BACKTRACKING

- Descomponer un nuemro finito de subtareas parciales que deben ser exploradas exhaustivamente.
- Backtracking construye y atraviesa gradualmente un arbol de subtareas
- La técnica de backtracking no sigue una regla fija de cálculos.
- Los pasos para llegar a la solución final se prueban y se registran, si los pasos no conducen a la solución estos son eliminados del registro
- Cuando la búsqueda en el arbol de soluciones crece rápidamente es necesario utilizar algoritmos aproximados o heurísticos que no garantizan la solución óptima, pero son rapidos.

- **Algoritmos aproximados**: son comúnmente usados para resolver problemas que no tienen una solución polinomial; estos deben correr en tiempo polinomial dentro de límites "probables" de calidad absoluta o asintótica.
- **Heurística**: tienen como objetivo brindar soluciones sin un límite formal de calidad, generalmente evaluados empíricamente en términos de complejidad de las solucione. Este está diseñado para obtener ganancias computacionales o simplicidad conceptual posiblemente a costa de la precisión.
- **Comentarios finales**:
  1. Técnica usada comúnmente cuando no se sabe que camino seguir para encontrar una solución.
  2. No garantiza una solución óptima.
  3. Al analizar un algoritmo que utiliza backtracking, también se debe analizar el crecimiento del espacio de solución.
  4. El backtranking también puede ser visto como una variante de la recursividad.

DIVIDE Y VENCERAS:

- Consiste en dividir problemas en partes más pequeñas, encontrar soluciones para esas partes (supuestamente más fáciles) y combinarlas en una solución global, generalmente conduce a soluciones eficientes y elegante especialmente cuando se usa recursividad.
- Esta técnica consiste en los siguientes pasos:
  1. Dividir el problema original en subproblemas similares al original, pero de menor tamaño.
  2. Resolver cada subproblema sucesiva e independientemente (generalmente recursiva).
  3. Combinar las soluciones individuales en una solución global para todo el problema.
- El algoritmo divide y vencerás es normalmente relacionado a una ecuación de recurrencia que contiene los términos referentes al propio problema:

$$T(n) = aT(n/b) + f(n)$$

  Donde "a" es el numero de subproblemas generados, "b" el tamaño de esos subproblemas y f(n) el costo de hacer la división.

- Este paradigma no solo se aplica a problemas recursivos.
- Hay al menor 3 escenarios donde se aplica divide y venceras:
  1. Procesar independientemente partes de conjunto de datos, ejemplo el mergesort
  2. Eliminar partes del conjunto de datos a examinar, ejemplo la búsqueda binaria

3. Procesar partes del conjunto de datos por separado, pero donde la solución de una parte influya en el resultado de la otra.

PROGRAMACION DINAMICA

- Este paradigma está basado en el método de soluciones basado en tablas.
- Programación dinámica resuelve todos los subproblemas menores, pero solo reutiliza soluciones optimas.
- Divide y vencerás divide el problema en subproblemas mas pequeños.
- Cundo la sumatoria de los tamaños de los subproblemas = O(n) es probable que el algoritmo recursivo tenga complejidad polinomial.
- En ese caso DP puede llevar a una solución más eficiente.
- DP calcula una solución para todos los subproblemas, desde el mas pequeño hasta el mas grande y almacena los resultados en una tabla.
- La ventaja es que una vez que se resuelve un subproblema, la respuesta se almacena en una tabla y nunca se vuelve a calcular.

- **Principio de optimalidad:** en una secuencia optima de elecciones o decisiones, cada subsecuencia también debe ser optima. Cada subsecuencia representa un costo mínimo, por lo tanto, todos los valores de la tabla representan elecciones optimas.
- **Aplicación del principio de optimalidad:**
  1. El principio de optimalidad no puede ser aplicado indistintamente.
  2. Si el principio no se aplica es probable que el probable que el problema no se pueda resolver con éxito usando DP.
  3. Por ejemplo, el problema utiliza recursos limitados y el total de recursos utilizados en las sub-instancias es mayor a los recursos disponibles.
- **¿Cuándo aplicar DP?:**
  1. El problema debe tener una formulación recursiva.
  2. No debe haber ciclos en la formulación (usualmente el problema debe ser reducido a problemas menores).
  3. El número total de instancias del problema debe ser pequeño (n).
  4. El tiempo de ejecución es O(n) × tiempo para resolver la recursividad.
  5. PD tiene una subestructura óptima: La solución óptima al problema contiene soluciones óptimas a los subproblemas.
  6. Superposición de subproblemas: El número total de subproblemas distintos es pequeño en comparación con el tiempo de ejecución recursivo.

  Cuando se empieza del todo hacia la partícula se llama top down y cunado es de la partícula hacia arriba se llama botón up.

NP COMPLETOS

- Los problemas intratables o difíciles son comunes en la naturaleza y en las áreas de conocimiento.
- Problemas “fáciles”: resueltos mediante algoritmos polinómicos.
- Problemas “difíciles”: por el momento sólo conocemos algoritmos exponenciales para resolverlos.
- La complejidad temporal de la mayoría de los problemas es polinomial o exponencial.

- La teoría de la complejidad que se presentará no muestra cómo obtener algoritmos polinómicos para problemas que exigen algoritmos exponenciales, ni afirma que no existen.
- Es posible demostrar que los problemas para los que no existe un algoritmo polinomial conocido están relacionados computacionalmente.
- Forman la clase conocida como N P.
- Propiedad: un problema de clase N P se puede resolver en tiempo polinomial si y sólo si todos los demás problemas en N P también se pueden resolver.
- Este hecho es una fuerte indicación de que casi nadie podrá encontrar un algoritmo eficiente para un problema de clase N P

- **CLASE NP: PROBLMAS SI/NO:**
  1. Para el estudio teórico de la complejidad de los algoritmos se consideran problemas cuyo resultado de cálculo es “sí” o “no”.
  2. Característica de la clase N P: problemas de “sí/no” para los que se puede verificar fácilmente una solución dada.
  3. La solución puede ser muy difícil o imposible de obtener, pero una vez conocida se puede verificar en tiempo polinomial.

- **ALGORITMOS DETERMINISTAS Y NO DETERMINISTAS:**
  1. **Algoritmos deterministas:** el resultado de cada operación se define de forma única.
  2. En un marco teórico, es posible eliminar esta restricción.
  3. A pesar de parecer poco realista, este es un concepto importante y generalmente se usa para definir la clase N P.
  4. En este caso, los algoritmos pueden contener operaciones cuyo resultado no está definido de forma única.
  5. **Algoritmo no determinista:** capaz de elegir una de varias alternativas posibles en cada paso.
  6. Los algoritmos no deterministas contienen operaciones cuyo resultado no está definido de manera única, aunque se limita a un conjunto definido de posibilidades.

- **FUNCION ESCOGER (C):**
  1. Los algoritmos no deterministas utilizan una función de elección (C), que elige arbitrariamente uno de los elementos del conjunto C.
  2. El comando de asignación X ← elegir(1 : n) puede resultar en la asignación a X de cualquiera de los enteros en el rango [1, n].
  3. La complejidad de tiempo para cada elección de llamada de función es O(1).
  4. En este caso, no hay una regla que especifique cómo se hace la elección.
  5. Si un conjunto de posibilidades conduce a una respuesta, siempre se elige este conjunto y el algoritmo terminará con éxito.
  6. Por otro lado, un algoritmo no determinista termina sin éxito si y sólo si no hay un conjunto de opciones que indiquen el éxito.

- **COMANDOS DE ÉXITO Y FRACASO:**
  1. Los algoritmos no deterministas también utilizan dos comandos, a saber: – fallo: indica una terminación fallida. – éxito: indica finalización satisfactoria.
  2. Los comandos de falla y éxito se utilizan para definir la ejecución de un algoritmo.
  3. Estos comandos son equivalentes a un comando de parada de un algoritmo determinista.

4. Los comandos de falla y éxito también tienen una complejidad de tiempo O(1).

- **MAQUINA NO DETERMINISTA:**
  1. Una máquina capaz de realizar la función de selección admite capacidad de cálculo no determinista.
  2. Una máquina no determinista es capaz de producir copias de sí misma frente a dos o más alternativas, y continuar computando independientemente para cada alternativa.
  3. La máquina no determinista que acabamos de definir no existe en la práctica, pero aun así proporciona una fuerte evidencia de que ciertos problemas no pueden resolverse mediante algoritmos deterministas en tiempo polinomial, como se muestra en la definición de la clase N P-completa (más adelante).
  4. Búsqueda no determinista: Determina un índice j tal que A[j] = x para una terminación exitosa o no exitosa cuando x no está presente en A. • El algoritmo tiene una complejidad O(1) no determinista. • Para un algoritmo determinista la complejidad es (n).
  5. Ordenación no determinista: • Se utiliza un vector auxiliar B[1 : n]. Al final, B contiene el conjunto en orden ascendente. • La posición correcta en B de cada entero de A se obtiene de forma no determinista mediante la función de elección. • A continuación, el comando de decisión comprueba si aún no se ha utilizado la posición B[j]. • La complejidad es O(n). (Para un algoritmo determinista la complejidad es Ω(n log n))

- **CARACTERISTICAS DE LAS CLASES P Y NP:**
  1. P: conjunto de todos los problemas que pueden resolverse mediante algoritmos deterministas en tiempo polinomial.
  2. NP: conjunto de todos los problemas que pueden resolverse mediante algoritmos no deterministas en tiempo polinomial.
  3. Para mostrar que un problema dado está en NP, basta con presentar un algoritmo no determinista que se ejecute en tiempo polinomial para resolver el problema.
  4. Otra forma es encontrar un algoritmo determinista polinomial para verificar que una solución dada es válida.

- **¿EXISTE DIFERENCIAS ENTRE P Y NP?:**
  1. P ⊆ NP, ya que los algoritmos deterministas son un caso especial de los no deterministas.
  2. La pregunta es si P = NP o P != NP.
  3. Este es el problema sin resolver más famoso en el campo de la informática.
  4. Si existen algoritmos polinómicos deterministas para todos los problemas en N P, entonces P = N P.
  5. Por otro lado, la demostración de que P != NP parece requerir técnicas que aún se desconocen.
  6. Descripción tentativa del mundo N P, donde la clase P está contenida en la clase NP.
  7. Se cree que NP >> P, porque para muchos problemas en NP, no existen algoritmos polinómicos conocidos, ni un límite inferior no polinomial probado.

- **¿NP ⊃ P o NP = P? CONSECUENCIAS:**
  1. Muchos problemas prácticos en NP pueden o no pertenecer a P (no conocemos ningún algoritmo determinista eficiente para ellos).

2. Si podemos probar que un problema no pertenece a P, entonces no necesitamos buscarle una solución eficiente.
3. Como aún no existe tal prueba, siempre hay esperanza de que alguien descubra un algoritmo eficiente.
4. Casi nadie cree que NP = P.
5. Hay un esfuerzo considerable para demostrar lo contrario, ¡pero la pregunta sigue abierta!

- **PROBLEMAS NP COMPLETO Y NP HARD:**
1. Dos problemas Π1 y Π2 son polinomialmente equivalentes si y sólo si Π1 $\propto$ Π2 y Π2 $\propto$ Π1.
2. Ejemplo: Problema de satisfacibilidad. Si SAT $\propto$ Π1 y Π1 $\propto$ Π2, entonces SAT $\propto$ Π2.
3. Un problema Π es N P-difícil si y sólo si SAT $\propto$ Π (la satisfacibilidad es reducible a Π).
4. Un problema de decisión Π se denomina N P-completo cuando: 1. Π $\in$ NP; 2. Todo problema de decisión Π0 $\in$ N P-completo satisface Π0 $\propto$ Π.
5. Se puede demostrar que un problema de decisión Π que es N P-difícil es NP completo mostrando un algoritmo polinomial no determinista para Π.
6. Sólo los problemas de decisión ("sí/no") pueden ser NP-completos.
7. Los problemas de optimización pueden ser N P-difíciles, pero generalmente, si Π1 es un problema de decisión y Π2 un problema de optimización, es muy posible que Π1 $\propto$ Π2.
8. La dificultad de un problema N P-difícil no es menor que la dificultad de un problema N P-completo.

Entonces el problema que consigue resolver a SAT se le denomina NP HARD

Un problema de decisión Π cuando:
1. Π pertenece a NP Y para ello tiene que encontrar un algoritmo no determinístico que lo resuelva en tiempo polinomial.
2. Todo problema de decisión Π' pertenece a NP completo satisface Π' $\propto$ Π

NPC se reduce a Π' ósea que el problema np completo se reduce a ese problema por ejemplo sudoku.

En el caso del sat, no existe algoritmo polinomial determinístico que lo pueda resolver porque sat pertenece a np completo