

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
DEPARTAMENTO DE INGENIERÍA

Algoritmia

Laboratorio 3

Backtracking

Backtracking

En el libro “Estructuras de Datos y Algoritmos” de Niklaus Wirth en la sección 3.4 donde expone el algoritmo del *backtracking* (o rastreo hacia atrás) se presentan dos esquemas genéricos: el primero para hallar una solución y el segundo para hallar todas las soluciones. A continuación se muestran estos esquemas.

- esquema para una solución:

```
procedimiento ensayar (paso : TipoPaso)
  repetir
    | seleccionar_candidato
    | if acceptable then
    | begin
    |   anotar_candidato
    |   if solucion_incompleta then
    |     begin
    |       ensayar(paso_siguiente)
    |       if no acertado then borrar_candidato
    |     end
    |   else begin
    |     anotar_solucion
    |     acertado <- cierto;
    |   end
    | end
  hasta que (acertado = cierto) o (candidatos_agotados)
fin procedimiento
```

- esquema para todas las soluciones:

```
procedimiento ensayar (paso : TipoPaso)
  para cada candidato hacer
    |seleccionar candidato
    |if acceptable then
    |begin
    |  anotar_candidato
    |  if solucion_incompleta then
    |    ensayar(paso_siguiente)
    |  else
    |    almacenar_solucion
    |    borrar_candidato
    |end
  hasta que candidatos_agotados
fin procedimiento
```

La diferencia se encuentra en que en el primer caso una vez que se encuentra una solución, no sólo se debe terminar esa rama sino todas las que quedaron pendientes por eso la invocación recursiva debe tener una variable *boolean* que nos advierta de si ya se encontró una solución para cancelar el resto de llamadas o en su defecto seguir intentando y normalmente se programa con una instrucción *while*, para condicionar el bucle a cuando se encuentra una solución o cuando se acabaron las posibilidades.

En el segundo ejemplo como se desea buscar todas las soluciones es necesario recorrer todas las posibilidades, aunque en el camino podemos ir descartando las que no cumplan algunas condiciones. Por eso en este caso se programa con un *for*.

Otro aspecto a tener en cuenta son las restricciones que debe tener el candidato, si estas son muy complejas normalmente se invoca a una función para que se haga cargo de la verificación, en su defecto se llevan a acabo en el mismo código. A veces se pueden emplear arreglos auxiliares como una idea ingeniosa para ayudarnos a la solución como veremos en un ejemplo a continuación. También podría existir el caso en que no hay restricción alguna.

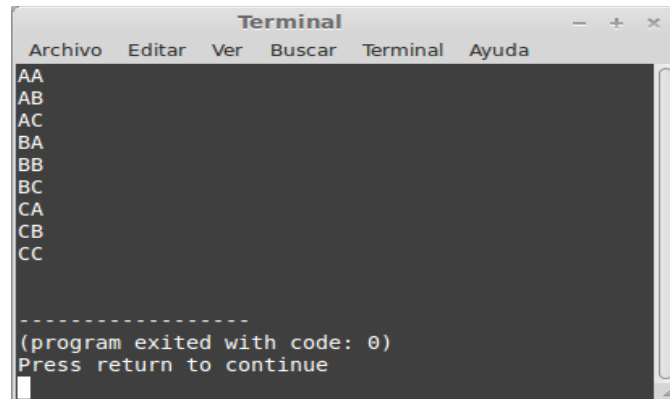
Ejemplo1.

Se tiene una cadena con tres caracteres: A, B y C, se desea un programa que empleando el *backtracking* imprima todas las combinaciones posibles de 2 caracteres, sin importar si se repiten, la salida deberá imprimir: AA, AB, AC, BB, BA, BC, CC, CA, CB..

En este caso todos los candidatos son admitidos, pues no hay ninguna restricción que tengan que cumplir los candidatos. El conjunto de posibilidades es 3, por tanto este será nuestro límite para el lazo *for*, y los elementos a obtener son 2, esta será la profundidad de nuestra llamada recursiva. Siguiendo el esquema arriba mostrado podemos obtener el siguiente programa:

```
combina.c ✕
1  #include <stdio.h>
2
3  char cadena[]={ 'A', 'B', 'C' };
4
5  void backtracking(char *letras, int n) {
6      int k;
7      for(k=0; k<3; k++){
8          letras[n]=cadena[k];
9          if(n<1) backtracking(letras, n+1);
10         else {
11             letras[2]='\0';
12             printf("%s\n",letras);
13         }
14     }
15 }
16 int main(void) {
17     char letras[3];
18
19     backtracking(letras,0);
20     return 0;
21 }
22
```

Al ejecutar obtenemos la salida deseada:

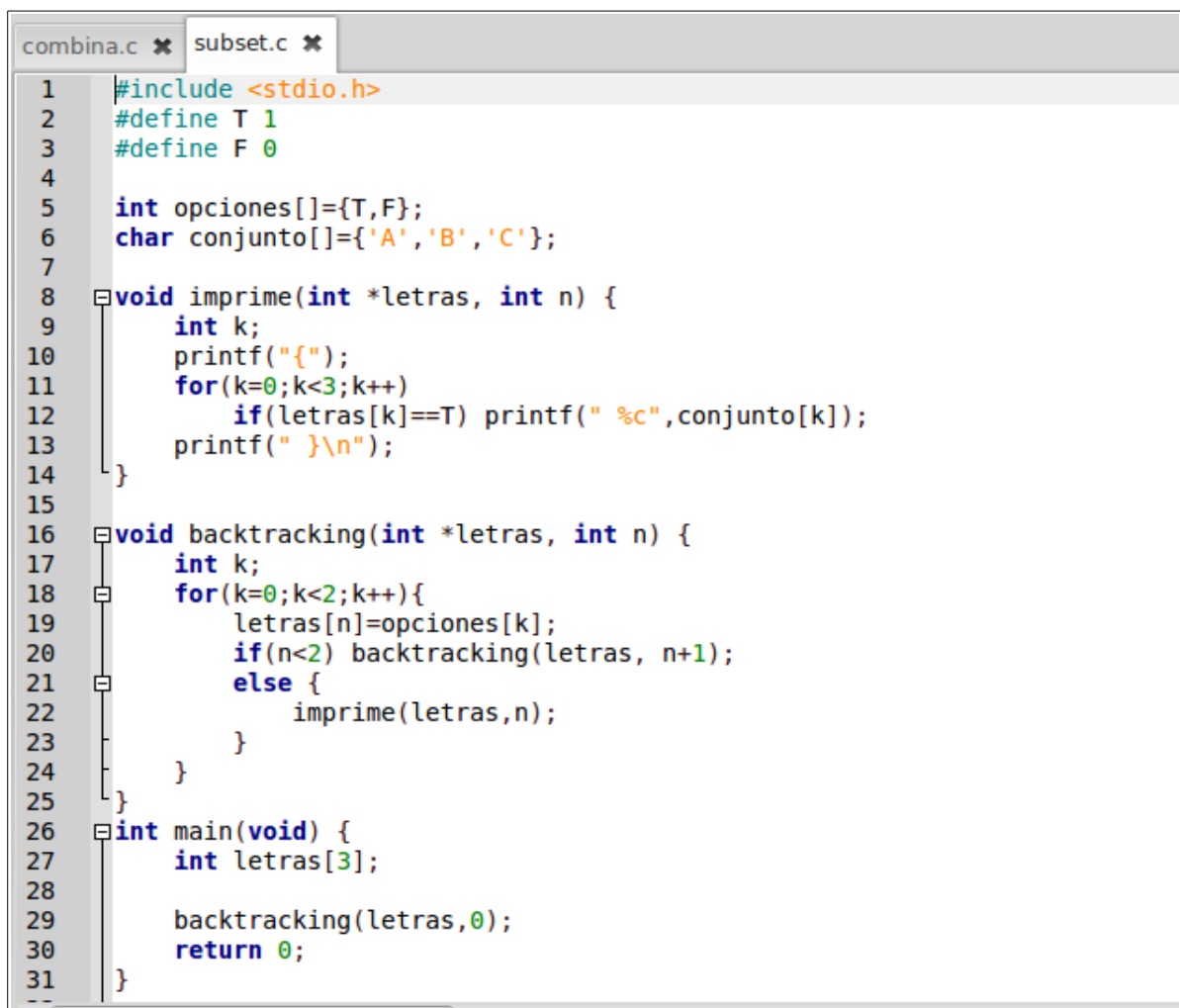


```
Terminal
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
AA
AB
AC
BA
BB
BC
CA
CB
CC

-----
(program exited with code: 0)
Press return to continue
```

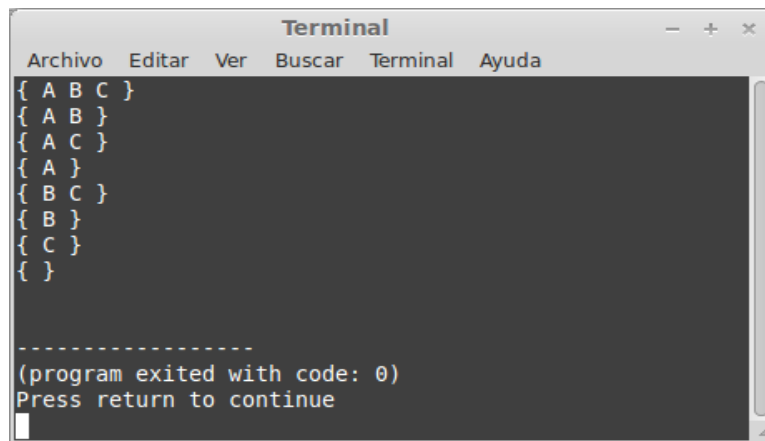
Ejemplo 2

Ahora se desea obtener todos los subconjuntos de un conjunto dado. Si el conjunto es {A,B,C} entonces los subconjuntos que se obtienen son: { {},{A}, {B},{C},{A,B}, {A,C}, {B,C}, {A,B,C}}. En este caso se empleará un arreglo auxiliar de booleanos. Como el conjunto es de 3 elementos el arreglo auxiliar también será de 3 y trataremos de llevar a cabo todas las combinaciones de T y F en dicho arreglo. Cada candidato obtenido nos servirá para imprimir los conjuntos, cuando sea un T se imprimirá el carácter, cuando sea un F no se imprimirá. A continuación el programa:



```
combina.c x subset.c x
1  #include <stdio.h>
2  #define T 1
3  #define F 0
4
5  int opciones[]={T,F};
6  char conjunto[]={'A','B','C'};
7
8  void imprime(int *letras, int n) {
9      int k;
10     printf("{");
11     for(k=0;k<3;k++)
12         if(letras[k]==T) printf(" %c",conjunto[k]);
13     printf(" }\n");
14 }
15
16 void backtracking(int *letras, int n) {
17     int k;
18     for(k=0;k<2;k++){
19         letras[n]=opciones[k];
20         if(n<2) backtracking(letras, n+1);
21         else {
22             imprime(letras,n);
23         }
24     }
25 }
26
27 int main(void) {
28     int letras[3];
29
30     backtracking(letras,0);
31     return 0;
32 }
```

Y la respectiva ejecución:




```
Terminal
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
{ A B C }
{ A B }
{ A C }
{ A }
{ B C }
{ B }
{ C }
{ }

-----
(program exited with code: 0)
Press return to continue
```

Ejemplo 3

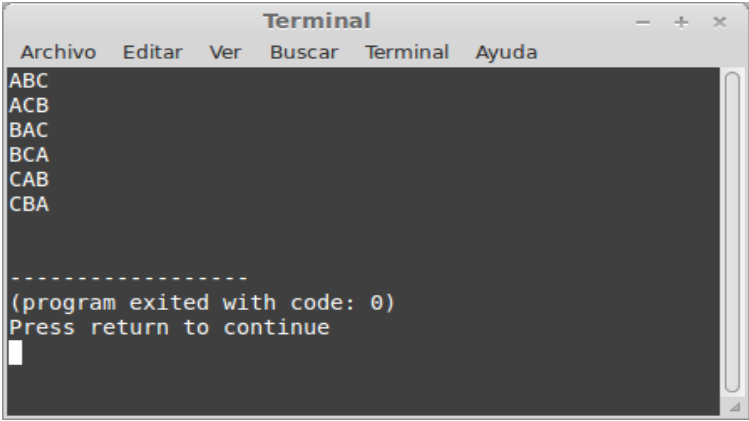
Se desea obtener todas las permutaciones posibles de un conjunto. Sea el conjunto A,B,C se deberían obtener: ABC, ACB, BCA, BAC, CAB, CBA.

Para resolver el problema emplearemos el segundo esquema, como lo hemos venido haciendo hasta ahora, sin embargo hay elementos que no deseamos, por ejemplo uno de los candidatos (que rechazaremos, por supuesto) es AAA. Es decir si una letra ya salió elegida no debe de volver a elegirse. Para lograr este objetivo emplearemos un arreglo auxiliar de booleanos, donde cada posición indica un elemento del conjunto, si ha sido seleccionado le asignaremos T. Al inicio como ningún elemento ha sido seleccionado todas las posiciones serán F. A continuación el programa:



```
combina.c x subset.c x permutar.c x
1  #include <stdio.h>
2  #define T 1
3  #define F 0
4
5  int auxiliar[]={F,F,F};
6  char conjunto[]={'A','B','C'};
7
8  void backtracking(char *letras, int n) {
9      int k;
10     for(k=0;k<3;k++){
11         if(auxiliar[k]==F){
12             letras[n]=conjunto[k];
13             auxiliar[k]=T;
14             if(n<2) backtracking(letras, n+1);
15         } else {
16             letras[3]='\0';
17             printf("%s\n",letras);
18         }
19         auxiliar[k]=F;
20     }
21 }
22
23 int main(void) {
24     char letras[4];
25
26     backtracking(letras,0);
27     return 0;
28 }
29
```

Y su respectiva salida



```
Terminal
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
ABC
ACB
BAC
BCA
CAB
CBA

-----
(program exited with code: 0)
Press return to continue
```

Ejemplo 4

A continuación se transcribe del libro “Técnicas de Diseño de Algoritmos” de Rosa Guerequeta y Antonio Vallecillo el problema del matrimonio estable.

“Supongamos que tenemos n hombres y n mujeres y dos matrices M y H que contienen las preferencias de los unos por los otros. Más concretamente, la fila $M[i, \cdot]$ es una ordenación (de mayor a menor) de las mujeres según las preferencias del i -ésimo hombre y, análogamente, la fila $H[i, \cdot]$ es una ordenación (de mayor a menor) de los hombres según las preferencias de la i -ésima mujer.

El problema consiste en diseñar un algoritmo que encuentre, si es que existe, un emparejamiento de hombres y mujeres tal que todas las parejas formadas sean estables. Diremos que una pareja (h, m) es estable si no se da ninguna de estas dos circunstancias:

- 1) Existe una mujer m' (que forma la pareja (h', m')) tal que el hombre h la prefiere sobre la mujer m y además la mujer m' también prefiere a h sobre h' .
- 2) Existe un hombre h'' (que forma la pareja (h'', m'')) tal que la mujer m lo prefiere sobre el hombre h y además el hombre h'' también prefiere a m sobre la mujer m'' .

Solución

Para este problema vamos a disponer de una n -tupla X que vamos a ir rellenando en cada etapa del algoritmo, y que contiene las mujeres asignadas a cada uno de los hombres. En otras palabras, x_i indicará el número de la mujer asignada al i -ésimo hombre en el emparejamiento. El algoritmo que resuelve el problema trabajará por etapas y en cada etapa k decide la mujer que ha de emparejarse con el hombre k .

Analicemos en primer lugar las restricciones del problema. En una etapa cualquiera k , el k -ésimo hombre escogerá la mujer que prefiere en primer lugar, siempre y cuando esta mujer aún esté libre y la pareja resulte estable. Para saber las mujeres aún libres utilizaremos un vector auxiliar denominado libre. Por simetría, aparte de la n -tupla X , también dispondremos de otra n -tupla Y que contiene los hombres asignados a cada mujer, que necesitaremos al comprobar las restricciones.

Por último, también son necesarias dos tablas auxiliares, ordenM y ordenH. La primera almacena en la posición $[i,j]$ el orden de preferencia de la mujer i por el hombre j , y la segunda almacena en la posición $[i,j]$ el orden de preferencia del hombre i por la mujer j .

Con todo esto, el procedimiento que resuelve el problema puede ser implementado como sigue:

```

TYPE  PREFERENCIAS = ARRAY [1..n],[1..n] OF CARDINAL;
      ORDEN = ARRAY [1..n],[1..n] OF CARDINAL;
      SOLUCION = ARRAY [1..n] OF CARDINAL;
      DISPONIBILIDAD = ARRAY [1..n] OF BOOLEAN;

VAR   M,H:PREFERENCIAS;
      ordenM,ordenH:ORDEN;
      X,Y:SOLUCION;
      libre:DISPONIBILIDAD;

PROCEDURE Parejas(hombre:CARDINAL;VAR exito:BOOLEAN);
  VAR mujer,prefiere,preferencias:CARDINAL;
  BEGIN
    prefiere:=0; (* recorre las posibles elecciones del hombre *)
    REPEAT
      INC(prefiere);
      mujer:=M[hombre,prefiere];
      IF libre[mujer] AND Estable(hombre,mujer,prefiere) THEN
        X[hombre]:=mujer;
        Y[mujer]:=hombre;
        libre[mujer]:=FALSE;
        IF hombre<n THEN
          Parejas(hombre+1,exito);
          IF NOT exito THEN
            libre[mujer]:=TRUE
          END
        ELSE
          exito:=TRUE;
        END
      END
    UNTIL (prefiere=n) OR exito;
  END Parejas;

```

La función Estable queda definida como:

```

PROCEDURE Estable(h,m,p:CARDINAL):BOOLEAN;
  VAR mejormujer,mejorhombre,i,limite:CARDINAL;s:BOOLEAN;
  BEGIN
    s:=TRUE; i:=1;
    WHILE (i<p) AND s DO (* es estable respecto al hombre? *)
      mejormujer:=M[h,i];
      INC(i);
      IF NOT(libre[mejormujer]) THEN
        s:=ordenM[mejormujer,h]>ordenM[mejormujer,Y[mejormujer]];
      END
    END;
    i:=1; limite:=H[m,h]; (* es estable respecto a la mujer? *)
    WHILE (i<limite) AND s DO
      mejorhombre:=H[m,i];
      INC(i);
      IF mejorhombre<h THEN
        s:=ordenH[mejorhombre,m]>ordenH[mejorhombre,X[mejorhombre]];
      END
    END;
    RETURN s
  END Estable;

```

El problema se resuelve mediante la inicialización apropiada de las matrices de preferencias y una invocación a `Parejas(1, exito)`. Tras su ejecución, las variables X e Y contendrán la asignaciones respectivas siempre que la variable `exito` lo indique.”

Ejercicios

- 1) Modifique los tres primeros programas para que los datos sean ingresados por teclado en lugar de estar definidos en el programa. Considere una longitud máxima para el arreglo que contendrá los valores permitidos.
- 2) Elabore un programa que pida un número por teclado e imprima la lista de naturales que sume dicho número. Por ejemplo si el número ingresado por teclado es 5, entonces se debe de imprimir {2,3}, {1,4}.
- 3) Escriba en lenguaje C una solución completa para el ejemplo 4. Considere el tipo *cardinal* como entero.
- 4) Dado un mapa, cada uno de las regiones o países se puede representar como un nodo de un grafo. Se pide implementar un programa en C (empleando la técnica del *backtracking*) de forma que se pueda pintar cada región del mapa de forma que dos regiones consecutivas no estén pintadas del mismo color. La acción de pintar puede estar representada por asignar un número a cada nodo del grafo. Además sólo existen 4 colores.
- 5.- Se tiene un arreglo de 0's y 1's. Donde 0 indica que la celda está libre y 1 que está ocupada. La idea es representar un laberinto. Escriba un programa en C (empleando la técnica del *backtracking*) halle a) una salida del laberinto. b) todas las salidas del laberinto, si las hubiera.