Búsqueda exhaustiva y backtracking

Sebastian Mestre

¿Cuándo alcanza con fuerza bruta?

- Para problemas con entrada pequeña (por ejemplo $N < 20$) podemos probar **todas las posibilidades**.
- La idea es explorar el **espacio de soluciones completo** y quedarnos con la mejor / alguna válida.
- Aunque parezca simple, muchos equipos se complican al implementarlo y pierden puntos en problemas fáciles.
- Objetivo de la clase:
  - Ver formas estándar de implementar fuerza bruta eficientemente.
  - Aprender trucos para **escribir poco código** y evitar errores.

Fuerza bruta sobre subconjuntos

- Para un conjunto de n elementos, hay 2^n subconjuntos posibles.
- Podemos representar cada subconjunto con un entero de 0 a $2^n - 1$:
  - El bit i indica si el elemento i está presente en el subconjunto.
- Esto permite recorrer **todos los subconjuntos** con un for.

Recorriendo todos los subconjuntos

Esquema básico

```
1 // mk representa un subconjunto de {0,1,2,...,n-1}
2 forn(mk, 1 << n) {
3   forn(i, n) {
4     if (mk & (1 << i)) {
5       // i es elemento del subconjunto mk
6     }
7   }
8 }
```

- Costo: $O(2^n \cdot n)$, aceptable para $n \leq 20$ aprox.

Ejemplo: Apple Division (CSES 1623)

- Dado un arreglo, dividirlo en dos grupos con suma lo más parecida posible.
- Fuerza bruta: probamos todos los subconjuntos como **un lado** de la partición.

Implementación típica

```
ll best = LLONG_MAX;
forn(mk, 1 << n) {
  ll s = 0;
  forn(i, n) if (mk & (1 << i)) s += a[i];
  ll resto = total - s;
  ll diff = abs(s - resto);
  best = min(best, diff);
}
cout << best << endl;
```

Fuerza bruta sobre permutaciones

- A veces queremos probar todos los órdenes posibles de un arreglo.
- En C++ la función `std::next_permutation` genera la siguiente permutación lexicográfica.

Uso típico

```
sort(v.begin(), v.end());
do {
  // usar la permutaci n actual de v
} while (next_permutation(v.begin(), v.end()));
```

- Complejidad: $O(n! \cdot \text{costo de procesar una permutación})$.
- Usable para $n \leq 10$ aprox.

Ejemplo: Codeforces 431B

- Problema 431B - Shower Line.
- 5 estudiantes se ordenan en una fila para usar la ducha.
- Mientras esperan, conversan por pares (1 con 2, 3 con 4, el 5 queda solo).
- Cada vez que alguien entra, la fila se acorta y se vuelven a formar parejas.
- Cada conversación entre i y j suma felicidad según una matriz $g[i][j]$.
- Objetivo: encontrar el orden inicial que maximiza la felicidad total.

Solución por fuerza bruta

- Como $n = 5$, alcanza con probar las 5! permutaciones.
- Costo total $O(n! \cdot n^2)$, más que suficiente para este límite.

Implementación

```
int const n = 5;
int g[n][n], p[] = {0, 1, 2, 3, 4};
int main() {
  forn(i, n) forn(j, n) cin >> g[i][j];
  ll best = 0;
  do {
    ll happiness = 0;
    forn(i, n) for (int j = i+1; j < n; j += 2)
      happiness += g[p[j-1]][p[j]] + g[p[j]][p[j-1]];
    best = max(best, happiness);
  } while (next_permutation(p, p + n));
  cout << best << endl;
}
```

Más allá de la fuerza bruta pura

- Para n un poco más grande podemos intentar:
  - Optimizar el cálculo interno a $O(n)$ por permutación.
  - O usar DP sobre subconjuntos ($O(2^n \cdot n)$) cuando el problema lo permite.
- Idea general: **aprovechar estructura** del problema para reducir el espacio de búsqueda.

Fuerza bruta recursiva

- Otra forma de fuerza bruta es construir soluciones **paso a paso** recursivamente.
- Problema clásico: N reinas (Chessboard and Queens, CSES 1624).
- Queremos colocar N reinas en un tablero $N \times N$ sin que se ataquen entre sí.

Versión muy bruta

Recorriendo todas las configuraciones

```
1  int const maxn = 10;
2  int N; // tama o del tablero
3  bool ocupado[maxn * maxn];
4  bool cumple_condiciones(); // false si dos reinas se atacan entre si
5
6  bool reinas(int n) {
7    if (n == 0) return cumple_condiciones();
8    forn(p, N * N - (n - 1)) {
9      if (ocupado[p]) continue;
10     ocupado[p] = true;
11     if (reinas(n - 1)) return true;
12     ocupado[p] = false;
13   }
14   return false;
15 }
16
```

Evitando duplicados innecesarios

- La versión anterior considera diferentes órdenes para colocar reinas en las mismas casillas.
- Ejemplo: trata como distintos

- Pero ambas configuraciones tienen una reina en la casilla 0 y otra en la casilla 3: son equivalentes.
- Podemos imponer un **orden** para no repetir subconjuntos.

Reinas en orden creciente de posición

Forzando orden en las posiciones

```
1  bool reinas(int n, int last) {
2    if (n == 0) return cumple_condiciones();
3    forr(p, last + 1, N * N - (n - 1)) {
4      ocupado[p] = true;
5      if (reinas(n - 1, p)) return true;
6      ocupado[p] = false;
7    }
8    return false;
9  }
10
```

- Ahora cada conjunto de casillas se visita una sola vez.

Una reina por columna

- En una solución válida solo puede haber **una reina por columna**.
- Podemos colocar la reina n-ésima siempre en la columna $n - 1$.

Iterando por columnas

```
bool reinas(int n) {
  if (n == 0) return cumple_condiciones();
  forr(j, 0, N) {
    int p = (n-1) * N + j;
    ocupado[p] = true;
    if (reinas(n - 1)) return true;
    ocupado[p] = false;
  }
  return false;
}
```

- Solo con estas optimizaciones pasamos de horas a milisegundos para $N = 10$.

Backtracking: podar temprano

- El backtracking mejora la fuerza bruta recursiva descartando **ramas imposibles** lo antes posible.
- En N reinas, cuando colocamos una reina en una fila solo avanzamos si:
  - la columna no está ocupada,
  - la diagonal principal no está ocupada,
  - la diagonal secundaria no está ocupada.
- Si alguna falla, retrocedemos (*backtrack*) y probamos otra opción.

Chequeando la condición en cada paso

Versión ingenua de backtracking

```
1  bool reinas(int n) {
2    if (!cumple_condiciones()) return false;
3    if (n == 0) return true;
4    forr(j, 0, N) {
5      ocupado[(n - 1) * N + j] = true;
6      if (reinas(n - 1)) return true;
7      ocupado[(n - 1) * N + j] = false;
8    }
9    return false;
10 }
11
```

- Mejor que chequear solo al final, pero todavía podemos optimizar.

Marcando columnas y diagonales

- En lugar de revisar todo el tablero en cada paso, marcamos:
  - columnas ocupadas (`c[j]`),
  - diagonal principal (`d1`),
  - diagonal secundaria (`d2`).
- Si alguna está ocupada, ni siquiera hacemos la llamada recursiva.

Backtracking eficiente

```
1  bool reinas(int n) {
2    if (n == 0) return true;
3    forn(j, 0, N) {
4      int p = (n-1) * N + j;
5      if (c[j] || d1[n + j] || d2[n - j + N]) continue;
6      ocupado[p] = c[j] = d1[n + j] = d2[n - j + N] = true;
7      if (reinas(n - 1)) return true;
8      ocupado[p] = c[j] = d1[n + j] = d2[n - j + N] = false;
9    }
10   return false;
11 }
```

Idea de Branch & Bound

- Técnica para problemas de optimización combinatoria.
- **Branch**: dividimos el problema en subproblemas recursivos (como backtracking).
- **Bound**: calculamos una **cota superior** de la mejor solución posible en cada rama.
- Si la cota ya es peor que la mejor solución conocida, podamo la rama sin seguir explorando.

Ejemplo: tractores en una grilla

- Problema de Codeforces 143E.
- Tenemos una grilla $N \times M$ y queremos colocar la mayor cantidad de **tractores**.
- Cada tractor ocupa 5 casillas en forma de T:

```
###
 #
 #
```

- Se pueden rotar 90 grados.
- Hay que dar la cantidad máxima de tractores y una configuración que lo logre.

Ejemplo: tractores en una grilla

- backtracking donde vamos poniendo tractores hasta que no se pueda mas
- fijamos un orden
- en cada paso hay 5 opciones
  - poner tractor hacia abajo
  - poner tractor hacia arriba
  - poner tractor hacia la izquierda
  - poner tractor hacia la derecha
  - no poner tractor

Backtracking básico para tractores

Probar todas las formas de poner / no poner

```
int cnt = 0, board[maxn][maxn]; // manejado por ok(), poner(), sacar()
int go(int p) {
  if (p == n * m) return cnt;
  int const i = p/m, j = p%m;
  int ans = cnt;
  forn(r, 5) if (ok(i, j, r)) {
    poner(i, j, r);
    ans = max(ans, go(p + 1));
    sacar(i, j, r);
  }
  return ans;
}
```

Guardando la mejor configuración

- El enunciado pide también **recuperar** una configuración óptima.
- Guardamos el mejor tablero encontrado hasta ahora.

Versión con memoria de la mejor solución

```
1  int best = 0, best_board[maxn][maxn];
2  int go(int p) {
3
4    ...
5
6    if (ans > best) {
7      best = ans;
8      memcpy(best_board, board, sizeof(board));
9    }
10
11   return ans;
12 }
13
```

Podando con una cota superior

- Lo anterior funciona, pero el algoritmo es muy lento.
- Idea: si incluso llenando todas las casillas vacías no podemos superar `best`, no vale la pena seguir.

Branch & bound

```
1  int go(int p) {
2    if (p == n * m) return cnt;
3
4    int rem = 0;
5    forr(q, p, n * m) rem += (tablero[q / m][q % m] == 0);
6    int opt = cnt + rem / 5;
7    if (opt <= best) return opt; // no se puede mejorar
8
9    ...
10 }
11
```

Importancia del orden de búsqueda

- Mientras construimos soluciones recursivamente, el **orden** en que probamos opciones importa mucho.
- Un buen orden puede hacer que encontremos soluciones rápido y podamos podar más.
- En el problema anterior es importante que la opcion "no poner tractor"sea la ultima que probamos. Esto es asi la busqueda encuentra rapidamente soluciones con "bastantes"tractores y la idea branch&bound puede podar mucho
- Ejemplo clásico: el paseo de un caballo en un tablero de ajedrez (Knight's tour).

Heurística para Knight's tour

- Dado un tablero $N \times M$, queremos que un caballo visite todas las casillas exactamente una vez y vuelva al inicio.
- Backtracking básico es demasiado lento.
- Heurística: siempre movernos a la casilla con **menos vecinos libres**.
- Para $N = M = 6$:
  - backtracking básico tarda ~ 20 segundos;
  - con la heurística tarda $\sim 0{,}01$ segundos.
- Para $N = M = 8$, el backtracking básico es prácticamente imposible de correr.
- Con la heurística podemos llegar a $N = M = 200$ en milisegundos.

Orden aleatorio en N reinas

- Otro ejemplo: N reinas con backtracking ya optimizado.
- Si probamos siempre las columnas en orden creciente, a veces caemos en casos muy malos.
- Si probamos las columnas en orden **aleatorio**, muchas instancias se resuelven muchísimo más rápido.

Probar columnas en orden aleatorio

```
1  bool go(int i) {
2    if (i == n) return true;
3    int idx[n];
4    forn(j, n) idx[j] = j;
5    random_shuffle(idx, idx + n);
6    for (int j : idx) {
7      ...
8    }
9    return false;
10 }
```

Resumen

- La **búsqueda exhaustiva** (fuerza bruta) es simple, **si se implementa bien**.
- Técnicas clave:
  - bitmasks para subconjuntos,
  - `next_permutation` para permutaciones,
  - recursión con buenas representaciones del espacio de estados,
  - backtracking y branch & bound para podar muchas ramas,
  - elegir bien el **orden de búsqueda**.

Preguntas

Q&A