Soluciones OIFem II Nivel 2

Entrenamiento 2

Subsets (1)

Solución

Como vimos en clase, mantenemos un vector con los índices que cogemos para el subonjunto y vamos explorando todas las posibilidades (coger o no coger cada elemento).

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;
using vvi = vector<vi>;

int n;
vector<string> v;
vi subset;

void bt(int i) {
    if (i == n) {
        int k = subset.size();
        cout << "{";
        for (int j = 0; j < k; ++j) {
            if (j > 0) cout << ",";
            cout << v[subset[j]];
        }
        cout << "}" << endl;
    }
    else {
        subset.push_back(i);
        bt(i+1);
        subset.pop_back();
        bt(i+1);
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    cin >> n;
    v = vector<string>(n);
```

```
35      for (int i = 0; i < n; ++i) {
36          cin >> v[i];
37      }
38      bt(0);
39  }
```

From one to en (1)

Solución

Como vimos en clase, mantenemos qué elementos hemos usado y la permutación generada y exploramos todas las opciones (coger cualquiera de los no usados en cada paso).

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;
using vvi = vector<vi>;

int n;
vi perm;

void bt() {
    if ((int)perm.size() == n) {
        cout << "(";
        for (int i = 0; i < n; ++i) {
            if (i > 0) cout << ",";
            cout << perm[i];
        }
        cout << ")" << endl;
    }
    else {
        for (int i = 1; i <= n; ++i) {
            int k = perm.size();
            bool avaliable = true;
            for (int j = 0; j < k; ++j) {
                if (perm[j] == i) avaliable = false;
            }
            if (avaliable) {
                perm.push_back(i);
                bt();
                perm.pop_back();
            }
        }
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cin >> n;
    bt();
}
```

Balance beam (1)

Solución

Como vimos en clase, vamos a cada paso probando las únicas dos opciones (salto a la izquierda / derecha).

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;
using vvi = vector<vi>;

int m, n;
vi l;

void bt(int pos, int i) {
    if (i == n) {
        cout << pos << endl;
    }
    else {
        if (pos + l[i] <= m/2)
            bt(pos + l[i], i+1);
        if (pos - l[i] >= -m/2)
            bt(pos - l[i], i+1);
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    cin >> m >> n;
    l = vi(n);
    for (int i = 0; i < n; ++i) {
        cin >> l[i];
    }
    bt(0, 0);
}
```

Més Sudokus

Solución

En clase lo explicamos con detalle. La idea es ir probando cada elemento de cada fila y mirar de forma eficiente que no ponemos un número inválido.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;
using vvi = vector<vi>;
using vvvi = vector<vvi>;

vvi s;
bool solved;
vvi usado_fila, usado_col;
vvvi usado_grupo;

void print() {
    solved = true;
    for (int i = 0; i < 9; ++i) {
        if (i == 3 or i == 6)
            cout << "------+-------+------" << endl;
        for (int j = 0; j < 9; ++j) {
            if (j == 3 or j == 6)
                cout << " |";
            if (j > 0) cout << " ";
            cout << s[i][j];
        }
        cout << endl;
    }
    cout << endl;
}

void bt(int i, int j) {
    if (j == 9) {
        if (i == 8) {
            print();
        }
        else {
            bt(i+1, 0);
        }
    }
    else {
        if (s[i][j] == 0) {
            for (int k = 1; k <= 9; ++k) {
                bool ok = (not usado_fila[i][k])
                    and   (not usado_col[j][k])
                    and   (not usado_grupo[i/3][j/3][k]);
                if (ok) {
                    usado_fila[i][k] = true;
                    usado_col[j][k] = true;
                    usado_grupo[i/3][j/3][k] = true;
                    s[i][j] = k;
                    bt(i, j+1);
```

```
                    s[i][j] = 0;
                    usado_fila[i][k] = false;
                    usado_col[j][k] = false;
                    usado_grupo[i/3][j/3][k] = false;
                }
            }
        }
        else {
            bt(i, j+1);
        }
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int t;
    cin >> t;
    while (t--) {
        string basura;
        s = vvi(9, vi(9));
        usado_fila = usado_col = vvi(9, vi(10, false));
        usado_grupo = vvvi(3, vvi(3, vi(10, false)));
        for (int i = 0; i < 9; ++i) {
            if (i == 3 or i == 6) cin >> basura;
            for (int j = 0; j < 9; ++j) {
                if (j == 3 or j == 6) cin >> basura;
                cin >> s[i][j];
                if (s[i][j] != 0) {
                    usado_fila[i][s[i][j]] = true;
                    usado_col[j][s[i][j]] = true;
                    usado_grupo[i/3][j/3][s[i][j]] = true;
                }
            }
        }
        solved = false;
        bt(0, 0);
        if (not solved) cout << "no solution" << endl;
        cout << "******************************" << endl;
    }
}
```

Equal sums (3)

Solución

Como vimos en clase, gracias a que los números son naturales podemos establecer condiciones (poda en backtracking) para dejar de explorar un camino que no lleva a nada.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;
using vvi = vector<vi>;
using vvvi = vector<vvi>;

int s;
int n;
vi x;
vi subset;
vi sum_derecha;

void bt(int i, int sum, bool last) {
    if (last and s == sum) {
        cout << "{";
        int k = subset.size();
        for (int j = 0; j < k; ++j) {
            if (j > 0) cout << ",";
            cout << x[subset[j]];
        }
        cout << "}" << endl;
    }
    if (i < n and sum + sum_derecha[i] >= s) {
        subset.push_back(i);
        if (sum+x[i] <= s) bt(i+1, sum+x[i], true);
        subset.pop_back();
        bt(i+1, sum, false);
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    cin >> s >> n;
    x = vi(n);
    for (int i = 0; i < n; ++i) {
        cin >> x[i];
    }

    sum_derecha = vi(n);
    sum_derecha[n-1] = x[n-1];
    for (int i = n-2; i >= 0; --i) {
        sum_derecha[i] = x[i] + sum_derecha[i+1];
    }

    bt(0, 0, true);

```

```
50      cerr << "FIN" << endl;
51  }
```

Optimal separation

Solución

El problema es similar al de equal sums, en este caso queremos separar un conjunto en dos conjuntos de forma que las sumas sean lo más parecidas posibles (minimizar la diferencia en valor absoluto). Si T es el total de la suma de todos los elementos, seguro que uno de los dos conjuntos tiene suma $\leq T/2$, el más pequeño.
Haremos el backtracking buscando el mejor conjunto pequeño, es decir, que no supere la suma de $T/2$ pero que se acerce todo lo posible. Por ejemplo si tuviéramos exactamente $T/2$ sería óptimo ya que el otro sería igual y por tanto, la diferencia sería 0.
Para optimizar usaremos la misma optimización que en *Equal sums (3)*, mirar si con lo que nos queda a la derecha podemos superar el mejor resultado hasta el momento.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;

int n, total, best;
vi v, sum_derecha;

void bt(int i, int sum) {
    best = max(best, sum);
    if (i < n) {
        if (sum+v[i] <= total/2 and sum+sum_derecha[i] > best)
            bt(i+1, sum+v[i]);
        if (sum+sum_derecha[i]-v[i] > best)
            bt(i+1, sum);
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    while (cin >> n) {
        v = vi(n);
        total = 0, best = 0;
        for (int i = 0; i < n; ++i) {
            cin >> v[i];
            total += v[i];
        }
        sum_derecha = vi(n);
        sum_derecha[n-1] = v[n-1];
        for (int i = n-2; i >= 0; --i) {
            sum_derecha[i] = v[i] + sum_derecha[i+1];
        }
        bt(0, 0);
        cout << (total-best) - best << endl;
    }
}
```

Queens (2)

Solución

Para resolver este problema, iremos avanzando en el número de la fila, y buscando en qué columnas podemos poner una reina.
Hay que tener en cuenta 3 cosas al poner una reina:

1. No hay otra reina en la misma columna
2. No hay otra reina en la misma diagonal izquierda
3. No hay otra reina en la misma diagonal derecha

Nota: Con diagonal izquierda me refiero a la que va de arriba a la izquierda a abajo a la derecha, y con diagonal derecha a la que va de arriba a la derecha a abajo a la izquierda.

Verificar que la columna está libre puede hacerse fácilmente con un vector.

Para verificar que la diagonal izquierda necesitamos la siguiente observación:
Dos casillas de la misma diagonal izquierda cumplen que (fila)-(columna) $= k$, para cada diagonal un cierto valor de k. Podrían dar números negativos por lo que identificaremos la posición (i, j) con la diagonal izquierda $n + i - j$, y así podemos en un vector representar las diagonales usadas.

Para verificar que la diagonal derecha necesitamos una observación similar:
Dos casillas de la misma diagonal derecha cumplen que (fila)+(columna) $= k$, para cada diagonal un cierto valor de k. Identificaremos la posición (i, j) con la diagonal derecha $i + j$, y así podemos en un vector representar las diagonales usadas.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;

int n;
vi pos_by_row, col, ldiag, rdiag;

void bt(int i) {
    if (i == n) {
        for (int r = 0; r < n; ++r) {
            for (int j = 0; j < n; ++j) {
                if (pos_by_row[r] == j)
                    cout << 'Q';
                else
                    cout << '.';
            }
            cout << endl;
        }
        cout << endl;
    }
    else {
        for (int j = 0; j < n; ++j) {
            if (not col[j] and not ldiag[n+i-j] and not rdiag[i+j]) {
                pos_by_row[i] = j;
```

```
                col[j] = ldiag[n+i-j] = rdiag[i+j] = true;
                bt(i+1);
                col[j] = ldiag[n+i-j] = rdiag[i+j] = false;
            }
        }
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    while (cin >> n) {
        pos_by_row = col = ldiag = rdiag = vi(2*n);
        bt(0);
    }
}
```

Every change

Solución

Mantenemos el índice de la moneda más grande usada para mantener el orden ascendente y vamos probando combinaciones de usar monedas. También mantenemos un vector de monedas usadas. Si llegamos a $n = 0$ hemos conseguido dar el cambio exacto e imprimimos la solución.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;

const vi coins = {50, 20, 10, 5, 2, 1};
vi change;

void solve(int n, int ci) {
    if (n == 0) {
        cout << change[0];
        for (int i = 1; i < change.size(); ++i) {
            cout << ' ' << change[i];
        }
        cout << endl;
    }
    else {
        for (int i = ci; i < coins.size(); ++i) {
            int c = coins[i];
            if (c <= n) {
                change.push_back(c);
                solve(n-c, i);
                change.pop_back();
            }
        }
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n;
    while (cin >> n) {
        solve(n, 0);
        cout << endl;
    }
}
```

Two rows of numbers

Solución

El problema se simplifica cuando ordenamos los números de menor a mayor. Tras ordenar solo tenemos que ir colocando de forma que en Y nunca haya más elementos que en X, ya que solo cuando se da ese caso no podremos meter en X uno más correspondiente en el mismo índice cumpliendo la restricción $X_i < Y_i$.

Código

```cpp
#include <bits/stdc++.h>
using namespace std;

using vi = vector<int>;

int n;
vi v, x, y;

void bt(int i) {
    int a = x.size();
    int b = y.size();
    if (a+b == 2*n) {
        for (int j = 0; j < n; ++j) {
            if (j > 0) cout << ' ';
            cout << x[j];
        }
        cout << endl;
        for (int j = 0; j < n; ++j) {
            if (j > 0) cout << ' ';
            cout << y[j];
        }
        cout << endl << endl;
    }
    else {
        if (a+1 <= n) {
            x.push_back(v[i]);
            bt(i+1);
            x.pop_back();
        }
        if (b+1 <= a) {
            y.push_back(v[i]);
            bt(i+1);
            y.pop_back();
        }
    }
}

int main() {
    cin >> n;
    v = vi(2*n);
    for (int i = 0; i < 2*n; ++i) {
        cin >> v[i];
    }
    sort(v.begin(), v.end());
    bt(0);
}
```

Numerology

Solución

Para este problema mantenemos un vector con los números que hemos separado, pero lo hacemos con un pair para guardar en la primera posición los ceros a la izquierda y en la segunda posición el valor. En cada paso probamos a seguir o a partir en ese punto el número.

Código

```
/#include <bits/stdc++.h>
using namespace std;

using ll = long long;

ll a;
string b;
bool some_solution;
vector<pair<int,int>> v;

void bt(ll sum, int lzeros, ll number, int i) {
    if (i == b.size()) {
        sum += number;
        v.push_back({lzeros, number});

        if (sum == a) {
            some_solution = true;
            cout << a << " = ";
            for (int j = 0; j < v.size(); ++j) {
                if (j != 0) cout << " + ";
                for (int k = 0; k < v[j].first; ++k) cout << '0';
                if (v[j].second != 0) cout << v[j].second;
            }
            cout << endl;
        }

        v.pop_back();
    }
    else if (sum + number <= a) {
        int d = b[i] - '0';
        if (i > 0) { // case 1 (add separator)
            v.push_back({lzeros, number});
            if (d == 0)
                bt(sum+number, 1, 0, i+1);
            else
                bt(sum+number, 0, d, i+1);
            v.pop_back(); // restore
        }

        if (d == 0 and number == 0)  // case 2 (no separator)
            bt(sum, lzeros+1, number, i+1);
        else
            bt(sum, lzeros, 10*number+d, i+1);
    }
}

int main() {
    ios::sync_with_stdio(0);
```

```
49      cin.tie(0);
50
51      while (cin >> a >> b) {
52          some_solution = false;
53          bt(0, 0, 0, 0);
54          if (not some_solution)
55              cout << "No solution for " << a << " " << b << "." << endl;
56      }
57  }
```

Partition to K Equal Sum Subsets

Solución

En lugar de a cada paso decidir a cuál de los K conjuntos mandamos cada elemento, es más eficiente generar primero el primer conjunto, luego el segundo, luego el tercero... Como sabemos la suma objetivo (suma total / K), es fácil que el algoritmo pare antes de considerar la mitad o más de los conjuntos, mientras que de la otra forma los consideramos todos a la vez.

Código

```
vector<int> nums, cnt;
int k, n, obj;
bool possible;

void bt(int i, int group, int sum) {
    if (i == n) {
        if (sum == obj) {
            if (group == k-1) {
                possible = true;
            }
            else {
                bt(0, group+1, 0);
            }
        }
    }
    else {
        if (cnt[nums[i]] > 0 and sum + nums[i] <= obj) {
            --cnt[nums[i]];
            bt(i+1, group, sum+nums[i]);
            ++cnt[nums[i]];
        }
        bt(i+1, group, sum);
    }
}

bool canPartitionKSubsets(vector<int>& nums_, int k_) {
    nums = nums_;
    k = k_;
    n = nums.size();
    cnt = vector<int>(1e4+9, 0);
    obj = 0;
    for (int x : nums) {
        cnt[x]++;
        obj += x;
    }
    if (obj % k != 0) return false;
    obj /= k;
    possible = false;
    bt(0, 0, 0);
    return possible;
}
```