Resumen Teoría de Algoritmos

Facultad de Ingeniería de la Universidad de Buenos Aires

Teoría de Algoritmos I

Cátedra Buchwald - Genender

Molina, Taiel Alexis

2023

Indice

Propiedades de grafos

Un grafo es **simple** si a lo más existe una arista uniendo dos vértices cualquiera.

Un grafo **completo** es un grafo simple donde cada par de vértices está conectado por una arista.

En todo grafo no dirigido, la cantidad de vértices con un grado impar, es par.

Un grafo **d-regular** es un grafo en el que todos los nodos tienen grado d. No implica que el grafo sea conexo.

Un grafo es un **árbol** si cumple las siguientes propiedades:

- El grafo es conexo.
- El grafo es acíclico
- El grafo tiene E = V - 1.

Si cumple dos de ellas, necesariamente cumple la tercera.

Camino de Euler: Un camino que pasa por todas las aristas

Ciclo de Euler: Idem anterior, pero que empieza y termina en el mismo vértice.

Un grafo tiene ciclo de Euler sii todos sus nodos tienen grado par. Un grafo tiene camino de Euler (no ciclo) sii tiene exactamente 2 vértices de grado impar.

Camino/Ciclo Hamiltoniano: similar, pero es un camino/ciclo que pasa por todos los vértices una única vez (en caso de ciclo, salvo al final, cerrando)

G y H son dos grafos con matrices de adyacencia A y B, respectivamente. G y H son isomórficos sii existe matriz de Permutación P tal que $B = PAP - 1$

División y Conquista

DyC es una técnica de diseño la cual consiste en dividir un problema en problemas más pequeños y cuando se llega a un tamaño en el cual el problema es resoluble se resuelve, y luego se van uniendo las respuestas volviendo en la recursión.

La fórmula del teorema maestro es

$\mathcal{T}(n) = A \cdot \mathcal{T}(\frac{n}{B}) + \mathcal{O}(n^C)$

siendo

- A: cantidad de llamados recursivos.
- B: proporción en la cual se dividen los problemas.
- $\mathcal{O}(n^C)$: el costo por fuera de los llamados recursivos.

Tenemos tres casos posibles:

- $\log_B(A) < C \rightarrow \mathcal{T}(n) = \mathcal{O}(n^C)$
- $\log_B(A) = C \rightarrow \mathcal{T}(n) = \mathcal{O}(n^C \cdot \log_B(n))$
- $\log_B(A) > C \rightarrow \mathcal{T}(n) = \mathcal{O}(n^{\log_B(A)})$

Problemas vistos en clase

Mulitiplicación de numeros muy grandes

La multiplicación común y corriente tiene un costo de $\mathcal{O}(n^2)$. Utilizando el algoritmo de Karatsuba-Offman podemos reducir dicha complejidad. El algoritmo es el siguiente

```
def multiplicacionBigInt(x, y):
    si largo de x e y son pequenios devolvemos x*y, si no:
    x = x1 2^(n/2) + x0
    y = y1 2^(n/2) + y0
    p = multiplicacionBigInt(x1 + x0, y1 + y0)
    x0y0 = multiplicacionBigInt(x0, y0)
    x1y1 = multiplicacionBigInt(x1, y1)
    return x1y1 + (p - x1y1 -x0y0) 2^(n/2) +x0y0
```

siendo n el largo de los números (por simplicidad decimos que ambos tienen el mismo largo).

La complejidad de este algoritmo se calcula como:

$\mathcal{T}(n) = 3\mathcal{T}(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.6})$

Puntos más cercanos en dos dimensiones

> Dado n puntos en el plano, se desea buscar la pareja que se encuentre más cercana.

Algoritmo sencillo: ver todas las distancias contra todas $\rightarrow \mathcal{O}(n^2)$

Buscando un algoritmo mejor, asumimos que ningún punto tienen la misma coordenada x o y.

```
def closest_pairs(P):
  Construimos px y py (O(n log n))
  p0, p1 = closest_pairs_rec(px, py)
  return p0, p1

def closest_pairs_rec(px, py):
  if len(px) <= 3: return el minimo de comparar cada punto
  Construir Qx, Qy, Rx, Ry (O(n))
  q0, q1 = closest_pairs_rec(Qx, Qy)
    r0, r1 = closest_pairs_rec(Rx, Ry)
    d = min(d(q0, q1), d(r0, r1))
    x* = maxima coordenada x de Qx
    S = puntos de P que estan a distancia <= d de la recta x = x*
    Construir Sy (O(n))
```

```
    por cada punto s de Sy computar distancia contra los siguientes 15 puntos quedarse con s y s' que minimizan esa distancia
  if d(s, s') < d:
        return s, s'
  elif d(q0, q1) < d(r0, r1):
        return q0, q1
  else:
        return r0, r1
```

La complejidad queda $\mathcal{T}(n) = 2 \cdot \mathcal{T}(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n \cdot \log n)$

Multiplicación de matrices

> Dado A y B matrices de nxn. Deseamos calcular la matriz resultante de multiplicar A por B.

Un algoritmo sencillo para multiplicar matrices consume $\mathcal{O}(n^3)$

Algoritmo de Strassen

Podemos dividir a cada matriz de nxn en 4 submatrices de $\frac{n}{2}x\frac{n}{2}$ (por simplicidad asumimos n potencia de 2).

Haciendo un poco de magia en el medio y cosas parecidas al algoritmo de Karatsuba, termina haciendo 7 llamados recursivos en vez de 8, nos queda una complejidad de:

$\mathcal{T}(n) = 7\mathcal{T}(\frac{n}{2}) + \mathcal{O}(1) = \mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.8})$

FFT: Transformación rápida de Fourier

> Tenemos dos vectores A y B, y queremos obtener la convolución entre ambos.

Un algoritmo fácil resuelve el problema en $\mathcal{T}(n) = \mathcal{O}(n^2)$. Con DC podemos obtener el resultado en $\mathcal{O}(n \cdot \log n)$.

Vamos a ver a los vectores como polinomios del tipo:

$a(x) = a_0 + a_1 x + ... + a_{n-1} x^{n-1}$

obtenemos $C(x) = A(x) \cdot C(x)$ donde $c = (c_0, c_1, ..., c_{2n-1})$ son los coeficiente de $C(x)$.

Obtener extremo de un polígono

> Dado P un polígono convexo de n vértices y un vector dirección u. Deseamos encontrar el punto extremo de P según la dirección u.

Un polígono es convexo cuando sus ángulos interiores miden a lo sumo 180 grados. Adicionalmente se puede ver que si es convexo también es monótono respecto a cualquier recta u. Es decir que cualquier línea ortogonal a u corta al polígono a lo sumo en dos puntos.

```
    BuscarMaximo(P,a,b):
        Si b-a <= 3
            Retornar maximo entre los puntos en P entre a y b
    proyectados a u

  Sea c el valor intermedio entre inicio y fin

  Si p[a] es positivo proyectado en u
    Si p[c] es negativo proyectado en
            u retornar BuscarMaximo(P,a,c)
    Si P[c] ubicado sobre P[a] en relacion a proyeccion u
      retornar BuscarMaximo(P,c,b)
    retornar BuscarMaximo(P,a,c)

  Si p[c] es positivo proyectado en u
    retornar BuscarMaximo(P,c,b)
  Si P[c] ubicado debajo de P[a] en relacion a proyeccion u
    retornar BuscarMaximo(P,c,b)
  retornar BuscarMaximo(P,a,c)

BuscarMaximo(P,0,n-1)
```

Complejidad: $\mathcal{T}(n) = \mathcal{O}(\log n)$

Conteo de inversiones

> Tengo un conjunto de n elementos + 2 arreglos/listas ordenados por diferentes criterios (A y B).

Dar una medida de semejanza entre dichas listas.

Nombramos los elementos en A como 1, 2, ..., n y B como correspondiente a eso.

¿Qué tan diferente está el orden B (de forma ascendente) respecto al A?

Pregunta: ¿Qué pasa si $b_i < b_{i+1}$ para todo i?

Dos elementos están invertidos si $b_i > b_j$ (con $i < j$)

Usamos mergesort y en el momento de hacer merge vamos contando las inversiones que realiza el algoritmo.

La complejidad es $\mathcal{T}(n) = \mathcal{O}(n \cdot \log n)$.

Envolvente convexa

> Dado un conjunto P de n puntos en el plano. Deseamos obtener el menor polígono convexo que contenga a todo el conjunto.

```
    Sea P un conjunto de n puntos en el plano ordenados por
    coordenadas x

CalcularCH(P):
```

```
  Si |P| <= 5
    Retornar CH construida por fuerza bruta

  Sean P1 los primeros |P|/2 puntos
  Sean P2 los ultimos|P|/2 puntos

  ch1 = CalcularCH(P1)
  ch2 = CalcularCH(P2)

  Retornar Unir(ch1,ch2)

Unir(ch1, ch2)
  Sea a punto mas a la derecha de ch1
  Sea b punto mas a la izquierda de ch2
  buscar tangente superior entre ch1 y ch2 partiendo de a y b
  buscar tangente inferior entre ch1 y ch2 partiendo de a y b

  Quitar de ch1 los puntos entre la tangente superior e inferior
  Quitar de ch2  los puntos entre la tangente superior e inferior
  Crear ch con puntos de ch1 y ch2 unidos por tangentes.

  Retornar ch
```

Greedy

Un algoritmo greedy es aquel que busca los óptimos locales para mi estado actual y aplicando iterativamente esa regla llegaremos a un óptimo global.

Ventajas y desventajas

- No siempre dan el resultado óptimo.
- Demostrar que dan el resultado óptimo es díficil.
- Son intuitivos de pensar. Son fáciles de entender.
- Suelen funcionar rápido.
- Para problemas complejos pueden ser buenas aproximaciones.

Problemas vistos en clase

Algoritmo Dikstra

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista.

```
def dijkstra(grafo, origen)
    dist, padre = {}, {}
    for v in grafo:
        dist[v] = float("inf")
```

```
    dist[origen] = 0
    padre[origen] = None
    q = []
    heapq.heappush(q, (0, origen))
    while len(q) > 0:
        _, v = heapq.heappop(q)
        for w in grafo.adyacentes(v):
            distancia_nuevo_camino = dist[v] + grafo.peso(v, w)
            if distancia_nuevo_camino < dist[w]:
                dist[w] = dist[v] + grafo.peso(v, w)
                padre[w] = v
                heapq.heappush(q, (dist[w], w))

    return padre, dist
```

Complejidad temporal: $\mathcal{O}(E \log(V))$

Algoritmo Prim

El algoritmo de Prim es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

```
def arbol_tendido_minimo_prim(grafo):
    cantidad_de_aristas = 0
    v = grafo.obtener_vertice_al_azar()
    visitados = set()
    visitados.add(v)
    q = []
    for w in grafo.adyacentes(v):
            heapq.heappush(q, (grafo.peso(v,w), (v, w)))
    arbol = Grafo(es_pesado = True)
    for v in grafo:
        arbol.agregar_vertice(v)

    while len(q) > 0:
        peso, (v, w) = heapq.heappop(q)

        if w in visitados:
            continue

        arbol.agregar_arista(v, w, peso)
        cantidad_de_aristas += 1
        visitados.add(w)

        for u in grafo.adyacentes(w):
            if u not in visitados:
                heapq.heappush(q, (grafo.peso(w,u), (w, u)))

    return arbol, cantidad_de_aristas
```

Complejidad temporal: $\mathcal{O}(E \log(V))$

Algoritmo Kruskal

El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado.

```
def kruskal(grafo):
    conjuntos = UnionFind(grafo.obtener_vertices())
    aristas = sort(obtener_aristas(grafo))
    arbol = grafo_crear(grafo.obtener_vertices())
    for a in aristas:
        v, w, peso = a
        if conjuntos.find(v) == conjuntos.find(w): continue
        arbol.agregar_arista(v, w, peso)
        conjuntos.union(v, w)
    return arbol
```

Complejidad temporal: $\mathcal{T}(n) = \mathcal{O}(n \cdot \log n)$

Problema de scheduling 1

> Tengo un aula donde quiero dar charlas, que cada una tiene un horario de inicio y un horario de fin. Quiero dar la mayor cantidad de charlas.

Solución: ordenar la charla por horario de fin (de menor a mayor) e ir agregando las charlas mientras no se superpongan con otras charlas ya agregadas a la solución.

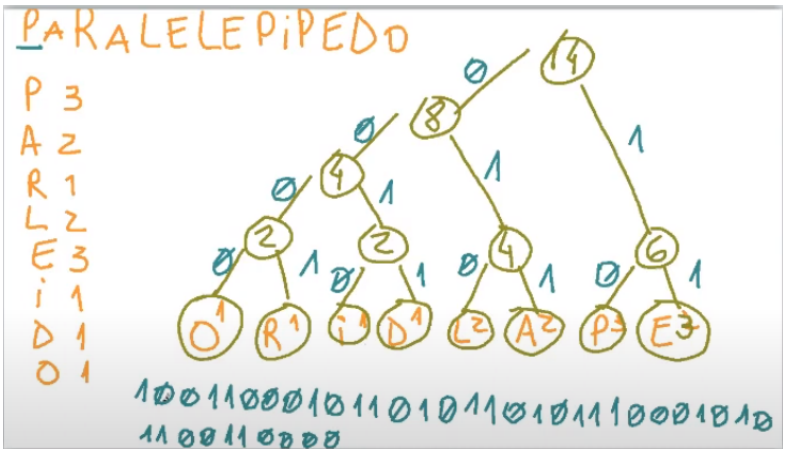
```
    def scheduling(horarios):
        horarios_ordenados = ordenar_por_horario_fin(horarios)
        charlas = []
        for horario in horarios_ordenados:
            if len(charlas) == 0 or not hay_interseccion(charlas
    [-1], horario):
            charlas.append(horario)
        return charlas
```

Árboles de Huffman

Es un algoritmo de compresión de datos. Consta de encolar en un heap de mínimos las letras con su respectiva frecuencia (la cantidad de veces que aparece la letra en la palabra). Voy desencolando del heap y voy haciendo hojas en un arbol binario. Luego voy uniendo las hojas desde las de menor frecuencia hacia la de mayor.

Para construir el binario se inicia desde la primera letra, y desde la raiz se anotan los números que se encuentran hasta llegar al nodo de la letra en particular, y luego se sigue con las siguientes letras con la misma lógica.

```
def huffman(texto):
    frecuencias = calcular_frecuencias(texto)
    q = heap_crear()
    for caracter in frecuencia:
        q.encolar(Hoja(caracter, frecuencia))
    while q.cantidad() > 1:
        t1 = q.desencolar()
        t2 = q.desencolar()
        q.encolar(Arbol(t1, t2, t1.frecuencia + t2.frecuencia))
    return codificar(q.desencolar())
```



Problema del Cambio

> Se tiene un sistema de monetario. Se quiere dar cambio de una determinada cantidad de plata. Implementar un algoritmo que devuelva el cambio pedido, usando la mínima cantidad de monedas/billetes.

Si el sistema monetario es *como la gente* lo fácil es ir devolviendo el vuelto con los billetes de mayor valor posible. Si fuera un sistema monetario malo, este algoritmo no va dar la respuesta óptima.

Problemas de compra con inflación

> Tenemos n productos en un arreglo R, donde R[i] representa el precio del producto i. Cada día debemos comprar un solo próducto, y cada dia aumentan los precios. El precio del producto i el dia j es $R[i]^{j+1}$ (con j comenzando en 0).
> Implementar un algoritmo que nos indique el precio mínimo al que podemos comprar todos los productos.

Lo que conviene es comprar los productos de mayor precio primero.

Problema de la carga de combustible

> Un camión debe viajar desde una ciudad a otra deteniéndose a cargar combustible cuando sea necesario. El tanque de combustible le permite viajar hasta K kilómetros. Las estaciones se encuentran distribuidas a lo largo de la ruta siendo d_i la distancia desde la estación i-1 a la estación i.
> Implementar un algoritmo que decida en qué estaciones debe cargar combustible de manera que se detenga la menor cantidad de veces posible.

Lo que se puede hacer es ir hasta la estación más lejana que dé la nafta.

Problema de la mochila

> Tenemos una mochila con una capacidad W (peso, volumen). Hay elementos a guardar. Cada elemento tiene un peso y un valor. Queremos maximizar el valor de lo que nos llevamos sin pasarnos de la capacidad.

Calcular ordenando por $\frac{v}{w}$ y mayor valor y quedarse con la mejor.

V2: Suponiendo que se pueden agarrar proporciones de los elementos, utilizar la relación $\frac{v}{w}$ da la solución óptima.

Problema de scheduling 2

> Ahora tenemos tareas con una duración y un deadline, pero pueden hacerse en cualquier momento, siempre que se hagan antes del deadline. Para este problema, buscamos minimizar la latencia en el que las tareas se ejecuten. Es decir, si definimos que una tarea i empieza en si, entonces termina en $f_i = s_i + t_i$, y su latencia es $l_i = f_i - d_i$ (si $f_i > d_i$, sino 0).

Hacemos que los trabajos que se necesiten terminar primero, se hagan primero. Esta solución también es óptima.

Problema de scheduling 3

> Contamos con un conjunto de n actividades. Cada actividad x tiene una fecha de inicio i_x y una fecha de finalización f_x. Además contamos con un número r de recursos donde se pueden llevar a cabo estas actividades. Determinar si es posible cumplir con todas las actividades utilizando la menor cantidad de recursos posibles. Además, dar una programación posible para llevarlas a cabo.

Este problema se puede ver como tener n charlas y r aulas, y se quiere saber si alcanzan o no las r aulas para dar todas las charlas, y en el caso de alcanzar, cuál es la mínima cantidad de aulas que se pueden utilizar.

Solución:

```
Interval coloring problem solution:
    Ordenar las actividades por fecha de inicio
    Sea H cola de prioridad maxima
    Sea k = 0
    Por cada actividad act
      obtener recurso rec de H con maxima clave
      Si act.fechaInicio >  rec.fechaLiberacion
        asignar act a recurso rec
        agregar (rec, act.fechaFinalizacion) a H
      Sino
        k++
        Si k<=r
          obtener un nuevo recurso rnew
          asignar act a recurso rnew
          agregar (rnew, act.fechaFinalizacion) a H
        Sino
          Retornar no hay suficientes recursos
    Retornar las asignaciones.
```

La solución es óptima, su complejidad temporal es de $\mathcal{O}(n \log(nr))$ y su complejidad espacial es de $\mathcal{O}(n + r)$

Problema de Caché

Podemos tener hasta k elementos de memoria bien a mano, el resto tendríamos que ir a RAM. Hay que decidir qué guardamos en la memoria caché.
Tenemos un conjunto de datos U en memoria general (n en total), una memoria caché de $k < n$ elementos. Tenemos una secuencia de pedidos de datos d_i. Si d_i está en la caché, accedemos muy rápido. Si no está $\rightarrow$ cache miss + ahora la tenemos que traer a la caché (y si la caché está llena, tenemos que evictear un dato previo). Queremos minimizar la cantidad de caché misses.

Suponiendo que sabemos en qué orden se van a pedir las cosas, sacamos de la caché aquel elemento que se van a pedir último entre los que estan en la caché.

Sin saber en qué orden se van a pedir las cosas, sacamos de la caché aquel elemento que se haya usado hace más tiempo.

Generación de un grafo

Dado una lista de n números naturales, implementar un algoritmo en tiempo polinomial que cree un grafo de n nodos cuyos grados sean los indicados por esa lista. G debe ser simple y sin bucles.

Ordenamos la lista de los n números y agarramos el último número (digamos de valor k) y le restamos 1 a los k primeros números, y asi sucesivamente.

Programación dinámica

Bottom-up: ir de los casos más fáciles a los más díficiles.

Top-down: ir de los casos más díficiles a los más fáciles.

Memoization es la técnica de guardarse los resultados calculados previamente con el fin de llegar al resultado final.

Cómo saber que podemos utilizar PD:

1. Hay un número polinomial de subproblemas
2. La solución al problema original puede ser construido a partir de soluciones a subproblemas
3. Hay un orden natural de los subproblemas de menor a mayor. Los subproblemas mayores son resueltos mediante la composición de problemas menores. La versión recursiva resuelve en formato Top-Down. La versión iterativa es de un formato Bottom-Up, es la que preferimos a partir de ahora al resolver ejercicios

Problemas vistos en clase

Fibonacci

Solución $\mathcal{O}(n)$ en complejidad espacial y temporal.

```
def fib_iterativo(n):
    M_FIB = [None] * (n+1)
    M_FIB[0] = 0
    M_FIB[1] = 1
    i = 2
    while i <= n:
        M_FIB[i] = M_FIB[i-1] + M_FIB[i-2]
        i+=1
    return M_FIB[n]
```

Solución $\mathcal{O}(n)$ en complejidad temporal y $\mathcal{O}(1)$ en complejidad espacial.

```
def fib_dinamico(n):
    if n == 0: return 0
    if n == 1: return 1
    anterior = 0
    actual = 1
    i = 1
    while i < n:
        nuevo = actual + anterior
        anterior = actual
        actual = nuevo
        i+=1
    return actual
```

Problemas de scheduling con peso

> Tengo un aula/sala donde quiero dar charlas. Las charlas tienen horario de inicio y fin, y un peso asociado al valor de cada charla. Quiero utilizar el aula para maximizar la sumatoria de pesos de las charlas dadas.

La idea será ordenar las charlas por horario de fin, y luego ir viendo en el cado de cada charla que nos conviene, si no dar la charla y tener el resto de charlas a disposición, o darla y tener a disposición solo las charlas que no se superpongan con la charla que decidí dar.

La ecuación de recurrencia se reduce a:

$$\rightarrow OPT(j) = max \begin{cases} \nearrow \text{dar la charla: } V_j + OPT(p(j)) \\ \searrow \text{no dar la charla: } OPT(j-1) \end{cases}$$

```
    def che_dinamico(n):
    if n == 0:
        return 0
    M_CHE = [None] * n
    M_CHE[0] = 0
    for j in range (1, n+1):
        M_CHE[j] = max(valor[j]+M_CHE[p[j]], M_CHE[j-1])
    return M_CHE[n]
```

Para reconstruir la solución hacemos lo siguiente:

```
def che_solucion(M_CHE, valor, p, j, solucion):
    if j == 0:
        return solucion
    if valor[j]+M_CHE[p[j]] >= M_CHE[j-1]:
        solucion.append(j)
        return che_solucion(M_CHE, valor, p, p[j], solucion)
    return che_solucion(M_CHE, valor, p, j-1, solucion)
```

Juan el vago

> Juan es ambicioso pero también algo vago. Dispone de varias ofertas de trabajo diarias, pero no quiere trabajar dos días seguidos. Dado un arreglo con el monto esperado a ganar cada día, determinar por programación dinámica el máximo monto a ganar, sabiendo que no aceptará trabajar dos días seguidos.

La idea para afrontar esta problemática, es analizar lo que se puede hacer con un día en particular: trabajarlo y no poder trabajar el día anterior, o no trabajarlo y sí poder utilizar el día anterior.

La ecuación de recurrencia se reduce a:

$$\rightarrow OPT(j) = max \begin{cases} \nearrow \text{trabajar un día: } V_j + OPT(j-2) \\ \searrow \text{no trabajar el día: } OPT(j-1) \end{cases}$$

```
def juan_el_vago_con_memoria(M, dias):
    G = [0] * dias
    G[0] = M[0]
    G[1] = max(M[0], M[1])
    for d in range(2, dias):
        G[d] = max(M[d] + G[d-2], G[d-1])
    return G
```

Para reconstruir la solución se puede ejecutar el siguiente algoritmo:

```
def construir_elecciones(G,M):
    elecciones = []
    d = len(G)-1
    while d > 1:
        if M[d] + G[d-2] >= G[d-1]:
            elecciones.insert(0, d)
            d -= 2
        else:
            d -= 1
    if G[1] == M[1]:
        elecciones.insert(0, 1)
    else:
        elecciones.insert(0, 0)}

    return elecciones
```

Caminos posibles en un laberinto

> Dado un laberinto representado por una grilla, queremos calcular la cantidad de caminos posibles que existen para llegar desde la posición (0,0) hasta la posición NxM Los movimientos permitidos son, desde la esquina superior izquierda (el 0,0), nos podemos mover hacia abajo o hacia la derecha

Las formas de llegar a una posición es, o bien llegar desde la posición de la izquierda, o bien la posición de arriba de la celda actual.

La ecuación de recurrencia se reduce a $Posibles[i][j] = Posibles[i-1][j] + Posibles[i][j-1]$.

Si existieran obstáculos en el laberinto, la forma de llegar a dicha celda con obstáculo es 0.

Si pasar por las posiciones diera una ganancia, no hay que hacer la sumatoria de los pasos de [i-1][j] y [i][j-1], si no calcular el máximo entre ambos, eligiendola como opción para llegar a la posición i, j.

Caminos a través del teclado del telefono

> Dado el teclado numérico de un celular, y un número inicial k, encontrar la cantidad de posibles números de longitud N empezando por cierto botón. Restricción: solamente se puede presionar un botón si está arriba, abajo, a izquierda, o derecha del botón actual.

Utilizamos un grafo para saber los adyacentes de un número.

La ecuación de recurrencia del problema es: $C[pasos = i][desde = v] = C[pasos = i - 1][desde = v] + \sum C[pasos = i - 1][desde = vecino]$

El algoritmo es

```
def cant_combinaciones(grafo, pasos, tecla_inicial):
    cant = [][]
    for tecla in range(teclas del 0 al 9):
        cant[0][tecla] = 0
        cant[1][tecla] = 1
    for i in range(2, pasos+1):
        for tecla in range(teclas del 0 al 9):
            contador = cant[i-1][tecla]
            for vecino in grafo.adyacentes(tecla):
                contador += cant[i-1][vecino]
            cant[i][tecla] = contador
    return cant[pasos][tecla_inicial]
```

Problema de la mochila

> Tenemos una mochila con una capacidad W (peso, volumen). Hay elementos a guardar. Cada elemento tiene un peso y un valor. Queremos maximizar el valor de lo que nos llevamos sin pasarnos de la capacidad.

Lo que podemos hacer frente a cada nuevo elemento que entre en la mochila es: o bien utilizar el objeto y hacer que $W = W - P_i$ o no utilizarlo.

La ecuación de recurrencia (siempre y cuando $P_i \leq W$) se escribe entonces como:

$$\rightarrow OPT(n, W) = max \begin{cases} \nearrow \text{utilizar un elemento: } OPT(n-1, W - P_i) + V_i \\ \searrow \text{no utilizarlo: } OPT(n-1, W) \end{cases}$$

Un algoritmo se ejecuta en tiempo Pseudo-Polinomial si su tiempo de ejecución es proporcional a un polinomio en el valor numérico de la entrada (el valor entero más grande presente en la entrada), pero no necesariamente en la longitud de la entrada (la cantidad de bits necesarios para representarlo), que es el caso de los algoritmos de tiempo polinomial.

El problema de la mochila es pseudopolinomial.

Problema del cambio

> Se tiene un sistema monetario (ejemplo, el nuestro). Se quiere dar cambio de una determinada cantidad de plata. Implementar un algoritmo que devuelva el cambio pedido, usando la mínima cantidad de monedas/-billetes.

```
def cant_monedas(sist_monetario, dinero):
    cant[0] = 0
    for i in range(1, dinero+1):
        minimo = i
        for moneda in sist_monetario:
            if moneda > i:  continue
            cantidad = 1 + cant[i - moneda]
            if cantidad < minimo:   minimo = cantidad
        cant[i] = minimo
    return cant[dinero]
```

Algoritmo de Floyd-Warshall

Sirve para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución.

```
def floyd_warshall(G):
    distance = list(map(lambda i: list(map(lambda j: j, i)), G))

    # Adding vertices individually
    for k in range(G.vertexes()):
        for i in range(G.vertexes):
            for j in range(G.vertexes):
                distance[i][j] = min(distance[i][j], distance[i][k]
     + distance[k][j])
```

Complejidad temporal: $\mathcal{O}(V^3)$
Es mejor que hacer V veces el algoritmo de Bellman-Ford.

Subset Sum

> Tenemos un conjunto de números v1, v2, ... Vn, y queremos obtener un subconjunto de todos esos números cuya suma sea igual a un valor V, o bien aproximar lo mayor posible a ese valor V.

La ecuación de recurrencia es:

$$\rightarrow OPT(n, V) = max \begin{cases} \nearrow \text{utilizar un elemento: } OPT(n-1, V-V_i) + V_i \\ \searrow \text{no utilizarlo: } OPT(n-1, V) \end{cases}$$

Tú a Londres y yo a California

Manejamos un negocio que atiende clientes en Londres y en California. Nos interesa cada mes decidir si operar en una u otra ciudad. Los costos de operación para cada mes pueden variar y son dados: Li y Ci para todo n.
Naturalmente, si en un mes operamos en una ciudad, y al siguiente en una distinta, habrá un costo fijo M por la mudanza.
Dados los costos de operación en Londres (L) y California (C), indicar la secuencia de las n localizaciones en las que operar durante n meses, sabiendo que queremos minimizar los costos de operación. Se puede empezar en cualquier ciudad

Para un mes n, un plan que termina en Londres puede tener los siguientes costos:

- Ln + los costos operativos que hayamos tenido en meses anteriores terminando en Londres
- Ln + los costos operativos que hayamos tenido en meses anteriores terminando en California + M

Para un mes n, un plan que termina en California puede tener los siguientes costos:

- Cn + los costos operativos que hayamos tenido en meses anteriores terminando en California
- Cn + los costos operativos que hayamos tenido en meses anteriores terminando en Londres + M

Las ecuaciones de recurrencia son:
OPT londres [n] = L[n] + min(OPT londres [n-1], M + OPT california [n-1])
OPT california [n] = C[n] + min(OPT california [n-1], M + OPT londres [n-1])

Fraccionamiento del aceite de oliva

Contamos con un depósito de L litros lleno de aceite de oliva. Queremos fraccionarlo y venderlo en el mercado. De acuerdo a las especificaciones vigentes solo se puede fraccionar en valores enteros de litros. Una tabla regulatoria determina el valor de venta para cada i litros como V_i. Podemos fraccionar como queramos, pero deseamos maximizar la ganancia.

Muy similar al problema de la mochila.

```
Sea L la cantidad de litros disponibles
Sea v[i] la ganancia por vender i litros de aceite
Sea M[i] la maxima ganancia posible por vender i litros

M[0]=0
Desde j=1 a L
  M[j]=0
  Desde i=1 a j
    gan = v[i] + M[j-i]
    Si gan > M[j]
      M[j] = gan

Retornar M[L]
```

Máxima suma de segmento

> Dado una lista L de n elementos. Cada elemento tiene asociado un valor numérico (positivo o negativo). Queremos encontrar el subconjunto contiguo de elementos que sume el mayor valor posible.

Solución por DyC:

```
Sea L[] el vector de n elementos
MaximoSubArreglo(L,inicio,final):
  Si final<inicio
    Retornar inicio
  Si final=inicio
    Retornar L[inicio]

  Sea mitad el valor intermedio entre inicio y fin

  maxM1 = MaximoSubArreglo(L,inicio,mitad-1)
  maxM2 = MaximoSubArreglo(L,mitad,final)
  maxPasando = MaximiIntermedio(L,inicio,mitad,final)

  Retornar maximo entre maxM1, maxM2 y maxPasando

MaximiIntermedio(L,inicio,mitad,final):
  MaxInicio=0
  AcumInicio=0
  Desde j=mitad-1 hasta inicio
    AcumInicio += L[j]
    Si AcumInicio>MaxInicio
      MaxInicio=AcumInicio

  MaxFinal=0
  AcumFinal=0
  Desde j=mitad hasta final
    AcumFinal += L[j]
    Si AcumFinal>MaxFinal
      MaxFinal=AcumInicio

  Retornar MaxInicio+MaxFinal

MaximoSubArreglo(L,1,n):
```

que utilizando el Teorema Maestro, llegamos a:

$\mathcal{T}(n) = 2\mathcal{T}(\frac{n}{2}) + \mathcal{O}(n) \rightarrow \mathcal{T}(n) = \mathcal{O}(n \cdot \log n)$

Usando PD, la idea es arrancando desde la posición 0, analizar el máximo subconjunto tomando como fin esa posición.

La decisión estará entre: quedarse con el máximo hasta la posición $i-1$ y sumarle $L[i]$ o quedarse con $L[i]$.

```
MaximoGlobal = L[0]
MaximoLocal = L[0]
IdxFinMaximo = 0
Desde i=1 a n // elementos
  MaximoLocal = max(MaximoLocal,0) + L[i]
  if MaximoLocal > MaximoGlobal
    MaximoGlobal = MaximoLocal
    IdxFinMaximo = i

Retornar MaximoGlobal
```

Esta solucion tiene complejidad temporal $\mathcal{O}(n)$ y complejidad espacial $\mathcal{O}(1)$

Mínimos cuadrados segmentados

> Sea un conjunto P de n puntos en el plano. Tal que para cada par de puntos p_i=(x_i,y_i) y p_j=(x_j,y_j) se cumple que $x_i > x_j$ si $i > j$. Queremos aproximar mediante segmentos los puntos de P minimizando el error cometido con la menor cantidad posibles de segmentos.

```
OPT[0] = 0

Para todo par i,j con i<=j
  Calcular e[i][j]

Desde j=1 a n
  OPTIMO[j] = +infinito

  Desde i=1 a n
    segmento = e[i][j] + C + OPT[i-1]

    si OPTIMO[j] > segmento
      OPTIMO[j] = segmento

Retornar OPT[n]
```

Fuerza Bruta y Backtracking

Fuerza Bruta es frente a un problema combinatorio, probar todas las combinaciones posibles.

Backtracking es una estrategia que frente a un algoritmo que utiliza Fuerza Bruta, establecer ciertas condiciones en las cuales no seguimos analizando una

posible solución del problema porque ya sabemos que no va ser la óptima. Se le llama *poda* porque literalmente corta el árbol de posibilidades.

Receta Backtracking:

1. Si ya encontre una solucion, la devuelvo y termino.
2. Avanzo si puedo
3. Pruebo si la solución parcial es válida
   (a) Si no lo es, retrocedo y vuelvo a 2.
   (b) Si lo es, llamo recursivamente y vuelvo a 1.
4. Si llegue hasta aca, ya probe con todo y no encontre una solucion (no válido para todos los casos, pero el esquema suele ser similar)

Problemas vistos en clase

N Reinas

> Dado un tablero de ajedrez NxN, ubicar (si es posible) a N reinas de tal manera que ninguna pueda comerse con ninguna.

```python
def es_compatible(grafo, puestos):
    for v in puestos:
        for w in puestos:
            if v == w: continue
            if grafo.hay_arista(v, w):
                return False
    return True

def _ubicacion_BT(grafo, vertices, v_actual, puestos, n):
    if v_actual == len(grafo):
        return False
    if len(puestos) == n:
        return es_compatible(grafo, puestos)

    if not es_compatible(grafo, puestos):
        return False

    # Mis opciones son poner aca, o no
    puestos.add(vertices[v_actual])
    if _ubicacion_BT(grafo, vertices, v_actual + 1, puestos, n):
        return True
    puestos.remove(vertices[v_actual])
    return _ubicacion_BT(grafo, vertices, v_actual + 1, puestos, n)
```

Independent Set

> Quiero guardar en un grafo N elementos. Debo elegir N vértices en los cuales guardar cada uno. Restricción! Queremos ver de ubicar N elementos sin que hayan dos adyacentes con elementos.

Camino Hamiltoniano

> Un camino hamiltoniano es un camino de un grafo, que visita todos los vértices del grafo una sola vez. Si además el último vértice visitado es adyacente al primero, el camino es un ciclo hamiltoniano.

```
def camino_hamiltoniano_dfs(grafo, v, visitados, camino):
    vistitados.add(v)
    camino.append(v)
    if len(visitados) == len(grafo): return True
    for w in grafo.adyacentes(v):
        if w not in visitados:
            if camino_hamiltoniano_dfs(grafo, w, visitados, camino)
    : return True
    visitados.remove(v)
    camino.pop()
    return False

def camino_hamiltoniano(grafo):
    camino = []
    visitados = set()
    for v in grafo:
        if camino_hamiltoniano(grafo, v, visitados, camino): return
     camino
    return None
```

K coloreo

> Dado un grafo y K colores diferentes, ¿es posible pintar los vértices de tal forma que ningún par de vértices adyacentes tengan el mismo color?

Si k=2 es el problema de si el grafo es bipartito.

```
1) Si todos los paises estan coloreados, devuelvo True

2) Pruebo colorear con un color el siguiente pais:
3) Verifico si la solucion parcial es valida
    a. Si no lo es, retrocedo y vuelvo a 2) a probar con otro color
    b. Si lo es, llamo recursivamente y vuelvo a 1)

4) Si llego hasta aca, ya probe con todo y no encontre una solucion
```

```
def es_compatible(grafo, colores, v):
    for w in grafo.adyacentes(v):
        if w in colores and colores[w] == colores[v]:
            return False
    return True

def _coloreo_rec(grafo, k, colores, v):
    for color in range(k):
        colores[v] = color
        if not es_compatible(grafo, colores, v):
            continue
        correcto = True

        for w in grafo.adyacentes(v):
            if w in colores: continue
            if not _coloreo_rec(grafo, k, colores, w):
                correcto = False
                break
        if correcto:
            return True

    del colores[v]
    return False

def coloreo(grafo, k):
    colores = {}
    return _coloreo_rec(grafo, k, colores, grafo.random_vertex())
```

Sudoku

Se desea encontrar si un tablero de sudoku tiene solución o no.

```
def sudoku(cant_elem, M):
    cant_elem = sig_pos_a_usar(cant_elem, M)
    if cant_elem >= 9*9: return True
    for num in range(1, 10):
        if puedo_poner(num, cant_elem, M):
            M[fila(cant_elem)][columna(cant_elem)] = num
            if sudoku(cant_elem, M): return True
    M[fila(cant_elem)][columna(cant_elem)] = 0
    return False
```

Podríamos también utilizar un grafo para modelar este problema: los vértices serían las celdas del sudoku, y las aristas unirán a aquellos que compartan subgrupo y/o columna y/o fila. Una vez creado el grafo, hacemos un 9-Coloreo.

Knight-Tour

Dado una pieza de caballo de ajedrez (knight o caballero) dentro de un tablero, determinar los movimientos a hacer para que el caballo logre pasar por todos los casilleros una vez.

```
def caballo(paso = 0):
    if completo():  return True
    x, y = obtener_posicion_actual_caballo()
    for fila, col in movimientos_caballo(x,y):
        if not dentro_de_tablero(fila, col): continue
        if casillero_ya_marcado(fila, col): continue
        mover_a_posicion(fila, col, paso)
        if (caballo(paso + 1)):
            return True
        volver_a_posicion(x,y)
    return False
```

Si ya tuvieramos un grafo con los vértices y aristas creados, siendo los vértices las celdas del casillero y creando aristas por cada movimiento de una celda a otra que realiza el caballero, la forma de encontrar la solución es buscar si existe un camino hamiltoniano.

Materias Compatibles

> Se tiene una lista de materias que deben ser cursadas en el mismo cuatrimestre, cada materia está representada con una lista de cursos/horarios posibles a cursar (solo debe elegirse un horario por cada curso). Cada materia puede tener varios cursos.
> Implementar un algoritmo de backtracking que devuelva un listado con todas las combinaciones posibles que permitan asistir a un curso de cada materia sin que se solapen los horarios. Considerar que existe una función son_compatibles($curso_1$, $curso_2$) que dados dos cursos devuelve un valor booleano que indica si se pueden cursar al mismo tiempo.

```
def solucion_posible(horarios):
    ultimo = horarios.ver_ultimo()
    for curso in horarios:
        if curso == ultimo: continue
        if not son_compatibles(curso, ultimo):
            return False
    return True

def horarios_posibles(materias, solucion_parcial):
    if len(materias) == 0:
        if solucion_posible(solucion_parcial):
            return [solucion_parcial]
        else:
            return []
    if not solucion_posible(solucion_parcial):
        return []
    materia_actual = materias.ver_primero()
    materias.borrar_primero()
    soluciones = []
    for curso in materia_actual:
        soluciones.extend(horarios_posibles(materias_restantes,
    solucion_parcial + [curso]))
    materias.guardar_primero(materia_actual)
```

```
23
24      return soluciones
```

Sumatoria de dados

> Escribir un algoritmo de tipo Backtracking que reciba una cantidad de dados n y una suma s. La función debe devolver todas las tiradas posibles de n dados cuya suma es s. Por ejemplo, con n = 2 y s = 7, debe imprimir [1, 6] [2, 5] [3, 4] [4, 3] [5, 2] [6, 1].

```
1  def suma_dados(cant, n, cant_faltan, soluciones):
2      if sum(solucion_parcial) == n and cant_faltan == 0:
3      soluciones.append(solucion_parcial[:])
4      return
5
6      if sum(solucion_parcial) + cant_faltan*MIN_DADO > n: return
7      if sum(solucion_parcial) + cant_faltan*MAX_DADO < n: return
8
9      for valor in range(MIN_DADO, MAX_DADO):
10         solucion_parcial.append(valor)
11         suma_dados(cant, n, cant_faltan-1, soluciones)
12         solucion_parcial.pop()
13     return
```

Subset Sum

> Escribir una función que, utilizando backtracking, dada una lista de enteros positivos L y un entero n devuelva todos los subconjuntos de L que suman exactamente n.

```
1  def subset_sum(L, index, n, solucion_parcial, soluciones):
2      if sum(solucion_parcial) == n:
3    soluciones.append(solucion_parcial[:])
4    return
5
6      if sum(solucion_parcial) > n:
7      return
8
9      for i in range(index, len(L)):
10   solucion_parcial.append(L[i])
11   subset_sum(L, index+1, n, solucion_parcial, soluciones)
12   solucion_parcial.pop()
13     return
```

Redes de flujo

Una red de flujo es un modelo que se realiza con un grafo que representa el flujo de materiales (cloacas, información en redes, etc).

El grafo es dirigido y tenemos una única fuente y un único sumidero, por lo que tenemos una única componente débilmente conexo.

- Fuente: vértice con grado de entrada 0.
- Sumidero: vértice con grado de salida 0.

Cada vértice intermedio lo único que hace es trasladar lo que pasan. No produce ni consume (conservación del material).

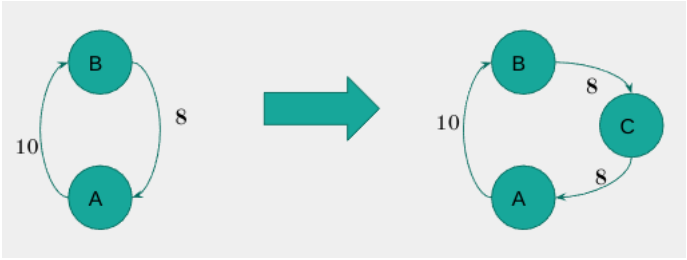
Cada arista tiene un peso que representa la capacidad de transporte de esa vía.

Restricciones:

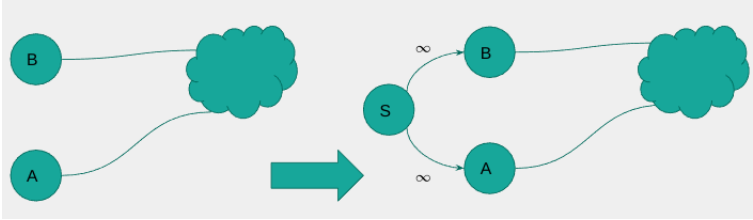
- No se aceptan bucles.
- No pueden haber ciclos de dos vértices.
- Todos los vértices pueden llegar al sumidero de alguna forma.
- Solo hay una fuente.
- Solo hay un sumidero.

Formas de respetar estas restricciones:

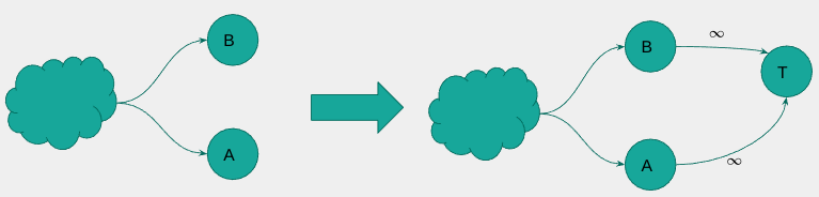
- Si hay bucles, los sacamos.
- Si hay un ciclo de dos vértices, agregamos un vértice en el medio de una de las aristas, y unimos "siguiendo el camino" ya existente, con el mismo peso en ambas aristas.



- Si hay un vértice que no tiene la capacidad de llegar al sumidero, lo sacamos.
- Si tengo más de una fuente, creo una superfuente.



- Si tengo más de un sumidero, creo un supersumidero.



Flujo máximo

La red residual es el grafo con los mismos vértices, pero tiene como aristas:

1. Las mismas del original, al que aún les quede capacidad por utilizar. El peso es esa capacidad restante
2. La arista opuesta, con peso la capacidad utilizada.
3. Si alguno de los anteriores es 0, no hay arista.

En el peor de los casos el grafo residual tiene el doble de aristas.

Un grafo residual es una red de flujo, salvo por las aristas antiparalelas.

Si encontramos un camino de la fuente (s) al sumidero (t) en el grafo residual, entonces encontramos un camino por el que podemos aumentar el flujo.

El algoritmo de Ford-Fulkerson encuentra el flujo máximo y su código es el siguiente:

Se aclara que cuando se llama a obtener_camino() se trata de un recorrido BFS.

```
1  def flujo(grafo, s, t):
2      flujo = {}
3      for v in grafo:
4          for w in grafo.adyacentes(v):
5              flujo[(v, w)] = 0
6      grafo_residual = copiar(grafo)
7      while (camino = obtener_camino(grafo_residual, s, t)) is not
       None:
8          capacidad_residual_camino = min_peso(grafo, camino)
9          for i in range(1, len(camino)):
10             if grafo.hay_arista(camino[i-1], camino[i]):
11                 flujo[(camino[i-1], camino[i])] +=
       capacidad_residual_camino
12                 actualizar_grafo_residual(grafo_residual, camino[i
       -1], camino[i], capacidad_residual_camino)
13             else:
14                 flujo[(camino[i], camino[i-1])] -=
       capacidad_residual_camino
15                 actualizar_grafo_residual(grafo_residual, camino[i
       ], camino[i-1], capacidad_residual_camino)
16
17     return flujo
```

```python
def actualizar_grafo_residual(grafo_residual, u, v, valor):
    peso_anterior = grafo_residual.peso(u, v)
    if peso_anterior == valor:
        grafo_residual.remover_arista(u, v)
    else:
        grafo_residual.cambiar_peso(u, v, peso_anterior - valor)
    if not grafo_residual.hay_arista(v, u):
        grafo_residual.agregar_arista(v, u, valor)
    else:
        grafo_residual.cambiar_peso(v, u, peso_anterior + valor)
```

La complejidad del algoritmo Ford-Fulkerson es $\mathcal{O}(V \cdot E^2)$

Corte mínimo

El corte mínimo de una red es el peso total (mńimo) que necesitamos desconectar para que un grafo deje de estar conectado (conexo para grafos no dirigidos, débilmente conexo para dirigido).

Esto se aplica a cualquier tipo de grafo, pero para redes de flujo tenemos algo bastante interesante. Si el grafo corresponde a una red de flujo, entonces el corte mínimo tiene capacidad igual al flujo máximo. Va a suceder que la fuente y el sumidero se encuentren en sets opuestos.

Para obtener el corte mínimo hacemos:

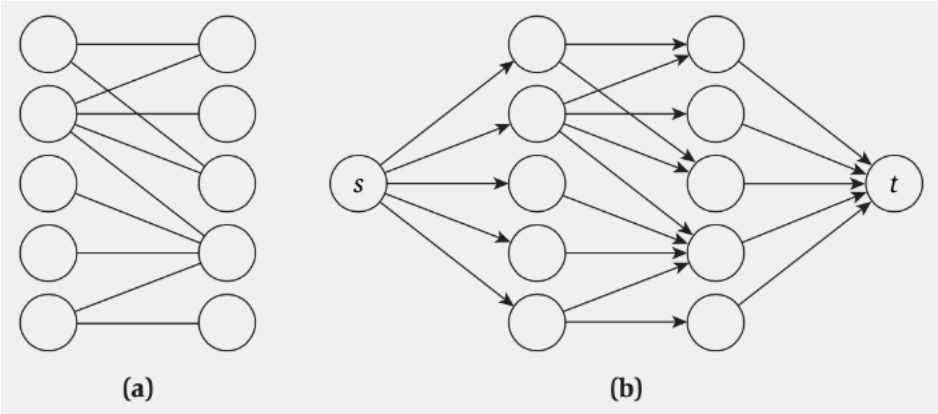
1. Agarramos nuestro grafo residual.
2. Vemos todos los vértices a los que llegamos desde la fuente.
3. Todas las aristas (del grafo original) que vayan de un vértice al que podamos llegar (en el residual) a uno que no (idem), son parte del corte.

Aplicaciones de redes de flujo

Perfect Bipartite Matching

> Dado un grafo no dirigido, un match es un subconjunto de las aristas en el cual para todo vértice V a lo sumo una arista del match incide en V (en el match, tienen grado de entrada a lo sumo 1). Decimos que el vértice v está matcheado si hay alguna arista que incida en él (sino, está unmatcheado). El matching máximo es aquel en el que tenemos la mayor cantidad de aristas (matcheamos la mayor cantidad posible).

La idea es transformar las aristas en dirigidas (de un subconjunto al otro, indistinto de cuál a cuál) y luego unir una fuente a cada uno de los vértices pertenecientes al subconjunto de *salida* y cada vértice del subconjunto de *llegada* unirlo a un sumidero.



Con Ford-Fulkerson se halla la solución a esta problemática en $\mathcal{O}(V \cdot E)$ por ser el grafo bipartito.

Disjoints Paths

> Decimos que dos caminos son disjuntos si no comparten aristas (pueden compartir nodos). Dado un grafo dirigido y dos vértices s y t, encontrar el máximo número de caminos disjuntos s-t en G.

Decimos que los pesos de las aristas son todos 1, y luego aplicamos FF tomando como fuente a s y como sumidero a t. El flujo máximo representará la cantidad de caminos disjuntos existentes.

La complejidad del algoritmo es $\mathcal{O}(V \cdot E)$

Si fuera un grafo no dirigido, agregamos vértices intermedios y las aristas en ambos sentidos y funciona.

Disjoints Paths II

> Dado grafo G = (V, E) dirigido, un nodo $o \in V$ origen y un nodo $d \in V$ destino. Deseamos conocer la cantidad de caminos nodos-disjuntos que se pueden trazar entre ellos.

Dos caminos son nodo-disjuntos entre sí, si no comparten nodos entre ellos. Es decir que queremos conocer la máxima cantidad de caminos independientes entre sí que se pueden trazar desde un nodo origen a uno destino.

En este caso aprovecharemos el trabajo realizado para resolver el problema de los caminos arco-disjuntos. Realizaremos una reducción polinomial a este problema. Realizaremos una modificación en la instancia para asegurarnos el uso de cada nodo en un camino solo una vez.

En la reducción, la primera transformación consistirá en la generación de un nuevo grafo donde reemplazamos cada nodo v (excepto el nodo origen y destino) por un par de nodos v1 y v2 unidos por un eje dirigido. Cada una de las aristas e = (u,v) que arriba al nodo v se modificará para sea (u, v1). De forma similar, cada eje e = (v,u) que sale de v se reemplazará por (v2,u). La diferencia en la manipulación de los nodos origen y destino se explica por la necesidad de evitar que el primer camino encontrado impida que nuevos se generen al entender que estos nodos ya fueron utilizados.

Una vez obtenido el grafo modificado utilizaremos como caja negra un algoritmo que resuelve el problema de caminos arco-disjuntos. Alternativamente podemos resolverlo utilizando la reducción que se analizó en la sección anterior que termina haciendo uso del problema de flujo máximo. Obtendremos como resultado la cantidad de caminos arco-disjuntos $|P|$ y los caminos en sí.

Problema de maximización de emparejamientos

> Dados un par de conjuntos A y B, un conjunto T de tuplas (a,b) con $a \in A$ y $b \in B$ de posibles emparejamientos. Deseamos construir la mayor cantidad de parejas posibles sin que los elementos aparezcan más de una vez en ellas.

Para resolver este problema transformaremos la instancia del problema en un grafo bipartito. Un grafo es bipartito si sus nodos se pueden separar en dos conjuntos disjuntos, donde los ejes existentes únicamente conectan nodos que se encuentran en diferente conjunto. Luego el grafo se transformará en una red de flujo donde la fuente se conectará mediante ejes con los elementos de uno de los conjuntos y el sumidero con los del otro. Todos los ejes tendrán una capacidad unitaria.

La primera transformación comienza al crear 2 nodos para representar los ejes y sumideros. Continuaremos con $|A| + |B|$ nodos que representan a cada uno de los conjuntos. Luego se crean $|A|$ ejes uniendo la fuente con los nodos que representan a los elementos de A. Similarmente se crean $|B|$ ejes uniendo los nodos que representan a los elementos de B con el sumidero. Finalmente se representa con ejes entre los nodos que representan a los elementos de A y B que pueden conformar una posible pareja. Estos nodos serán direccionados. En el caso más completo existirán $|A| * |B|$ de estos ejes. A todos los ejes se les otorgará, como se explicó, como capacidad el valor 1.

Transformación a instancia de problema de flujo máximo

```
Sea A y B conjuntos
Construir G=(V,E) red de flujo

Crear s fuente, insertarla en V
Crear t sumidero, insertalo en V
Por cada elemento a en A
  Crear un nodo n, insertarlo en V
  Crear un eje e=(s,n), insertarlo en E

```

```
Por cada elemento b en B
  Crear un nodo n, insertarlo en V
  Crear un eje e=(n,t) con capacidad Ce=1, insertarlo en E

Por cada tuplo t=(a,b) en T
  Crear un eje e=(a,b) con capacidad Ce=1, insertarlo en E

Retornar red de flujo G
```

Una vez obtenida la red de flujo, debemos utilizar como caja negra un algoritmo para resolver el problema de flujo máximo. La complejidad de esta ejecución no debemos medirla en función de vértices y nodos, sino en función de los parámetros de nuestro problema original. Por la forma de construir la red podemos decir que la instancia de la red G=(V,E), tendrá $|A| + |B| + 2$ nodos y como mucho $|A| * |B| + |A| + |B|$ ejes. Supondremos que la caja negra ejecuta Ford-Fulkerson. Entonces la complejidad $\mathcal{O}(C(V + E))$ se puede reemplazar en forma simplificada por $\mathcal{O}(C(A * B))$. C es la suma de las capacidades que salen de la fuente y en este caso son capacidades unitarias. Tenemos $|A|$ ejes que salen de la fuente y por lo tanto podemos expresar C como $|A|$ y la expresión de Ford Fulkerson como $\mathcal{O}(A2B)$.

Transformación de la solución del problema de flujo máximo a solución

```
Sea G=(V,E) red de flujo
Sea f flujo en la red

Establecer |P| como cantidad de parejas la suma de los flujos que
    salen de la fuente.

Por cada eje e=(u,v) en G con u!=s y v!=t
  Agregar pareja en P el elemento de A que representa u y el
    elemento de B que representa v.

Retornar |P| y P
```

La complejidad de esta segunda transformación es $O(A * B)$. La suma de todas las complejidades de las partes involucradas conforman la complejidad final para resolver el problema utilizando la reducción.

Problema de segmentación de imagen

> Sea una imagen representada como una cuadrícula de "n" píxeles. Donde cada pixel i tiene una probabilidad a_i de pertenecer al primer plano y una probabilidad b_i de pertenecer al segundo plano. Cada pixel i vecino de un pixel j tiene una penalidad p_{ij} por estar en diferente plano. Queremos separarla en primer y segundo plano de forma de maximizar la calidad de la segmentación.

A continuación elaboramos la reducción polinomial al problema del corte mínimo. La transformación de la instancia comenzará creando un grafo G=(V,E). Luego:

- Creamos el nodo fuente s en V como representación del segundo plano
- Creamos el nodo sumidero t en V como representación del primer plano
- Creamos un nodo p_i por cada píxel i de la imagen
- Un eje e=(s,p_i) por cada nodo "píxel" p_i con capacidad a_i.
- Un eje e=(p_i,t) por cada nodo "píxel" pi con capacidad b_i.
- Un eje e=(p_i,p_j) por cada píxel vecino con capacidad p_{ij} (esto genera un eje ida y otro vuelta entre cada par de píxeles vecinos).

Pseudocodigo de la transformación a instancia del problema de corte mínimo

```
Sea P los proyectos
Sea T el tope de ganancia
Construir G=(V,E) red de flujo

Crear s fuente, insertarla en V
Crear t sumidero, insertalo en V

Por cada nodo i
  Crear el nodo pi
  Crear el eje e=(s,pi) con capacidad ai
  Crear el eje e=(pi,t) con capacidad bi

Por cada nodo i
  Por cada nodo j correspondiente a un pixel vecino de i
    Crear el eje e=(pi,pj) con capacidad pij
```

Transformación de la solución de corte mínimo a solución

```
Sea G=(V,E) red de flujo
Sea f flujo en la red
Sea Gr grafo residual resultante de aplicar f en G
Sea Q la suma de todas las probabilidades ai y bi

Establecer como A (primer plano) a pixeles que representan los
    nodos de Gr accesibles desde la fuente (excluyendola) en G
Establecer como Calidad de corte q(A,B) a Q menos al flujo maximo
    de la red.

Retornar A y q(A,B)
```

La complejidad de esta transformación corresponde a la del DFS $\mathcal{O}(V+E)$ y la de obtener el flujo máximo sumando el flujo de los ejes que salen de la fuente $\mathcal{O}(E)$. Lo expresaremos en función de los parámetros del problema original. La cantidad de nodos en el grafo es $\mathcal{O}(n)$ y de igual manera es la cantidad de ejes $\mathcal{O}(n)$. Por lo que la complejidad total de la transformación es $\mathcal{O}(n)$.

Circulaciones con demandas

Supongamos que tenemos varias "fuentes" con un suministro, y "sumideros" con una demanda. Ahora cada nodo tiene una demanda (positiva, negativa o 0). Nuevas condiciones:

- Condición de capacidad: $0 \leq f(e) \leq C_e$
- Condición de demanda: $f_{in}(v) - f_{out}(v) = d_v$

Queremos ver si esto es factible y cómo.

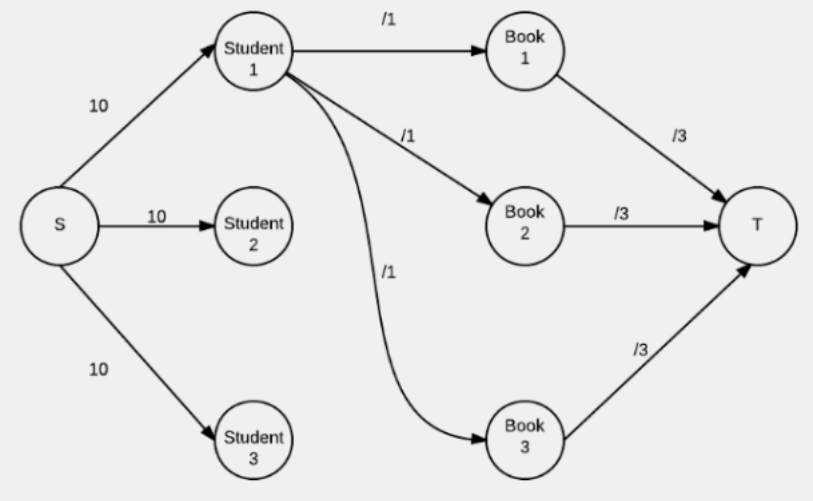
Primera condición: la suma de los suministros debe ser igual a la suma de las demandas.

Solución

Ejemplo

Supongamos que tenemos un sistema de una facultad en el que cada alumno puede pedir hasta 10 libros de la biblioteca. La biblioteca tiene 3 copias de cada libro. Cada alumno desea pedir libros diferentes. Implementar un algoritmo que nos permita obtener la forma de asignar libros a alumnos de tal forma que la cantidad de préstamos sea máxima.

1. Creamos superfuente y supersumídero.
2. Para los sumíderos, agregamos una arista de t a t* con capacidad = demanda.
3. Para las fuentes, agregamos una arista de s* a s con capacidad = suministro.



Un grafo G tiene una circulación factible con demandas sii para todos los cortes (A, B):

$$\sum_{v \in B} d_v \leq c(A, B)$$

Circulaciones con demandas y cotas mínimas

Ahora para cada arista, además de tener una capacidad tenemos una cota inferior que debe cumplirse, y cada nodo puede tener una demanda. Nuevas condiciones:

1. $L_e \leq f(e) \leq C_e$
2. $f_{in}(v) - f_{out}(v) = d_v$

Solución
Transformamos al problema en uno con demandas sin cotas inferiores:

1. Definimos uno flujo que cumpla las capacidades (incluyendo cotas).
2. Creamos un grafo nuevo con mismos vértices y aristas, con *capacidad* = C_v − *consumido* (salvo que la demanda se cumpla, entonces no ponemos al vértice y sus aristas).

Ejemplo

Suponer que queremos schedulear cómo los aviones van de un aeropuerto a otro para cumplir horarios y demás.
Podemos decir que podemos usar un avión para un segmento/vuelo i y luego para otro j si:

- El destino de i y el origen de j son el mismo.
- Podemos agregar un vuelo desde el destino de i al origen de j con tiempo suficiente.

Decimos que el vuelo j es alcanzable desde el vuelo i si es posible usar el avión del vuelo i y después para el vuelo j. Problema: ¿Podemos cumplir con los m vuelos usando a lo sumo k aviones?

Solución

1. Nuestras unidades de flujo son literalmente los aviones.
2. Ponemos las aristas de los vuelos que queremos si o si cumplir con cota mínima y capacidad = 1 (para forzar que se usen).
3. Si otro vuelo es alcanzable por las reglas anteriores, ponemos otra arista de capacidad 1.

4. Ponemos una fuente con aristas de capacidad 1 a los orígenes.
5. Ponemos un sumidero con aristas de capacidad 1 desde los destinos.
6. Ponemos una arista de la fuente al sumidero con capacidad K (los aviones que sobren no se usan).
7. La fuente tiene demanda -K y el sumidero K.

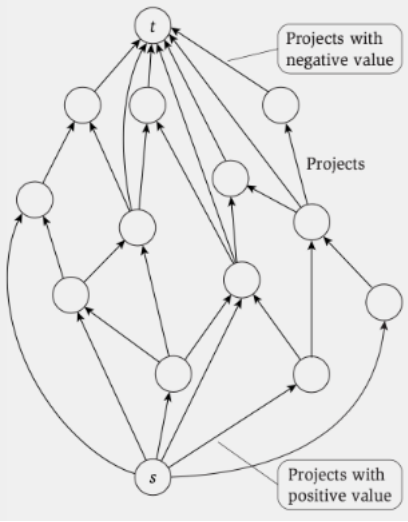
Ejemplo 2

Tenemos proyectos que podemos realizar. Algunos dan ganancia positiva y otros negativa. Hay dependencias: puede que para hacer un proyecto i también tengamos que hacer un proyecto j, sin ciclos.
Por supuesto un proyecto con ganancia negativa sin otro proyecto (positivo) que dependa de este, no se hará. Un proyecto que tiene ganancia positiva y no depende de uno negativo, se hará.

Solución

1. Ponemos vértices = proyectos. Arista (i, j) cuando el proyecto i depende del j.
2. Ponemos fuente, y unimos a proyectos positivos con capacidad igual al beneficio del proyecto.
3. Ponemos sumidero, y unimos los proyectos negativos con capacidad -beneficio del proyecto.
4. A las aristas de las dependencias le ponemos "capacidad infinita" (o la suma de los proyectos positivos + 1).

Obtenemos el corte mínimo (A', B') y nos quedamos con A' - fuente.



Reducciones

P y NP

Decimos que el problema Y se reduce polinomialmente al problema X ($Y \leq_p X$) si podemos traducir una entrada del problema Y a una del problema X en tiempo polinomial, resolver el problema X y establecer la correspondencia entre la salida del problema X con la salida del problema Y.

Las reducciones son transitivas.

P: problemas que se pueden resolver en tiempo polinomial (de forma eficiente). $\rightarrow$ Pueden ser resueltos en tiempo polinomial por una máquina de Turing determinística.

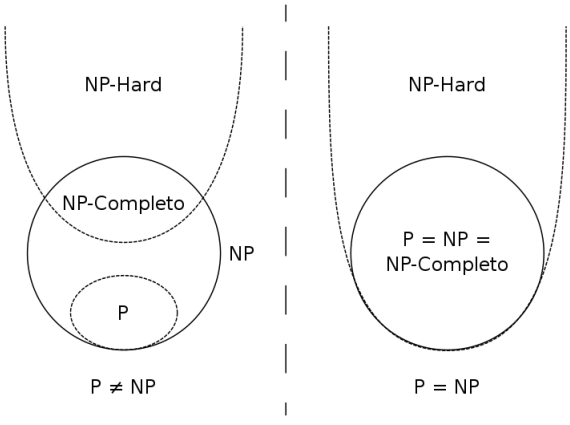
NP: problemas para los que existe un certificador eficiente, es decir que se pueden validar en tiempo polinomial $\rightarrow$ Pueden ser resueltos en tiempo polinomial por una máquina de Turing indeterminística.

Respecto a estos últimos, es evidente que todos los problemas que pertenezcan a P también pertenecen a NP, ya que si se puede encontrar una solución en *tp*, también se puede validar una solución en *tp*. Eso no implica que todos los problemas NP pertenezcan a P. Un problema *díficil* de resolver, puede ser *fácil* de validar.

Resumiendo, $P \subseteq NP$ pero $NP \nsubseteq P$ (a menos que $P = NP$).

Un problema pertenece a NP-Hard (también escrito como NP-H) si todo problema que pertenece a NP se puede reducir polinomialmente a este. Expresado matemáticamente $X \in NP - H \longleftrightarrow \forall Y \in NP,\ Y \leq_p X$.

Un problema es NP-Completo si cualquier problema perteneciente a NP puede ser reducido polinomialmente al problema NP-Completo. Esto, por consecuente implica que los problemas NP-Completos son todos reducibles entre sí (en ambos sentidos).



Para demostrar que un problema Y es NP-Completo, se debe demostrar que Y efectivamente pertenece a NP, y además se debe demostrar que cualquier

problema perteneciente a NP es reducible polinomialmente a este problema. Para ello, la forma más sencilla de hacerlo es reduciendo polinomialmente un problema NP-Completo ya conocido a nuestro problema Y.

Problemas NP-Completos

Listado de problemas NP-Completos:

- SAT y 3-SAT.
- Independent Set.
- Vertex Cover.
- N-Reinas.
- Ciclo Hamiltoniano.
- Camino Hamiltoniano.
- Coloreo de Grafos (con $k \geq 3$).
- Subset Sum.
- Problema de la mochila.

SAT y 3-SAT

El problema de SAT se basa en dar todas las soluciones a una expresion booleana descripta por la unión de cláusulas.

Cada clásula se compone de la siguiente forma:

$C_1 = X_1 \vee X_2 \vee \neg X_3 \vee ... \vee X_n$

y la unión sería $S = C_1 \wedge C_2 \wedge C_3 \wedge ... \wedge C_k$

3-SAT es el problema en el cual cada clásula tiene 3 literales solamente. ($C_1 = X_1 \vee X_2 \vee X_3$)

K-Clique

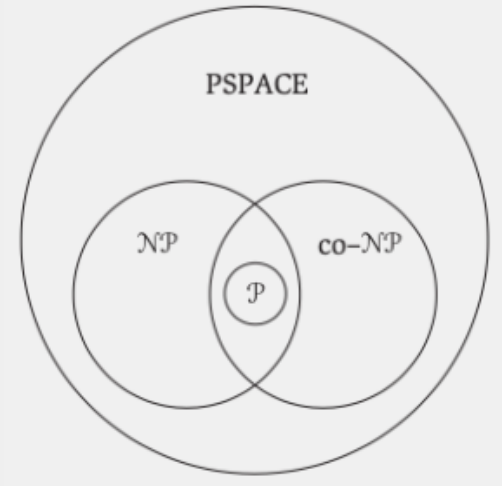
> Dado un grafo no dirigido G y un entero positivo k, se busca determinar si existe un conjunto de k vértices en G tal que todos los vértices en ese conjunto estén conectados entre sí por una arista (es decir, si existe un subgrafo completo de tamaño k vértices).

Reducimos de Independent Set → K-Clique:

1. Obtenemos el grafo complemento → aristas que no están en G.
2. Definimos que hay un Independent Set de al menos K vértices si hay un clique de al menos K vértices en el grafo complemento.

PSPACE

Problemas que se resuelven con un algoritmo que consume una cantidad polinomial de espacio.



PSPACE-completo es el subconjunto de los problemas de decisión en PSPACE y todo problema en PSPACE puede ser reducido a él en tiempo polinomial.

Algoritmos de aproximación

Se utilizan para encontrar soluciones que garantizan estar cerca del óptimo para problemas que no se conoce una solución en tiempo polinomial.

Problemas vistos en clase

Problema de cargas

Dadas m Máquinas y un conjunto n de trabajos, donde cada trabajo j toma un tiempo Tj, se desea asignar el trabajo en las Máquinas de forma balanceada.
Dada una asignación A(i) para la Máquina i, su tiempo de trabajo es $T_i = \sum_{j \in A(i)} t_j$
Y queremos encontrar la asignación que minimice el máx valor de Ti, que también representa el tiempo que se tardará en finalizar todos los trabajos.

Solución greedy

- Iterar sobre todos los trabajos.

- Para cada trabajo, asignarlo a la máquina con menos trabajo al momento.

Si se ordena los trabajos de mayor a menor y se los va colocando, se aproxima a una solución más cercana a la óptima.

Mochila aproximada

> Tenemos una mochila con una capacidad W. Hay elementos a guardar. Cada elemento tiene un peso y un valor. Queremos maximizar el valor de lo que llevamos sin excedernos de la capacidad W.

Propondremos un nuevo algoritmo el cual buscará la capacidad mínima que debe tener una mochila para alcanzar ciertos valores, a diferencia del algoritmo previamente visto, el cual buscaba el máximo valor para cierta capacidad de la mochila.

La ecuación de recurrencia queda como:

$$OPT(n,V) = min \begin{cases} \nearrow \text{utilizar un elemento: } Pi + OPT(n-1, max(0, V - Vi) \\ \searrow \text{no utilizarlo: } OPT(n-1, V) \end{cases}$$

```
def mochila(valor, peso, N, W, T=sum(valor)):
    OPT[0..N+1][0..T+1]
    for i in range(0, n+1):
        OPT[i][0] = 0
    for i in range(1, n+1):
        sum_valor = sum(valor[:i+1])
        for V in range(1, sum_valor):
            if V > sum_valor-valor[i]:
                OPT[i][V] = peso[i] + OPT[i-1][V-valor[i]]
            else:
                OPT[i][V] = min(OPT(n-1, V), peso[i] + OPT(n-1, max
    (0, V-valor[i])))
    return max(V donde OPT[N][V] <= W)
```

Maximizar sumatoria

> Dado un límite W y un conjunto de valores enteros V, se busca un subconjunto de V que maximice la sumatoria de valores sin exceder el valor W.

Greedy propuesto: agregar "como vienen", siempre y cuando no se exceda el valor W.

Algoritmos randomizados

Le vamos a permitir al algoritmo tomar decisiones al azar Independientemente de los valores de entrada El rol de esta aleatoriedad es un factor completamente interno al algoritmo

- Hacer modelos más potentes
  - Un algoritmo determinístico eficiente que calcula la respuesta correcta puede verse como un caso particular de un algoritmo randomizado que da la respuesta correcta con alta probabilidad
  - Un algoritmo determinístico eficiente que calcula la respuesta correcta puede verse como un caso particular de un algoritmo randomizado que da la siempre la respuesta correcta y tiene una expectativa de complejidad temporal eficiente
- Un Algoritmo Randomizado podría resolver un problema que podría no ser resuelto de forma eficiente por un algoritmo determinístico
- Conceptualmente más sencillos de entender e implementar
- Funcionamiento sin requerir mantener estado interno o memoria del pasado
- Para sistemas distribuidos es una forma de quebrar la simetría entre procesos que corren el mismo algoritmo

Algunos conceptos de proba:

- Probabilidad de que un evento ocurra: p
- Probabilidad de que un evento no ocurra: 1-p
- Esperanza, valor esperado de un evento: $E[X] = \sum_{j=0}^{\inf} j \cdot Pr[X = j]$

Problemas vistos en clase

Problema de encontrar la mediana

Para un conjunto de números S = a1, a2,..., an, la Mediana es aquel valor que estaría en la posición del medio si el conjunto de números estuviera ordenado. La Mediana está en la posición k del arreglo ordenado tal que:

- si la cantidad n es impar, k es (n+1)/2
- si la cantidad n es par, k es n/2

Este problema se puede resolver fácilmente ordenando. En $\mathcal{O}(n \cdot \log n)$ podemos ordenar, luego acceder a la posición k en $\mathcal{O}(1)$.

De hecho, esta solución nos permitiría Seleccionar cualquier valor k en el arreglo ordenado.

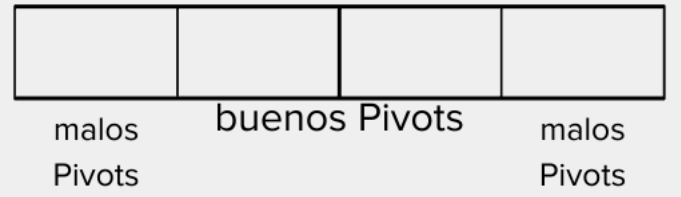
Deseamos un algoritmo de División y Conquista con tiempo esperado $\mathcal{O}(n)$ y será un Algoritmo Randomizado

```
def select(S, k):
    ai = elegir_pivot(S, k)
    for elem in S:
        Agregar elem a S- si elem<ai
        Agregar elem a S+ si elem>ai
    if |S-| == k-1: return ai
    if |S-| >= k: return select(S-, k)
    else:
        return select(S+, k-1-|S-|)
```

(Las | | representan el tamaño de cierta colección)

Para que la complejidad del algoritmo realmente sea $\mathcal{O}(n)$ la idea es que el pivot esté bastante centrado, es decir, que se acerque al valor de la mediana.

Para asegurarnos de esto lo que haremos es elegir el pivot al azar, ya que hay mas probabilades de elegir un buen pivot que uno malo:



Decimos que el Algoritmo está en fase j, si la cantidad de elementos del set en consideración está entre $n(\frac{3}{4})^j$ y $n(\frac{3}{4})^{(j+1)}$

Se logra para la mitad central de Pivots, con probabilidad es $\frac{1}{2}$.

La cantidad esperada de iteraciones para avanzar de fase es 2.

Podemos definir que X es la cantidad de trabajo del algoritmo hasta atravesar todas las fases, será la suma $X = X_0 + X_1 + X_2 + \ldots$

La cantidad de trabajo en una fase es alguna constante veces el tamaño del set en esa fase, que es como mucho $n(\frac{3}{4})^j$, esperable sean hasta 2 iteraciones $E[X_j] \leq 2cn(\frac{3}{4})^j$

Gracias a la Linealidad de la Esperanza, y la sumatoria geométrica convergente

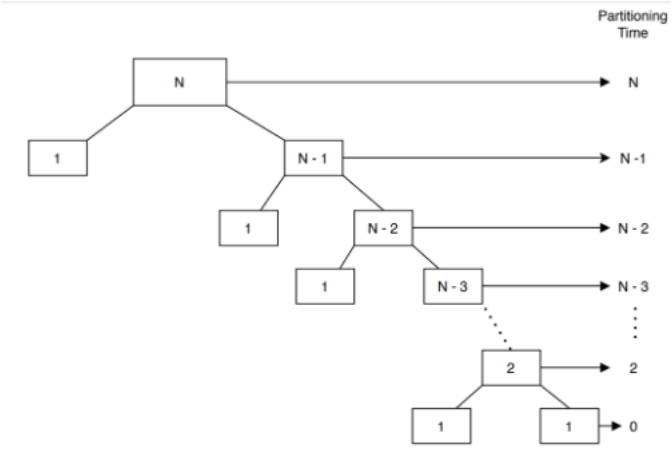
$E[X] = \sum_j E[X_j] \leq \sum_j 2cn(\frac{3}{4})^j = 2cn \sum_j (\frac{3}{4})^j \leq 8cn$

Nos queda que la complejidad esperada es $\mathcal{O}(n)$

Quicksort

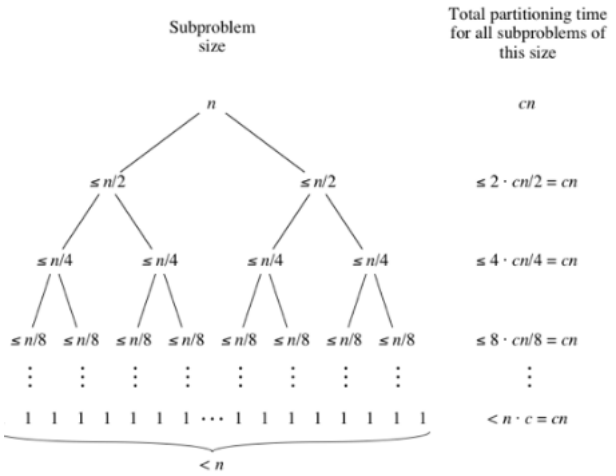
```
def quicksort(S):
    if |S|<=2: #caso base: return orden trivial
    ai = elegir_pivot(S)
    for elem in S:
        Agregar elem a S- si elem<ai
        Agregar elem a S+ si elem>ai
    return quicksort(S-) + [ai] + quicksort(S+)
```

Peor caso:



Complejidad: $\mathcal{O}(n^2)$

Mejor caso:



Complejidad: $\mathcal{O}(n \cdot \log n)$

El tiempo esperado del algoritmo para un Set S, sin contar llamadas recursivas, es linealmente proporcional a S, repetido una cantidad esperada de hasta 2. Es $O(|S|)$

Las llamadas recursivas las agrupamos por tamaño en fase j, si la cantidad está entre $n(\frac{3}{4})^j$ y $n(\frac{3}{4})^{(j+1)}$. Fuera de la recursividad tenemos en fase j: $\mathcal{O}(n(\frac{3}{4})^j)$

Partir con un pivot central, genera Sets del tamaño de la siguiente fase

El número máximo de problemas de fase j es $(\frac{4}{3})^{(j+1)}$, con costo $\mathcal{O}(n(\frac{3}{4})^j)$ cada uno.

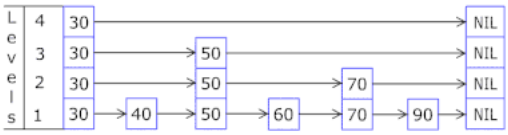
$E[X_j] = \mathcal{O}(n)$ para cada j, y por linealidad de la esperanza podemos sumar las fases.

Cantidad máxima de fases en función de n $\log_{\frac{4}{3}} n = \mathcal{O}(\log n) \rightarrow \mathcal{O}(n \cdot \log n)$

- El tiempo del algoritmo es independiente del orden de los elementos de entrada
- No hay un conjunto de datos de entrada que pueda garantizar el peor caso
- El peor caso solamente surge de un mal caso de las selecciones aleatorias realizadas

Skip-Lists

Las Skip Lists son estructuras que incorporan una randomización para permitir de forma eficiente, con alta probabilidad, la búsqueda, guardado y borrado en una lista ordenada



Se busca simil búsqueda binaria, empezando por las capas superiores.

Al insertar siempre se insertará en la base, y se sube de capa con probabilidad p.

Las capas superiores tienden a tener menos elementos.

Las capas superiores funcionan como ruta express para llegar al elemento.

Orden temporal promedio, con alta probabilidad: $\frac{1}{p} \log_{\frac{1}{p}} n$.

- Búsqueda: $\mathcal{O}(\log n)$.
- Inserción: $\mathcal{O}(\log n)$.
- Borrado: $\mathcal{O}(\log n)$.
- Orden temporal de los peores casos: $\mathcal{O}(n)$.
- Orden espacial en el peor caso: $\mathcal{O}(n \cdot \log n)$.

Funciones de hashing

Perfect Hashing
FKS Hashing

- Se aplica cuándo sabemos cuáles/cuántos elementos habrán.

- Tabla de tamaño número primo, con prob de colisión 1/n (Clase Universal).
- Para los elementos que colisionan en una posición, se crea una nueva sub-tabla de Hash solamente para esos elementos, de tamaño n_i^2.
- Para esta nueva sub-tabla se usa cualquier función de la clase universal que cumpla que no genera colisiones para los n_i elementos.
- Al tomar cualquier función al azar de la clase, la cantidad esperada de colisiones es $\frac{1}{2}$. Encontraremos una función adecuada rápidamente.
- El espacio utilizado es aproximadamente 2n, es decir, $\mathcal{O}(n)$.

Metodologías

Las Vegas

- Siempre dan la respuesta correcta.
- El tiempo en encontrar la solución puede variar aleatoriamente

Ejemplos de Algoritmos tipo Las Vegas

- Selección con pivote aleatorio.
- Quicksort con pivote aleatorio.
- Bogosort: mezcla todos los elementos del arreglo, revisa si están ordenados, en ese caso, termina.

Monte Carlo

- Puede dar la respuesta incorrecta con cierta probabilidad
- El tiempo en el que corre un algoritmo Montecarlo está siempre acotado polinomialmente
- Como parte de un esquema podemos correr un algoritmo Montecarlo varias veces para incrementar la confianza en la solución

Ejemplo:

El corte mínimo de un grafo es el peso total (mínimo) que necesitamos desconectar para que un grafo deje de estar conectado (conexo para grafos no dirigidos, débilmente conexo para dirigido). Y también vimos una resolución para grafos que son red de flujo El Algoritmo de Karger busca el corte mínimo de un grafo, en general

- El algoritmo hace unas “contracciones”
- Las aristas que contrae son al azar

- La probabilidad de que una corrida del algoritmo encuentre exactamente el corte mínimo es: $\binom{n}{2}^{-1}$
- Como parte del esquema, se hacen varias corridas aleatorias del algoritmo $\binom{n}{2} \ln n$
- Repitiendo el algoritmo una cantidad de veces:
- Y quedándose siempre con el mejor resultado, la probabilidad de que no se encuentre el Corte Mínimo es acotada por $\frac{1}{n}$

Heurísticas

- Métodos para aproximarse más rápidamente a una solución
- Usado en casos de algoritmos de búsqueda u optimización
- Una función heurística es una función que permite rankear diferentes opciones basada en la información disponible
- Los algoritmos Greedy pueden verse como heurísticas que tratan de aproximar la solución óptima de un problema
- Ejemplo de problema de la mochila: heurística de guardar primero los elementos con mejor función valor/peso