

Backtracking

Ese que **NO** es fuerza bruta

Agustín Santiago Gutiérrez

Universidad Nacional de Buenos Aires - FCEN

Training Camp Argentina 2024



Gracias Sponsors!

Organizador



Diamond



Gold



- 1 Backtracking vs Brute Force
- 2 Principio “Generar, no filtrar”
- 3 Performance
- 4 Filosófico
- 5 Máximo conjunto independiente
- 6 Problema de regional reciente
- 7 Branch and Bound

- 1 Backtracking vs Brute Force
- 2 Principio “Generar, no filtrar”
- 3 Performance
- 4 Filosófico
- 5 Máximo conjunto independiente
- 6 Problema de regional reciente
- 7 Branch and Bound

Brute Force

Fuerza bruta es:

- Un método de búsqueda exhaustiva
- Para cada solución... (de alguna forma iterarlas)
- Ver si anda / evaluarla / procesarla

Ejemplos típicos:

- ```
do { eval(v); }
 while (next_permutation(begin(v), end(v)));
```
- ```
for (int i = 1; i <= N; i++) best = max(best, eval(i));
```
- ```
for (int i = 1; i <= N; i++)
 for (int j = 1; j <= N; j++)
 for (int k = 1; k <= N; k++)
 if (cuentaLoca(i,j,k))
 ret++;
```

Backtracking es:

- **Proceso incremental de construcción** de posibles soluciones
- Es decir **paso a paso**
- En cada paso **evaluamos**, no solamente al final

# for for if if vs for if for if

Notar que la definición de backtracking nunca habla de recursividad.

- // FUERZA BRUTA

```
for (int i = 1; i <= N; i++)
for (int j = 1; j <= N; j++)
for (int k = 1; k <= N; k++)
if (cuentaLoca(i,j,k))
 ret++;
```

- // BACKTRACKING

```
for (int i = 1; i <= N; i++)
if (okI(i))
for (int j = 1; j <= N; j++)
if (okJ(i,j))
for (int k = 1; k <= N; k++)
if (okK(i,j,k))
 ret++;
```

# Ejemplo: prefi-primos

- Un número es prefi-primo si todos sus prefijos son primos
- 233 es prefi-primo porque 2, 23 y 233 son primos
- 251 es primo, pero no prefi-primo
- Sea  $f_n$  = cantidad de prefi-primos de  $n$  dígitos

¿Cómo computamos  $f_n$  y hallamos prefi-primos eficientemente?



# Opción fuerza bruta

- ```
for (int a = 1; a <= 9; a++)  
  for (int b = 0; b <= 9; b++)  
    for (int c = 0; c <= 9; c++)  
      for (int d = 0; d <= 9; d++)  
        if (primo(a) && primo(10*a+b) && primo(100*a+10*b+c) &&  
            primo(1000*a+100*b+10*c+d) )  
          ret++;
```
- Asumiendo por simplicidad que el check de primo es $O(1)$, computar f_n toma tiempo $O(n \cdot 10^n)$

Opción backtracking

- ```
for (int a = 1; a <= 9; a++)
 if (primo(a))
 for (int b = 0; b <= 9; b++)
 if (primo(10*a+b))
 for (int c = 0; c <= 9; c++)
 if (primo(100*a+10*b+c))
 for (int d = 0; d <= 9; d++)
 if (primo(1000*a+100*b+10*c+d))
 ret++;
```
- Complejidad  $O(n(f_1 + f_2 + \dots + f_n))$

# Hay **MUY** pocos prefri-primos

$$f(1) = 4$$

$$f(2) = 9$$

$$f(3) = 14$$

$$f(4) = 16$$

$$f(5) = 15$$

$$f(6) = 12$$

$$f(7) = 8$$

$$f(8) = 5$$

$$f(x \geq 9) = 0$$

# Mejora de complejidad por eval incremental

- El ejemplo anterior es extremo, pasa de  $10^8$  operaciones a 664
- Aun así en muchísimos otros ejemplos también se reduce la complejidad asintótica
- Incluso si no se poda casi nada, se suele ahorrar el costo de hacer “un solo eval caro al final”

# Mejora por eval incremental (prefi-primos)

- Pasa con los prefi-primos: se va el factor  $O(n)$  gracias a ir acumulando los números
- ```
for (int a = 1; a <= 9; a++)  
  if (primo(a))  
    for (int b = 0; b <= 9; b++)  
      if (primo(ab = 10*a+b))  
        for (int c = 0; c <= 9; c++)  
          if (primo(abc = 10*ab+c))  
            for (int d = 0; d <= 9; d++)  
              if (primo(abcd = 10*abc+d))  
                ret++;
```
- Complejidad $O(f_1 + f_2 + \dots + f_n)$

Mejora por eval incremental (permutaciones)

Otra ejemplo clásico donde se evita un factor $O(N)$ del eval final

- ```
do { process(accumulate(begin(v), end(v), 0)); }
 while (next_permutation(begin(v), end(v))); // $O(N * N!)$
```
- ```
void go(i) { //  $O(N!)$   
    if (i == n) {  
        process(suma);  
        return;  
    }  
    for (int j = i; j < n; j++) {  
        swap(v[i], v[j]);  
        suma += v[i];  
        go(i+1);  
        suma -= v[i];  
        swap(v[i], v[j]);  
    }  
}
```

- 1 Backtracking vs Brute Force
- 2 Principio “Generar, no filtrar”
- 3 Performance
- 4 Filosófico
- 5 Máximo conjunto independiente
- 6 Problema de regional reciente
- 7 Branch and Bound

Ejemplos de “generar, no filtrar”

- Buscando números que deben cumplir ser múltiplos de x
 - `for (int i = 1; i <= N; i++) if (i % x == 0) print(x);`
 - `for (int i = x; i <= N; i += x) print(x);`

Ejemplos de “generar, no filtrar” (cont)

- Números con solo 2 y 3 como factores primos
 - ```
for (int i = 1; i <= N; i++)
 if (soloDosTres(x))
 print(x);
```
  - ```
set<int> pending; pending.insert(1);  
while (true) {  
    int x = *begin(pending);  
    if (x > N) break;  
    pending.erase(begin(pending));  
    pending.insert(2*x);  
    pending.insert(3*x);  
}
```

Generar de forma única

- Mejor que filtrar es generar
- Mejor que generar es **generar de forma única**
- En el ejemplo anterior era necesario el set para eliminar duplicados
- Regla: $x \rightarrow 3x$ solo si x es impar

- ```
vector<int> pending; pending.push_back(1);
while (true) {
 int x = pending.back();
 pending.pop_back();
 if (x <= N) {
 pending.push_back(2*x)
 if (x % 2 != 0) pending.push_back(3*x);
 }
}
```

# Generar de forma única: estudio caso real

- En un problema de icpc reciente en latinoamerica, había que determinar si un número  $n$  dado era producto de fibonaccis o no
- Muchos equipos obtenían TLE durante la prueba por no hacer ni dp (eliminar duplicados) ni asegurarse de generar de forma única

- ① Backtracking vs Brute Force
- ② Principio “Generar, no filtrar”
- ③ Performance
- ④ Filosófico
- ⑤ Máximo conjunto independiente
- ⑥ Problema de regional reciente
- ⑦ Branch and Bound

# Consejos para Backtracking eficiente

- Usar bitmasks/bitsets y operaciones de bits
  - Matriz de adyacencia = un bitmask por nodo
  - Apagar/prender vecinos = ands y ors de máscaras
  - Contar grado restante = popcount de máscaras
  - `__builtin_clz` para ir al primer nodo “disponible”
- Evitar a toda costa reservas de memoria dinámica (nunca tener un vector local)
- Modificar in-place un estado global en lugar de copiar
  - Aunque empezar copiando para hacer más eficiente luego es muy buena práctica
  - Utilizar una clase struct `State` para copiar fácil y abstraer operaciones
- Truco: contar, no flaggear
- Precomputar mucho para alivianar al máximo la parte exponencial

- 1 Backtracking vs Brute Force
- 2 Principio “Generar, no filtrar”
- 3 Performance
- 4 Filosófico
- 5 Máximo conjunto independiente
- 6 Problema de regional reciente
- 7 Branch and Bound

# Qué es un buen problema de backtracking

- Buen problema de backtracking:
  - Se puede demostrar que el backtracking en concreto corre suficientemente rápido, o bien
  - Se puede probar correrlo localmente razonablemente rápido para toda entrada relevante, y saber con eso si entra o no, sin depender de los casos de prueba
- Mal problema de backtracking:
  - “Que hagan un backtracking con tales optimizaciones, eso pasa los casos de prueba”
  - No sabemos como autores o no queda claro, si no existirá un caso de prueba alternativo donde no alcanzan esas optimizaciones, y capaz que el participante tiene una solución mejor enfocada para esos casos más difíciles pero que no están en el input

- 1 Backtracking vs Brute Force
- 2 Principio “Generar, no filtrar”
- 3 Performance
- 4 Filosófico
- 5 Máximo conjunto independiente**
- 6 Problema de regional reciente
- 7 Branch and Bound



- Fuerza bruta:  $O(2^n \cdot n^2)$
- Backtracking directo:  $O(2^n \cdot n)$
- Backtracking eficiente con bits:  $O(2^n)$

# Poda: nodos aislados

- Si el nodo es aislado (grado 0), no considerar no elegirlo
- $O(\phi^n) = O(1.618^n)$

# Poda: nodos de grado 1

- Si el nodo tiene grado  $\leq 1$ , no considerarlo no elegirlo
- $O(1.466^n)$

## Poda: nodos de grado 2

- Si todos los nodos son de grado 2, son ciclos simples: resolver ad-hoc
- Ahora cada vez que el backtracking se ramifica, lo hace con un nodo de grado al menos 3
- $O(1.38^n)$
- Programando esto eficientemente, llegamos a resolver sin problemas cualquier caso de  $n = 42$
- Le gana al combo “DP con máscaras y meet in the middle” que sería  $O(1.41^n \cdot n)$  y es conceptualmente mucho más complicado

- 1 Backtracking vs Brute Force
- 2 Principio “Generar, no filtrar”
- 3 Performance
- 4 Filosófico
- 5 Máximo conjunto independiente
- 6 Problema de regional reciente**
- 7 Branch and Bound

# Almost Origami (Regional 2020)

- Se reduce a mínimo número de productos para formar  $x^n$  a partir de  $x$
- Eso no se sabe resolver en tiempo polinomial en  $n$
- $n \leq 100$
- Solamente 100 inputs, se puede testear localmente la eficiencia
- También se puede correr un backtracking local suficientemente podado y hardcodear los 100 valores
- Posibilidad que ejecuta muy rápido: un BFS sobre el espacio de estados de un buen backtracking (computa los 100 valores a la vez)
- Lo resolvió un solo equipo de todo Latam: [UNAM/FC] PumOMM

- 1 Backtracking vs Brute Force
- 2 Principio “Generar, no filtrar”
- 3 Performance
- 4 Filosófico
- 5 Máximo conjunto independiente
- 6 Problema de regional reciente
- 7 Branch and Bound**

- En cada nodo del árbol de backtracking se computan cotas inferior y superior  $L, U$  (con  $L \leq U$ ) a las posibles soluciones desde ese nodo
- Se mantiene una variable global `best` con el valor de la mejor solución encontrada
- Para un problema de minimización, `best` tiene en todo momento el mínimo  $U$  que hemos visto en todo el algoritmo
- Para un problema de minimización, **si  $L \geq \text{best}$  podemos descartar ese nodo y todo su subárbol**
- Lo esencial: meter un `if (L >= best) return;` al comienzo de la función recursiva
- Casi nunca usamos  $U$  en nodos no hoja en problemas tipo ICPC