Instituto Politécnico Nacional
Escuela Superior de Cómputo
Análisis de Algoritmos

Tarea 06

Tecnica divide y venceras

Alumno:

López Manríquez Ángel (2017630941)

M. en C. Edgardo Adrián Franco Martínez

Grupo: 3CM3
Fecha: 22 de junio de 2019

Índice

1. Maximum subarray sum

1.1. Descripcion

La tarea es simple, dado un arreglo A de números enteros debes imprimir cual es la suma máxima en cualquier subarreglo contiguo.

1.2. Codigo

```cpp
// Angel Lopez Manriquez

#include <iostream>
#include <vector>

#include <math.h>
#include <stdint.h> // int64_t defined

using namespace std;

int64_t max_subarr_between(const vector<int64_t> &v, const long l, const long r,
        const long mid) {
    int64_t leftSum = INTMAX_MIN; // -infty
    int64_t rightSum = INTMAX_MIN;
    int64_t sum = 0;

    // obtain the highest value from the left
    for (int i = mid; i >= l; --i) {
        sum += v[i];
        if (sum > leftSum) leftSum = sum;
    }

    // obtain the highest value from the right
    sum = 0;
    for (int i = mid + 1; i <= r; ++i) {
        sum += v[i];
        if (sum > rightSum) rightSum = sum;
    }

    return leftSum + rightSum;
}

int64_t max_subarr_subseq(const vector<int64_t> &v, const long l, const long r) {
    if (l == r) {
        return v[l];
    }

    // compute the left subarray, right and the middle one and return the maximum
    const long mid = (r - l) / 2 + l;
    const int64_t leftSum = max_subarr_subseq(v, l, mid);
    const int64_t rightSum = max_subarr_subseq(v, mid + 1, r);
    const int64_t midSum = max_subarr_between(v, l, r, mid);
    return max(leftSum, max(rightSum, midSum));
```

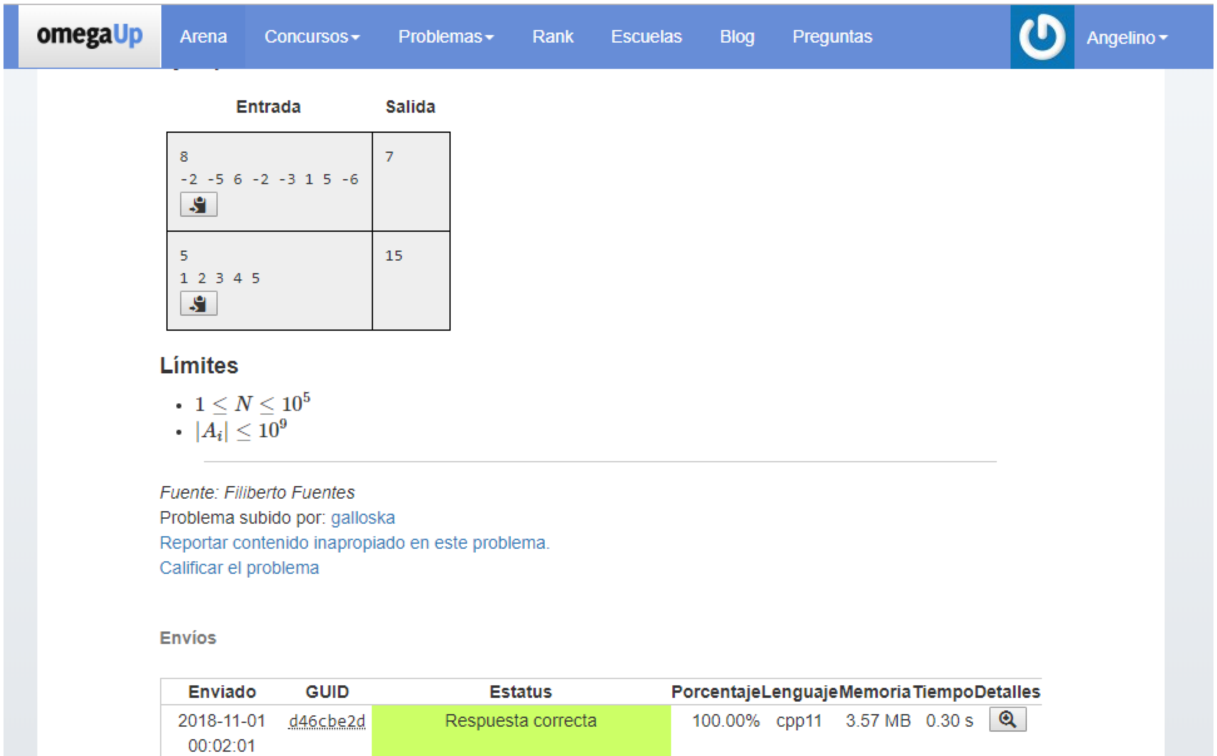
```
}

inline int64_t max_subarr_subseq(vector<int64_t> &v) {
    return max_subarr_subseq(v, 0, v.size() - 1); // begin the recurrence
}

vector<int64_t> fillVec() {
    long n;
    cin >> n;
    vector<int64_t> v(n); // allocate a vector with n elements
    for (int i = 0; i < n; ++i) cin >> v[i];
    return v;
}

int main(void) {
    vector<int64_t> v = fillVec(); // read the vector

    int64_t ans = max_subarr_subseq(v);
    cout << ans;
    return 0;
}
```

1.3. Explicacion

Dado un arreglo $A =< a_0, a_1, ..., a_{n-1} >$, este problema se resolvio con la tecnica divide y venceras (divide and conquer, en ingles) en donde, separamos al vector en dos mitades L, R y obtenemos el valor de estas recursivamentes teniendo muy en cuenta que posiblemente la suma maxima se encuentre en el medio de estos dos subvectores, tomamos este caso aparte obteniendo la suma maxima empezando desde el medio hasta la izquierda y de manera analoga con la derecha, solo obtenemos la suma maxima de estas mitades y al final sumamos estas pues al estar concatenadas la suma forma parte de un subarreglo, claro, el caso base es cuando el arreglo A es tal que $|A| \leq 1$ pues es el caso mas sencillo. Al final de la recurrencia encontramos el maximo entre sus costados y el medio.

Por lo anterior explicado la complejidad es $O(n \lg n)$.

1.4. Validacion del juez



2. Closest pair

2.1. Descripción

Te encuentras con un mapa del cúmulo de estrellas R136. En el mapa, cada estrella aparece como un punto ubicada en un plano cartesiano. Te asalta de pronto una pregunta, ¿cuál será la distancia mínima entre dos estrellas en el mapa?

2.2. Codigo

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // copy_if
#include <iomanip> // fixed, setprecision

#include <stdint.h>
#include <float.h>
#include <math.h>
#include <stdlib.h>

using namespace std;
```

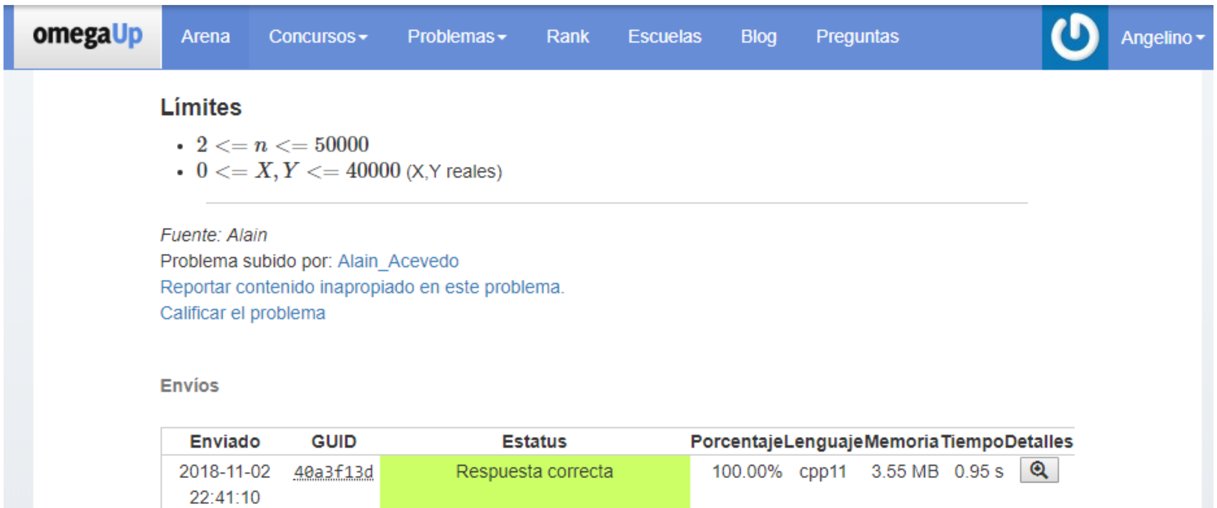
```cpp

typedef pair<double, double> double_pair;

inline double dist(const double_pair &p, const double_pair &q) {
    return hypot((p.first - q.first), (p.second - q.second));
}

int cmp_y_axis(const double_pair &p, const double_pair &q) {
    return p.second < q.second;
}

double min_dist_brute_force(const vector<double_pair> &pairs) {
    double min = DBL_MAX;
   const int n = pairs.size();
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            const double d = dist(pairs[i], pairs[j]);
            if (d < min) min = d;
        }
    }
    return min;
}

/**
 * Obtains the minimum distance in the middle of the set of computed points.
 * We know that if the distance is bigger than the minimum computed in the sides
 * we don't even have to compute it.
 **/
double min_middle(const vector<double_pair> &xsorted, const double &offset, const double &d) {

    double min = DBL_MAX;
    vector<double_pair> center_pairs;
    auto in_limit = [&](const double_pair p) { return abs(p.first - offset) <= d; };
    copy_if(xsorted.begin(), xsorted.end(), back_inserter(center_pairs), in_limit); // it may be optimized

    const int n = center_pairs.size();
    for (int i = 0; i < n; ++i)
        for (int j = i + 1; j < n; ++j) {
            if (abs(center_pairs[i].second - center_pairs[j].second) < d) {
                double curr_dist = dist(center_pairs[j], center_pairs[i]);
                if (curr_dist < min) min = curr_dist;
            }
        }

    return min;
}

// always returns a positive value
double min_dist(const vector<double_pair> &xsort) {
    const int n = xsort.size();

    // apply brute force for this little case
    if (n <= 3) return min_dist_brute_force(xsort);
    const double_pair mid_pair = xsort[n / 2];

```

```
    vector <double_pair> left;
    vector <double_pair> right;

    for (auto &p: xsort) {
        if (p.first <= mid_pair.first) left.push_back(p);
        else right.push_back(p);
    }

    const double min_left = min_dist(left), min_right = min_dist(right);
    const double min_sides = min(min_left, min_right);
    const double min_mid = min_middle(xsort, mid_pair.first, min_sides);

    return min(min_sides, min_mid);
}

double get_min_dist(vector<double_pair> &pairs) {
    sort(pairs.begin(), pairs.end()); // it'll sort by the first element by default

    return min_dist(pairs);
}

void print_pairs(const vector<double_pair> &pairs) {
    for (auto &p: pairs) cout << p.first << ", " <<  p.second << endl;
}

vector<double_pair> scanv() {
    int n;
    cin >> n;
    vector<double_pair> pairs(n);
    for (int i = 0; i < n; ++i) {
        cin >> pairs[i].first >> pairs[i].second;
    }
    return pairs;
}

int main(int argc, char const *argv[]) {
    vector<double_pair> pairs = scanv();
    cout << fixed << setprecision(3);
    cout << get_min_dist(pairs);
    return 0;
}

// compilation:
// g++ -g closest_pair.cpp -std=c++11
```

2.3. Explicacion

Esta pregunta es un clasico y, como todo programa, puede tener multiples soluciones, aqui, dado un conjunto de pares ordenados $\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ primero procedemos a ordenarlos con respecto al primer par del eje x, luego nos concentramos en resolver el problema recursivamente para la primer mitad de puntos y para la siguiente mitad, obtenemos el minimo de ambas junto con la franja que se encuentra en el medio de estas tomando en cuenta que esta franja debe abarcar un ancho tal que es el minimo de las soluciones anteriores pues no hay necesidad de buscar mas alla por la desigualdad del triangulo. El caso base es cuando tenemos solo tres puntos, aqui procedemos a resolverlos mediante fuerza bruta. La implementacion solo funciona si para toda $x \in \mathbf{R}$ le corresponde un unico valor, afortunadamente en el juez no se presento tal caso. La complejidad es $O(n \lg n)$.

2.4. Validacion del juez



3. Inversion count

3.1. Descripción

Dado un arreglo $A[0,\ldots,n-1]$ de enteros positivos ($n \leq 2 \times 10^5$, $A[i] \leq 10^7$), definimos una *inversión* como un par ordenado (i, j) tal que $i < j$ y $A[i] > A[j]$; es decir, dos elementos distintos que se encuentran "desordenados". El problema es hallar todas las inversiones en A.

3.2. Codigo

```
#include <iostream>
#include <vector>
#include <array>
#include <functional>
#include <stack>

#include <stdint.h>

using namespace std;

template<typename t>
int merge(vector<t> &dest, vector<t> &buff, const int left, const int mid, const int right) {
    int i = left, j = mid, k = left, count = 0;
```

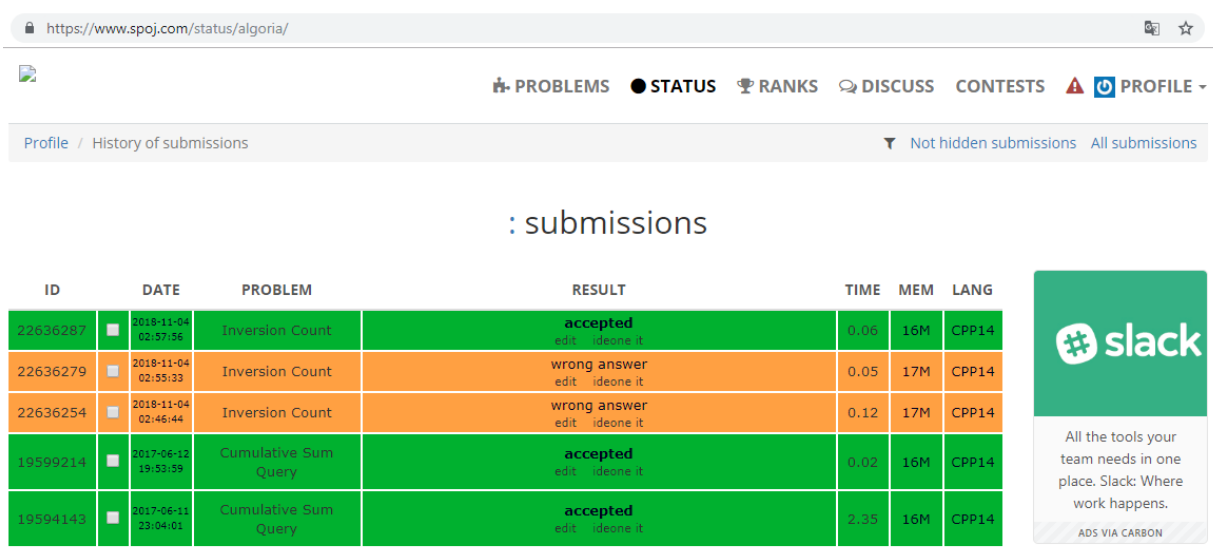
```
    while ((i <= mid - 1) && (j <= right)) {
        if (dest[i] <= dest[j]) buff[k] = dest[i++];
        else {
            buff[k] = dest[j++];
            count += mid - i;
        }
        ++k;
    }

    while (i <= mid - 1) {
        buff[k] = dest[i++];
        ++k;
    }

    while (j <= right ) {
        buff[k] = dest[j++];
        ++k;
    }

    for (i = left; i <= right; ++i) dest[i] = buff[i];
    return count;
}

template<typename t>
int mergesort(vector<t> &v, vector<t> &buff, const int l, const int r) {
    if (!(l < r)) return 0;
    int mid = l + (r - l) / 2;
    return mergesort(v, buff, l, mid)
        +  mergesort(v, buff, mid + 1, r)
        +  merge(v, buff, l, mid + 1, r);
}

template<typename t>
int mergesort(vector<t> &v) {
    vector<t> buff(v.size());
    return mergesort(v, buff, 0, v.size() - 1);
}

vector<int64_t> scanv() {
    int n;
    cin >> n;
    vector<int64_t> v(n);
    for (int i = 0; i < n; ++i)
        cin >> v[i];
    return v;
}

template<typename t>
void print(vector<t> v) {
    cout << endl;
    for (auto &x: v) cout << x << " ";
    cout << endl;
}

int main(int argc, char const *argv[])
{
```

```
    ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    int t;
    cin >> t;

    while (t--) {
        vector<int64_t> v = scanv();
        cout << mergesort(v) << endl;
    }


    return 0;

}
```

3.3. Explicacion

Para resolver el ejercicio usaremos el algoritmo *mergesort* modificado ligeramente, es bien sabido que el mergersort tiene una complejidad $O(n \lg n)$ por lo que su ejecucion es relativamente rapida. Basicamente, ordenamos como de costumbre solo que esta vez retornamos un entero el cual da el numero de inversiones que hay en el arreglo, cada que llamamos la funcion `merge` le pasamos los indices del arreglo i, j, m en el cual el numero de inversiones es representado por $\sum_i (m - i)$ cuando se encuentra a una inversion.

3.4. Validacion del juez



ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
22636287		2018-11-04 02:57:56	Inversion Count	accepted edit ideone it	0.06	16M	CPP14
22636279		2018-11-04 02:55:33	Inversion Count	wrong answer edit ideone it	0.05	17M	CPP14
22636254		2018-11-04 02:46:44	Inversion Count	wrong answer edit ideone it	0.12	17M	CPP14
19599214		2017-06-12 19:53:59	Cumulative Sum Query	accepted edit ideone it	0.02	16M	CPP14
19594143		2017-06-11 23:04:01	Cumulative Sum Query	accepted edit ideone it	2.35	16M	CPP14

4. Mega inversions

4.1. Descripción:

Dado un arreglo $A[0,\ldots,n-1]$ de enteros positivos ($n \leq 10^5$, $A[i] \leq n$), definimos una *mega inversión* como una terna ordenada (i,j,k) tal que $i<j<k$ y $A[i] > A[j] > A[k]$. El problema es hallar todas las mega inversiones en A.

4.2. Codigo

```
#include <iostream>
#include <vector>
#include <algorithm>

#include <stdint.h>

using namespace std;

template<typename T>
class FenwickTree {
private:
   int n;
   vector<T> bit;

public:
   FenwickTree(int N) {
      this->n = N;
      bit.assign(N, 0);
   }

   void update(int pos, T value) {
      while(pos < n) {
         bit[pos] += value;
         pos |= pos + 1;
      }
   }

   T query(int r) {
      T res = 0;
      while (r >= 0) {
         res += bit[r];
         r = (r & (r + 1)) - 1;
      }
      return res;
   }

   T query(int l, int r) {
      return query(r) - query(l - 1);
   }
};

int64_t megaInversions(vector<int> & v) {
```

```
    int64_t count = 0;
    int maximum = *max_element(v.begin(), v.end());
    FenwickTree<int64_t> bit1(maximum + 1);
    FenwickTree<int64_t> bit2(maximum + 1);
    for (int & a: v) {
        bit1.update(a, 1);
        bit2.update(a, bit1.query(a + 1, maximum));
        count += bit2.query(a + 1, maximum);
    }
    return count;
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; ++i) cin >> v[i];
    cout << megaInversions(v) << "\n";
    return 0;
}
```

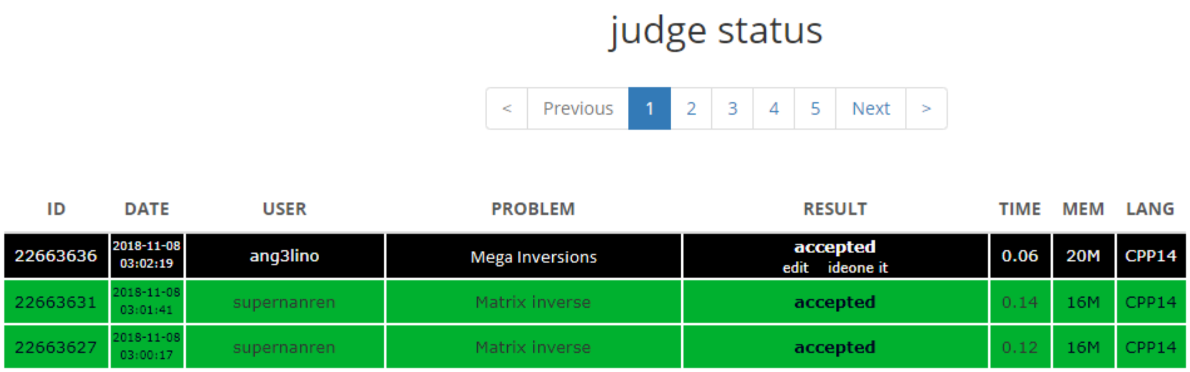
4.3. Explicacion

Usamos dos Fenwick Tree B_1 y B_2 inicializados en cero. ¿Que es un Fenwick tree? Es una estructura de dato tipo arbol T la cual, dado un arreglo A es capas de obtener $\sum_{i=a}^{b} a_i$, es decir, la "funcion acumulada"de A de manera bastante rapida, a diferencia de un arbol de segmentos o usar barridos, en fin, ahora ya explicado el Fenwick Tree tenemos que iterar sobre los elementos de A y definimos:

1. $B_1[a_i]$ es el número de veces que ha aparecido el elemento a_i hasta la i-ésima iteración. Tal y como lo hicimos en el problema anterior.
2. Por lo tanto, $\displaystyle\sum_{j=a_i+1}^{n} B_1[j]$ nos dirá cuántos elementos a_j a la derecha de a_i son mayores a a_i, es decir, los elementos a_j tal que $i < j$ y $A[i] > A[j]$.

Para el analiss, sea $n = |A|$, asi, la construccion del arbol cuesta $O(n)$ tenemos un bucle que incrementa en una unidad en donde actualizamos y consultamos el arbol, cada una de estas operaciones cuesta $\lg n$, por lo que la funcion `megaInversion` cuesta $O(n + n(4 \lg n)) = O(n \lg n)$.

4.4. Validacion del juez


judge status

< Previous 1 2 3 4 5 Next >

ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
22663636	2018-11-08 03:02:19	ang3lino	Mega Inversions	accepted edit ideone it	0.06	20M	CPP14
22663631	2018-11-08 03:01:41	supernanren	Matrix inverse	accepted	0.14	16M	CPP14
22663627	2018-11-08 03:00:17	supernanren	Matrix inverse	accepted	0.12	16M	CPP14

5. Cobertura

5.1. Descripción

Considera una pieza con forma de L formada por 3 cuadros adyacentes.

Se requiere cubrir una cuadricula con piezas como la antes descrita dejando un solo cuadro de 1×1 vacío. La cuadrícula consta de 2^n filas (numeradas de 1 a 2^n) por 2^n columnas (numeradas también de 1 a 2^n).

Para cubrir la cuadrídula no es válido colocar una pieza sobre otra ni dejar una o más piezas parcialmente fuera de la cuadrícula. Es posible comprobar que toda cuadrícula de dichas dimensiones puede ser cubierta siguiendo las restricciones anteriores sin importar donde se encuentre el espacio vacío.

5.2. Codigo

```cpp
#include <iostream>
using namespace std;

void solve(int r0, int r1, int c0, int c1, int x, int y) {
   if (r1 - r0 == 1 && c1 - c0 == 1) {// base case
      if (!(r0 == x && c0 == y)) cout << r0 << " " << c0 << " ";
      if (!(r0 == x && c1 == y)) cout << r0 << " " << c1 << " ";
      if (!(r1 == x && c0 == y)) cout << r1 << " " << c0 << " ";
      if (!(r1 == x && c1 == y)) cout << r1 << " " << c1 << " ";
      cout << "\n";
   } else { // inductive case
      int mid_r = (r0 + r1) / 2;
      int mid_c = (c0 + c1) / 2;
      if (r0 <= x && x <= mid_r && c0 <= y && y <= mid_c) {
         //first quadrant
         cout << mid_r << " " << mid_c + 1 << " ";
```

```
            cout << mid_r + 1 << " " << mid_c << " ";
            cout << mid_r + 1 << " " << mid_c + 1 << "\n";
            solve(r0, mid_r, c0, mid_c, x, y);
            solve(r0, mid_r, mid_c + 1, c1, mid_r, mid_c + 1);
            solve(mid_r + 1, r1, c0, mid_c, mid_r + 1, mid_c);
            solve(mid_r + 1, r1, mid_c + 1, c1, mid_r + 1, mid_c + 1);
        } else if (r0 <= x && x <= mid_r && mid_c + 1 <= y && y <= c1) {
            //second quadrant
            cout << mid_r << " " << mid_c << " ";
            cout << mid_r + 1 << " " << mid_c << " ";
            cout << mid_r + 1 << " " << mid_c + 1 << "\n";
            solve(r0, mid_r, c0, mid_c, mid_r, mid_c);
            solve(r0, mid_r, mid_c + 1, c1, x, y);
            solve(mid_r + 1, r1, c0, mid_c, mid_r + 1, mid_c);
            solve(mid_r + 1, r1, mid_c + 1, c1, mid_r + 1, mid_c + 1);
        } else if (mid_r + 1 <= x && x <= r1 && c0 <= y && y <= mid_c) {
            //third quadrant
            cout << mid_r << " " << mid_c << " ";
            cout << mid_r << " " << mid_c + 1 << " ";
            cout << mid_r + 1 << " " << mid_c + 1 << "\n";
            solve(r0, mid_r, c0, mid_c, mid_r, mid_c);
            solve(r0, mid_r, mid_c + 1, c1, mid_r, mid_c + 1);
            solve(mid_r + 1, r1, c0, mid_c, x, y);
            solve(mid_r + 1, r1, mid_c + 1, c1, mid_r + 1, mid_c + 1);
        } else if (mid_r + 1 <= x && x <= r1 && mid_c + 1 <= y && y <= c1) {
            //fourth quadrant
            cout << mid_r << " " << mid_c << " ";
            cout << mid_r << " " << mid_c + 1 << " ";
            cout << mid_r + 1 << " " << mid_c << "\n";
            solve(r0, mid_r, c0, mid_c, mid_r, mid_c);
            solve(r0, mid_r, mid_c + 1, c1, mid_r, mid_c + 1);
            solve(mid_r + 1, r1, c0, mid_c, mid_r + 1, mid_c);
            solve(mid_r + 1, r1, mid_c + 1, c1, x, y);
        }
    }
}

int main() {
    int n, x, y;
    cin >> n >> x >> y;
    solve(1, 1 << n, 1, 1 << n, x, y);
    return 0;
}
```

5.3. Explicacion

Para la implementacion del programa se tuvo que hacer uso de la demostracion de un lema, el lema es la descripcion del problema el cual asegura que existe una configuracion tal que un cuadrado de dimension de potencias de dos puede ser llenado usando piezas en forma de L (se llama triomino).

Prueba del paso inductivo: Para $n = k$, tenemos un tablero de ajedrez de tamaño $2^k \times 2^k$. , con un azulejo de la esquina eliminado. Podemos dividir este tablero en 4 partes, tres de las cuales son tableros de ajedrez completos de tamano $2^{k-1} \times 2^{k-1}$, y uno del mismo tamano con un azulejo de esquina eliminado. El tablero con el azulejo de la esquina eliminado se puede cubrir con triominos, utilizando la hipótesis inductiva. Los otros tres tableros pueden tener todos los mosaicos, excepto un azulejo de esquina, cada uno cubierto por triominos, nuevamente utilizando la hipótesis inductiva. Ahora tenemos tres tablas, de tamano $2^{k-1} \times 2^{k-1}$, cada uno con un azulejo de esquina descubierto. Podemos organizar las tres tablas para que las tres baldosas descubiertas se unan para formar una vacante con forma de triomino, y cubrirlo utilizando otro triomino. Entonces, hemos probado que si la declaración se mantiene para $n = k - 1$, también se mantiene para $n = k$. Por lo tanto, la afirmación está probada para todos n usando inducción. La complejidad es $O(2^{2n})$, lo cual explica el porque la maxima n que se nos puede dar es 10.

5.4. Validacion del juez