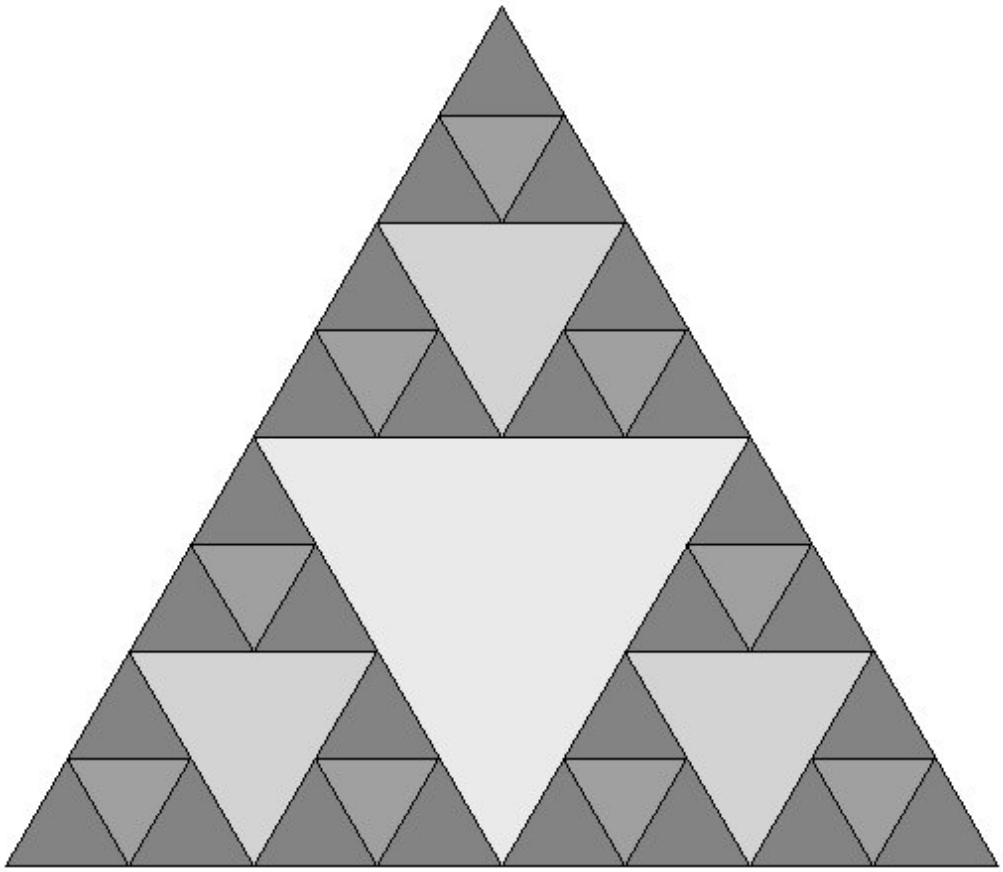
RECURSIÓN



Por Luis E. Vargas Azcona (Lobishomen)
Imagenes por Roberto López

Índice general

Capítulo 1

Introducción

La recursión es uno de los temas mas básicos en la Olimpiada de Informática. De hecho, muchas veces(no siempre) es el primer tema tratado en lo que se refiere a resolver problemas de tipo Olimpiada de Informática. Esto de "primer tema"puede parecer un poco confuso, ya que para llegar a recursión es necesario de antemano ya saber programar; y en muchos casos, al comenzar a ver recursión, ya se estudió y practicó el manejo de Karel. Entonces podrías estar preguntándote, ¿por qué primer tema si ya tuve que aprender Karel y un lenguaje de programación?.
La respuesta es sencilla: Resolver problemas no se limita a saber cómo escribir una solución en una computadora, hay que saber cómo encontrar la solución. El conocimiento de un lenguaje de programación se limita a expresiones lógicas para expresar las ideas en una computadora. Podría parecer para algunos una pérdida de tiempo leer acerca de cómo encontrar soluciones a problemas, y pueden también estarse preguntando ¿no basta con ser inteligente para poder encontrar la solución a un problema?. En teoría es cierto que cualquiera podría llegar a la solución de un problema simplemente entendiendo el problema y poniéndose a pensar.
Sin embargo, en la práctica, la gente tiende a comparar un problema con problemas que ya ha resuelto antes y de esa forma se resuelven los problemas mucho más ágilmente.
Por ejemplo, el matemático Rene Descartes no fue capaz de encontrar la solución al problema de encontrar una recta tangente a una curva; sin embargo, conociendo teoría de límites no parece tan difícil resolver el problema de la tangente a la curva. De hecho la solución a este problema(que recibe el nombre de derivada) es altamente conocida y aplicada en la actualidad.
De esta manera, pretender resolver un problema de olimpiada de cierto nivel sin haber resuelto nunca antes problemas mas fáciles resulta prácticamente imposible.
En el entendimiento claro y la buena aplicación de la recursión descansan las bases teóricas y prácticas de buena parte(casi me atrevería a decir que de la mayoría) de los algoritmos que se aprenden mas adelante.

No quiero decir con esto que las implementaciones recursivas sean la respuesta para todo ni para la mayoría; pero un razonamiento partiendo de una recursión es con mucha frecuencia utilizado para implementar algoritmos no recursivos.
Escribí este tutorial, al igual que los otros, para entrenar a participantes de la olimpiada de informática cuando no puedo hacerlo personalmente dándole el enfoque a lo que considero que es mas importante: resolver problemas.

Capítulo 2

Antes de empezar...

2.1. Requisitos Para Entender Este Tutorial

- Conocer algún lenguaje de programación(de preferencia C)
- Conocimientos básicos de álgebra

2.2. Un Poco de Notación

- Frecuentemente durante este tutorial se utilizarán puntos suspensivos(...) en algunas fórmulas. Estos puntos suspensivos deben de completar la fórmula con el patrón que se muestra. Por ejemplo:

  $1 + 2 + 3 + ... + 9 + 10$ significa $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$

- $n!$ se lee como *ene factorial* y

  $n! = (1)(1)(2)(3)(4)(5)...(n-1)(n)$

  Por ejemplo $5! = (1)(1)(2)(3)(4)(5) = 120$ y se lee como *cinco factorial*

- Una proposición es un enunciado que declara que algo es verdadero o falso(nunca ambas cosas a la vez).

- La expresión x_y significa x escrito en base y(si y no se especifica la base, se asume que es diez) Por ejemplo 101_2 significa el número binario 101, es decir, 5 en sistema decimal. Estos son los primeros números enteros positivos en sistema binario:

  $1_2, 10_2, 11_2, 100_2, 101_2, 110_2, 111_2, 1000_2, 1001_2, 1010_2, 1011_2, ...$

- El resto de la notación se especifica a lo largo del tutorial

2.3. Recomendaciones para Aprovechar Este Tutorial

Es muy común entre alumnos que comienzan en la olimpiada de informática, valorar más la habilidad de escribir código que las buenas ideas. En niveles un tanto mayores las ideas y no la habilidad de escribir código son lo que definen a los ganadores.
La solución para resolver un problema no es sólo una idea, requiere de muchas otras ideas y observaciones intermediarias para llegar a ella. Es por eso que la mayor parte de lo que viene en este tutorial no está relacionado directamente con la escritura de código en la computadora. Más sin embargo, tienen una gran utilidad en las "ideas intermedias"para llegar a la solución.
A lo largo de este tutorial aparecen muchos ejemplos, incluso si el lector ya entendió la idea sobre lo que tratan los ejemplos ¡son importantes!, el lector debe de intentar resolver los ejemplos por su propia cuenta y luego comparar la solución a la cual llegó con la solución escrita en el tutorial.
Si de pronto el lector no logra resolver un ejemplo, puede empezar a leer la solución para darse una idea y continuar por cuenta propia. Pero si no se intentan resolver los problemas por cuenta propia será imposible aprender de los errores, lo cual es muy importante.
Si encuentras algún error o algún término que no se defina en el tutorial a pesar de contar con los requisitos previos, manda un mensaje avisando a luison.cppgmail.com

Capítulo 3

Inducción Matemática

Primeramente, para aclarar posibles malentendidos, hay que avisar que la inducción no es necesariamente recursión. Pero, el buen entendimiento de la inducción puede ser un buen escalón para la recursión.Además, la inducción es un tema MUY importante(y desgraciadamente muy subestimado) en la Olimpiada de Informática y en algún momento es conveniente tratarlo; como la inducción sirve de escalón para la recursión el momento ideal para tratar este tema es ahora.
La inducción es una técnica muy útil para probar propiedades o resolver problemas de demostraciones.
Puede ser que de momento no se le encuentren aplicaciones prácticas en resolver problemas de olimpiada de informática. Sin embargo, en el momento de resolver problemas, es muy importante observar propiedades, y si no se está seguro si una propiedad es cierta, la inducción puede servir más de lo que te puedes imaginar.
Empezaremos por un ejemplo sencillo, luego pasaremos a ejemplos mas complejos para posteriormente definir formalmente la inducción matemática. También, aprovecharemos esta sección para poner como ejemplos algunas propiedades que luego serán muy útiles en el momento de resolver algunos problemas.

3.1. Ejemplos de Inducción

Ejemplo 1 *Prueba que para todo entero* $n \geq 1$

$$1 + 2 + 3 + 4 + 5 + ... + n - 1 + n = \frac{(n)(n+1)}{2}$$

Solución Si analizamos un poco la proposición para todo entero$n \geq 1, 1+2+3+4+5+...+n-1+n = \frac{(n)(n+1)}{2}$, ello quiere decir que dicha propiedad debe ser

cierta para $n = 1$, para $n = 2$, para $n = 3$, para $n = 4$, para $n = 5$, etc... Podemos comenzar verificando que es cierto para los primeros enteros positivos:

$$\frac{(1)(1+1)}{2} = 1, \frac{(2)(2+1)}{2} = 3 = 1+2, \frac{(3)(3+1)}{2} = 6 = 1+2+3, ...$$

Esta claro que si tratamos de aplicar este procedimiento para todos los números enteros positivos nunca acabaríamos a menos que encontráramos un número en el que no se cumpliera la proposición(lo cual no podemos asegurar).
Aquí muchos se sentirían tentados a pensar *esto es cierto porque funciona en todos los casos que verifiqué*, pues esa no es la salida correcta.
Existen varias razones, una de ellas es que muchas veces, lo que parece funcionar siempre, no siempre funciona, y los casos de prueba que se ponen en problemas de olimpiada están hechos para que no saquen puntos aquellos que *se van con la finta*.
Otra razón es que la solución completa de un problema(la solución de un problema incluye las comprobaciones de todas sus proposiciones) puede servir para descubrir propiedades que se aplican a muchos otros problemas. Y es ahí donde resolver un problema realmente sirve de algo.
Volviendo al tema, podemos transformar este problema de comprobar la proposición general en comprobar este conjunto infinito de proposiciones:

$$\frac{(1)(1+1)}{2} = 1, \frac{(2)(2+1)}{2} = 1+2, \frac{(3)(3+1)}{2} = 1+2+3, \frac{(4)(4+1)}{2} = 1+2+3+4, ...$$

Vamos a probar entonces:

- Que la primera proposición es verdadera.
- Que si una proposición es verdadera, la siguiente también lo es.

Probar que la primera proposición es verdadera es realmente fácil:

$$\frac{(1)(1+1)}{2} = \frac{(1)(2)}{2} = \frac{2}{2} = 1$$

Ahora, suponiendo que para alguna n

$$\frac{(n)(n+1)}{2} = 1+2+3+4+....+n-1+n$$

vamos a probar que:

$$\frac{(n+1)(n+2)}{2} = 1+2+3+4+....+n-1+n+n+1$$

Es decir, vamos a probar que si se cumple con n, se debe de cumplir con $n+1$.
Partimos de la ecuación inicial

$$\frac{(n)(n+1)}{2} = 1+2+3+4+...+n-1+n$$

Sumamos $n+1$ en ambos lados de la ecuación

$$\frac{(n)(n+1)}{2} + n + 1 = 1+2+3+4+...+n+n+1$$

$$\frac{(n)(n+1)}{2} + \frac{(2)(n+1)}{2} = 1+2+3+4+...+n+n+1$$

Y sumando por factor común se concluye que:

$$\frac{(n+2)(n+1)}{2} = 1+2+3+4+...+n+n+1$$

La proposición es correcta con 1, y si es correcta con 1 entonces será correcta con 2, si es correcta con 2 lo será con 3, si es correcta con 3 entonces lo será con 4, y así sucesivamente, podemos asegurar entonces que se cumple con todos los números enteros positivos.

Ejemplo 2 *En una fiesta hay n invitados, se asume que si un invitado conoce a otro, éste último conoce al primero. Prueba por inducción que el número de invitados que conocen a un número impar de invitados es par.*

Solución Nos encontramos ahora con un problema un poco mas abstracto que el anterior. ¿Cómo tratar este problema?.
Nuevamente imaginaremos el caso más simple que puede haber: una fiesta en la que nadie se conozca.
En este caso, ¿cuántos invitados conocen a un número impar de invitados?, la respuesta es obvia: ninguno, es decir, 0. Recordemos entonces que el 0 es par, si p es un número par $p+1$ es impar, y $p+2$ es par.
Ahora, si imaginamos en la fiesta que un dos personas se presentan mutuamente, habrá pues, 2 personas que conocen a un solo invitado y las demás no conocen a nadie. El número de invitados que conocen a un número impar de invitados será 2.
Nuevamente, si otras 2 personas que no se conocen entre sí se presentan mutuamente, entonces el número de invitados que conocen a un número impar de personas aumentará a 4.
Pero después de ello, ¿qué sucedería si una persona que conoce a un solo invitado se presenta con una persona que no conoce a nadie? La persona que no conoce a nadie conocería a un solo invitado, mientras que la persona que ya conocía a un solo invitado, pasará a conocer a dos invitados.
Nótese que una persona que conocía a un número impar de invitados(conocía a 1) pasó a conocer a un número par de invitados(acabó conociendo a 2), y la que conocía

a un número par de invitados(no conocía a ninguno) pasó a conocer a un número impar de invitados(acabó conociendo a 1) ¡el número de personas que conocen a un número impar de invitados no cambió!. Así entonces, olvidando las personas que se acaban de conocer, y solo sabemos que el número de personas que conocen a un número impar de invitados es par, si dos invitados que no se conocían, de pronto se conocieran, hay 3 posibilidades:

- Una persona conoce a un número impar de invitados y la otra conoce a un número par de invitados. Ambos aumentarán su número de conocidos en 1, el que conocía a un número impar de invitados pasará a conocer a un número par de invitados, el que conocía un número par de invitados, pasará a conocer un número impar de invitados. Por ello el número de personas que conocen a un número impar de invitados no cambia y sigue siendo par.

- Ambas personas conocen a un número par de invitados. En este caso, ambas personas pasarán a conocer a un número impar de invitados. El número de personas que conocen a un número impar de invitados aumenta en 2, por ello sigue siendo par.

- Ambas personas conocen a un número impar de invitados. En este caso, ambas personas pasarán a conocer un número par de invitados. El número de personas que conocen a un número impar de invitados disminuye en 2, por ello sigue siendo par.

Entonces, si al principio de sus vidas nadie se conoce, el número de personas que conocen a un número impar de invitados es par, y conforme se van conociendo, el número seguirá siendo par. Por ello se concluye que siempre será par.

Ejemplo 3 *Prueba que para todo entero* $n \geq 4$, $n! > 2^n$.

Solución Tal vez después de haber leído las soluciones de los dos primeros problemas te haya sido más fácil llegar a la solución de este.

$$4! = (1)(2)(3)(4) = 24$$
$$24 = (2)(2)(2)(2) = 16$$

Ya comprobamos que $4! > 24$, ahora, suponiendo que para alguna n

$$n! > 2^n$$

proseguimos a comprobar que

$$(n+1)! > 2^n + 1$$

Partiendo de la desigualdad inicial:

$$n! > 2^n$$

multiplicamos por $n+1$ el miembro izquierdo de la desigualdad y multiplicamos por 2 el lado derecho de la desigualdad. Como $n+1 > 2$, podemos estar seguros que el lado izquierdo de la desigualdad seguirá siendo mayor.

$$(n!)(n+1) > (2^n)2$$

Sintetizando:

$$(n+1)! > 2^n + 1$$

Ello implica que si se cumple para un número n, también se cumple para $n+1$, como se cumple para 4, entonces se cumplirá para 5, y si se cumple para 5, entonces se cumplirá para 6, etc. Esto indica entonces, que se cumplirá para todos los números enteros mayores o iguales a 4.

Ejemplo 4 *Prueba por inducción que para todo entero positivo n*

$$1^2 + 2^2 + 3^2 + 4^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Solución Nuevamente comenzaremos por el caso mas simple, cuando $n = 1$

$$\frac{1(1+1)(2+1)}{6} = \frac{6}{6} = 1$$

Ahora, suponiendo que para alguna n

$$1^2 + 2^2 + 3^2 + 4^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$$

veremos que sucede si se le suma $(n+1)^2$ a ambos miembros de la ecuación

$$\begin{aligned} 1^2 + 2^2 + ... + n^2 + (n+1)^2 &= n(n+1)(2n+1)/6 + 6(n+1)(n+1)/6 \\ 1^2 + 2^2 + ... + n^2 + (n+1)^2 &= (n+1)(n(2n+1) + 6n + 6)/6 \\ 1^2 + 2^2 + ... + n^2 + (n+1)^2 &= (n+1)(2n^2 + 7n + 6)/6 \\ 1^2 + 2^2 + ... + n^2 + (n+1)^2 &= (n+1)(n+2)(2(n+1)+1)/6 \end{aligned}$$

Nuevamente observamos que se cumple con 1, y si se cumple con n también se cumplirá con $n+1$ y por ello con todo número mayor n, por lo tanto se cumple con todos los enteros positivos.

Ejemplo 5 *Muestra que para cualquier cantidad de dinero mayor a 7 centavos puede ser formada usando solo monedas de 3 centavos y de 5 centavos.*

Solución La cantidad de 8 centavos puede ser formada con una moneda de 3 y una de 5; la cantidad de 9 centavos puede ser formada con 3 monedas de 9 centavos; la cantidad de 10 centavos puede ser formada con 2 monedas de 5 centavos.

Suponiendo que para algún entero positivo a, tal que $a > 7$ fuera posible formar las cantidades a, $a+1$ y $a+2$, si a cada una de esas cantidades se les añade una moneda de 3 centavos, resuelta que también será posible formar las cantidades $a+3$, $a+4$ y $a+5$, es decir, para cualquier tercia de números consecutivos, tal que el menor de ellos sea mayor que 7, la siguiente tercia también se podrá formar.

Con ello podemos asumir que cualquier tercia de números consecutivos se podrá formar y por ende, cualquier número entero positivo.

Ejemplo 6 *Prueba que si se tienen n personas, es posible elegir de entre* $2^n - 1$ *grupos de personas para hacer una marcha.*

Por ejemplo, con 3 personas A, B y C se pueden elegir 7 grupos:

$$\begin{aligned} A \\ B \\ C \\ A, B \\ A, C \\ B, C \\ A, B, C \end{aligned}$$

Solución Consideremos el caso de que solo hay una persona, con esta persona, solamente se puede elegir un grupo(de un solo integrante) para la marcha.

$$2^1 - 1 = 1$$

Ahora, suponiendo que con n personas se pueden elegir $2^n - 1$ grupos, para algún entero n, un desconocido va pasando cerca de las n personas y se une a ellas, puede formar parte de la marcha, o bien no formar parte.

Grupo que se podía formar con n personas habrá otros 2 grupos con $n + 1$ personas(uno en el caso de que el nuevo integrante participe y otro en el caso de que no participe).

Viendo que la marcha se puede crear solamente con la persona que acaba de llegar.Entonces con $n + 1$ personas habrá $2^{n+1} - 1$ grupos que se pueden elegir.

Con ello queda demostrada la proposición.

3.2. Errores Comunes

A veces al estar intentando probar algo por inducción se llega a caer en ciertos errores, por ejemplo:

Proposición Todos los perros son del mismo color

Pseudo-prueba Si tenemos un solo perro, es trivial que es de su propio color; ahora, si tenemos n perros del mismo color y un perro es negro, por construcción los demás también son negros. Por lo tanto queda probado.

Error Aquí se está asumiendo que se tienen n perros negros, pero no se verifica que si se tienen n perros negros, y se añade uno, el siguiente también será negro; en efecto, puede no ser negro.
Está bastante claro que esta *prueba* es incorrecta, sin embargo errores similares, pero menos obvios pueden suceder en la práctica.
Siempre hay que verificar que si un caso se cumple el siguiente también lo hará, así como verificar el caso mas pequeño.

3.3. Definición de Inducción

Después de estos ejemplos le será posible al lector abstraer lo que tienen en común y así hacerse de una idea clara de lo que es la inducción.

Definición 1 (Inducción) *Método para comprobar la relación entre 2 funciones* $f(n) = g(n)$*, comprobando que* $f(0) = g(0)$*, y que* $f(n-1) = g(n-1)$ *implica* $f(n) = g(n)$*.*

O dicho de una manera mas coloquial, la inducción es un método para comprobar una proposición matemática basándose en la verificación de la forma mas simple de la proposición y luego comprobando que una forma compleja de la proposición se cumple siempre y cuando una forma mas simple se cumpla.

Capítulo 4

Definición y Características de la Recursión

Ya vimos con la inducción que a veces una proposición se puede verificar en su forma más simple y luego probar que si se cumple para una de cierta complejidad también se cumplirá para otra con más complejidad.
Ahora, la recursividad o recursión se trata de algo parecido, se trata de definir explícitamente la forma más simple de un proceso y definir las formas mas complejas de dicho proceso en base a formas un poco más simples.
Wikipedia define la recursividad de la siguiente manera:

Definición 2 (Recursión) *Forma en la cual se especifica un proceso basado en su propia definición. Siendo un poco más precisos, y para evitar el aparente círculo sin fin en esta definición, las instancias complejas de un proceso se definen en términos de instancias más simples, estando las finales más simples definidas de forma explícita.*

4.1. Factorial

Recordemos la función factorial.
Sabemos que $n! = (1)(1)(2)(3)(4)...(n)$
Pero puede parecer incómodo para una definición seria tener que usar los puntos suspensivos(...).
Por ello, vamos a convertir esa definición en una definición recursiva.
La forma mas simple de la función factorial es:

$$0! = 1$$

Y ahora, teniendo $n!$, ¿Cómo obtenemos $(n+1)!$?, simplemente multiplicando $n!$

por $n+1$.

$$(n+1)! = (n!)(n+1)$$

De esta forma especificamos cada que se tenga un número, cómo obtener el siguiente y la definición queda completa, ésta ya es una definición recursiva, sin embargo, se ve un tanto sucia, puesto que define a $(n+1)!$ en términos de $n!$. Y para ir más de acuerdo con la idea de la recursividad hay que definir $n!$ en términos de $(n-1)!$. Así que, teniendo $(n-1)!$, para obtener $n!$ hay que multiplicar por n :). De esa forma nuestra definición recursiva queda así:

$$0! = 1 n! = (n-1)! n$$

Para quienes comiencen a pensar lo contrario, el autor de este tutorial no se ha olvidado que es para la olimpiada de informática ;). Al igual que las matemáticas, una computadora también soporta funciones recursivas, por ejemplo, la función factorial que acaba de ser definida puede ser implementada en lenguaje C de la siguiente manera:

Código 1: Función recursiva factorial

```
(1)    int factorial(int n){
(2)        if(n == 0) return 1;
(3)        else return factorial(n-1)*n;
(4)    }
```

Para los estudiantes de programación que comienzan a ver recursión suele causar confusión ver una función definida en términos de sí misma. Suele causar la impresión de ser una definición cíclica. Pero ya que construimos la función paso a paso es posible que ello no le ocurra al lector :). Pero de todas formas es buena idea ver cómo trabaja esta función en un depurador o ir simulándola a mano. Podemos notar algunas cosas interesantes:

- Si llamamos a factorial(5), la primera vez que se llame a la función factorial, n será igual a 5, la segunda vez n será igual a 4, la tercera vez n será igual a 3, etc.
- Después de que la función regresa el valor 1, n vuelve a tomar todos los valores anteriores pero esta vez en el orden inverso al cual fueron llamados, es decir, en lugar de que fuera 5, 4, 3, 2, 1, ahora n va tomando los valores 1, 2, 3, 4, 5; conforme el programa va regresando de la recursividad va multiplicando el valor de retorno por los distintos valores de n.
- Cuando la función factorial regresa por primera vez 1, la computadora recuerda los valores de n, entonces guarda dichos valores en algún lado.

La primera observación parece no tener importancia, pero combinada con la segunda se vuelve una propiedad interesante que examinaremos después. La tercera observación hace obvia la necesidad de una estructura para recordar, los valores de las variables cada que se hace una llamada a función. Dicha estructura recibe el nombre de pila o stack. Y como toda estructura de datos, la pila requiere memoria de la computadora y tiene un límite de memoria a ocupar(varía mucho dependiendo del compilador y/o sistema operativo), si dicho límite es superado, el programa terminará abruptamente por acceso a un área restringida de la memoria, este error recibe el nombre de desbordamiento de pila o stack overflow(ahora ya sabes qué quería decir esa pantalla azul ;)).

A veces, al abordar un problema, es mejor no pensar en la pila dado que el tamaño de la pila nunca crecerá mucho en ese problema; otras veces hay que tener cuidado con eso; y en ocasiones, tener presente que la pila recuerda todo, puede ayudar a encontrar la solución a un problema. Para observar mas claramente las propiedades mencionadas, conviene echar un vistazo a esta versión modificada de la función factorial:

Código 2: Función recursiva factorial modificada

```
(1)    int factorial(int n){
(2)        int fminus1;
(3)        printf(" %d \n", n); if(n == 0) return 1;
(4)        fminus1=factorial(n-1);
(5)        printf(" %d %d\n", n, fminus1);
(6)        return fminus1*n;
(7)    }
```

Si llamas factorial(5) con el código de arriba obtendrás la siguiente salida en pantalla:

5
4
3
2
1
0
11
21
32
46
524

Allí se ve claramente cómo n toma los valores en el orden inverso cuando va regresando de la recursividad y como la línea 3 se ejecuta 6 veces mientras se van haciendo las llamadas recursivas, y la linea 6 se ejecuta ya se hicieron todas las llamadas recursivas, no antes.

4.2. Imprimir Números Binarios

Con la función factorial ya vimos varias propiedades de las funciones recursivas cuando se ejecutan dentro de una computadora.
Aunque siendo sinceros, es mucho mas práctico calcular el factorial con un for que con una función recursiva.
Ahora, para comenzar a aprovechar algunas ventajas de la recursividad se tratará el problema de imprimir números binarios, el cual es fácilmente extensible a otras bases.

Ejemplo 7 *Escribe una función recursiva que imprima un número entero positivo en su formato binario(no dejes ceros a la izquierda).*

Solución La instancia mas simple de este proceso es un número binario de un solo dígito(1 ó 0). Cuando haya mas de un dígito, hay que imprimir primero los dígitos de la izquierda y luego los dígitos de la derecha.
También hay que hacer notar algunas propiedades de la numeración binaria:

- Si un número es par, termina en 0, si es impar termina en 1.
  Por ejemplo $101_2 = 5$ y $100_2 = 4$

- Si un número se divide entre 2(ignorando el residuo), el cociente se escribe igual que el dividendo, simplemente sin el último número de la derecha.

Por ejemplo $\frac{10011001_2}{10_2} = 1001100_2$
Tomando en cuenta todo lo anterior concluimos que si $n < 2$ entonces hay que imprimir n, sino hay que imprimir $n/2$ y luego imprimir 1 si n es impar y 0 si n es par. Por lo tanto, la función recursiva quedaría de esta manera:

Código 3: Función recursiva que Imprime un Número Binario

```
(1)    void imprime_binario(int n){
(2)        if(n>=2){
(3)            imprime_binario(n/2);
(4)            printf(" %d", n%2);
(5)        }else{
(6)            printf(" %d", n);
(7)        }
(8)    }
```

Vemos aquí que esta función recursiva si supone una mejoría en la sencillez en contraste si se hiciera iterativamente(no recursivo), ya que para hacerlo iterativamente se necesitaría llenar un arreglo y en cada posición un dígito, y luego recorrerlo en el orden inverso al que se llenó. Pero hay que hacer notar que un proceso recursivo funciona ligeramente mas lento que si se hiciera de manera iterativa. Esto es porque las operaciones de añadir elementos a la pila y quitar elementos de la pila toman tiempo. Con esto se concluye que la recursión hay que usarla cuando simplifique sustancialmente las cosas y valga la pena pagar el precio de un poco menos de rendimiento.

4.3. Los Conejos de Fibonacci

Había una vez cierto matemático llamado Leonardo de Pisa, apodado Fibonacci, que propuso el siguiente problema:

Ejemplo 8 *Alguien compra una pareja de conejos(un macho y una hembra), luego de un mes de haber hecho la compra esos conejos son adultos, después de dos meses de haber hecho la compra esa pareja de conejos da a luz a otra pareja de conejos(un macho y una hembra), al tercer mes, la primera pareja de conejos da a luz a otra pareja de conejos y al mismo tiempo, sus primeros hijos se vuelven adultos.*
Cada mes que pasa, cada pareja de conejos adultos da a luz a una nueva pareja de conejos, y una pareja de conejos tarda un mes en crecer. Escribe una función que regrese cuántos conejos adultos se tienen pasados n meses de la compra.

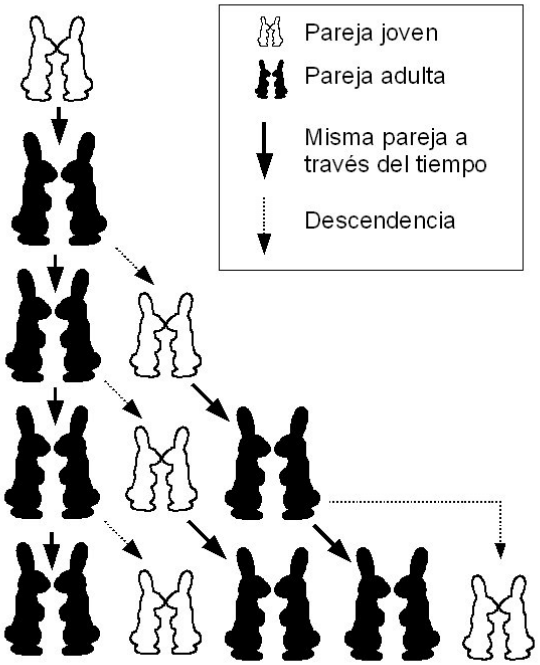


Figura 4.1: Reproducción de los Conejos de Fibonacci

Solución Sea $F(x)$ el número de parejas de conejos adultos pasados x meses. Podemos ver claramente que pasados 0 meses hay 0 parejas adultas y pasado un mes hay una sola pareja adulta. Es decir $F(0) = 0$ y $F(1) = 1$. Ahora, suponiendo que para alguna x ya sabemos $F(0)$, $F(1)$, $F(2)$, $F(3)$, ..., $F(x-1)$, en base a eso ¿cómo podemos averiguar $F(x)$? Si en un mes se tienen a parejas jóvenes y b parejas adultas, al siguiente mes se tendrán a+b parejas adultas y b parejas jóvenes. Por lo tanto, el número de conejos adultos en un mes n, es el número de conejos adultos en el mes n-1 más el número de conejos jóvenes en el mes n-1. Como el número de conejos jóvenes en el mes n-1 es el número de conejos adultos en el mes n-2, entonces podemos concluir que:

$$F(0) = 0 F(1) = 1 F(n) = F(n-1) + F(n-2)$$

Como curiosidad matemática, posiblemente alguna vez leas u oigas hablar sobre la serie o sucesión de Fibonacci, cuando suceda, ten presente que la serie de Fibonacci es

$$F(0), F(1), F(2), F(3), F(4), F(5), ...$$

O escrita de otra manera:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$$

Además de servir como solución a este problema, la serie de Fibonacci cumple también con muchas propiedades interesantes que pueden servir para resolver o plantear otros problemas. Por el momento no se mostrarán aquí, ya que ese no es el objetivo del tutorial, pero si te interesa investiga ;).
El siguiente código en C muestra una implementación de una función recursiva para resolver el problema planteado por Fibonacci:

Código 4: Función recursiva de la serie de Fibonacci

```
(1)    int F(int n){
(2)        if(n==0){
(3)            return 0;
(4)        }else if(n==1){
(5)            return 1;
(6)        }else{
(7)            return F(n-1)+F(n-2);
(8)        }
(9)    }
```

Si lo corres en una computadora, te darás cuenta que el tamaño de los números crece muy rápido y con números como 39 o 40 se tarda mucho tiempo en responder, mientras que con el número 50 parece nunca terminar.
Hemos resuelto matemáticamente el problema de los conejos de Fibonacci, sin embargo, en la olimpiada de informática hay tiempo límite de ejecución, ¡el cual casi siempre está entre 1 y 3 segundos!.

Capítulo 5

Recursión con Memoria o Memorización

La recursión con Memoria o Memorización es un método para evitar que una misma función recursiva se calcule varias veces ejecutándose bajo las mismas condiciones; consiste en tener en una estructura(por lo general un arreglo de una o varias dimensiones) para guardar los resultados ya calculados.

5.1. Mejorando el Rendimiento de Fibonacci

Ejemplo 9 *Recordando, la sucesión de Fibonacci se define como*

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

Escribe un programa que calcule e imprima los primeros 50 números de la serie de Fibonacci en menos de un segundo.

Solución Ya habíamos visto en la sección anterior que la función recursiva de Fibonacci(véase código 4) es extremadamente lenta, vale la pena preguntarnos ¿Qué la hace lenta?.
Si nos ponemos a ver su ejecución en un depurador o la realizamos mentalmente, nos daremos cuenta que para calcular $F(n)$, se está calculando $F(n-1)$ y $F(n-2)$, y para calcular $F(n-1)$ también se está calculando $F(n-2)$.
Una vez que se terminó de calcular $F(n-1)$ ya se había calculado $F(n-2)$, y sin embargo, se vuelve a calcular. Con ello, ya hemos visto que $F(n-2)$ se esta calculando 2 veces. Extrapolando este razonamiento $F(n-4)$ se está calculando al menos 4 veces, $F(n-6)$ se esta calculando al menos 8 veces, etc.

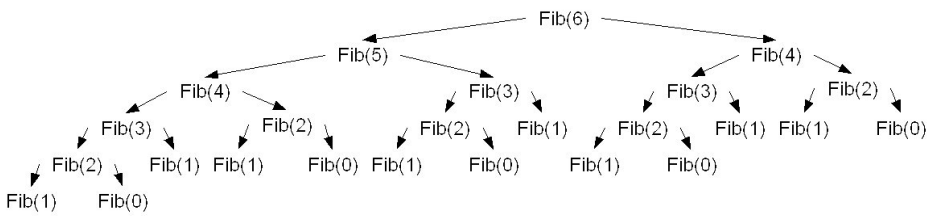


Figura 5.1: Llamadas a función que se realizan para calcular $F(6)$

¿Exactamente de cuántas veces estamos hablando que se calcula cada número de Fibonacci?. Por el momento no vale la pena ver eso, es suficiente con saber que es un número exponencial de veces.
Para solucionar esta problemática, vamos a declarar un arreglo v de tamaño 50 y con el tipo de datos long long, ya que a nadie le extrañaría que $F(49) > 2^32$.
Luego, cada que se ejecute la función $F(n)$, verificará si $v[n]$ ya fue usado(los valores de v se inicializan en 0, ya que es un arreglo global), si ya fue usado simplemente regresará $v[n]$ si no ha sido usado entonces calculará el valor de $F(n)$ y lo guardará en $v[n]$.
Así concluimos nuestros razonamientos para llegar a este código:

Código 5: Fibonacci utilizando recursión con memoria

```
(1)   #include <stdio.h>
(2)   int v[50];
(3)   int F(int n){
(4)      if(v[n]!=0){
(5)         return v[n];
(6)      }if(n==0){
(7)         return 0;
(8)      }if(n==1){
(9)         return 1;
(10)     }v[n]=F(n-1)+F(n-2);
(11)     return v[n];
(12)  }
(13)  int main(){
(14)     int i;
(15)     for(i=0;i<50;i++){
(16)        printf(" %lld\n", F(i));
(17)     }return 0;
(18)  }
```

Aunque este código es notablemente mas largo que el código 4 solamente las líneas

de la 2 a la 12 son interesantes, lo demás es solamente para imprimir en pantalla el resultado.
Si ejecutas esta nueva función $F(n)$ en una computadora te darás cuenta que es mucho mas rápida que la función $F(n)$ que se muestra en el código 4.

5.2. Error Común en la Memorización

La recursión con memoria solamente sirve cuando la función recursiva que se quiere optimizar no utiliza los valores de variables estáticas ni globales para calcular su resultado; de lo contrario si la función es llamada con los mismos parámetros no se garantiza que se esté calculando exactamente lo mismo mas de una vez y la recursión con memoria no sirve en ese caso.
Es común para alguien que va empezando en la recursión con memoria cometer este tipo de errores por ello hay que tomar precauciones adicionales.

5.3. Triangulo de Pascal

Probablemente alguna vez viste este triángulo:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
.....

Este triángulo se conoce como Triángulo de Pascal, la manera de construirlo es bastante simple, se coloca un 1 en el primer renglón, en cada renglón a partir del segundo, se forman números sumando cada par de números adyacentes del renglón anterior(es decir, cada número es la suma de los dos números que tiene arriba) y se coloca un 1 en cada extremo del renglón.
Vamos a llamar al j-ésimo número del i-ésimo renglón $P(i, j)$.

Ejemplo 10 *Escribe una función recursiva que regrese $P(i, j)$, para $i < 50$ y $j < 50$ y $P(i, j) < 2^31$.*

Primero hay que notar que $P(1, 1) = 1$, luego que el primer numero de cada renglón también es 1; es decir $P(i, 1) = 1$, y el último número de cada renglón también es 1, osea $P(i, i) = 1$; en otro caso es la suma de los dos números que tiene arriba, es

decir $P(i,j) = P(i-1,j-1) + P(i-1,j)$. De esta manera completamos la función recursiva así:

$$P(i,1) = 1$$

$$P(i,i) = 1$$

$$P(i,j) = P(i-1,j-1) + P(i-1,j) \text{ para } 1 < j < i$$

Aquí se ignoró $P(1,1) = 1$ ya que $P(i,1) = 1$ implica $P(1,1) = 1$ Nuevamente utilizaremos un arreglo llamado v para guardar los resultados ya calculados, y ya que ningún resultado puede ser 0, cuando $v[i][j]$ sea 0 sabremos que no se ha calculado aún. Ahora el código de la función en C resulta así:

Código 6: Triangulo de Pascal utilizando Recursión con Memoria

```
(1)  int v[51][51];
(2)  int P(int i, int j){
(3)      if(j==1 || i==j){
(4)          return 1;
(5)      }if(v[i][j]!=0){
(6)          return v[i][j];
(7)      }v[i][j]=P(i-1, j-1)+P(i-1, j);
(8)      return v[i][j];
(9)  }
```

Ejemplo 11 *Sea* $\binom{m}{n}$ *las combinaciones de* m *en* n*(la cantidad de formas de escoger* m *objetos entre un total de* n *objetos distintos), prueba que* $\binom{j-1}{i-1} = P(i,j)$.

Solución Recordemos cómo se calcula P:

$$P(i,1) = 1$$

$$P(i,i) = 1$$

$$P(i,j) = P(i-1,j-1) + P(i-1,j) \text{ para } 1 < j < i$$

De esa manera obtenemos 3 proposiciones:

- $P(i,1) = 1$ implica $\binom{0}{i-1} = 1$, es decir, el número de formas de elegir 0 objetos entre un total de $i-1$ objetos distintos es 1.
- $P(i,i) = 1$ implica $\binom{i-1}{i-1} = 1$, es decir, el número de formas de elegir $i-1$ objetos entre un total de $i-1$ objetos distintos es 1.

- $P(i,j) = P(i-1,j-1)+P(i-1,j)$ implica $\binom{j-1}{i-1} = \binom{j-2}{i-2}+\binom{j-1}{i-2}$ para $1 < j < i$ esta proposición es equivalente a:

$$\binom{j}{i} = \binom{j-1}{i-1} + \binom{j}{i-1}$$

La primera proposición es obvia, solo hay una manera de elegir 0 objetos entre un total de $i-1$ objetos distintos, y ésta es no eligiendo ningún objeto.
La segunda proposición también es obvia, la única manera de elegir $i-1$ objetos de entre un total de $i-1$ objetos es eligiendo todos.
La tercera proposición es más difícil de aceptar, digamos que se tienen n objetos numerados de 1 a n; y se quieren elegir m objetos entre ellos, en particular, se puede optar entre elegir el objeto n o no elegir el objeto n.
El número de formas de elegir m objetos de entre un total de n objetos distintos, es el número de formas de hacer eso eligiendo al objeto n más el número de formas de hacer eso sin elegir al objeto n.
Si se elige el objeto n, entonces habrá que elegir $m-1$ objetos de entre un total de $n-1$ objetos distintos; hay $\binom{m-1}{n-1}$ formas distintas de hacerlo.
Si no se elige el objeto n, entonces habrá que elegir m objetos de entre un total de $n-1$ objetos distintos(si ya se decidió no elegir a n, entonces quedan $n-1$ objetos que pueden ser elegidos); hay $\binom{m}{n-1}$ formas de hacerlo.
Por ello concluimos que $\binom{m}{n} = \binom{m-1}{n-1} + \binom{m}{n-1}$ lo que prueba tercera proposición.

5.4. Teorema del Binomio

El triángulo de Pascal tiene muchas propiedades interesantes, una de ellas esta estrechamente relacionada con el teorema del binomio; el cual se le atribuye a Newton. Para dar una idea de a qué se refiere el teorema del binomio; basta ver los coeficientes de las siguientes ecuaciones:

$$(a+b)^0 = 1$$
$$(a+b)^1 = a+b$$
$$(a+b)^2 = a^2+2ab+b^2$$
$$(a+b)^3 = a^3+3a^2b+3ab^2+b^3$$
$$(a+b)^4 = a^4+4a^3b+6a^2b^2+4ab^3+b^4$$
$$(a+b)^5 = a^5+5a^4b+10a^3b^2+10a^2b^3+5ab^4+b^5$$

Después de ver los coeficientes es natural que a uno le llegue a la mente el triángulo de Pascal.

En efecto, el teorema del binomio dice que para cualquier entero no negativo n:

$$(a+b)^n = \binom{0}{n}a^n b^0 + \binom{1}{n}a^{n-1}b^1 + \binom{2}{n}a^{n-2}b^2 + \binom{3}{n}a^{n-3}b^3 + ... + \binom{n}{n}a^0 b^n$$

Ejemplo 12 *Comprueba el teorema del binomio*

Solución Ya vimos que para n=0 si se aplica el teorema. Ahora, suponiendo que para algún número n se aplica el teorema ¿se aplicará para $n+1$? Por construcción vamos suponiendo que para alguna n:

$$(a+b)^n = \binom{0}{n}a^n b^0 + \binom{1}{n}a^{n-1}b^1 + \binom{2}{n}a^{n-2}b^2 + \binom{3}{n}a^{n-3}b^3 + ... + \binom{n}{n}a^0 b^n$$

Multiplicamos ambos miembros de la ecuación por $(a+b)$

$$((a+b)^n)(a+b) = (\binom{0}{n}a^n b^0 + \binom{1}{n}a^{n-1}b^1 + \binom{2}{n}a^{n-2}b^2 + ... + \binom{n}{n}a^0 b^n)(a+b)$$

$$(a+b)^{n+1} = \binom{0}{n}a^{n+1}b^0 + (\binom{1}{n} + \binom{0}{n})a^n b^1 + (\binom{2}{n} + \binom{1}{n})a^{n-1}b^2 + ... + \binom{n}{n}a^0 b^{n+1}$$

Recordamos que $\binom{m}{n} = \binom{m-1}{n-1} + \binom{m}{n-1}$, por lo tanto por inducción el teorema queda comprobado.
El Triangulo de Pascal aún tiene muchas más propiedades interesantes, pero por el momento esto es todo lo que hay en el tutorial, si algún lector se quedó intrigado por este famoso triángulo puede buscar en internet y encontrará muchas cosas :D.

Capítulo 6

Divide y Vencerás

ésta célebre frase para estrategias de guerra ha llegado a ser bastante popular en el campo de las matemáticas y sobre todo, en el de las matemáticas aplicadas a la computación.
La estrategia *Divide y Vencerás* se define de una manera bastante simple:
Divide un problema en partes mas pequeñas, resuelve el problema por las partes, y combina las soluciones de las partes en una solución para todo el problema.
En la olimpiada de informática es difícil encontrar un problema donde no se utilice ésta estrategia de una u otra forma.
Sin embargo, aquí se tratará exclusivamente de la estrategia Divide y Vencerás en su forma recursiva:

- *Divide*. Un problema es dividido en copias mas pequeñas del mismo problema.
- *Vence*. Se resuelven por separado las copias mas pequeñas del problema. Si el problema es suficientemente pequeño, se resuelven de la manera mas obvia.
- *Combina*. Combina los resultados de los subproblemas para obtener la solución al problema original.

La dificultad principal en resolver este tipo de problemas radica un poco en cómo dividirlos en copias mas pequeñas del mismo problema y sobre todo cómo combinarlos.
A veces la estrategia recursiva Divide y Vencerás mejora sustancialmente la eficiencia de una solución, y otras veces sirve solamente para simplificar las cosas.
Como primer ejemplo de Divide y Vencerás veremos un problema que puede resultar mas sencillo resolverse sin recursión, pero esto es solo para dar una idea de cómo aplicar la estrategia.

6.1. Máximo en un Arreglo

Ejemplo 13 *Escribe una función que dado un arreglo de enteros v y dados dos enteros a y b, regrese el número mas grande en $v[a..b]$(el número mas grande en el arreglo que esté entre los índices a y b, incluido este último).*

Solución Lo mas sencillo sería iterar desde a hasta b con un for, guardar el máximo en una variable e irla actualizando.
Pero una forma de hacerlo con Divide y Vencerás sería:

- Si $a < b$, dividir el problema en encontrar el máximo en $v[a..(a+b)/2]$ y el máximo en $v[(a+b)/2+1..b]$, resolver ambos problemas recursivamente.
- Si $a = b$ el máximo sería $v[a]$
- Una vez teniendo las respuestas de ambos subproblemas, ver cual respuesta de las dos respuestas es mayor y regresar esa.

Los 3 puntos de éste algoritmo son los 3 puntos de la estrategia Divide y Vencerás aplicados.
Esta claro que este algoritmo realmente encuentra el máximo, puesto que en $v[a..b]$ está compuesto por $v[a..(a+b)/2]$ y $v[(a+b)/2+1..b]$, sin quedar un solo número del intervalo excluido. También sabemos que si un número $x > y$ y $y > z$, entonces $x > z$, por ello podemos estar seguros que alguno de los dos resultados de los subproblemas debe de ser el resultado al problema original.
Para terminar de resolver este problema, hay que escribir el código:

Código 7: Máximo en un intervalo $[a, b]$

```
(1)   int maximo(int v[], int a, int b){
(2)       int maximo1, maximo2;
(3)       if(a<b){
(4)           maximo1=maximo(v, a, (a+b)/2);
(5)           maximo2=maximo(v, (a+b)/2+1, b);
(6)           if(maximo1>maximo2){
(7)               return maximo1;
(8)           }else{
(9)               return maximo2;
(10)          }
(11)      }else{
(12)          return v[a];
(13)      }
(14)  }
```

6.2. Búsqueda Binaria

Ahora, después de haber visto un ejemplo impráctico sobre el uso de Divide y Vencerás, veremos un ejemplo práctico.
Supongamos que tenemos un arreglo v con los números ordenados de manera ascendente. Es decir, $v[a] > v[a-1]$ y queremos saber si un número x se encuentra en el arreglo, y de ser así, ¿dónde se encuentra?
Una posible forma sería iterar desde el principio del arreglo con un for hasta llegar al final y si ninguno de los valores es igual a x, entonces no está, si alguno de los valores es igual a x, guardar dónde está.
¿Pero qué sucedería si quisiéramos saber un millón de veces dónde está algún número(el número puede variar)? Como ya te lo podrás imaginar, el algoritmo anterior repetido un millón de veces se volverá lento.
Es como si intentáramos encontrar el nombre de un conocido en el directorio telefónico leyendo todos los nombres de principio a fin a pesar de que están ordenados alfabéticamente.

Ejemplo 14 *Escribe una función que dado un arreglo ordenado de manera ascendente, y tres enteros a, b y x, regrese -1 si x no está en v[a..b] y un entero diciendo en qué índice se encuentra x si lo está. Tu programa deberá no deberá hacer mas de 100 comparaciones y puedes asumir que $b-a<1000000$.*

Solución La solución a este problema se conoce como el algoritmo de la búsqueda binaria.
La idea consiste en ver qué número se encuentra a la mitad del intervalo $v[a..b]$. Si $v[(a+b)/2]$ es menor que x, entonces x deberá estar en $v[(a+b)/2+1..b]$, si $v[(a+b)/2]$ es mayor que x, entonces x deberá estar en v$[a..(a+b)/2]$.
En caso de que $a=b$, si $v[a]=x$, entonces x se encuentra en a.
Así que, los 3 pasos de la estrategia Divide y Vencerás con la búsqueda binaria son los siguientes:

- Si $b > a$, comparar $v[(a+b)/2]$ con x, si x es mayor entonces resolver el problema con $v[(a+b)/2+1..b]$, si x es menor resolver el problema con $v[a..(a+b)/2-1]$, y si x es igual, entonces ya se encontró x.

- Si $b \geq a$, comparar $v[a]$ con x, si x es igual entonces se encuentra en a, si x es diferente entonces x no se encuentra.

- Ya sabiendo en qué mitad del intervalo puede estar, simplemente hay que regresar el resultado de ese intervalo.

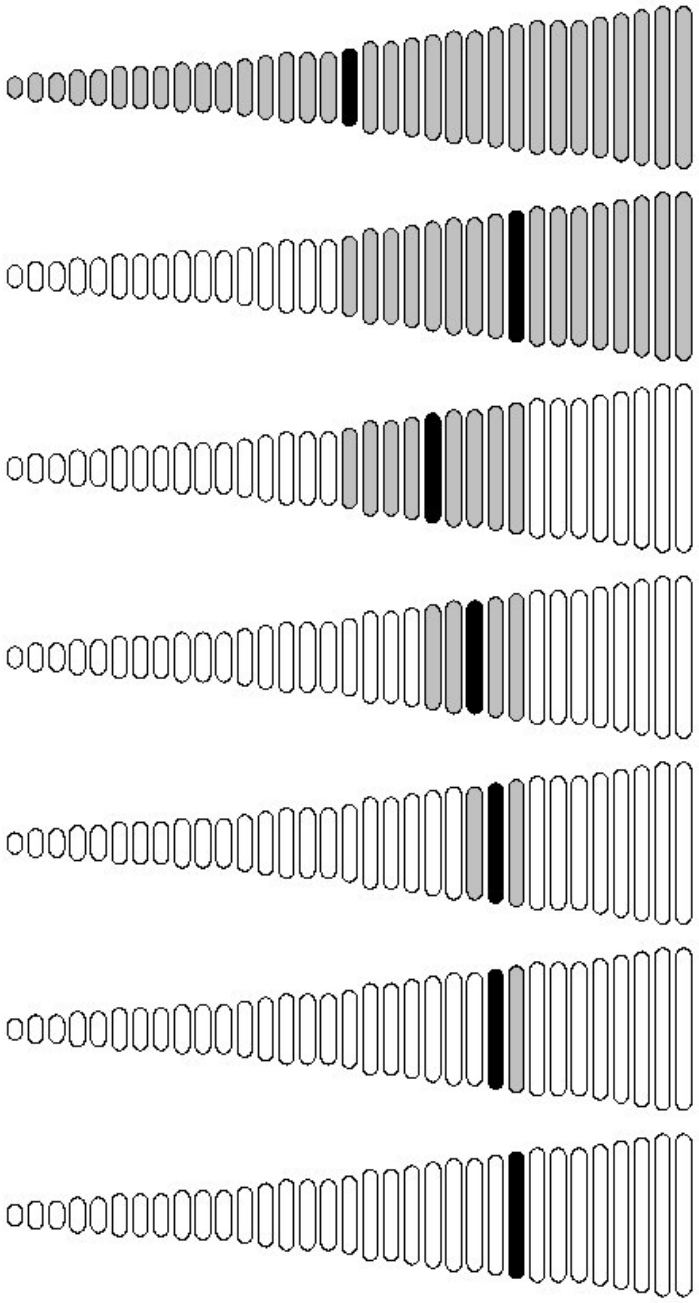


Figura 6.1: Rendimiento de la Búsqueda Binaria. El espacio de búsqueda(intervalo donde puede estar la solución) esta marcado con gris, y el valor que se esta comparando está marcado con negro.

Código 8: Búsqueda Binaria Recursiva

```
(1)   int Busqueda_Binaria(int v[], int a, int b, int x){
(2)       if(a>=b){
(3)           if(v[a]==x)
(4)           return a;
(5)           else
(6)           return -1;
(7)       }
(8)       if(v[(a+b)/2]==x){
(9)           return (a+b)/2;
(10)      }else if(v[(a+b)/2]<x){
(11)          return Busqueda_Binaria(v, (a+b)/2+1, b, x);
(12)      }else{
(13)          return Busqueda_Binaria(v, a, (a+b)/2-1, x);
(14)      }
(15)  }
```

A pesar de que la idea de la búsqueda binaria es puramente recursiva, también se puede eliminar por completo la recursión de ella:

Código 9: Búsqueda Binaria Iterativa

```
(1)   int Busqueda_Binaria(int v[], int a, int b, int x){
(2)       while(a<b)
(3)       if(v[(a+b)/2]==x){
(4)           return (a+b)/2;
(5)       }else if(v[(a+b)/2]<x){
(6)           a=(a+b)/2+1;
(7)       }else{
(8)           b=(a+b)/2-1;
(9)       }
(10)      if(v[a]==x){
(11)          return a;
(12)      }else{
(13)          return -1;
(14)      }
(15)  }
```

A pesar de que los códigos están del mismo tamaño, muchas veces resulta mas práctica la iterativa, ya que no es necesario declarar una nueva función para realizarla.

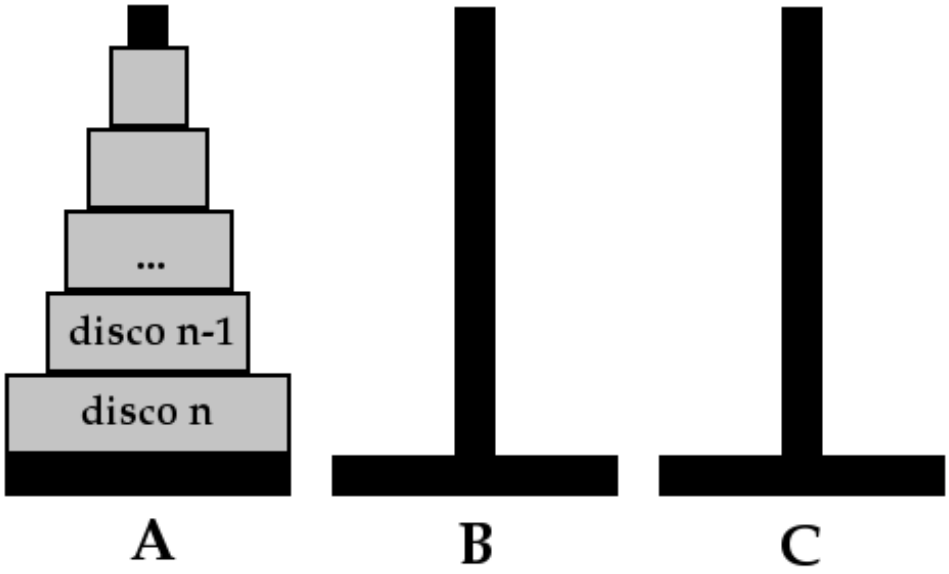


Figura 6.2: Tórres de Hanoi $F(6)$

6.3. Torres de Hanoi

El problema de las Torres de Hanoi es un problema utilizado frecuentemente como ejemplo de recursión.
Imagina que tienes 3 postes llamados A, B y C.
En el poste A tienes n discos de diferente diámetro, acomodados en orden creciente de diámetro desde lo más alto hasta lo más bajo.
Solamente puedes mover un disco a la vez desde un poste hasta otro y no esta permitido poner un disco mas grande sobre otro mas pequeño. Tu tarea es mover todos los discos desde el poste A hasta el poste C.

Ejemplo 15 *Escribe una función que reciba como parámetro n y que imprima en pantalla todos los pasos a seguir para mover los discos del poste A al poste C.*

Solución Pensando primero en el caso mas pequeño y trivial si $n = 1$, tendríamos un solo disco y solo habría que moverlo de la torre A a la C.
Ahora, suponiendo que para algún n ya sabemos cómo mover $n - 1$ discos de una torre a cualquier otra ¿qué deberíamos hacer?
Luego de hacerse esta pregunta es fácil llegar a la conclusión de que primero hay que mover los primeros $n - 1$ discos a la torre B, luego el disco n a la torre C, y posteriormente mover los $n - 1$ discos de la torre B a la torre C.

Podemos estar seguros que lo anterior funciona ya que los primeros $n-1$ discos de la torre siempre serán mas pequeños que el disco n, por lo cual se podrían colocarse libremente sobre el disco n si así lo requirieran.
Por inducción entonces, el procedimiento anterior funciona.
Así que nuestro algoritmo de Divide y Vencerás queda de la siguiente manera:
Sea x la torre original, y la torre a la cual se quieren mover los discos, y z la otra torre.

- Para $n > 1$, hay que mover $n-1$ discos de la torre x a la z, luego mover un disco de la torre x a la y y finalmente mover $n-1$ discos de la torre z a la y.

- Para $n = 1$, hay que mover el disco de la torre x a la y ignorando la torre z.

Nótese que aquí los pasos de *Divide* y *Combina* se resumieron en uno sólo, pero si están presentes ambos.
El siguiente código muestra una implementación del algoritmo anterior, y utiliza como parámetros los nombres de las 3 torres, utilizando parámetros predeterminados como A, B y C.

Código 10: Torres de Hanoi

```
(1)  int hanoi(int n, char x='A', char y='C', char z='B'){
(2)      if(n==1){
(3)          printf("Mueve de %c a %c.\n", x, y);
(4)      }else{
(5)          hanoi(n-1, x, z, y);
(6)          printf("Mueve de %c a %c.\n", x, y);
(7)          hanoi(n-1, z, y, x);
(8)      }
(9)  }
```

Una leyenda cuenta que en la ciudad de Hanoi hay 3 postes así y unos monjes han estado trabajando para mover 64 discos del poste A al poste C y una vez que terminen de mover los 64 discos el mundo se acabará.
¿Te parecen pocos 64 discos?, corre la solución a este problema con n=64 y verás que parece nunca terminar(ahora imagina si se tardaran 2 minutos en mover cada disco).

Ejemplo 16 *¿Cuántas líneas imprime hanoi(n) (asumiendo que esta función está implementada como se muestra en el código 10)?*

Solución Sea $H(n)$ el número de líneas que imprime $hanoi(n)$.
Es obvio que $H(1) = 1$ puesto que $hanoi(1)$ solamente imprime una línea.
Nótese que cada que se llama a $hanoi(n)$ se está llamando dos veces a $hanoi(n-1)$ una vez en la línea 5 y otra en la línea 7. Además, en la línea 6 imprime un movimiento.
Por lo tanto obtenemos la siguiente función recursiva:

$$H(1) = 1$$
$$H(n) = H(n-1) * 2 + 1$$

Ahora haremos unas cuantas observaciones para simplificar aún mas esta función recursiva:

$$\begin{aligned} H(1) &= 1 = 1 \\ H(2) &= 2 + 1 = 3 \\ H(3) &= 4 + 2 + 1 = 7 \\ H(4) &= 8 + 4 + 2 + 1 = 15 \\ H(5) &= 16 + 8 + 4 + 2 + 1 = 31 \\ H(6) &= 32 + 16 + 8 + 4 + 2 + 1 = 63 \\ &... \end{aligned}$$

Probablemente ya estés sospechando que

$$H(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^0$$

Por inducción podemos darnos cuenta que como con $H(1)$ si se cumple y

$$(2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^0)2 + 1 = 2^n + 2^{n-1} + 2^{n-2} + ... + 2^0$$

Entonces para cualquier número n la proposición se debe de cumplir.
O también puedes estar sospechando que:

$$H(n) = 2^n - 1$$

De nuevo por inducción

$$H(0) = 2^0 - 1$$

Y suponiendo que para alguna n, $H(n) = 2^n - 1$

$$(2^n - 1)(n) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$$

Por lo tanto, podemos concluir que la respuesta de este problema es sin lugar a dudas $2^n - 1$ Además de haber resuelto este problema pudimos darnos cuenta que

$$2^n - 1 = 2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^0$$

Esta propiedad de la sumatoria de potencias de 2 hay que recordarla.

Capítulo 7

Búsqueda Exhaustiva

A veces parece que no hay mejor manera de resolver un problema que tratando todas las posibles soluciones. Esta aproximación es llamada Búsqueda Exhaustiva, casi siempre es lenta, pero a veces es lo único que se puede hacer.
También a veces es útil plantear un problema como Búsqueda Exhaustiva y a partir de ahí encontrar una mejor solución.
Otro uso práctico de la Búsqueda Exhaustiva es resolver un problema con con un tamaño de datos de entrada lo suficientemente pequeño.
La mayoría de los problemas de Búsqueda Exhaustiva pueden ser reducidos a generar objetos de combinatoria, como por ejemplo cadenas de caracteres, permutaciones(reordenaciones de objetos) y subconjuntos.

7.1. Cadenas

Ejemplo 17 *Escribe una función que dados dos números enteros n y c, imprima todas las cadenas de caracteres de longitud n que utilicen solamente las primeras c letras del alfabeto(todas minúsculas), puedes asumir que $n < 20$.*

Solución Llamemos $cadenas(n, c)$ al conjunto de cadenas de longitud n usando las primeras c letras del alfabeto.
Como de costumbre, para encontrar la solución a un problema recursivo pensaremos en el caso mas simple.
El caso en el que $n = 1$ y $c = 1$. En ese caso solamente hay que imprimir "a", o dicho de otra manera, $cadenas(1, 1) = \{"a"\}$
En este momento podríamos pensar en dos opciones para continuar el razonamiento cómo de costumbre:

$$n = 1 \text{y} c > 1 \text{o}$$

$$n > 1 y c = 1$$

Si pensáramos en la segunda opción, veríamos que simplemente habría que imprimir las primeras n letras del abecedario.
Si pensáramos en la segunda opción, veríamos que simplemente habría que imprimir n veces áý posteriormente un salto de línea.
Ahora vamos a suponer que $n > 1$ y $c > 1$, para alguna n y alguna c, ¿sería posible obtener $cadenas(n, c)$ si ya se tiene $cadenas(n-1, c)$ ó $cadenas(n, c-1)$?
Si nos ponemos a pensar un rato, veremos que no hay una relación muy obvia entre $cadenas(n, c)$ y $cadenas(n, c-1)$, así que buscaremos la relación por $cadenas(n-1, c)$.
Hay que hacer notar aquí que si se busca una solución en base a una pregunta y ésta parece complicarse, es mejor entonces buscar la solución de otra forma y sólo irse por el camino complicado si no hay otra opción.
Volviendo al tema, buscaremos la relación de $cadenas(n, c)$ con $cadenas(n-1, c)$.
A veces algunos ejemplos sencillos pueden dar ideas. Observamos que

$$\begin{aligned} cadenas(1,3) = \{&a, \\ &b, \\ &c, \\ &\} \end{aligned}$$

$$\begin{aligned} cadenas(2,3) = \{&aa, ab, ac, \\ &ba, bb, bc, \\ &ca, cb, cc \\ &\} \end{aligned}$$

$$\begin{aligned} cadenas(3,3) = \{&aaa, aab, aac, aba, abb, abc, aca, acb, acc, \\ &baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, \\ &caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc, \\ &\} \end{aligned}$$

Luego de observar esto es posible prestar mas atención a la siguiente propiedad:
Toda cadena de n caracteres puede ser formada por una cadena de $n-1$ caracteres seguida de otro caracter.
Esto quiere decir que para cada caracter que pueda tener la cadena hay que generar todas las cadenas de $n-1$ caracteres y a cada cadena colocarle dicho caracter al final.

Partiendo de esta idea, que puede parecer un tanto complicada de implementar, se puede sustituir por la idea de primero colocar el ultimo caracter de la cadena, y luego generar todas las cadenas de $n-1$ caracteres posibles.
Para evitar meter muchos datos en la pila, es mejor tener la cadena guardada en un arreglo global de tipo char.

Código 11: Generador de Cadenas de Caracteres

```
(1)   char C[21];
(2)   void cadenas(int n, int c){
(3)     int i;
(4)     if(n==0){
(5)       printf(" %s\n", C);
(6)     }else{
(7)       for(i='a';i<'a'+c;i++){
(8)         C[n]=i;
(9)         cadenas(n-1, c);
(10)      }
(11)    }
```

Ejemplo 18 *¿Cuántas cadenas de longitud n que utilizan solamente las primeras c letras del abecedario(todas minúsculas) existen? O dicho de otra forma ¿Cuántas líneas imprime el código 11?*

Solución Llamémosle $cad(n, c)$ al número de líneas que imprime el código 11. Es obvio que $cad(1, c)$ imprime c líneas.
Nótese que si $n > 1$, $cadenas(n, c)$ llama c veces a $cadenas(n-1, c)$. Por lo tanto

$$cad(1, c) = c$$

$$cad(n, c) = cad(n-1, c) * c \text{para} n > 1$$

Ahora por inducción es fácil darnos cuenta que

$$cad(n, c) = c^n$$

Un algoritmo de cambio mínimo para generar cadenas binarias(de 0s y 1s) las debe generar en tal orden que que cada cadena difiera de su predecesor en solamente un caracter.
El siguiente código es una implementación de un algoritmo de cambio mínimo que genera las 2^n cadenas binarias.

Código 12: Generador de cadenas binarias con cambio mínimo

```
(1)   void genera(int n){
(2)     if(n==0){
(3)       imprime_cadena();
(4)     }else{
(5)       genera(n-1);
(6)       C[i]=!C[i];
(7)       genera(n-1);
(8)     }
```

El código anterior genera las cadenas en un arreglo llamado C, y asume que siempre será lo suficientemente grande para generar todas las cadenas, la línea 3 llama a una función para imprimir la cadena generada, dicha función no se muestra porque ocuparía demasiado espacio y realmente no tiene relevancia en el algoritmo.
Puede parecer un tanto confusa la línea 6, pero hay que recordar que !0 devuelve 1 y !1 devuelve 0.

Ejemplo 19 *Prueba que el código 12 genera todas las cadenas binarias y que cada cadena que genera difiere de la anterior en solamente un dígito.*

Solución Este problema requiere que se comprueben 2 cosas: una es que el algoritmo genera todas las cadenas binarias de longitud n, y otra es que las genera con cambio mínimo.
Si $n = 1$ basta con echar un vistazo al código para darse cuenta que funciona en ambas cosas.
Podemos observar también, que luego de ejecutarse la línea 6 se llama a $genera(n-1)$ y se sigue llamando recursivamente a la función genera sin pasar por la línea 6 hasta que n toma el valor de 0 y se imprime la cadena en la línea 3; de esta forma podemos asegurar que genera las cadenas con cambio mínimo.
Para probar el hecho de que se imprimen todas las cadenas binarias de longitud n, hay que hacer la siguiente observación que no es muy obvia a simple vista:
No importa si el arreglo C no está inicializado en 0s, siempre y cuando contenga únicamente 0s y 1s. La prueba de esto es que dada una cadena binaria de longitud n, se puede generar a partir de ella cualquier otra cadena binaria de longitud n, simplemente cambiando algunos de sus 1s por 0s y/o algunos de sus 0s por 1s. Si el algoritmo produce todas los conjuntos de cambios que existen entonces generará todas las cadenas sin importar la cadena inicial.
La capacidad de poder hacer este tipo de razonamientos tanto en la Olimpiada de Informática como en la ACM es muy importante, tan importante es poder reducir un problema como dividirlo.
De esa forma, suponiendo que para algún $n-1$ la función produce todas las cadenas de $n-1$ caracteres, ¿las producirá para n?.

Dado que la función no hace asignaciones sino solamente cambios(línea 6), podemos estar seguros que si se producen todas las cadenas llamando a la función con $n-1$ significa que el algoritmo hace todos los cambios para $n-1$. También se puede observar que la línea 5 llama a $genera(n-1)$ con $C[n] = a$ para alguna $0 \leq a \leq 1$, y la línea 7 llama a $genera(n-1)$ con $C[n] =$not a (si $a = 0$ entonces $C[n] = 1$ y si $a = 1$ entonces $C[n] = 0$). Como $C[n]$ está tomando todos los valores posibles y para cada uno de sus valores esta generando todas las cadenas de tamaño n-1 podemos concluir que el algoritmo genera todas las cadenas binarias.

7.2. Subconjuntos

Un conjunto es una colección o agrupación de objetos, a los que se les llama elementos.
Hay muchos ejemplos de la vida cotidiana de conjuntos, por ejemplo, el conjunto de los libros de una biblioteca, el conjunto de muebles de la casa, el conjunto de personas entre 25 y 30 años, etc.
Así como existen conjuntos tan concretos, las matemáticas(y por ende las ciencias de la computación) tratan por lo general con conjuntos más abstractos. Por ejemplo el conjunto de los números entre 0 y 10, el conjunto de los números pares, el conjunto de los polígonos regulares, entre otros. De hecho casi todos los objetos matemáticos son conjuntos.
Los conjuntos, por lo general se describen con una lista de sus elementos separados por comas, por ejemplo, el conjunto de las vocales:

$$\{a, e, i, o, u\}$$

El conjunto de los números pares positivos de un solo dígito:

$$\{2, 4, 6, 8\}$$

Dado que un conjunto es una agrupación de elementos, no importa el orden en el que se escriban los elementos en la lista. Por ejemplo:

$$\{1, 5, 4, 3\} = \{4, 5, 1, 3\}$$

Los conjuntos suelen representarse con letras mayúsculas y elementos de los conjuntos con letras minúsculas.
Si un objeto a es un elemento de un conjunto P, entonces se dice que a pertenece a P y se denota de la siguiente manera:

$$a \in P$$

y si un objeto a no pertenece a P se denota de la siguiente manera

$$a \notin P$$

Por ejemplo

$$1 \in 3, 1, 5 \text{ pero } 1 \notin 3, 4, 2$$

Algunos conjuntos muy renombrados son:

- El conjunto de los números reales $\mathbb{R}$
- El conjunto de los números enteros $\mathbb{Z} = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$
- El conjunto de los números naturales $\mathbb{N} = \{0, 1, 2, 3, 4, 5, ...\}$

Respecto a este último conjunto, hay algunos matemáticos que aceptan el 0 como parte de los números naturales y hay matemáticos que no lo aceptan. Por lo general los que no aceptan el 0 como parte de los números naturales es porque en teoría de números no se comporta como el resto de ellos; pero en las ciencias de la computación por lo general si se acepta.
Se dice que un conjunto A es subconjunto de B, si todos los elementos de A también son elementos de B. O dicho de otra manera:
Se dice que A es subconjunto de B si para todo elemento c tal que $c \in A$ se cumple que también $c \in B$ y se denota como $A \subseteq B$.
Por ejemplo:
El conjunto de las vocales es un subconjunto del conjunto del alfabeto.
El conjunto de los números pares es un subconjunto del conjunto de los números enteros.

$$\{a, b\} \subseteq \{b, c, d, a, x\}$$

$$\mathbb{N} \subseteq \mathbb{Z}$$

$$\mathbb{Z} \subseteq \mathbb{R}$$

Los subconjuntos son muy utilizados en la búsqueda exhaustiva, ya que muchos problemas pueden ser reducidos a encontrar un subconjunto que cumpla con ciertas características.
Ya sea para poder resolver un problema con búsqueda exhaustiva o para poder encontrar una mejor solución partiendo de una búsqueda exhaustiva es preciso saber cómo generar todos los subconjuntos.

Ejemplo 20 *Supongamos que hay un arreglo global C que es de tipo entero. Escribe una función que reciba un entero n como parámetro e imprima todos los subconjuntos del conjunto de primeros n elementos del arreglo C.*

Solución Este problema de generar todos los subconjuntos se puede transformar en el problema de generar cadenas de caracteres de una manera bastante sencilla: Cada elemento de C puede estar presente o ausente en un subconjunto determinado. Vamos a crear entonces un *arreglo de presencias* al que llamaremos P, es decir $P[i]$ será 0 si $C[i]$ está ausente, y será 1 si $C[i]$ está presente.
El problema entonces se reduce a generar todas las posibles cadenas binarias de longitud n en P. La técnica de expresar un problema en términos de otro es indispensable para resolver este tipo de problemas.

Código 13: Generación de subconjuntos

```
(1)   int imprime_subconjuntos(int n, int m=-1){
(2)       int i;
(3)       if(m<n){
(4)           m=n;
(5)       }if(n==0){
(6)           for(i=0;i<m;i++)
(7)           if(P[i]==1)
(8)           printf(" %d ", C[i]);
(9)           printf("\n");
(10)      }else{
(11)          P[n-1]=0;
(12)          imprime_subconjuntos(n-1, m);
(13)          P[n-1]=1;
(14)          imprime_subconjuntos(n-1, m);
(15)      }
(16)  }
```

El código anterior imprime todos los subconjuntos de C, se omite la declaración de P para no gastar espacio y se utiliza el parámetro m como el número de elementos de los que hay que imprimir subconjuntos para que no haga conflicto con el parámetro n que es el número de elementos de los que hay que generar los subconjuntos; nótese que si se llama a la función dejando el parámetro predeterminado para m, posteriormente m tomará el valor de n.

Ejemplo 21 *Considera la misma situación que en el ejemplo 20, escribe una función que reciba como parámetros n y m, e imprima todos los subconjuntos de los primeros n elementos del arreglo C tal que cada subconjunto contenga exactamente m elementos.*

Solución Como ya vimos en el ejemplo anterior, este problema se puede reducir a un problema de generación de cadenas binarias. Así que vamos a definir $P[i]$ como 1 si $C[i]$ esta presente y como 0 si $C[i]$ esta ausente del subconjunto.

Sea además $S(n, m)$ los subconjuntos de los primeros n elementos del arreglo tal que cada subconjunto tenga exactamente m elementos.

Por ello el problema se reduce a encontrar todas las cadenas binarias de longitud n que contengan exactamente m 1s.

Antes de encontrar el algoritmo para resolver esto es necesario hacer notar las siguientes propiedades:

- Si $m > n$ no hay cadena binaria que cumpla con los requisitos, ya que requiere tener exactamente m 1s, pero su longitud es demasiado corta(menor que m).

- Si $m = 0$ solo hay un subconjunto: el conjunto vacío.

- Si $n > 0$ y $n > m$ entonces $S(n, m)$ esta compuesto únicamente por $S(n-1, m)$ y por todos elementos de $S(n-1, m-1)$ añadiéndoles $C[n]$ a cada uno(véase Ejemplo 11).

Con estas propiedades ya se puede deducir el algoritmo.

Al igual que en el ejemplo anterior, utilizaremos un parámetro auxiliar al que llamaremos l que indicará de qué longitud era la cadena inicial. La intención es que a medida que se vayan haciendo las llamadas recursivas, el programa vaya formando una cadena que describa el subconjunto en P.

Dicho de otra manera definiremos una función llamada $imprime_subconjuntos(n, m, l)$, que generará todos los subconjuntos de tamaño m de los primeros n elementos del arreglo C. E imprimirá cada subconjunto representado en $P[0..n-1]$ junto con los elementos definidos en $P[n..l-1]$(los que ya se definieron en llamadas recursivas anteriores), es decir, imprimirá $P[0..l-1]$ para cada subconjunto.

Si $m > n$ entonces simplemente hay que terminar la ejecución de esa función pues no hay nada que imprimir.

Si $m = 0$ solamente queda el conjunto vacío, entonces solo hay que imprimir el subconjunto representado en $P[0..l-1]$.

En otro caso hay que asegurarse de que $C[n-1]$ este ausente en el subconjunto y llamar a $imprime_subconjuntos(n-1, m, l)$, esto generará todos los subconjuntos de los primeros n elementos del arreglo C, con exactamente m elementos, donde $C[n-1]$ no esta incluido.

Posteriormente, hay que poner a $C[n]$ como presente y llamar a $imprime_subconjuntos(n-1, m-1, l)$ para generar los subconjuntos donde $C[n]$ esta incluido.

Código 14: Generación de todos los subconjuntos de $C[0..n-1]$ con m elementos

```
(1)   void imprime_subconjuntos(int n, int m, int l=-1){
(2)      int i;
(3)      if(l<n)
(4)      l=n;
(5)      if(m>n){
(6)         return;
(7)      }if(m==0){
(8)         for(i=0;i<l;i++)
(9)         if(P[i]==1)
(10)        printf(" %d ", C[i]);
(11)        printf("\n");
(12)     }else{
(13)        P[n-1]=0;
(14)        imprime_subconjuntos(n-1, m, l);
(15)        P[n-1]=1;
(16)        imprime_subconjuntos(n-1, m-1, l);
(17)     }
(18)  }
```

Con este ejemplo cerramos el tema de subconjuntos.

7.3. Permutaciones

Así como a veces un problema requiere encontrar un subconjunto que cumpla con ciertas características, otras veces un problema requiere encontrar una secuencia de objetos que cumplan con ciertas características sin que ningún objeto se repita; es decir, una permutación.
Una permutación se puede definir como una cadena que no utilizan 2 veces un mismo elemento.
Las permutaciones, al igual que los conjuntos, suelen representarse como una lista de los elementos separada por comas. Sin embargo, a diferencia de los conjuntos, en las permutaciones el orden en el que se listan los elementos si importa. Por ejemplo, la permutación $\{3, 2, 5\}$ es diferente a la permutación $\{5, 3, 2\}$ y diferente a la permutación $\{2, 3, 5\}$.
Si se tiene un conjunto A, una permutación de A es una cadena que utiliza cada elemento de A una sola vez y no utiliza elementos que no pertenezcan a A.
Por ejemplo, sea A el conjunto $\{a, b, c, d\}$, una permutación de A es $\{a, c, d, b\}$ otra permutación es $\{d, a, c, b\}$.

Ejemplo 22 *Sea A un conjunto con n elementos diferentes. ¿Cuántas permutaciones de A existen?*

Solución Un conjunto con un solo elemento o ningún elemento tiene solamente una permutación.
Si para algún n se sabe el número de permutaciones que tiene un conjunto de n elementos, ¿es posible averiguar el número de permutaciones con un conjunto de $n+1$ elementos?
Consideremos una permutación de un conjunto con n elementos(aqui a_i representa el i-ésimo número de la permutación):

$$\{a_1, a_2, a_3, a_4, ..., a_n\}$$

Suponiendo que quisiéramos insertar un nuevo elemento en esa permutación, lo podríamos poner al principio, lo podríamos poner entre a_1 y a_2, entre a_2 y a_3, entre a_3 y a_4, ... , entre a_{n-1} y a_n, o bien, al final. Es decir, lo podríamos insertar en $n+1$ posiciones diferentes.
Nótese entonces que por cada permutación de un conjunto de n elementos, existen $n+1$ permutaciones de un conjunto de $n+1$ elementos conservando el orden de los elementos que pertenecen al conjunto de n elementos.
Y también, si a una permutación de un conjunto con $n+1$ elementos se le quita un elemento, entonces queda una permutación de un conjunto con n elementos.
Sea $p(n)$ el numero de permutaciones de un conjunto con n elementos, podemos concluir entonces que $p(0) = 1$ y $p(n) = p(n-1)n$, esta recurrencia es exactamente igual a la función factorial.
Por ello el número de permutaciones de un conjunto con n elementos diferentes es $n!$.

Ejemplo 23 *Escribe un programa que dado n, imprima todas las permutaciones del conjunto de los primeros n números enteros positivos.*

Solución Antes que nada tendremos un arreglo $p[] = \{1, 2, 3, ..., n\}$
Si $n = 0$ entonces solamente hay una permutación.
Si $n > 0$, hay que generar todas las permutaciones que empiecen con $p[0]$, las que empiecen con $p[1]$, las que empiecen con $p[2]$, las que empiecen con $p[3]$, ... , las que empiecen con $p[n-1]$(sobra decir a estas alturas que una permutación de un conjunto con n números es un solo número seguido de una permutación de un conjunto con $n-1$ números).
Esto se puede hacer recursivamente, ya que la función generará todas las permutaciones con n elementos, sin importar qué elementos sean. Es decir, si el programa intercambia el valor de $p[i]$ con el valor de $p[n-1]$ y manda a llamar a la función con

$n-1$, la función generará todas las permutaciones de los primeros $n-1$ elementos con el antiguo valor de $p[i]$ al final. Así que hay que repetir este procedimiento de intercambiar y mandar llamar a la función para toda i entre 0 y $n-1$.
Así que el programa queda de esta manera:

Código 15: Generación de permutaciones

```
(1)   #include <stdio.h>
(2)   int *p;
(3)   int N;
(4)   void permutaciones(int n){
(5)      int i, aux;
(6)      if(n==0){ //Si n=0 imprime la permutacion
(7)         for(i=0;i<N;i++)
(8)         printf(" %d ", p[i]);
(9)         printf("\n");
(10)     }else{ //Sino, realiza los intercambios correspondientes y en-
         tra en recursion
(11)        for(i=0;i<n;i++){
(12)           aux=p[i]; //Intercambia los valores de p[n-1] y p[i]
(13)           p[i]=p[n-1];
(14)           p[n-1]=aux;
(15)           permutaciones(n-1); //Hace la llamada recursiva
(16)           aux=p[i]; //Pone los valores de p[n-1] y p[i] en su lugar
(17)           p[i]=p[n-1];
(18)           p[n-1]=aux;
(19)        }
(20)     }
(21)  }
(22)  int main(){
(23)     int i;
(24)     scanf(" %d", &N); //Lee el tama\ño del conjunto de la en-
         trada
(25)     p=new int[N];
(26)     for(i=0;i<N;i++) //Inicializa el conjunto
(27)     p[i]=i+1;
(28)     permutaciones(N);//Ejecuta la recursion
(29)     return 0;
(30)  }
```

Nuevamente el código de la función es bastante corto aunque el resto del programa hace el código mas largo.

Bibliografía

[1] Ian Parberry y William Gasarch, **Problems on Algorithms** 2002

[2] Arthur Engel, **Problem Solving Strategies** Springer 2000

[3] Maria Luisa Pérez Seguí **Combinatoria** 2005

[4] Bernard Kolman, Robert C. Busby y Sharon Cutler Ross, **Estructuras de Matemáticas Discretas para la Computación** Editorial Pearson Educación 1995

[5] Francisco Javier Zaragoza MartÃnez**Una Breve Introducción a la Recursión** 2004