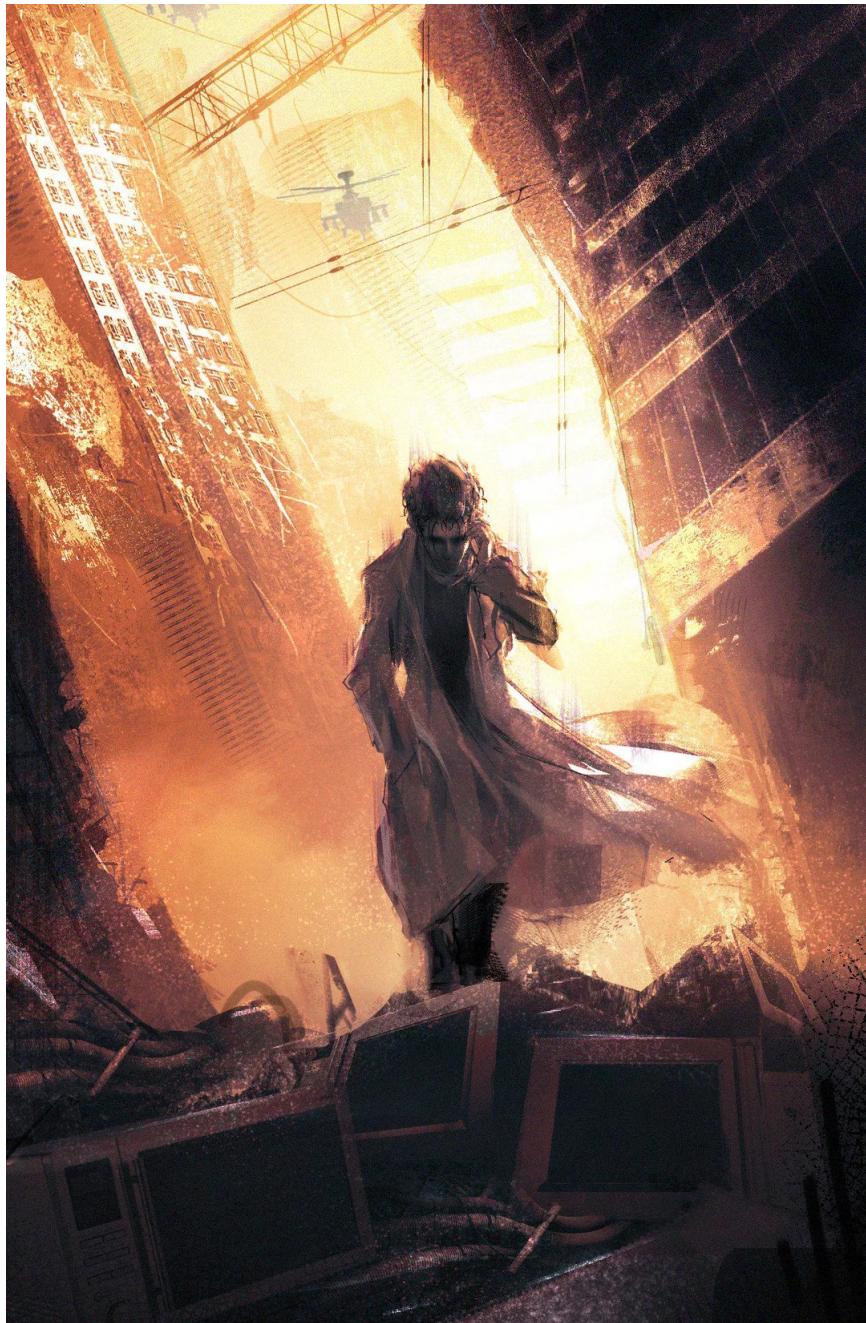


Resumen Algoritmos



“... En medio del odio descubrí que había, dentro de mí, un amor invencible. En medio de las lágrimas descubrí que había, dentro de mí, una sonrisa invencible. En medio del caos descubrí que había, dentro de mí, una calma invencible. Me di cuenta a pesar de todo eso... En medio del invierno descubrí que había, dentro de mí, un verano invencible. Y eso me hace feliz. Porque esto dice que no importa lo duro que el mundo empuja contra mí; en mi interior hay algo más fuerte, algo mejor, empujando de vuelta.”

— Viktor T.

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 → cantidad de llamadas recursivas
- b : factor de división del problema. **solo se puede dividir simétricamente.**
- $f(n)$ o 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

```
1 struct Activity {
2     int start, finish;
3 };
4
5 vector<Activity> activitySelection(vector<Activity>& acts) {
6     sort(acts.begin(), acts.end(),
7          [] (Activity a, Activity b) {
8              return a.finish < b.finish;
9          });
10
11    vector<Activity> result;
12    result.push_back(acts[0]);
13
14    int lastFinish = acts[0].finish;
15    for(int i = 1; i < acts.size(); i++) {
16        if(acts[i].start >= lastFinish) {
17            result.push_back(acts[i]);
18            lastFinish = acts[i].finish;
19        }
20    }
21    return result;
22 }
```

3. Programación Dinámica

3.1. Características

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

3.2. Ejemplo: Fibonacci con DP

```
1 // Bottom-up approach
2 int fib(int n) {
3     vector<int> dp(n + 1);
4     dp[0] = 0; dp[1] = 1;
5
6     for(int i = 2; i <= n; i++)
7         dp[i] = dp[i-1] + dp[i-2];
8
9     return dp[n];
10}
11
12 // Optimización de espacio
13 int fibOptimized(int n) {
14     int prev = 0, curr = 1;
15     for(int i = 2; i <= n; i++) {
16         int next = prev + curr;
17         prev = curr;
18         curr = next;
19     }
20     return curr;
21}
```

4. Algoritmos de Grafos

4.1. Dijkstra

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

```
1 vector<int> dijkstra(vector<vector<pair<int,int>>> &adj,
2                         int V, int src) {
3     vector<int> dist(V, INT_MAX);
4     priority_queue<pair<int,int>,
5                     vector<pair<int,int>>,
6                     greater<pair<int,int>>> pq;
7
8     dist[src] = 0;
9     pq.push({0, src});
10
11    while(!pq.empty()) {
12        int u = pq.top().second;
13        pq.pop();
14
15        for(auto& [v, weight] : adj[u]) {
16            if(dist[v] > dist[u] + weight) {
17                dist[v] = dist[u] + weight;
18                pq.push({dist[v], v});
19            }
20        }
21    }
22    return dist;
23}
```

4.2. Bellman-Ford

Para detectar ciclos negativos y encontrar caminos más cortos.

```
1 vector<int> bellmanFord(int V, int src,
2     vector<vector<int>>& edges) {
3     vector<int> dist(V, INT_MAX);
4     dist[src] = 0;
5
6     // Relajación V-1 veces
7     for(int i = 1; i <= V-1; i++) {
8         for(auto& edge : edges) {
9             int u = edge[0];
10            int v = edge[1];
11            int weight = edge[2];
12            if(dist[u] != INT_MAX &&
13                dist[u] + weight < dist[v])
14                dist[v] = dist[u] + weight;
15        }
16    }
17
18    // Detectar ciclo negativo
19    for(auto& edge : edges) {
20        int u = edge[0];
21        int v = edge[1];
22        int weight = edge[2];
23        if(dist[u] != INT_MAX &&
24            dist[u] + weight < dist[v])
25            return {}; // Hay ciclo negativo
26    }
27    return dist;
28 }
```

4.3. Prim

Para árbol de expansión mínima.

```
1 vector<int> prim(vector<vector<pair<int,int>>>& adj,
2                     int V) {
3     vector<bool> visited(V, false);
4     vector<int> parent(V, -1);
5     vector<int> key(V, INT_MAX);
6
7     priority_queue<pair<int,int>,
8                     vector<pair<int,int>>,
9                     greater<pair<int,int>>> pq;
10
11    key[0] = 0;
12    pq.push({0, 0});
13
14    while(!pq.empty()) {
15        int u = pq.top().second;
16        pq.pop();
17        visited[u] = true;
18
19        for(auto& [v, weight] : adj[u]) {
20            if(!visited[v] && weight < key[v]) {
21                parent[v] = u;
22                key[v] = weight;
23                pq.push({key[v], v});
24            }
25        }
26    }
27    return parent;
28 }
```

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

```
1 int maxSumSubarray(vector<int>& arr, int k) {
2     int n = arr.size();
3     if(n < k) return 0;
4
5     int maxSum = 0;
6     // Primera ventana
7     for(int i = 0; i < k; i++)
8         maxSum += arr[i];
9
10    int windowSum = maxSum;
11    // Deslizar ventana
12    for(int i = k; i < n; i++) {
13        windowSum = windowSum - arr[i-k] + arr[i];
14        maxSum = max(maxSum, windowSum);
15    }
16    return maxSum;
17 }
```

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

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

8.2. Ejemplo Prim

```
1 ll prim(ll s){
2     ll totalSum = 0;
3     priority_queue<pair<ll, ll>, vector<pair<ll, ll>>, greater<pair<ll, ll>>> pq;
4     vector<bool> visited = vector<bool> (n+1, false);
5     pq.push({0, s});
6
7     while (!pq.empty()){
8         auto edge = pq.top();
9         pq.pop();
10        ll src = edge.second;
11        if (visited[src]){
12            continue;
13        }
14        totalSum += edge.first;
15        visited[src] = true;
16        visitedCount++;
17        if (visitedCount >= n){
18            break;
19        }
20
21        for (auto edge : adjList[src]){
22            ll cost = edge.first;
23            ll to = edge.second;
24            if (!visited[to]){
25                pq.push({cost, to});
26            }
27        }
28    }
29    return totalSum;
30}
```

8.3. Ejemplo DP

```
1 long cacheLoader(vector<long> &cost, long P, table& cache){
2
3     long n = cost.size();
4     /*
5      We have have a n*P cache table and we're filling it with bottom-up
6
7      We will have 2 choices everytime we add a new item (i) from the menu:
8      use it to get to the desired amount (p) or not
9
10     (1) useIt:
11     if we choose to use it, then we check if p - cost[i] >= 0, so it makes sense and
12     stays in the bounds.
13     if it is, then the cost of using i is precomputed at cache[i][p-cost[i-1]].
14
15     (2) dontUseIt:
16     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
```

```

17     located right above (cache[i-1][p])
18
19
20     finally, the total amount of ways is given by adding up the possible ways of
21     both cases.
22 */
23 long i, p, dontUseIt, useIt;
24 for(i = 0; i<= n; i++){
25
26     for(p = 0; p <= P; p++){
27         /*
28         Base cases:
29         if we have 0 items in the menu, we CAN'T satisfy ANY order
30
31         if the order's value is 0, there's only ONE way to satisfy that equation:
32         not asking for anything in the menu.
33         */
34         if(i == 0){
35             cache[i][p] = 0;
36         } else if (p == 0){
37             cache[i][p] = 1;
38         } else {
39             // We follow the rules from the beginning
40             dontUseIt = cache[i-1][p];
41             useIt = 0;
42             if (p >= cost[i-1]) useIt = cache[i][p-cost[i-1]];
43             cache[i][p] = useIt + dontUseIt;
44         }
45     }
46     return cache[n][P]; // Optional return of the maximum solution
47 }
48
49
50 vector<long> getItemsFromCombination(vector<long> &cost, long P, table& cache){
51     long n = cost.size();
52     long i, p, dontUseIt, useIt;
53     p = P;
54     vector<long> solution;
55     // We need to reconstruct the menu items which make up the solution at
56     // cache[n][P]
57     for(i = n; i > 0; i--){
58         // If we include cost[i-1] in the solution, can we still get a possible
59         // combination?
60         if (p >= cost[i-1] && cache[i][p-cost[i-1]] != 0){
61             // Yes if there's at least one way to use the item (i-1)
62             solution.push_back(i);
63             p = p-cost[i-1];
64             i++; // We need to check for repetition so we stay in the same row
65         }
66         // If we don't get a possible combination including cost[i-1], then we simply
67         // don't add it to the solution
68         // and go to the next combination that doesn't include the current item
69         // (going one row up).
70         if (p == 0){
71             break; // Safety check
72         }
73     }
74     return solution;
75 }
```

8.4. Ejemplo Bellman y ciclos negativos

```
1 struct Edge{
```

```

2     int from;
3     int to;
4     int cost;
5 };
6
7 bool NegativeCicleBellmanFord(int V, vector<Edge>& edges, int source, vector<int>
8   &minDist, vector<bool>& reachableByNC){
9   minDist = vector<int>(V, INT_MAX);
10  reachableByNC = vector<bool>(V, false);
11  minDist[source] = 0;
12
13 // Primera fase: encontrar las distancias mas cortas
14 for (int i = 0; i < V-1; i++){
15   for (Edge edge : edges){
16     if (minDist[edge.from] != INT_MAX &&
17       minDist[edge.from] + edge.cost < minDist[edge.to]){
18       minDist[edge.to] = minDist[edge.from] + edge.cost;
19     }
20   }
21
22 // Segunda fase: detectar y propagar ciclos negativos
23 queue<int> q;
24 vector<bool> inQueue(V, false);
25
26 // Identificar nodos iniciales con ciclos negativos
27 for (Edge edge : edges){
28   if (minDist[edge.from] != INT_MAX &&
29     minDist[edge.from] + edge.cost < minDist[edge.to]){
30     if (!inQueue[edge.to]){
31       q.push(edge.to);
32       inQueue[edge.to] = true;
33       reachableByNC[edge.to] = true;
34     }
35   }
36 }
37
38 if(inQueue.empty()) return false; // No hay elementos en los ciclos negativos
39
40 // Propagar el efecto de los ciclos negativos
41 while (!q.empty()){
42   int v = q.front();
43   q.pop();
44   inQueue[v] = false;
45
46   for (Edge edge : edges){
47     if (edge.from == v && !reachableByNC[edge.to]){
48       reachableByNC[edge.to] = true;
49       if (!inQueue[edge.to]){
50         q.push(edge.to);
51         inQueue[edge.to] = true;
52       }
53     }
54   }
55 }
56
57 return true; // Llegamos hasta aca solo si hay ciclos negativos
58 }

```

9. Guía de Complejidades por Tamaño de Entrada

9.1. Rangos de Tipos de Datos

Importante:

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

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)

- 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

```

1 #include <vector>
2 vector<T> vec;           // Declaración
3 vec.push_back(ele);       // Agrega al final O(1)
4 vec.pop_back();           // Elimina del final O(1)
5 vec.insert(vec.begin() + p, e); // Inserta en pos p O(n)
6 vec.erase(vec.begin() + p); // Elimina de pos p O(n)
7 vec[i];                  // Acceso directo O(1)
8 vec.size();               // Tamaño O(1)

```

Stack y Queue

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

Deque

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

Set

```
1 #include <set>           // Ordenado
2 set<T> st;
3 st.insert(ele);          // O(log n)
4 st.erase(ele);           // O(log n)
5 st.find(k);              // O(log n)
6 st.lower/upper_bound(ele); // O(log n)
7
8 // Unordered - O(1) promedio, O(n) peor caso
9 #include <unordered_set>
10 unordered_set<T> ust;
```

Map

```
1 #include <map>           // Ordenado
2 map<T1,T2> mp;
3 mp.insert({k,v});        // O(log n)
4 mp.erase(k);             // O(log n)
5 mp.find(k);              // O(log n)
6 mp.lower/upper_bound(k); // O(log n)
7 iterator->first/second; // Acceso key/value
8
9 // Unordered - O(1) promedio, O(n) peor caso
10 #include <unordered_map>
11 unordered_map<T1,T2> ump;
```

Priority Queue

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

Algoritmos

```
1 #include <algorithm>
2 // Sort -  $O(n \log n)$ 
3 sort(vec.begin(), vec.end());
4 sort(vec.begin(), vec.end(), comp);
5 sort(arr, arr+n);
6
7 // Binary Search -  $O(\log n)$ 
8 lower_bound(vec.begin(), vec.end(), v);
9 upper_bound(vec.begin(), vec.end(), v);
```