Resumen Algoritmos

1. Teorema Maestro

Para analizar la complejidad de algoritmos recursivos de la forma:

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

donde:

- a: número de subproblemas $\rightarrow$ cantidad de llamadas recursivas
- b: factor de división del problema. **solo se puede dividir simétricamente**.
- $f(n) \quad o \quad d$: costo de dividir y combinar

Comparamos $n^{\log_b a}$ con $f(n)$:

1. Si $f(n) = O(n^{\log_b a - \epsilon})$, entonces $T(n) = \Theta(n^{\log_b a})$
2. Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \log n)$
3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$, entonces $T(n) = \Theta(f(n))$

O en español:

1. Si $a = b^d \rightarrow O(n^d log(n))$
2. Si $a < b^d \rightarrow O(n^d)$
3. Si $a > b^d \rightarrow O(n^{(log_b(a))})$

2. Algoritmos Greedy

2.1. Características Principales

- Toma decisiones localmente óptimas
- No reconsidera decisiones previas
- Requiere demostrar que las decisiones locales llevan al óptimo global

2.2. Ejemplo: Problema de Scheduling

```
struct Activity {
    int start, finish;
};

vector<Activity> activitySelection(vector<Activity>& acts) {
    sort(acts.begin(), acts.end(),
         [](Activity a, Activity b) {
             return a.finish < b.finish;
         });

    vector<Activity> result;
    result.push_back(acts[0]);

    int lastFinish = acts[0].finish;
    for(int i = 1; i < acts.size(); i++) {
```

```
        if(acts[i].start >= lastFinish) {
            result.push_back(acts[i]);
            lastFinish = acts[i].finish;
        }
    }
    return result;
}
```

3. Programación Dinámica

3.1. Características

- Superposición de subproblemas
- Subestructura óptima
- Memorización (top-down) o tabulación (bottom-up)

3.2. Ejemplo: Fibonacci con DP

```
// Bottom-up approach
int fib(int n) {
    vector<int> dp(n + 1);
    dp[0] = 0; dp[1] = 1;

    for(int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];

    return dp[n];
}

// Optimizaci n de espacio
int fibOptimized(int n) {
    int prev = 0, curr = 1;
    for(int i = 2; i <= n; i++) {
        int next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

4. Algoritmos de Grafos

4.1. Dijkstra

Para caminos más cortos desde un origen en grafos con pesos no negativos.

```
vector<int> dijkstra(vector<vector<pair<int,int>>>& adj,
                    int V, int src) {
    vector<int> dist(V, INT_MAX);
    priority_queue<pair<int,int>,
                   vector<pair<int,int>>,
                   greater<pair<int,int>>> pq;

    dist[src] = 0;
    pq.push({0, src});

    while(!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for(auto& [v, weight] : adj[u]) {
```

```
            if(dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

4.2. Bellman-Ford

Para detectar ciclos negativos y encontrar caminos más cortos.

```
vector<int> bellmanFord(int V, int src,
    vector<vector<int>>& edges) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    // Relajaci n V-1 veces
    for(int i = 1; i <= V-1; i++) {
        for(auto& edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];
            if(dist[u] != INT_MAX &&
               dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Detectar ciclo negativo
    for(auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];
        int weight = edge[2];
        if(dist[u] != INT_MAX &&
           dist[u] + weight < dist[v])
            return {}; // Hay ciclo negativo
    }
    return dist;
}
```

4.3. Prim

Para árbol de expansión mínima.

```
vector<int> prim(vector<vector<pair<int,int>>>& adj,
                 int V) {
    vector<bool> visited(V, false);
    vector<int> parent(V, -1);
    vector<int> key(V, INT_MAX);

    priority_queue<pair<int,int>,
                  vector<pair<int,int>>,
                  greater<pair<int,int>>> pq;

    key[0] = 0;
    pq.push({0, 0});

    while(!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        visited[u] = true;

        for(auto& [v, weight] : adj[u]) {
```

```
            if(!visited[v] && weight < key[v]) {
                parent[v] = u;
                key[v] = weight;
                pq.push({key[v], v});
            }
        }
    }
    return parent;
}
```

5. Sliding Window Technique

5.1. Características

- Útil para problemas de subarreglos/subcadenas
- Mantiene una ”ventana”que se desliza sobre los datos
- Puede ser de tamaño fijo o variable

5.2. Ejemplo: Máxima suma de subarray de tamaño k

```
int maxSumSubarray(vector<int>& arr, int k) {
    int n = arr.size();
    if(n < k) return 0;

    int maxSum = 0;
    // Primera ventana
    for(int i = 0; i < k; i++)
        maxSum += arr[i];

    int windowSum = maxSum;
    // Deslizar ventana
    for(int i = k; i < n; i++) {
        windowSum = windowSum - arr[i-k] + arr[i];
        maxSum = max(maxSum, windowSum);
    }
    return maxSum;
}
```

6. Complejidades y Casos Especiales

6.1. Complejidades

- Dijkstra: $O((V + E)\log V)$ con cola de prioridad
- Bellman-Ford: $O(VE)$
- Prim: $O((V + E)\log V)$ con cola de prioridad
- Sliding Window: Generalmente $O(n)$ donde n es tamaño del input

6.2. Casos Especiales

- Dijkstra falla con pesos negativos
- Bellman-Ford detecta ciclos negativos
- Prim requiere grafo conexo
- Sliding Window requiere que la propiedad sea monotónica

7. Patrones de Identificación

- **Greedy**: Problemas de scheduling, selección de actividades
- **DP**: Optimización con subproblemas superpuestos
- **Dijkstra**: Camino más corto sin pesos negativos
- **Bellman-Ford**: Cuando puede haber pesos negativos
- **Prim**: Conectar puntos con costo mínimo
- **Sliding Window**: Subarreglos consecutivos, strings

8. Ejemplos

8.1. Template para Competencia

```
#pragma GCC optimize("Ofast")
#pragma GCC optimize("unroll-loops")
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define fast_cin() ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
```

8.2. Ejemplo Prim

```
ll prim(ll s){
    ll totalSum = 0;
    priority_queue<pair<ll, ll>, vector<pair<ll, ll>>, greater<pair<ll, ll>>> pq;
    vector<bool> visited = vector<bool> (n+1, false);
    pq.push({0, s});

    while (!pq.empty()){
        auto edge = pq.top();
        pq.pop();
        ll src = edge.second;
        if (visited[src]){
            continue;
        }
        totalSum += edge.first;
        visited[src] = true;
        visitedCount++;
        if (visitedCount >= n){
            break;
        }

        for (auto edge : adjList[src]){
            ll cost = edge.first;
            ll to = edge.second;
            if(!visited[to]){
                pq.push({cost, to});
            }
        }
    }
    return totalSum;
}
```

8.3. Ejemplo DP

```
long cacheLoader(vector<long> &cost, long P, table& cache){

    long n = cost.size();
    /*
    We have have a n*P cache table and we're filling it with bottom-up

    We will have 2 choices everytime we add a new item (i) from the menu:
    use it to get to the desired ammount (p) or not

    (1) useIt:
    if we choose to use it, then we check if p - cost[i] >= 0, so it makes sense and stays in the bounds.
    if it is, then the cost of using i is precomputed at cache[i][p-cost[i-1]].

    (2) dontUseIt:
    if we choose not to use it, we have to look up to the precalculated
    ways of getting to the value without the current item, which value is
    located right above (cache[i-1][p])


    finally, the total ammount of ways is given by adding up the possible ways of both cases.
    */
    long i, p, dontUseIt, useIt;
    for(i = 0; i<= n; i++){

        for(p = 0; p <= P; p++){
            /*
            Base cases:
            if we have 0 items in the menu, we CAN'T satisfy ANY order

            if the order's value is 0, there's only ONE way to satisfy that ecuation:
            not asking for anything in the menu.
            */
            if(i == 0){
                cache[i][p] = 0;
            } else if (p == 0){
                cache[i][p] = 1;
            } else {
                // We follow the rules from the beginning
                dontUseIt = cache[i-1][p];
                useIt = 0;
                if (p >= cost[i-1]) useIt = cache[i][p-cost[i-1]];
                cache[i][p] = useIt + dontUseIt;
            }
        }
    }
    return cache[n][P]; // Optional return of the maximum solution
}


vector<long> getItemsFromCombination(vector<long> &cost, long P, table& cache){
    long n = cost.size();
    long i, p, dontUseIt, useIt;
    p = P;
    vector<long> solution;
    // We need to reconstruct the menu items which make up the solution at cache[n][P]
    for(i = n; i > 0; i--){
        // If we include cost[i-1] in the solution, can we still get a possible combination?
        if (p >= cost[i-1] && cache[i][p-cost[i-1]] != 0){
            // Yes if there's at least one way to use the item (i-1)
            solution.push_back(i);
            p = p-cost[i-1];
```

```
            i++; // We need to check for repetition so we stay in the same row

        }
        // If we dont get a possible combination including cost[i-1], then we simply
            dont add it to the solution
        // and go to the next combination that doesn't include the current item
            (going one row up).
        if (p == 0){
            break; // Safety check
        }
    }
    return solution;
}
```

8.4. Ejemplo Bellman y ciclos negativos

```
struct Edge{
    int from;
    int to;
    int cost;
};

bool NegativeCicleBellmanFord(int V, vector<Edge>& edges, int source, vector<int>
    &minDist, vector<bool>& reachableByNC){
    minDist = vector<int>(V, INT_MAX);
    reachableByNC = vector<bool>(V, false);
    minDist[source] = 0;

    // Primera fase: encontrar las distancias mas cortas
    for (int i = 0; i < V-1; i++){
        for (Edge edge : edges){
            if (minDist[edge.from] != INT_MAX &&
                minDist[edge.from] + edge.cost < minDist[edge.to]){
                minDist[edge.to] = minDist[edge.from] + edge.cost;
            }
        }
    }

    // Segunda fase: detectar y propagar ciclos negativos
    queue<int> q;
    vector<bool> inQueue(V, false);

    // Identificar nodos iniciales con ciclos negativos
    for (Edge edge : edges){
        if (minDist[edge.from] != INT_MAX &&
            minDist[edge.from] + edge.cost < minDist[edge.to]){
            if (!inQueue[edge.to]){
                q.push(edge.to);
                inQueue[edge.to] = true;
                reachableByNC[edge.to] = true;
            }
        }
    }

    if(inQueue.empty()) return false; // No hay elementos en los ciclos negativos

    // Propagar el efecto de los ciclos negativos
    while (!q.empty()){
        int v = q.front();
        q.pop();
        inQueue[v] = false;

        for (Edge edge : edges){
            if (edge.from == v && !reachableByNC[edge.to]){
                reachableByNC[edge.to] = true;
```

```
                if (!inQueue[edge.to]){
                    q.push(edge.to);
                    inQueue[edge.to] = true;
                }
            }
        }
    }

    return true; // Llegamos hasta aca solo si hay ciclos negativos
}
```

9. Guía de Complejidades por Tamaño de Entrada

Tamaño (n)	Complejidad Máxima	Algoritmos Posibles
$n \leq 10$	$O(n!)$ o $O(2^n)$	Backtracking, Fuerza Bruta
$n \leq 20$	$O(2^n)$	DP con bitmask, Backtracking
$n \leq 100$	$O(n^3)$	Floyd-Warshall, DP
$n \leq 1000$	$O(n^2)$	DP simple, Dijkstra denso
$n \leq 10^5$	$O(n \log n)$	Sorting, Dijkstra sparse, Greedy
$n \leq 10^6$	$O(n)$	Sliding Window
$n \leq 10^9$	$O(\log n)$	Binary Search

Cuadro 1: Guía de complejidades aceptables según tamaño de entrada

9.1. Rangos de Tipos de Datos

Tipo	32-bits (bytes)	64-bits (bytes)	Rango	Potencia de 10
short int	2	2	$[-2^{15}, 2^{15}-1]$	$\pm 10^4$
unsigned short	2	2	$[0, 2^{16}-1]$	10^5
int	4	4	$[-2^{31}, 2^{31}-1]$	$\pm 10^9$
unsigned int	4	4	$[0, 2^{32}-1]$	10^{10}
long	4	8	32-bit: $[-2^{31}, 2^{31}-1]$ 64-bit: $[-2^{63}, 2^{63}-1]$	32-bit: $\pm 10^9$ 64-bit: $\pm 10^{18}$
unsigned long	4	8	32-bit: $[0, 2^{32}-1]$ 64-bit: $[0, 2^{64}-1]$	32-bit: 10^{10} 64-bit: 10^{20}
long long	8	8	$[-2^{63}, 2^{63}-1]$	$\pm 10^{18}$
unsigned long long	8	8	$[0, 2^{64}-1]$	10^{20}

Cuadro 2: Rangos y tamaños de tipos enteros en C++ (32-bits vs 64-bits)

Importante:

- El tipo `long` varía según la arquitectura del sistema
- Para portabilidad, preferir `long long` sobre `long`
- Para operaciones con valores grandes:
  - Suma: considerar que el resultado puede exceder el máximo
  - Multiplicación: revisar si el producto cabe en el tipo de dato
  - División: cuidado con división por cero y precisión en divisiones enteras

9.2. Notas Importantes

- Para $n \leq 10^8$: Operaciones simples por elemento
- Para grafos: Considerar tanto V (vértices) como E (aristas)
- En problemas de strings: Considerar tanto la longitud como el tamaño del alfabeto
- Para DP: Considerar el número total de estados posibles

10. EDDs y Algoritmos STL C++

Vector

```
#include <vector>
vector<T> vec;              // Declaraci n
vec.push_back(ele);         // Agrega al final O(1)
vec.pop_back();            // Elimina del final O(1)
vec.insert(vec.begin()+p,e);// Inserta en pos p O(n)
vec.erase(vec.begin()+p);   // Elimina de pos p O(n)
vec[i];                     // Acceso directo O(1)
vec.size();                 // Tama o O(1)
```

Stack y Queue

```
#include <stack>            // #include <queue>
stack<T> st;               // queue<T> qu;
st.push(ele);              // qu.push(ele);
st.pop();                  // qu.pop();
st.top();                  // qu.front();
st.size();                 // qu.size();
// Todas las operaciones O(1)
```

Deque

```
#include <deque>
deque<T> dq;
dq.push_front/back(ele);    // Insertar O(1)
dq.pop_front/back();        // Eliminar O(1)
dq.front()/back();          // Acceso O(1)
```

Set

```
#include <set>              // Ordenado
set<T> st;
st.insert(ele);             // O(log n)
st.erase(ele);             // O(log n)
st.find(k);                // O(log n)
st.lower/upper_bound(ele);  // O(log n)

// Unordered - O(1) promedio, O(n) peor caso
#include <unordered_set>
unordered_set<T> ust;
```

Map

```
#include <map>              // Ordenado
map<T1,T2> mp;
mp.insert({k,v});          // O(log n)
mp.erase(k);               // O(log n)
mp.find(k);                // O(log n)
mp.lower/upper_bound(k);    // O(log n)
iterator->first/second;     // Acceso key/value

// Unordered - O(1) promedio, O(n) peor caso
#include <unordered_map>
unordered_map<T1,T2> ump;
```

Priority Queue

```cpp
#include <queue>
priority_queue<T> pq;        // Max heap
priority_queue<T,vector<T>,greater<T>> pq2;// Min heap
pq.push(ele);               // O(log n)
pq.pop();                   // O(1)
pq.top();                   // O(1)
```

Algoritmos

```cpp
#include <algorithm>
// Sort - O(n log n)
sort(vec.begin(), vec.end());
sort(vec.begin(), vec.end(), comp);
sort(arr, arr+n);

// Binary Search - O(log n)
lower_bound(vec.begin(), vec.end(), v);
upper_bound(vec.begin(), vec.end(), v);
```