La magia de la programación competitiva

Comunidad new liberty

24 de julio de 2017

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

recursividad

1.1. Descripción y Motivación

Existen problemas que para resolverlos tenemos que ejecutar el mismo bloque de instrucciones varias veces, esto se puede lograr con ciclos iterativos o con recursividad. Todos los algoritmos iterativos pueden ser programados recursivamente y viceversa, aun que debemos aprender a elegir cual es la técnica correcta a utilizar. La implementación de un algoritmo iterativo consiste en repetir el cuerpo del bucle, la implementacion de un algoritmo recursivo consiste en ejecutar repetidamente el mismo metodo. Los principales criterios a la hora de elegir entre programar algo iterativamente o recursivamente son: el rendimiento y la simpleza del codigo generado. Supongamos que debemos resolver el problema de sumar los primeros n numeros, dos algoritmos que solucionan este problema son los siguientes:

Iterativo

Listing 1.1: sumaIterativa.cpp

```
int sumaIterativa(int n){
    int resultado = 0;
    for(int i=1;i<=n;i++){
        resultado += i;
    }
    return resultado;
}
```

Recursivo

Listing 1.2: sumaRecursiva.cpp

```
int sumaRecursiva(int n){
    //Caso base
    if(n==1){
        return 1;
    }else{
        return n + sumaRecursiva(n-1);
    }
}
```

Algo muy importante a tener en cuenta en los algoritmos recursivos es el caso base, al igual que en los algoritmos iterativos se debe saber cuando detener la ejecución en los algoritmos recursivos necesitamos saber en donde detenernos. En realidad los dos algoritmos que mostramos tienen una ligera diferencia aun que dan el mismo resultado. En nuestro algoritmo iterativo sumamos desde 0 hasta n de la siguiente manera: 0+1+2+3+...+n, pero en el recursivo sumamos desde n hasta 0: n+(n-1)+(n-2)...+0. Si quisieramos que tuvieran un comportamiento mas similar podriamos programar el algoritmo recursivo de la siguiente manera:

Listing 1.3: sumaRecursiva2.cpp

```
int sumaRecursiva(int actual, int n){
    //Caso base
    if(actual==n){
        return actual;
    }else{
        return actual + sumaRecursiva(actual+1,n);
    }
}
```

El caso base esta muy ligado a la manera en que hacemos la recursividad, por lo general la recursividad se hace disminuyendo los parametros del problema, pero no siempre es asi como vimos en el segundo ejemplo, al igual que podemos hacer algoritmos iterativos con el contador ascendente o descendente y tenemos que generar la condicion de detener en base a este, en la recursividad también lo hacemos asi.

Quizas el ejemplo mas claro de recursividad es factorial de n. Ya que la solucion de factorial de n es n * factorial de (n-1), y la solución de factorial de (n-1) es (n-1) * factorial de (n-2) y asi constantemente.

Por ejemplo factorial de 5 es
f(5) = 5*f(4)
f(4) = 4*f(3)
f(3) = 3*f(2)
f(2) = 2*f(1)
f(1) = 1
por lo tanto
f(2) = 2*f(1) = 2*1 = 2
f(3) = 3*f(2) = 3*2 = 6
f(4) = 4*f(3) = 4*6 = 24
f(5) = 5*f(4) = 5*12 = 120

Mas o menos de esa manera funciona la recursividad en código, se van guardando cada llamada al metodo en una cola, al retornar regresa al metodo que la llamo. asi f(5)-¿f(4)-¿f(3)-¿f(2)-¿f(1)
Iterativo

Listing 1.4: factorialIterativo.cpp

```
int factorial(int n){
    if(n==0)return 1;
    int resultado = 1;
    for(int i=n;i>=1;i--){
        resultado*=i;
    }
    return resultado;
}
```

Iterativo

Listing 1.5: factorialRecursivo.cpp

```
int factorial(int n){
    //Caso base
    if(n==0){
        return 1;
    }
    if(n==1){
        return 1;
    }
    return n * factorial(n-1);
}
```

Se debe tener cuidado al usar recursividad en no calcular muchas veces la misma solución, por ejemplo con el algoritmo de fibbonacci. su formula recursiva es f(n) = f(n-1) + f(n-2). Si ejecutamos por ejemplo f(5) sucederia lo siguiente:

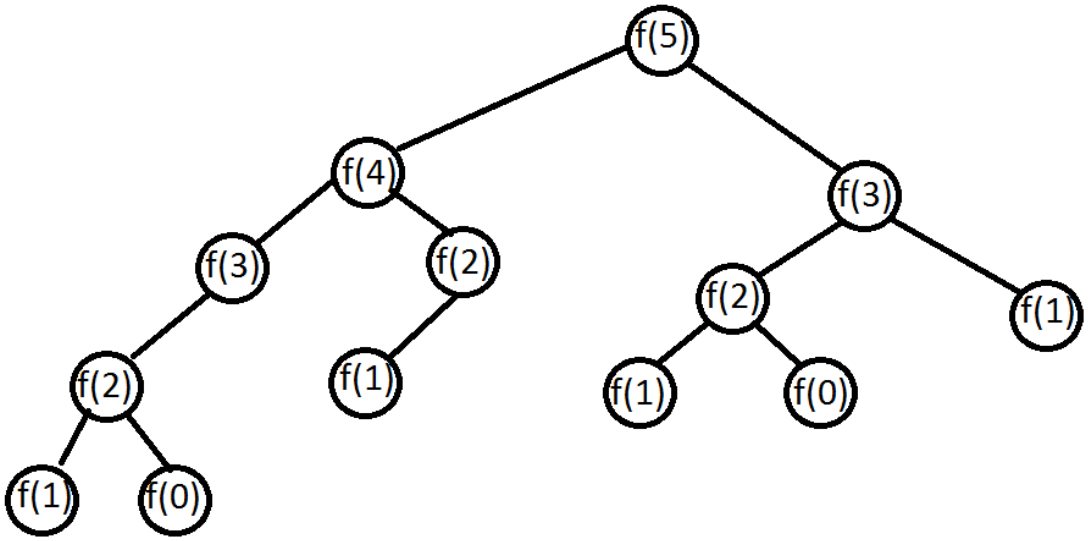

Figura 1.1: fibonacci.png

Podemos observar que recalculamos mucho, esto se puede resolver aplicando tecnicas de DP, pero eso lo veremos en otro capitulo.

1.2. Ejemplos

Ya vimos uno de los ejemplos mas tipicos de recursividad, el del factorial, en esta sección veremos el algoritmo de fibonacci, el algoritmo de Euclides (para hallar el máximo común divisor) y un algoritmo para solucionar las torres de hanoi.

Empecemos por el algoritmo de fibonacci, por definición la suseción de fibonacci comienza de la siguiente forma : 0,1,1,2,3,5,8 ... , cada elemento es la suma de sus dos anteriores. Más formalmente:

f(n) = f(n-1) + f(n-2)

Veamos primero como seria el algoritmo de fibonacci sin hacer uso de la recursión.

Listing 1.6: fibonacciIterativo.cpp

```
int fibonacci(int n){
    if(n==0)return 0;
    if(n==1)return 1;
    int a = 0;
    int b = 1;
    int c = a+b;
    for(int i=2;i<=n;i++){
        c = a+b;
        a = b;
        b = c;
    }
    return c;
}
```

Y ahora como seria usando recursión

Listing 1.7: fibonacciRecursivo.cpp

```
int fibonacci(int n){
    if(n==0)return 0;
    if(n==1)return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Mucho más simple, ¿no lo creen?. Ahora veamos el algoritmo de euclides
Iterativo

Listing 1.8: euclidesIterativo.cpp

```
int euclides(int a,int b){
    int temporal = a;
    while(a>0){
        temporal = a;
        a = b%a;
        b = temporal;
    }
    return b;
}
```

Recursivo

Listing 1.9: euclidesRecursivo.cpp

```
int euclides(int a,int b){
    if(b==0)return a;
    return euclides(b,a%b);
}
```

Por último mi ejemplo favorito para demostrar el potencial de la recursividad,las torres de hannoi. Si no conoces este juego, te recomiendo que primero busques en google “torres de hannoi online”, te saldran multiples opciones para jugarlo, es bastante simple e interesante.

En este caso no pondre una solución iterativa puesto que no se me ocurre ninguna, excepto simulando el comportamiento de la recursividad con una cola, (para saber más detalles al respecto, te invito a profundizar en como funciona internamente la recursividad).
El caso base de esta solución consiste en tener unicamente dos piezas apiladas, saber donde estan apiladas, hacia donde se dirigen, y el otro palo sera nuestro auxiliar.
La solución al caso base es muy sencilla, unicamente debemos desplazar la ficha superior a nuestro palo auxiliar, la ficha base a nuestro palo destino y por ultimo la ficha superior a nuestro destino, y

asi logramos resolver la torre de hannoi de nuestro caso base.

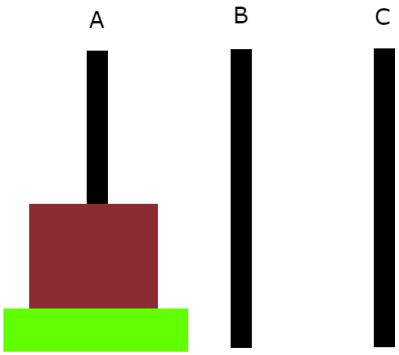

Figura 1.2: torre1.png

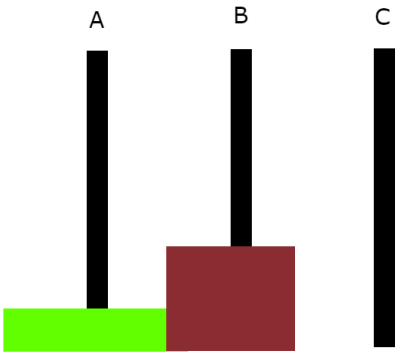

Figura 1.3: torre2.png

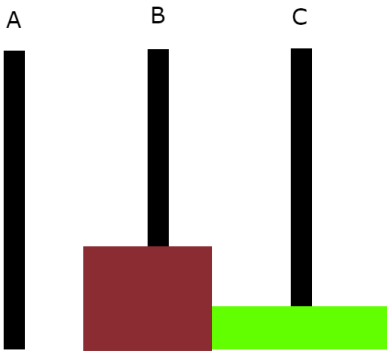

Figura 1.4: torre3.png

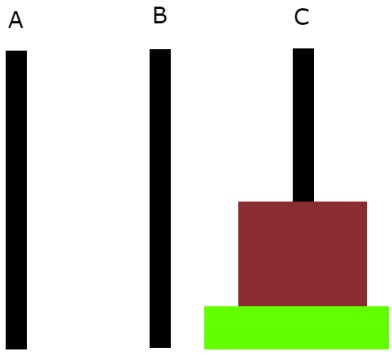

Figura 1.5: torre4.png

Pero que pasaria si fueran mas de dos fichas, he aqui donde viene la recursividad sucederia algo asi

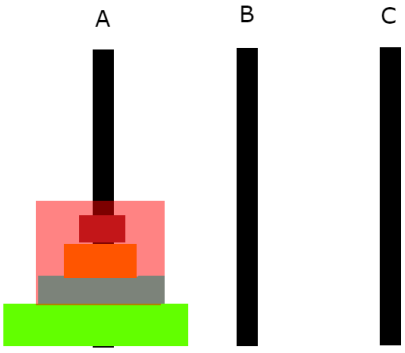

Figura 1.6: torre1-2.png

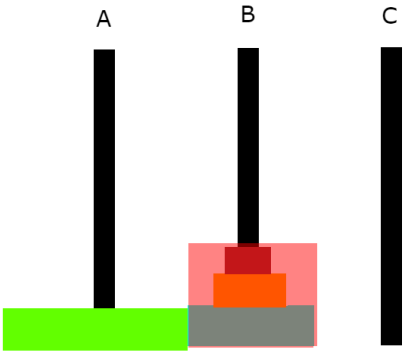

Figura 1.7: torre2-2.png

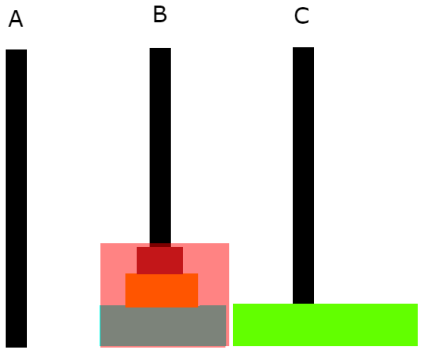

Figura 1.8: torre3-2.png

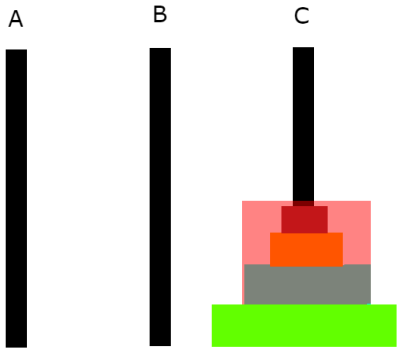

Figura 1.9: torre4-2.png

Internamente la recursion de las fichas sombreadas en rojo desde "torre1-2.png" hacia "torre2-2.png" funcionarian de la siguiente manera:

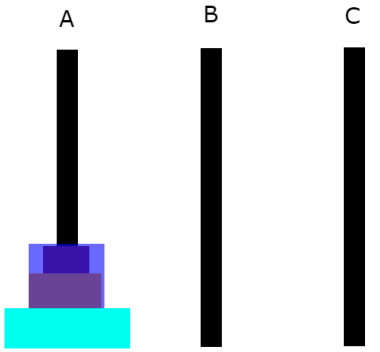

Figura 1.10: torre1-3.png

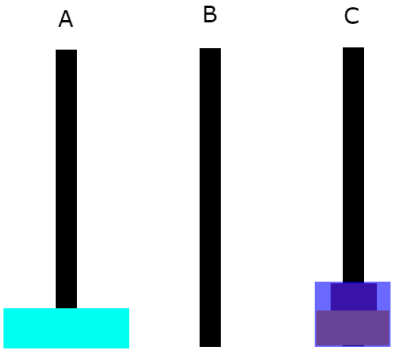

Figura 1.11: torre2-3.png

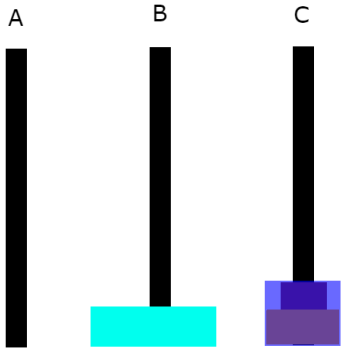

Figura 1.12: torre3-3.png

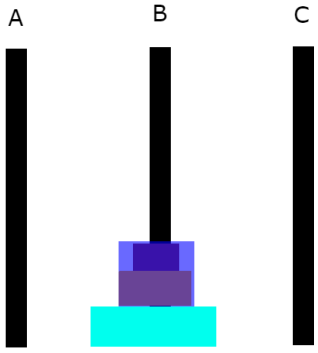

Figura 1.13: torre4-3.png

Y asi sucesivamente (recursivamente). La solución recursiva de la torre de hannoi consiste en llevar la parte superior (todas las piezas menos la base) hacia el palo auxiliar, mover la base al palo destino y finalmente mover la parte superior al palo destino. Cuando la parte superior es de mas de una pieza, se realiza la recursión cambiando invirtiendo el palo destino y el auxiliar. En código seria asi:

Recursivo

Listing 1.10: hannoi.cpp

```
1  void Hanoi(int disco, char origen, char intermedio, char destino){
2
3    if(disco == 1){
4      //caso base, solo movemos el disco a su destino
5      cout << "Mover disco " << disco << " desde " << origen << " hasta " <<
           destino << endl;
6    }else{
7      //movemos la parte superior al intermedio
8      Hanoi(disco-1, origen,destino,intermedio);
9      cout << "Mover disco " << disco << " desde " << origen << " hasta " <<
           destino << endl;
10     //movemos la parte superior al destino
11     Hanoi(disco-1,intermedio,origen,destino);
12   }
13 }
14
15 int main(){
16   int discos;
17   cout << "Ingrese la cantidad de discos: " << endl;
18   cin >> discos;
19   Hanoi(discos, 'A', 'B', 'C');
20
21   system("pause");
22 }
```

Capítulo 2

matemáticas

2.1. sucesiones y series

2.1.1. sucesión aritmética

las sucesiones aritméticas son aquellas que restando un elemento con su antecesor siempre da una constante se representan de la siguiente manera.
$an + b$
donde a es la resta entre dos elementos consecutivos y b es el primer elemento

2.1.2. sucesión geométrica

las sucesiones geométricas son aquellas que el cociente de un elemento con su antecesor siempre da una constante se representan de la siguiente manera.
ar^{n-1}
donde a es el primer termino y r es el cociente entre un numero y su anterior

2.1.3. serie aritmética

una serie aritmética es una sucesión creada con la suma de los términos de una sucesión aritmética, su formula es:
$a\frac{n(n+1)}{2} + nb$

2.1.4. serie geométrica

una serie geométrica es una sucesión creada con la suma del los términos de una sucesión geométrica, su formula es:
$a\frac{1-r^n}{1-r}$

Capítulo 3

geometricos

3.1. formulas de geometría

- $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$

3.2. estructuras geométricas

3.2.1. puntos

punto de enteros

```
// struct point_i { int x, y; };
// basic raw form, minimalist mode
struct point_i { int x, y;
// whenever possible, work with point_i
point_i() { x = y = 0; }
// default constructor
point_i(int _x, int _y) : x(_x), y(_y) {} };
// user-defined
```

punto de reales

```
struct point { double x, y;
// only used if more precision is needed
point() { x = y = 0.0; }
// default constructor
point(double _x, double _y) : x(_x), y(_y) {} };
// user-defined
```

ordenamiento de puntos

```
struct point { double x, y;
point() { x = y = 0.0; }
point(double _x, double _y) : x(_x), y(_y) {}
bool operator < (point other) const { // override less than operator
if (fabs(x - other.x) > EPS)
// useful for sorting
        return x < other.x;
// first criteria , by x-coordinate
return y < other.y; } };
```

```
// second criteria, by y-coordinate
// in int main(), assuming we already have a populated vector<point> P
sort(P.begin(), P.end());
// comparison operator is defined above
```

saber si dos puntos son iguales

```
struct point { double x, y;
point() { x = y = 0.0; }
point(double _x, double _y) : x(_x), y(_y) {}
// use EPS (1e-9) when testing equality of two floating points
bool operator == (point other) const {
return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };
```

distancia euclídea entre 2 puntos

```
double dist(point p1, point p2) {
// Euclidean distance
// hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
return hypot(p1.x - p2.x, p1.y - p2.y); }
// return double
```

rotar un punto con respecto al origen

```
// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
double rad = DEG_to_RAD(theta);
// multiply theta with PI / 180.0
return point(p.x * cos(rad) - p.y * sin(rad),
p.x * sin(rad) + p.y * cos(rad)); }
```

3.2.2. lineas

```
struct line { double a, b, c; };
// a way to represent a line
```

hallar una recta con 2 puntos

```
// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
if (fabs(p1.x - p2.x) < EPS) {
// vertical line is fine
l.a = 1.0;
l.b = 0.0;
l.c = -p1.x;
// default values
} else {
l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
l.b = 1.0;
```

```
12 // IMPORTANT: we fix the value of b to 1.0
13 l.c = -(double)(l.a * p1.x) - p1.y;
14 } }
```

saber si dos lineas son paralelas

```
1 bool areParallel(line l1, line l2) {
2 // check coefficients a & b
3 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }
```

saber si 2 lineas son iguales

```
1 bool areSame(line l1, line l2) {
2 // also check coefficient c
3 return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }
```

intersección entre 2 lineas

3.2.3. vectores

3.2.4. círculos

ángulos en una circunferencia

3.2.5. triángulos

3.2.6. vectores

3.3. representación de un polígono

3.4. perímetro de un polígono

3.5. area de un polígono

3.6. comprobar si un punto esta dentro de un polígono

3.7. comprobar que un polígono es convexo

3.8. cortar un polígono con una linea recta

3.9. cubierta convexa

Capítulo 4

Estructura de datos

4.1. Descripción y Motivación

Una estructura de datos es la manera en la cual se organiza la información, por esta razon es posible que este capítulo sea el que más uses en tu vida cotidiana como programador.
Comencemos imaginando dos bibliotecas, la primera es muy estricta con sus reglas y todas las personas que leen un libro, deben regresarlo a su ubicación. En cambio la segunda biblioteca no tiene un orden, los libros estan regados por todas partes y las personas que los utilizan los dejan tirados donde sea. A primera vista pareciera que la segunda biblioteca no sirve para nada, pero en realidad si tu solo deseas ir a leer cualquier cosa y luego no tener que preocuparte de donde dejar el libro la segunda biblioteca seria ideal. A lo que quiero llegar es que hay distintas formas de ordenar la información, y algunas sirven para mejorar el desempeño en algunas areas sacrificando otras, no existe una estructura perfecta que haga bien todo al mismo tiempo.
Las principales operaciones sobre las estructuras son:

- Insertar
- Buscar
- Borrar
- Actualizar

Conocer las principales estructuras de datos y entender muy bien el problema al que nos enfrentemos seran la clave para idear una solución optima.

4.2. Complejidad

No es la intención de este libro dar una explicación detallada de lo que es la complejidad de algoritmos, solo daremos una descripcion por encima de la notación big O. Esta notación nos dice cuantas ejecuciones realizaria un algoritmo en el peor de los casos, por ejemplo si tenemos que buscar un libro dentro de la biblioteca desordenada, la complejidad seria big $O(n)$ siendo n la cantidad de libros, ya que en el peor de los casos tendriamos que buscar uno por uno hasta el último libro.

4.3. Estructuras de datos lineales

Una estructura de datos es considerada lineal si todos sus elementos estan organizados en linea, por ejemplo en un arreglo de izquierda a derecha.
En la mayoria de lenguajes de programación podemos distinguir entre arreglos estaticos y arreglos dinamicos, a los arreglos estaticos les definimos un tamaño y es inalterable, su ventaja es que acceder a un elemento conociendo su indice es instantaneo, por ejemplo

Listing 4.1: arregloEstatico.cpp

```
int main(){
    string palabras[] = {"hola","adios","tres"};
    cout<<palabras[2]<<endl;
}
```

Los arreglos comienzan con el indice 0 ciendo palabras[0] = “hola”, palabras[1] = “adios” y palabras[2]=“tres”. Los arreglos estaticos son muy utiles cuando sabemos exactamente el tamaño de elementos que usaremos, su complejidad en las diferentes operaciones es:

- Insertar/Actualizar $O(1)$ si conocemos la casilla donde insertaremos o actualizaremos, si no $O(n)$
- Buscar $O(1)$ (cuando conocemos el indice), si no $O(n)$
- Borrar $O(1)$ o $O(n)$ esta es una operacion complicada, ya que al borrar un elemento dejamos el espacio vacio, y lo más tipico seria correr todos los elementos de la derecha a la izquierda

Para entender un poco más esto imaginemos una estanteria de libros, donde solo caben 10 libros. Esta vacia y podemos empezar a meter libros donde queramos, pero si no tenemos un orden a la hora de ponerlos cuando esta más llena nos tomara más tiempo encontrar un espacio vacio, en cambio si vamos metiendo en orden siempre sabremos donde meter el proximo. La operación de buscar seria similar a agarrar el libro de la estanteria, si sabemos exactamente donde esta solo debemos tomarlo y ya, si no empezar a mirar uno por uno hasta encontrar el que buscamos, la operación de borrar es muy simple si solo quitamos el libro, pero hay dos cosas que podrian complicarla, la primera seria saber que libro quitaremos y la segunda si queremos que no quede el espacio vacio, pues nos tocaria correr todos los libros de la derecha hacia la izquierda para llenar el agujero. La operación de actualizar sera como una mezcla entre borrar e insertar.
Pero no nos asustemos, para usos prácticos es muy simple, solo usaremos arreglos estaticos para guardar información que recorreremos completa a menudo, por ejemplo si tenemos muchos amigos y a todos les queremos dar regalos:

Listing 4.2: arregloAmigos.cpp

```
int main(){
    string amigos[5] = {"ana","brian","cesar","daniel","eliana"};
    string regalos[3] = {"abrazo","reloj","perfume"};

    for(int i=0;i<5;i++){
        for(int j=0;j<3;j++){
            cout<<"le regalo un "<<regalos[j]<<" a "<<amigos[i]<<endl;
        }
    }
}
```

Los arreglos dinamicos son iguales a los estaticos, excepto por que pueden agrandarse todo lo que quieran (que lo soporte la RAM), otra gran diferencia es que ya trae por defecto la implementación de inserción y eliminación, esta estructura no permite huecos, por lo que su complejidad es la siguiente:

- Insertar $O(1)$
- Buscar $O(1)$ (cuando conocemos el indice), si no $O(n)$
- Borrar $O(n)$
- actualizar $O(1)$ (cuando conocemos el indice), si no $O(n)$

Como podemos observar, sus complejidades son muy efectivas, y por eso son muy usadas en la mayoria de las ocasiones, de hecho casi cualquier problema que requiera estructura de datos se puede solucionar aplicando esta estructura, solo que obviamente no siempre es la solución optima. Supongamos una

base de datos que solo usara arreglos, seria muy lenta y poco práctica. Un ejemplo de uso de arreglo dinámico es el siguiente:

Listing 4.3: arregloDinamicoAmigos.cpp

```
int main(){
    vector<string> amigos;
    vector<string> regalos;
    string amigo,regalo;
    cout<<"ingrese todos sus amigos, uno por uno , si ya termino ingrese 0"
       <<endl;
    while(cin>>amigo){
        if(amigo=="0")break;
        amigos.push_back(amigo);
    }
    cout<<"ingrese todos los regalos, uno por uno , si ya termino ingrese 0"
       <<endl;
    while(cin>>regalo){
        if(regalo=="0")break;
        regalos.push_back(regalo);
    }

    for(int i=0;i<amigos.size();i++){
        for(int j=0;j<regalos.size();j++){
            cout<<"le regalo un "<<regalos[j]<<" a "<<amigos[i]<<endl;
        }
    }
}
```

Generalmente la unica manera de conocer el indice del elemento que estamos buscando, es que queramos recorrer el arreglo, como lo hemos hecho en los ejemplos. Asi que en la mayoria de ocasiones cuando buscamos un elemento y no estamos recorriendo el arreglo la complejidad es de $O(n)$, pero podemos mejorar esto, ordenando el arreglo. Como en el ejemplo de la biblioteca tener la información ordenada nos permite encontrar las cosas más rapidamente, pero sacrificamos otras cosas a cambio. Como ya lo mencionamos en estructuras de datos no hay nada perfecto para todo, tenemos dos opciones. La primera es ordenar el arreglo antes de hacer la consulta, la otra es siempre tenerlo ordenado. Ordenar un arreglo no es una tarea fácil, por suerte la mayoria de lenguajes de programación nos provee herramientas para hacer esto, los mejores algoritmos de ordenamiento genericos tienen una complejidad de $O(n \log n)$, y buscar un elemento en un arreglo ordenado nos toma $O(\log n)$ por medio de busqueda binaria, la busqueda binaria funciona parandonos en la mitad, decidiendo si el elemento que buscamos se encuentra hacia la derecha o hacia la izquierda (lo sabemos por que estan ordenados) y repitiendo el proceso. Es ineficiente ordenar un arreglo para hacer una única búsqueda, pero se vuelve efectivo a partir de una cantidad, vamos a calcular en que momento se vuelve efectivo: S busquedas en un arreglo desordenado tiene una complejidad de $O(S \times n)$ y S busquedas en un arreglo ordenado tiene una complejidad de $O(n \log n + S \log n)$. Si igualamos y despejamos S, obtenemos $S = \frac{nlog(n)}{n-log(n)}$ por lo tanto si nuestra cantidad de busquedas es mayor a S, vale la pena ordenar el arreglo antes de realizarlas.