Algoritmos ávidos

ANÁLISIS Y DISEÑO DE ALGORITMOS

MTI. PEDRO O. PÉREZ M.
TECNOLÓGICO DE MONTERREY, CAMPUS QUERÉTARO

Contenido

- ¿En qué consiste el paradigma de Algoritmos ávidos?
- Algoritmos que aplican este paradigma.
  - Cambio en monedas.
  - Criba de Eratóstenes.
  - Asignación de recursos (y algunas variantes).
  - Union-disjoint set.
  - Código de Huffman y compresión de datos.

¿En qué consiste el paradigma de Algoritmos ávidos?

- Un algoritmo ávido sugiere la construcción de una solución a través de una secuencia de pasos, cada uno expande una solución parcial previamente generada, hasta llegar a la solución completa del problema.
- Cada paso (punto central de este paradigma) una decisión debe ser hecha con base en:
  - Probable: que satisfaga las restricciones del problema.
  - Óptima local: Es la mejor opción de todas las opciones probables disponibles en ese paso en particular.
  - Irrevocable: una vez tomada una determinada opción, no es posible cambiar.

¿En qué consiste el paradigma de Algoritmos ávidos?

- Los algoritmos ávidos son aplicables solamente a problemas de optimización.
- Un algoritmo ávido puede producir una solución cercana a la óptima.
- ¿Conoce algún algoritmo de este tipo?
  - El camino más corto (algoritmo de Djikstra).
  - El árbol de expansión mínima (algoritmos de Prim y Kruskal).

Cambio de monedas

- Problema: Dado un sistema monetario S con N monedas de diferentes denominaciones y una cantidad de cambio C, indique el menor número de monedas de S equivalente a C.

Cambio de monedas

```
procedure COIN_CHANGE(C, S[N]):
amount ← 0
Ordenar las monedas en S de mayor a menor ( O(n log n) )
foreach s in S do
begin
    while s < C do
    begin
        amount ← amount + 1
        C ← C - s
     end
end
return amount
```

¿Nos da la mejor opción? Prueba el algoritmo con el conjunto de monedas {4,3,1} y el cambio es 6.

Asignación de recursos

- Tenemos un recurso (un salón de clases, una supercomputadora, etc.) y mucha gente requiere usarlo por cierto intervalo de tiempo. El recurso solo puede ser usado por un persona a la vez. La meta es maximizar el número de peticiones aceptadas.
- Sea $P = \{p_1, p_2, ..., p_1\}$, el conjunto de peticiones. Cada petición i tienen un inicio de uso, s_i, y un tiempo final, f_i.

Asignación de recursos

```
procedure INTERVAL_SCHEDULING(R):
A ← {}
Ordenar las peticiones en P de por el tiempo final
foreach p in P do
begin
    Agrega p a A
    Elimina cualquier petición en P que sea incompatible con p
end
return A
```

Variante 1

- En el problema anterior, sólo hay un recurso y muchas peticiones; pero que pasa si tenemos varios recursos del mismo tipo. Este problema es conocido como la Partición de intervalos (Interval partitioning).
- Sea $P = \{p_1, p_2, \ldots, p_1\}$, el conjunto de peticiones. Cada petición i tienen un inicio de uso, s_i, y un tiempo final, f_i.
- Sea $R = \{r_1, r_2, \ldots, r_n\}$, el conjunto de recursos disponibles.

Variante 1

```
procedure INTERVAL_PARTITIONING(P, R):
Ordenar las peticiones en P por el tiempo inicial
foreach p in P do
begin
    A ← P
    foreach p' in A do
        if p' precede y/o se superpone a p then
            elimina p'
    A ← A ∪ p
    foreach r in R do
    begin
        Sea a un elemento en A, asigna el intervalo a al recurso r
        Elimina r de R
        Elimina a de A
    end
end
```

Variante 2

- Considera ahora una situación en la cual tenemos un sólo recurso y un conjunto de peticiones para usar el recurso por un cierto intervalo de tiempo. Sin embargo, ahora las peticiones son mas flexibles. En vez de un tiempo de inicio y fin, la petición *i* tiene un límite de tiempo d_i, y requiere de un intervalo de tiempo ininterrumpido para su terminación, t_i.

Variante 2

- Sea $P = \{p_1, p_2, \ldots, p_n\}$, el conjunto de peticiones. Cada petición i tienen un tiempo límite, d_i, y un tiempo de ejecución, t_i.

Variante 2

```
procedure LATENESS_PARTITIONING(P):
Ordenar las peticiones en P por su tiempo límite
scheduled ← {}
f ← start_time
foreach p in P do
begin
    Asigna el trabajo p al intervalo de tiempo [s_i, f_i] a scheduled donde
        s_i ← f
        f_i ← f + t_i
    f ← f + t_i
end
return scheduled
```

Mantenimiento del cache

- Problema: Esta trabajando en un importante trabajo de investigación, y tu bibliotecario sólo te permite tener prestado ocho libros a la vez. Tú sabes que probablemente necesitarás más que esa cantidad para terminar tu investigación; pero en cualquier momento, te gustaría tener acceso a los ocho libros que sean más relevantes de ese momento. ¿Cómo debes decidir cuáles libros pedir prestado, y cuándo debes regresar alguno por intercambio de otro, minimizando el número de veces que ir a la biblioteca?

Mantenimiento del cache

```
procedure CACHING():
if d_i necesita ser traído a cache then
   elimina el elemento que va a tardar más en ser utilizado.
```

Union-find Set

```
procedure KRUSKAL( G(v,e) ):
A = Ø
foreach v in V do
    MAKE-SET(v)
        foreach (u, v) ordered by weight(u, v), increasing do
            if FIND-SET(u) ≠ FIND-SET(v) then
                    A = A U {(u, v)}
                    UNION(u, v)
    return A
```

Union-find Set

- La estructura de datos Union-Find nos permite mantener conjuntos disjuntos. Tiene dos operaciones muy simples:
  - FIND(p, q) regresa verdadero si el conjunto al cual pertenece p es mismo conjunto de q.
  - UNION(p, q) permite juntar el conjunto al cual pertenece p con el conjunto al cual pertenece q.
- ¿Cómo podríamos implementar esta estructura de datos?

Union-find Set

```
public class QuickFind {
	private int ids[];

	public QuickFind(int n) {
		ids = new int[n];
		for (int i = 0; i < ids.length; i++) {
			ids[i] = i;
		}
	}

	public boolean find(int p, int q) {
		return ids[p] == ids[q];
	}

	public void union(int p, int q) {
		int pid;

		pid = ids[p];
		for (int i = 0; i < ids.length; i++) {
			if (ids[i] == pid) {
				ids[i] = ids[q];
			}
		}
	}
}
```

Union-find Set

```
public class QuickUnion {
    private int ids[];

    public QuickUnion(int n) {
        ids = new int[n];
        for (int i = 0; i < ids.length; i++) {
            ids[i] = i;
        }
    }

    private int root(int p) {
        while(p != ids[p]) {
            p = ids[p];
        }
        return p;
    }

    public boolean find(int p, int q) {
        return (root(p) == root(q));
    }

    public void union(int p, int q) {
        int pid = root(p);
        int qid = root(q);
        ids[pid] = qid;
    }
}
```

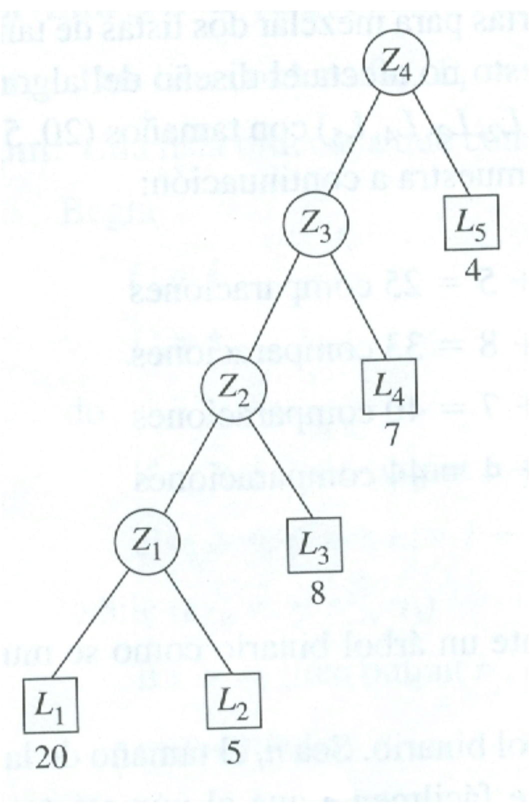
Union-find Set

```
public class WeightQuickUnion {
private int ids[];
private int szs[];

public WeightQuickUnion(int n) {
      ...
}

private int root(int p) {
      while (p != ids[p]) {
            ids[p] = ids[ids[p]];
            p = ids[p];
      }
      return p;
}

public boolean find(int p, int q) {
      return (root(p) == root(q));
}

public void union(int p, int q) {
      int pid = root(p);
      int qid = root(q);

      if (szs[pid] < szs[qid]) {
            ids[pid] = qid;
            szs[qid] += szs[pid];
      } else {
            ids[qid] = pid;
            szs[pid] += szs[qid];
      }
}
```

Mezclar 2 listas

- Problema:
  - Cuando se cuentan con dos listas ordenadas $L_1 = \{a_1, a_2, ..., a_m\}$ y $L_2 = \{b_1, b_2, ..., b_n\}$, pueden fusionarse en una lista ordenada aplicando el algoritmo de Merge Sort.
  - El número de comparaciones requeridas es $(m + n - 1)$ en el peor caso.
  - Pero, ¿qué pasa cuando hay más de dos listas ordenadas y queremos combinarlas en una sola lista?
- Por ejemplo, supongamos que tenemos un conjunto de listas $\{L_1, L_2, L_3, L_4, L_5\}$ con tamaños {20, 5, 8, 7, 4} respectivamente. ¿De qué forma se pueden mezclas?
  - L_1 y $L_2 \rightarrow Z_1$, 20 + 5 = 25
  - Z_1 y $L_3 \rightarrow Z_2$, 25 + 8 = 33
  - Z_2 y $L_4 \rightarrow Z_3$, 33 + 7 = 40
  - Z_3 y $L_5 \rightarrow Z_4$, 40 + 4 = 44
  - Nos da un total de 142 comparaciones.

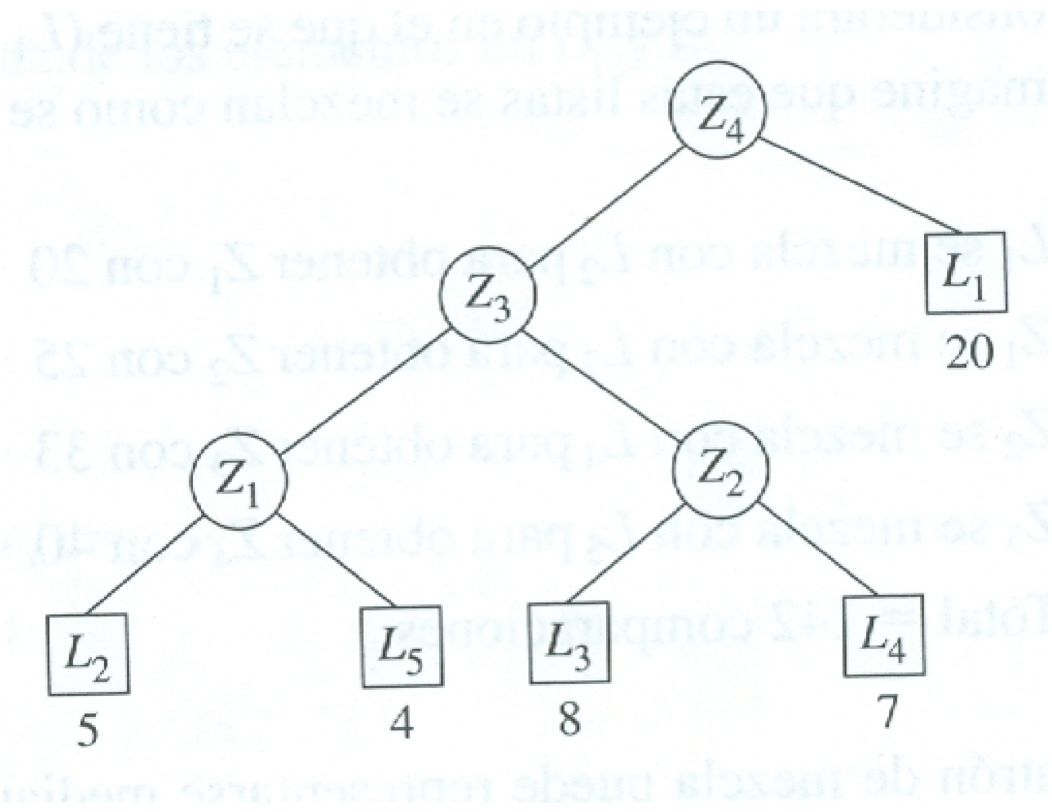
Mezclar 2 listas



Mezclar 2 listas

```
procedure TWO_WAY_MERGE(L):
Ordenar L por el tamaño de la listas
while SIZE(L) <> 1 do
begin
   Obtener (y remover) los dos primeros elementos, a y b, de L
   Juntar los tamaños de a y b en c.
   Agregar c a L
end
return el primer elemento de L
```

Mezclar 2 listas



Mezclar 2 listas