos (Es como Kruskal o Prim, pero para vertices, no arboles).