Cuaderno de problemas
Fundamentos de algoritmia.

Vuelta atrás (*Backtracking*)

Prof. Isabel Pita

27 de enero de 2021

t

Índice

1. Generar variaciones con repetición de m elementos tomados de n en n.

Generar todas las variaciones con repetión de un conjunto de elementos

Las variaciones con repetición de m elementos tomados de n en n son los distintos grupos de n elementos, iguales o distintos, que podemos formar con los m elementos, de forma que dos grupos se diferencian en alguno de sus elementos o en su colocación.

El número de variaciones con repetición de m elementos tomados de n en n es $\mathrm{VR}_{m,n} = \mathrm{m}^n$.

Por ejemplo dadas las letras `a`, `b`, y `c`, las variaciones con repetición de estos tres elementos tomados de dos en dos son `aa`, `ab`, `ac`, `bb`, `ba`, `bc`, `cc`, `ca`, `cb`.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso tiene una línea donde se indica el número de letras que se consideran m y el tamaño de la palabra n.

Se consideran únicamente las letras del alfabeto anglosajón, cogiéndose las m primeras. Se garantiza que $n <= m$.

Salida

Para cada caso de prueba se muestran todas las posibles palabras que se pueden formar una por línea. Después de cada caso se muestra una línea en blanco.

Entrada de ejemplo

```
3 2
1 1
```

Salida de ejemplo

```
aa
ab
ac
ba
bb
bc
ca
cb
cc

a
```

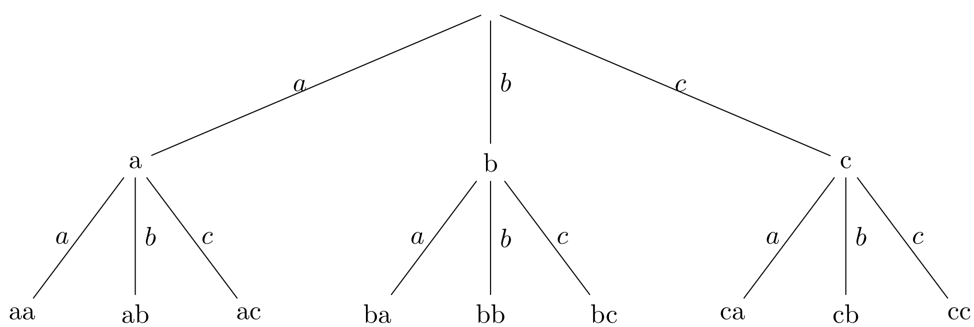
Autor: Isabel Pita

1.1. Objetivos del problema.

- Aprender a generar permutaciones de elementos mediante la técnica de vuelta atrás (*backtracking*).
- Conocer los conceptos de *espacio de búsqueda* y *árbol de exploración*

1.2. Ideas generales.

- El problema se resuelve con la técnica de vuelta atrás.
- Antes de comenzar a resolver estos problemas, es necesario tener claro cual es el espacio de búsqueda, cómo es el árbol de exploración y cómo es la solución que queremos construir.
    - El *espacio de búsqueda* son todas las soluciones potenciales del problema, de entre ellas obtendremos las soluciones reales que serán las que cumplan las restricciones del enunciado. En nuestro problema son soluciones potenciales todas las palabras de n letras que podemos formar con un conjunto de m letras. Por ejemplo si tenemos 3 letras a,b,c ($m = 3$) y queremos formar palabras de 2 letras ($n = 2$), las soluciones potenciales son: `aa`, `ab`, `ac`, `bb`, `ba`, `bc`, `cc`, `ca`, `cb`. En total tenemos m^n soluciones potenciales.
    - El *espacio de búsqueda* se puede estructurar como un *árbol de exploración*. En el ejemplo, en la primera etapa se genera la primera letra de la palabra, y en la segunda etapa la segunda letra de la palabra. En general, el número de etapas será el número de letras de la palabra, de forma que en la etapa i-ésima se genera la letra i-ésima de la palabra.



    - En este problema no se dan restricciones sobre las soluciones potenciales, por lo que estas coinciden con las soluciones reales. (ver el problema de generar variaciones sin repetición para un ejemplo de restricciones en el problema)
    - La solución se genera siguiendo el esquema dado por el árbol de exploración

- El esquema general de vuelta atrás para encontrar todas las soluciones a un problema es:

```
void vueltaAtras (TDatos const& datos, int k, Tupla & sol) {
   for(auto i = primerHijoNivel(k); i < ultHijoNivel(k); i = sigHijoNivel(k)){
      sol[k] = i;
      if (esValida(sol, k)){
         if (esSolucion(sol, k))
            tratarSolucion(sol);
         else
            vueltaAtras(datos, k + 1, sol);
      }
   }
}
```

- Los parámetros indican:
    - `TDatos const & datos`, son los datos de entrada al problema que se necesiten para resolverlo.
    - `Tupla & sol`, vector en el que se va construyendo la solución.
    - `int k`, etapa del árbol en que se encuentra la ejecución.
- En este problema:

- `primerHijoNivel(k)` es el carácter `'a'`.
- `ultHijoNivel(k)` es el carácter `'c'`.
- `sigHijoNivel(k)` debe sumar uno al carácter.
- `esValida(sol, k)` es siempre cierto, ya que no hay ninguna restricción para construir la palabra, por lo tanto no es necesario hacer esta comprobación.
- `esSolucion(sol, k)` es cierto cuando la etapa `k` coincide con el tamaño de la palabra que se quiere construir.

1.3. Algunas cuestiones sobre implementación.

- Usaremos un parámetro por referencia para pasar el vector solución a la función de vuelta atrás, por cuestión de eficiencia. Devolver el vector como resultado de la función supondría hacer una copia del mismo en cada llamada recursiva.

- En el problema se piden variaciones de letras del alfabeto, por lo que la solución deberíamos construirla sobre un `string` (vector de caracteres), sin embargo para mostrar el método general, en el que la solución se construye sobre un vector, utilizaremos un vector de caracteres.

1.4. Coste de la solución.

Se calculan todas las soluciones posibles, por lo tanto el coste es cómo mínimo m^n.

En los problemas de vuelta atrás el coste de los programas es por lo menos exponencial, ya que se recorre el árbol de exploración. El número de nodos del árbol es $(m^h - 1)/(m - 1)$ siendo m el número de hijos de cada nodo y h la altura (número de etapas) del árbol.

Para mejorar el coste veremos cómo se pueden aprovechar las restricciones que puedan poner los problemas sobre las soluciones para no explorar el árbol completo.

1.5. Modificaciones al problema.

- Generar las variaciones sin repetición de m elementos tomados de n en n. El problema es semejante al anterior, pero no deben generarse palabras con letras repetidas. Para ello se utilizará la técnica de marcaje.

- Generar todas las permutaciones de n elementos. Este problema es equivalente al anterior cuando los valores de m y n coinciden.

- Generar todas las combinaciones de elementos. En este caso el orden de los elementos es indiferente, es decir, la palabra `abc` es la misma que la palabra `bca`.

1.6. Implementación.

```
// Escribe una solucion
void escribirSolucion(std::vector<char> const& v) {
    for (char c : v) std::cout << c;
    std::cout << '\n';
}

// Calcula de forma recursiva las variaciones de los elementos
void variaciones (int m, int n, int k, std::vector<char> & sol) {
    for (char i = 'a'; i < 'a' + m; ++i){
        sol[k] = i;
        if (k == n-1){ // Es solucion
            escribirSolucion(sol);
        }
        else { // Sigue completando la solucion
            variaciones(m,n,k+1,sol);
        }
    }
```

```
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuracion, y escribiendo la respuesta
bool resuelveCaso() {
    // Lectura de los datos de entrada
    int m,n; std::cin >> m >> n;
    if (!std::cin) return false;
    // Generar las soluciones
    std::vector<char> sol(n);
    variaciones(m,n,0,sol);
    std::cout << '\n';
    return true;
}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

2. Generar variaciones sin repetición de m elementos tomados de n en n.

Generar todas las variaciones sin repetión de un conjunto de elementos

Las variaciones sin repetición de m elementos tomados de n en n son los distintos grupos de n elementos distintos, que podemos formar con los m elementos, de forma que dos grupos se diferencian en alguno de sus elementos o en su colocación.

El número de variaciones sin repetición de m elementos tomados de n en n es $\mathrm{V}_{m,n}$ = m! / (m-n)!.

Por ejemplo dadas las letras `a`, `b`, y `c`, las variaciones sin repetición de estos tres elementos tomados de dos en dos son `ab`, `ac`, `ba`, `bc`, `ca`, `cb`.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso tiene una línea donde se indica el número de letras que se consideran m y el tamaño de la palabra n.

Se consideran únicamente las letras del alfabeto anglosajón, cogiéndose las m primeras. Se garantiza que $n <= m$.

Salida

Para cada caso de prueba se muestran todas las posibles palabras que se pueden formar una por línea. Después de cada caso se muestra una línea en blanco.

Entrada de ejemplo

```
3 2
1 1
```

Salida de ejemplo

```
aa
ab
ac
ba
bb
bc
ca
cb
cc

a
```

Autor: Isabel Pita

1

2.1. Objetivos del problema.

- Aprender a generar permutaciones de elementos sin repetición mediante la técnica de vuelta atrás (*backtracking*).
- Aprender la técnica de *marcaje* para mejorar la eficiencia de la solución.

2.2. Ideas generales.

- El problema se resuelve con la técnica de vuelta atrás.
- Para resolver este problema es necesario haber comprendido antes el problema 1, sobre generación de variaciones con repetición de m elementos tomados de n en n.
- Modificamos ligeramente el esquema de vuelta atrás para incluir las marcas:

```
void vueltaAtras (TDatos const& datos, int k, Tupla & sol, Tmarcas & marcas) {
   for(auto i = primerHijoNivel(k); i < ultHijoNivel(k); i = sigHijoNivel(k)){
      sol[k] = i;
      marcar(marcas);
      if (esValida(sol, k)){
         if (esSolucion(sol, k))
            tratarSolucion(sol);
         else
            vueltaAtras(datos, k + 1, sol, marcas);
      }
      desmarcar(marcas);
   }
}
```

- La solución no admite elementos repetidos, por lo tanto en este problema las soluciones reales no coinciden con las soluciones potenciales. Las soluciones potenciales son las variaciones con repetición de los m elementos tomados de n en n, mientras que las soluciones reales son únicamente las soluciones potenciales que no tienen letras repetidas.
- Por razones de eficiencia se generan únicamente las soluciones reales, no todas las soluciones potenciales. Esto implica que cuando en el árbol de exploración tenemos que añadir una letra repetida no realizaremos la llamada recursiva y por lo tanto no se generan las soluciones por debajo de ese punto en el árbol. Esto se denomina *podar el árbol*.
- Para saber cuando una letra está repetida utilizaremos un vector de booleanos de dimensión m (todas las letras que se consideran). Un valor cierto en ese vector indica que la letra correspondiente ya ha sido usada en la solución. A este vector le llamaremos vector de *marcas*
- Debe evitarse comprobar si la letra está repetida recorriendo el vector de solución, ya que esto se realizaría en todas las llamadas recursivas añadiendo un coste innecesario a la ejecución del programa.

2.3. Ideas detalladas.

- Las acciones de marcado y desmarcado deben realizarse en el mismo bloque de código.

2.4. Algunas cuestiones sobre implementación.

- El vector de marcas se define entre los índices $[0..m)$, pero el índice del vector debe representar cada una de las letras que se consideran: 'a' ,..., 'a'+m. Por lo tanto cuando consultamos o modificamos las componentes del vector, no debemos acceder con el valor del carácter correspondiente, sino con la posición que el carácter ocupa en el alfabeto: x - 'a', donde x es el carácter que queremos modificar en el vector.

2.5. Coste de la solución.

En este tipo de problemas es difícil estimar el número de ramas del árbol de exploración que se podan (no se recorren), y por lo tanto no se pedirá calcular el coste de la solución en los problemas de vuelta atrás.

2.6. Implementación.

```
struct tDatos {
    int m,n;
};

struct tSol {
    tVector<char> sol;
    tVector<bool> marcas;
};

// Escribe una solucion
void escribirSolucion(std::vector<char> const& v) {
    for (char c : v) std::cout << c;
    std::cout << '\n';
}

void variaciones (tDatos const& datos, int k, tSol<char> & s) {
    for (char i = 'a'; i < 'a' + datos.m; ++i){
        s.sol[k] = i;
        if (!s.marcas[i-'a']){ // Si la letra no esta utilizada
            s.marcas[i-'a'] = true;
            if (k == datos.n-1){ // Es solucion
                escribirSolucion(s.sol);
            }
            else { // Genera el resto de la solucion
                variaciones(datos,k+1,s);
            }
            s.marcas[i-'a'] = false;
        }
    }
}
// Resuelve un caso de prueba, leyendo de la entrada la
// configuracion, y escribiendo la respuesta
bool resuelveCaso() {
    int m,n; std::cin >> m >> n;
    if (!std::cin) return false;
    tSol<char> s;
    s.sol.resize(n);
    s.marcas.assign(m, false);
    variaciones({m,n},0,s);
    std::cout << '\n';
    return true;
}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
```

```
    system("PAUSE");
#endif

    return 0;
}
```

29. Necesito otro abrigo

Cuando volvía del trabajo he visto un abrigo precioso en un escaparate. Realmente necesito un abrigo para este invierno y este era precioso. Sin embargo, mi compañera ha asegurado que tengo suficientes y que ni siquiera tengo necesidad de llevar dos días seguidos el mismo, impidiéndome entrar en la tienda. Lo que ella no tiene en cuenta es que no todos los abrigos se pueden llevar todos los días. Cuando llueve no puedo llevar el de ante, y si la lluvia es fuerte solo sirven las gabardinas.

Después de mucho discutir me ha prometido que si puedo demostrar que no tengo forma de combinar los abrigos sin repetir dos días seguidos alguno me acompañará a comprarlo. Me he puesto manos a la obra y he conseguido una estimación de las precipitaciones de cada día del invierno. Ahora me toca decidir qué abrigo puedo llevar cada día para no mojarme y no tener que repetir prenda dos días consecutivos.

Además tengo una serie de manías que no me permiten ponerme cualquier cosa con tal de no mojarme. Para empezar, no me gusta utilizar un abrigo muchos más días que otro porque se desgasta y se pone feo. Por ello, no consideraré aquellas combinaciones en las cuales el abrigo que más haya utilizado supere en un día o más a un tercio de los días que van transcurridos, es decir, la combinación de abrigos será válida si:

$$\forall i : 1 <= i <= \textit{número de días de la temporada} \Rightarrow dias(a, i) \leq 2 + i/3.$$

Siendo a el abrigo que más he usado hasta el día i-ésimo y *dias(a,i)* el número de días que he utilizado el abrigo a en el periodo $[1..i]$.

Además el abrigo que utilizo el último día del invierno debe ser diferente al que utilicé el primer día.

Requisitos de implementación.

El problema se debe resolver utilizando la técnica de vuelta atrás. Se valorará que se eviten las llamadas recursivas que no producirán solución válida.

Poner comentarios e indicar el coste de la función *esValida* que se haya implementado. El coste debe ser lo menor posible.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba tiene 3 líneas. En la primera se muestra el número de días n sobre el que se hace el estudio, seguido del número de abrigos a que tengo. En la segunda línea se muestran n valores correspondientes a la estimación de la precipitación de cada día. En la tercera línea se muestran a valores que indican la cantidad de precipitación que puede soportar cada uno de los abrigos. La entrada termina con una línea con cero días y cero abrigos.

El número de días es un entero, 1 <= n <= 12 y el número de abrigos es un entero 1 <= a <= 5. La estimación de precipitación es un entero 0 <= p <= 300.

Salida

Para cada caso de prueba se muestra en un línea el número de combinaciones posibles que tengo para ponerme los abrigos. Si no existe ninguna combinación se escribirá *Lo puedes comprar.*

3.1. Objetivos del problema.

- Encontrar todas las soluciones a un problema mediante la técnica de vuelta atrás (*backtracking*).
- Practicar la técnica de *marcaje* para mejorar la eficiencia de la solución.
- Aprender a generar únicamente las soluciones que cumplen una serie de propiedades.
- Diferenciar entre las propiedades que debe cumplir la solución mientras se construye y las propiedades que cumple la solución completa.

3.2. Ideas generales.

- El problema se resuelve con la técnica de vuelta atrás.
- Modificamos ligeramente el esquema de vuelta atrás para que la función devuelva el resultado pedido:

```
int vueltaAtras (TDatos const& datos, int k, Tupla & sol, Tmarcas & marcas) {
   int numSol = 0;
   for(auto i = primerHijoNivel(k); i < ultHijoNivel(k); i = sigHijoNivel(k)){
      sol[k] = i;
      marcar(marcas);
      if (esValida(sol, k, marcas)){
         if (esSolucion(sol, k))
            tratarSolucion(sol);
         else
            numSol += vueltaAtras(datos, k + 1, sol, marcas);
      }
      desmarcar(marcas);
   }
   return numSol;
}
```

- Las condiciones que se deben cumplir mientras se construye la solución se deben comprobar en la función `esValida`.
- Las condiciones que debe cumplir la solución final se comprueban únicamente cuando ya se tiene una solución, dentro del condicional `if (esSolucion) { // Aqui se comprueban las condiciones}`

3.3. Ideas detalladas.

- Para comprobar que no se repite el abrigo, debemos consultar dos valores de la solución, el de la etapa actual y el de la etapa anterior. Sólo se debe realizar esta comprobación si no estamos en la primera etapa.
- Para poder comprobar la condición de que un abrigo no se utiliza muchos más días que otro:
  - Debemos conocer el número de días que se utiliza cada abrigo. Para ello utilizaremos un vector de dimensión el número de abrigos, donde en cada componente se guarda el número de días que se ha utilizado ese abrigo en la solución que estamos formando.
  - No es necesario conocer el abrigo que más veces se ha utilizado. Cuando estamos tratando un abrigo en una solución intermedia, incrementamos el uso que hacemos de este abrigo. Si este abrigo pasa a ser el más utilizado, al comprobar que este abrigo cumple la propiedad que nos piden estaríamos comprobando que la propiedad la cumple el mas utilizado. Si por el contrario el abrigo no pasa a ser el más utilizado, es porque el abrigo más utilizado no ha cambiado. No es necesario verificar la propiedad para el abrigo más utilizado, porque ya se verificó en una llamada recursiva anterior y no ha cambiado. Por otro lado podemos verificar la propiedad para el abrigo actual, porque se cumplirá trivialmente (está menos utilizado que el máximo que si la cumple).

3.4. Algunas cuestiones sobre implementación.

- Aunque nos piden únicamente el número de soluciones encontradas, necesitamos construir la solución para poder comprobar que no se repite el abrigo dos días seguidos y que no se pone el mismo abrigo el primer y último día del invierno.
- Devolveremos la solución como resultado de la función, mientras que para el vector con la solución utilizaremos un parámetro por referencia.
- En estos problemas, aunque no se pida la solución, a veces es conveniente calcularla para poder depurar el programa y comprobar cómo se va formando.

3.5. Coste de la solución.

En este tipo de problemas es difícil estimar el número de ramas del árbol de exploración que se podan (no se recorren), y por lo tanto no se pedirá calcular el coste de la solución en los problemas de vuelta atrás.

3.6. Implementación.

```
// Datos de entrada
struct tDatos {
    std::vector<int> tiposAbrigos;
    std::vector<int> probPrecipitacion;
};

bool es_valida(int k, int i, tDatos const& d, std::vector<int> const& sol,
     std::vector<int> const& marcas){
    // El abrigo soporta la cantidad de lluvia que va a caer
    if (d.tiposAbrigos[i] < d.probPrecipitacion[k]) return false;
    // No llevo dos dias seguidos el mismo abrigo
    if (k > 0 && sol[k] == sol[k-1]) return false;
    // No me pongo un abrigo muchos mas dias que otro
    if (marcas[i] > k/3+2) return false;
    return true;
}

// d - datos entrada. probabilidad precipitacion y lluvia que aguanta cada abrigo.
// k - etapa
// sol - abrigo que me pondre cada dia.
// marcas: dias que me he puesto cada abrigo.
int abrigos (tDatos const& d, int k, std::vector<int> & sol, std::vector<int> & marcas) {
    int numSoluciones = 0; // numero de soluciones diferentes encontradas
    for (int i = 0; i < d.tiposAbrigos.size(); ++i) {
        sol[k] = i;
        // marcar
        ++marcas[i];
        if (es_valida(k, i, d,sol,marcas)){ // es valida
            if (k == d.probPrecipitacion.size()-1){ // es solucion
                if (sol[0] != sol[k]) { // El primer abrigo no coincide con el ultimo
                    ++numSoluciones;
                    // Para escribir las soluciones cuando se pidan
                    //for (int x : sol) std::cout << x << ' ';
                    //std::cout << '\n';
                }
            }
            else {
                numSoluciones += abrigos(d,k+1,sol,marcas);
            }
        }
        // desmarcar
```

```
            --marcas[i];
        }
        return numSoluciones;
    }

    // Resuelve un caso de prueba, leyendo de la entrada la
    // configuracion, y escribiendo la respuesta
    bool resuelveCaso() {
        // leer los datos de la entrada
        int numDias, numAbrigos;
        std::cin >> numDias;
        if (numDias == 0) return false;
        std::cin >> numAbrigos;
        // leer probabilidad de precipitacion
        tDatos d;
        d.probPrecipitacion.resize(numDias);
        for (int& i : d.probPrecipitacion) std::cin >> i;
        // leer caracteristicas de los abrigos
        d.tiposAbrigos.resize(numAbrigos);
        for (int & i : d.tiposAbrigos) std::cin >> i;
        // Declaracion de tipos para llamar a la funcion
        int k = 0;
        std::vector<int> sol(numDias);
        std::vector<int> marcas(numAbrigos);
        int numSoluciones = abrigos(d,k,sol,marcas);
        // Escribir resultado
        if (numSoluciones == 0) std::cout << "Lo puedes comprar\n";
        else std::cout << numSoluciones << '\n';
        return true;
    }

    int main() {
    #ifndef DOMJUDGE
        std::ifstream in("datos.txt");
        auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
    #endif

        while (resuelveCaso()) {} //Resolvemos todos los casos

    #ifndef DOMJUDGE // para dejar todo como estaba al principio
        std::cin.rdbuf(cinbuf);
        system("PAUSE");
    #endif

        return 0;
    }
```

4. Papa Noel reparte juguetes. Versión con optimización.

Papa Noel reparte juguetes. Versión con optimización

Cada año son más los niños que le piden regalos a Papa Noel. Esta noche debe repartir los regalos y todavía no tiene preparado lo que le dará a cada niño. Para poder llegar a tiempo los elfos han diseñado un programa informático, que asignará a cada niño un juguetes entre todos los disponibles. Cada juguete sólo se puede asignar a un niño. Como quieren que los niños queden contentos han elaborado una lista con la satisfacción que le produce a cada niño cada uno de los juguetes que tienen en la fábrica. El objetivo es maximizar el grado de satisfacción del conjunto de todos los niños. La suma de la satisfacción de todos los niños debe ser máxima.

El jefe elfo de informática ha puesto a su equipo a trabajar en un programa que obtenga la satisfacción máxima que pueden conseguir.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de *n+1* líneas. En la primera se indica el número juguetes que se fabrican, m (1 <= m <= 10), y el número de niños a los que se les reparten juguetes, n (1 <= n <= 9). En las n líneas siguientes se indica la satisfacción de cada uno de los n niños con cada uno de los m juguetes. Después de cada caso hay una línea en blanco para facilitar la identificación de los casos en el ejemplo.

Se garantiza que el número de juguetes es siempre igual o superior al número de niños. La satisfacción es un número entero que puede ser negativo si el niño aborrece el juguete.

Salida

Para cada caso de prueba se escribe la satisfacción máxima que se puede conseguir.

Entrada de ejemplo

```
4 3
8 9 3 1
6 4 5 3
2 2 9 9

4 3
8 9 3 1
6 4 5 3
2 2 9 9

4 2
10 12 12 15
9 10 20 7
```

Salida de ejemplo

```
24
24
35
```

Autor: Isabel Pita

1

4.1. Objetivos del problema.

- Utilizar el esquema de vuelta atrás para problemas de optimización.
- Aprender la técnica de estimación para podar el árbol de soluciones.

4.2. Ideas generales.

- El problema se resuelve con la técnica de vuelta atrás.
- Al tratarse de un problema de optimización (calcular la asignación que produce una satisfacción máxima) debemos guardar la solución mejor hasta el momento en que no encontramos.
- Cuando se encuentra una solución mejor que la que tenemos guardada, cambiaremos la solución mejor por la nueva que hemos encontrado.
- Es necesario inicializar la solución mejor antes de llamar a la función recursiva. El mejor valor inicial, porque nos permite podar más el árbol de soluciones es una solución válida, cuya satisfacción sea lo mayor posible. Atención porque debe ser una solución válida del problema. En este caso podemos asignar a cada niño un juguete diferente empezando por el niño 1 con el juguete 1, el niño 2 con el juguete 2 etc. La solución no será la mejor, pero es válida.

  Si no es posible encontrar una solución válida con facilidad, se inicializará al menor valor del tipo por tratarse de un problema de maximización. Si se tratase de un problema de minimización se inicializaría al valor máximo del tipo.

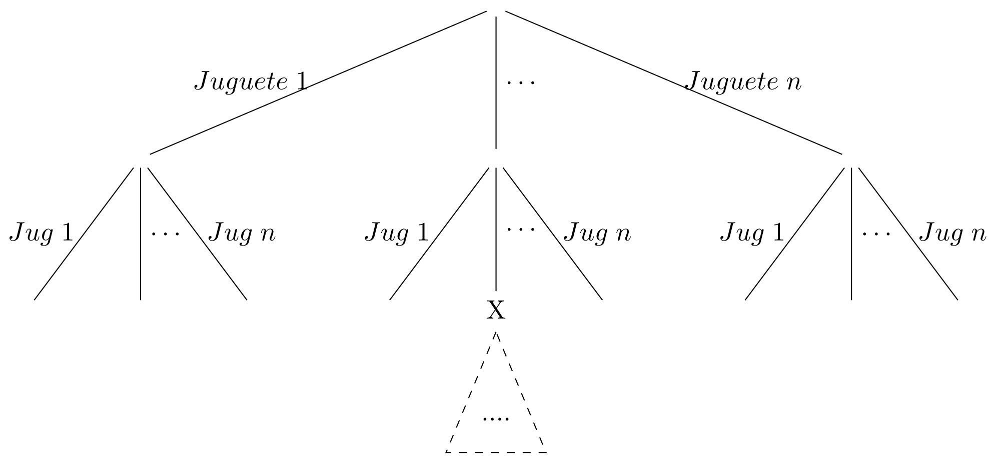
- El esquema general de vuelta atrás para los problemas de optimización es:

```
void vueltaAtras (TDatos const& datos, int k, Tupla & sol, int coste,
   Tupla & solMejor, int & costeMejor) {
   for(auto i = primerHijoNivel(k); i < ultHijoNivel(k); i = sigHijoNivel(k)){
      sol[k] = i;
      coste += ...
      if (esValida(sol, k, ..)){
         if (esSolucion(sol, k, ..)){
            if (esSolucionMejor(...))
                solMejor = sol; costeMejor = coste;
            }
         else if (estimacion(..))
                vueltaAtras(datos, k + 1, sol, coste, solMejor, costeMejor);
      }
      coste -= ....
   }
}
```

- Los parámetros indican:
  - `TDatos const & datos`, son los datos de entrada al problema que se necesiten para resolverlo.
  - `int k`, etapa del árbol en que se encuentra la ejecución.
  - `Tupla & sol`, vector en el que se va construyendo la solución.
  - `int coste`, coste de la solución que se está construyendo.
  - `Tupla & solMejor`, vector en el que se guarda la solución mejor hasta este momento.
  - `int costeMejor`, coste de la solución mejor.
- En este problema:
  - `primerHijoNivel(k)` es el primer juguete que se puede asignar, el cero.
  - `ultHijoNivel(k)` es el último juguete que se puede asignar, el m.
  - `sigHijoNivel(k)` debe sumar uno al juguete.

- `esValida(sol, k)` debe comprobar que el juguete no se ha asignado todavía (debe utilizarse la técnica de marcaje explicada en ...).
- `esSolucion(sol, k)` es cierto cuando la etapa `k` coincide con el último niño al que hay que asignar juguete.

- Los problemas de optimización permiten realizar podas en el árbol de solución basadas en estimar el valor que puede conseguirse con la parte de la solución que todavía no se ha construido.



- Por ejemplo, cuando estamos en el punto X construyendo la solución:

  1. Tenemos seleccionado un juguete para el niño 1 y otro para el niño 2, que producen una satisfacción hasta este momento de S.
  2. Supongamos que tenemos una solución anterior que ha asignado juguetes a todos los niños con una satisfacción M y que esta es la mejor solución encontrada hasta este momento.
  3. Podemos calcular la satisfacción que se obtendría si asignamos a cada uno de los niños que todavía no tienen juguete, el juguete que mayor satisfacción les produce.
  4. Esta asignación seguramente no será una solución válida, pero tenemos la seguridad de que ninguna solución válida que se pueda obtener desde este punto tendrá una satisfacción mayor.
  5. Por lo tanto si la satisfacción S más la satisfacción estimada para los niños que faltan no supera la satisfacción mejor hasta este momento M no debemos seguir explorando esta rama ya que no encontraremos una solución mejor.

- La estimación debe calcularse con la mayor eficiencia posible. Para ello se llevarán precalculados valores que se pasarán a la función de vuelta atrás como parámetro.

- Para este problema llevaremos precalculado la mejor satisfacción que se puede obtener desde cada uno de los niños hasta el último. (ver Sec. 4.3)

4.3. Algunas cuestiones sobre implementación.

- Para calcular la mejor satisfacción que podemos obtener desde cada niño hasta el último partimos de la matriz de satisfaciones que nos dan como datos de entrada.

- El vector de acumulados se calcula en dos etapas:

  1. Calculamos un vector con el máximo de cada fila, es decir la componente v[i] del vector tiene el máximo de la fila i de la matriz y representa el juguete que le produce mayor satisfacción al niño i.
  2. A partir del vector de máximos calculamos un vector de acumulados. Empezando por la penúltima componente del vector se va sumando cada componente con la siguiente.

4.4. Implementación.

```
// Datos de entrada
struct tDatos {
    int numJuguetes, numNinos;
    std::vector<std::vector<int> > satis;
};
// Solucion que se va construyendo
struct tSol {
    std::vector<int> sol;
    int sumaSatis;
};
// Parametros:
// datos de entrada
// k - etapa
// s - Solucion que se va construyendo
// asignados - Vector de dimension el numero de juguetes marca los juguetes ya asignados
// acum - vector con la satisfaccion acumulada desde cada ninno hasta el ultimo
// satisMejor - la satisfacion maxima
void resolver (tDatos const& datos,int k, tSol & s, std::vector<bool>& asignados, int& satisMejor
              std::vector<int> const& acum) {
    for (int i = 0; i < datos.numJuguetes; ++i) {
        s.sol[k] = i;
        if (!asignados[i]) { // es Valida
            s.sumaSatis += datos.satis[k][i];
            asignados[i] = true; // marca
            if (k == s.sol.size()-1) { // es solucion
                if (s.sumaSatis > satisMejor) satisMejor = s.sumaSatis; // solucion mejor
            }
            else { // No es solucion
                if (s.sumaSatis + acum[k+1] > satisMejor) // Estimacion
                    resolver(datos, k+1, s, asignados,satisMejor, acum);
            }
            asignados[i] = false; // desmarca
            s.sumaSatis -= datos.satis[k][i];
        }
    }
}

bool resuelveCaso() {
    // Lectura de los datos de entrada
    tDatos datos;
    std::cin >> datos.numJuguetes;
    if (!std::cin) return false;
    std::cin >> datos.numNinos;
    // Lee la satisfaccion que les proporcionan los juguetes a los ninnos
    for (int i = 0; i < datos.numNinos; ++i){
        std::vector<int> aux(datos.numJuguetes);
        for (int j = 0; j < datos.numJuguetes; ++j) std::cin >> aux[j];
        datos.satis.push_back(aux);
    }
    // Calcula el vector de maximos
    std::vector<int> acum(datos.numNinos);
    for (int i = 0; i < datos.numNinos; ++i){
        int auxMax = datos.satis[i][0];
        for (int j = 1; j < datos.numJuguetes; ++j)
            if (auxMax < datos.satis[i][j]) auxMax = datos.satis[i][j];
        acum[i] = auxMax;
    }
    // Vector acumulados
    for (int i = (int)acum.size()-1; i > 0; --i){
        acum[i-1] += acum[i];
```

```
    }
    // Obtiene una inicializacion posible para la solucion mejor
    int satisMejor = 0;
    for (int i = 0; i < datos.numNinos; ++i){
        satisMejor += datos.satis[i][i];
    }
    // Prepara los datos de la funcion
    tSol s;
    s.sol.assign(datos.numNinos,0);
    s.sumaSatis = 0;
    std::vector<bool> asignados(datos.numJuguetes,false);
    resolver(datos,0,s,asignados,satisMejor, acum);
    std::cout << satisMejor << '\n';
    return true;
}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf());
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

5. Adornando la casa por Navidad.

Adornando la casa por Navidad

Este año mis padres han decidido renovar los adornos que tradicionalmente ponemos en la casa por Navidad. Quieren comprar algunas figuras para el jardin delantero y para colgar en la fachada; un enorme árbol del que cuelguen bolas, luces y muñecos; y varias guirnaldas para las paredes y techos. Han seleccionado en un catálogo todos aquellos objetos que les gustaría poner. Ahora quieren maximizar la superficie cubierta por los adornos, sin sobrepasar el presupuesto que tienen. Debemos tener en cuenta que los adornos no se pueden partir, ya que correríamos el riesgo de provocar un cortocircuito eléctrico.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de $n+1$ líneas. En la primera se indica el número de objetos que se pueden comprar ($1 \leq n \leq 30$) y el presupuesto con que se cuenta. En las n siguientes se indican el coste y la superficie que ocupa cada uno de los objetos.

Salida

Para cada caso de prueba se escribe en una línea la máxima superficie que pueden cubrir.

Entrada de ejemplo

```
2 10
1 1
10 4
4 10
4 2
3 4
5 5
2 1
4 10
4 6
3 4
5 5
3 1
8 15
7 6
5 4
10 8
8 9
6 8
5 6
7 5
6 8
```

Salida de ejemplo

```
4
10
11
17
```

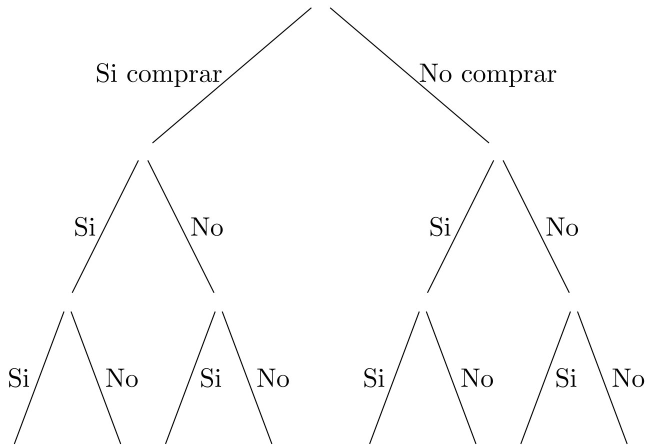
1

5.1. Objetivos del problema.

- Diferenciar los problemas que se resuelven con un árbol de soluciones binario de los que se resuelven con un árbol de soluciones general.
- Practicar la técnica de estimación para podar el árbol de soluciones.

5.2. Ideas generales.

- El problema se resuelve con la técnica de vuelta atrás con un árbol de soluciones binario.
- En la solución debemos obtener los adornos que queremos comprar. Para resolverlo utilizaremos un vector solución de dimensión el número de adornos y en cada etapa marcaremos si conviene comprar el adorno o no conviene.
- El árbol de soluciones es binario:



- Se trata de un problema de optimización, maximizar la superficie cubierta con los adornos, por lo que debemos guardar la solución mejor hasta el momento en que no encontramos.
- Cuando se encuentra una solución mejor que la que tenemos guardada, cambiaremos la solución mejor por la nueva que hemos encontrado.
- Es necesario inicializar la solución mejor antes de llamar a la función recursiva. El mejor valor inicial, porque nos permite podar más el árbol de soluciones es una solución válida, cuya superficie por unidad de precio sea lo mayor posible. Atención porque debe ser una solución válida del problema. Podemos ordenar los adornos por superficie por unidad de precio y comprar todos los adornos hasta cubrir el presupuesto en el orden obtenido y sin partir los adornos (si los partimos la solución ya no será válida). Cómo los adornos están ordenados por mayor superficie por unidad de coste, esta solución cubrirá una superficie bastante parecida a la de la solución final, por lo que las soluciones *malas* se podarán muy pronto.

  Si no es posible encontrar una solución válida con facilidad, se inicializará al menor valor del tipo por tratarse de un problema de maximización. Si se tratase de un problema de minimización se inicializaría al valor máximo del tipo.
- Al tratarse de un árbol binario no se utiliza un bucle para tratar cada una de las opciones, sino que se tratan secuencialmente.
- El esquema general de vuelta atrás para los problemas de optimización con árbol de exploración binario es:

```
void vueltaAtras (TDatos const& datos, int k, Tupla & sol, int coste,
   Tupla & solMejor, int & costeMejor) {
   // Se toma el objeto
   sol[k] = true;
   coste += ...
```

```
    if (esValida(sol, k,..)){
       if (esSolucion(sol, k,..)){
             if (esSolucionMejor(...)
                solMejor = sol; costeMejor = coste;
       }
       else
             vueltaAtras(datos, k + 1, sol, coste, solMejor, costeMejor);
    }
    coste -= ....
    // No se toma el objeto
    sol[k] = false;
    // Si no se toma el objeto normalmente no hay que actualizar el coste ni comprobar que es
    if (esSolucion(sol, k..)){
             if (esSolucionMejor(...)
                solMejor = sol; costeMejor = coste;
    }
    else  if (estimacion(...))
             vueltaAtras(datos, k + 1, sol, coste, solMejor, costeMejor);
 }
```

- Los parámetros indican:
  - `TDatos const & datos`, son los datos de entrada al problema que se necesiten para resolverlo.
  - `int k`, etapa del árbol en que se encuentra la ejecución.
  - `Tupla & sol`, vector en el que se va construyendo la solución.
  - `int coste`, coste de la solución que se está construyendo.
  - `Tupla & solMejor`, vector en el que se guarda la solución mejor hasta este momento.
  - `int coste`, coste de la solución mejor.
- En este problema:
  - `esValida(sol, k)` debe comprobar que el coste de los adornos no supera el presupuesto (debe utilizarse la técnica de marcaje explicada en ...).
  - `esSolucion(sol, k)` es cierto cuando la etapa `k` coincide con el último adorno que se puede comprar.
- En el árbol de exploración se puede explorar antes la rama en la que no se compra el objeto. Se debe elegir aquella opción en la cual se realizan más podas y por lo tanto el tiempo de ejecución es menor. En este problema la solución mejor debería estar cerca de la solución que contiene los adornos con mayor superficie por unidad de precio, por lo que al elegir el árbol de exploración que recorre primero la rama en la que si se compra el objeto encontramos las mejores soluciones *muy rápido* lo que nos permite podar antes, mediante la técnica de la estimación, las ramas que no pueden mejorar la solución que ya tenemos calculada
- Al ser un problema de optimización permite realizar estimaciones. Antes de llamar a la función recursiva de vuelta atrás se comprobará si estimando la superficie que se puede cubrir con los objetos que todavía no se han tratado se puede llegar a cubrir más superficie que con la mejor solución que se tiene hasta este momento. La estimación que se aplica en este caso es:
  - Tendremos los objetos ordenados por mayor superficie por unidad de coste en el vector de entrada a la función recursiva. Los objetos se irán tratando en este orden.
  - Antes de llamar a la función recursiva calcularemos la superficie que podríamos cubrir con los adornos que vienen a continuación en el vector de entrada sin superar el presupuesto que tenemos y suponiendo que el último adorno lo podríamos cortar para cubrir exactamente el presupuesto.
  - Si con estos objetos mejoramos la superficie que cubríamos con la solución mejor que teníamos hasta este momento realizaremos la llamada recursiva, Si no llegamos a cubrir la superficie que cubríamos con la solución mejor hasta este momento no realizaremos la llamada, ya que

ninguna compra puede dar una superficie mayor que la que hemos estimado con el presupuesto que tenemos.

- Observad que solo es necesario estimar en el caso de que no se coja el objeto. En la estimación de la etapa anterior se consideró que este objeto se iba a comprar y con esta consideración se concluyó que se debía realizar la llamada recursiva. Por lo tanto en esta etapa si se compra el objeto se está siguiendo la estimación ya realizada en la etapa anterior por lo que se debe realizar la llamada.

5.3. Algunas cuestiones sobre implementación.

- El problema pide únicamente la superficie que podremos cubrir, pero en la solución del problema calcularemos también el vector solución para facilitar la depuración de la solución.

5.4. Implementación.

```
// Se utiliza un vector de pares para almacenar el coste y la superficie
// para facilitar el ordenarlo por superficie por unidad de coste
struct tDatos {
    int costeMax;
    std::vector<std::pair<int,int>> costeSuperf;
};

// Para ordenar los objetos hacemos una clase comparador que
// sobrecarga el operador parentesis
class comparador {
public:
    bool operator()(std::pair<int, int> p1, std::pair<int, int> p2) {
        return (float)p1.second/p1.first > (float)p2.second / p2.first;
    }
};

// Para escribir la solucion
std::ostream& operator<< (std::ostream& salida, std::vector<bool> const& v){
    for (int i = 0; i < v.size(); ++i)
        if (v[i]) std::cout << i << ' ';
    std::cout << '\n';
    return salida;
}

// Para estimar la mayor superficie que se puede conseguir desde este momento
int estimar(tDatos const& datos, int costeAct, int k){
    int i = k+1; int sumaCoste = costeAct; int sumaSuperf = 0;
    while (i < datos.costeSuperf.size() &&
           sumaCoste + datos.costeSuperf[i].first <= datos.costeMax){
        sumaCoste += datos.costeSuperf[i].first;
        sumaSuperf += datos.costeSuperf[i].second;
        ++i;
    }
    // La parte que queda sin completar. Como la division es entera se suma 1 al resultado
    // para asignar una cota superior a la variable entera
    if ( i < datos.costeSuperf.size() && sumaCoste < datos.costeMax)
        sumaSuperf += datos.costeSuperf[i].second *
              (datos.costeMax-sumaCoste)/datos.costeSuperf[i].first + 1;
    return sumaSuperf;
}
// Parametros
// datos - datos de entrada
// k - etapa
// sol - vector de dimension el numero de adornos, guarda la solucion actual
// costeAct - coste de la solucion actual
```

```
// superficieAct - Superficie de la solucion actual
// solMejor - solucion mejor hasta este momento
// superficieMejor - superficie de la solucion mejor
void resolver (tDatos const& datos, int k, std::vector<bool> &sol, int costeAct,
               int superficieAct, std::vector<bool> & solMejor, int& superficieMejor) {
    // Compra el objeto
    sol[k] = true;
    costeAct += datos.costeSuperf[k].first;
    superficieAct += datos. costeSuperf[k].second;
    if (costeAct ≤ datos.costeMax) { // Si es valida
        if (k = sol.size() - 1) { // Es solucion
            if (superficieAct > superficieMejor) { // Solucion mejor
                superficieMejor = superficieAct;
                solMejor = sol;
            }
        }
        else {
           resolver(datos, k+1, sol, costeAct, superficieAct, solMejor, superficieMejor);
        }
    }
    costeAct -= datos.costeSuperf[k].first; // Recupera el valor
    superficieAct -= datos. costeSuperf[k].second; // Recupera el valor
    // No compra el objeto
    sol[k] = false;
    // Siempre es valido
    if (k = sol.size() - 1) { // Es solucion
        if (superficieAct > superficieMejor) { // Solucion mejor
            superficieMejor = superficieAct;
            solMejor = sol;
        }
    }
    else {
        if (estimar(datos, costeAct, k) + superficieAct > superficieMejor)
            resolver(datos, k+1, sol, costeAct, superficieAct, solMejor, superficieMejor);
    }
}

int inicializarSuperficie( tDatos const& Obj, int costeMax) {
    int i = 0; int sumaSuperficie = 0; int sumaCoste = 0;
    while (i < Obj.costeSuperf.size() && sumaCoste + Obj.costeSuperf[i].first ≤ costeMax) {
        sumaSuperficie += Obj.costeSuperf[i].second;
        sumaCoste += Obj.costeSuperf[i].first;
        ++i;
    }
    return sumaSuperficie;
}

bool resuelveCaso() {
    int numObjetos;
    std::cin >> numObjetos;
    if (!std::cin) return false;
    tDatos datos;
    std::cin >> datos.costeMax;
    // Lee el coste y la superficie de cada adorno
    for (int i = 0; i < numObjetos; ++i) {
        int coste, superficie;
        std::cin >> coste >> superficie;
        datos.costeSuperf.push_back({coste,superficie});
    }
    // ordena los datos por superficie por unidad de coste
    std::sort(datos.costeSuperf.begin(), datos.costeSuperf.end(), comparador());
    std::vector<bool> sol(numObjetos);
```

```cpp
    int costeAct = 0; int superficieAct = 0;
    int superficieMejor = inicializarSuperficie(datos, datos.costeMax);
    std::vector<bool> solMejor;
    resolver(datos, 0, sol, costeAct, superficieAct, solMejor, superficieMejor);
    //std::cout << solMejor;
    std::cout << superficieMejor << '\n';
    return true;
}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf());
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

6. Generar combinaciones sin repetición de m elementos tomados de n en n.

Generar todas las combinaciones sin repetición de un conjunto de elementos

Las combinaciones sin repetición de m elementos tomados de n en n son los distintos grupos de n elementos distintos, que podemos formar con los m elementos, de forma que dos grupos con los mismos elementos colocados de diferente forma se consideran la misma combinación.

El número de combinaciones sin repetición de m elementos tomados de n en n es $C_{m,n} = m! / (n!(m-n)!)$.

Por ejemplo dadas las letras `a`, `b`, y `c`, las combinaciones sin repetición de estos tres elementos tomados de dos en dos son `ab`, `ac`, y `bc`.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso tiene una línea donde se indica el número de letras que se consideran m y el tamaño de la palabra n.

Se consideran únicamente las letras del alfabeto anglosajón, cogiéndose las m primeras. Se garantiza que $n <= m$.

Salida

Para cada caso de prueba se muestran todas las posibles palabras que se pueden formar una por línea. Después de cada caso se muestra una línea en blanco.

Entrada de ejemplo

```
3 2
5 3
```

Salida de ejemplo

```
ab
ac
bc
abc
abd
abe
acd
ace
ade
bcd
bce
bde
cde
```

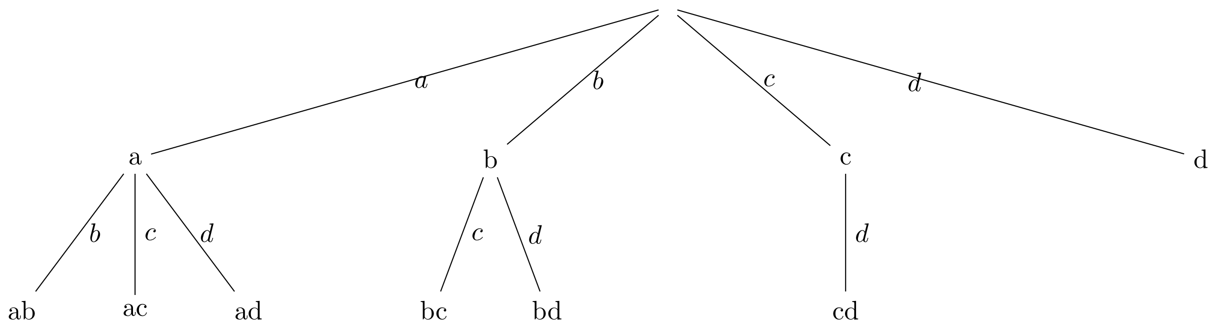
Autor: Isabel Pita

6.1. Objetivos del problema.

- Aprender a generar combinaciones de elementos mediante la técnica de vuelta atrás (*backtracking*).
- Manejar un *árbol de exploración* cuyas ramas dependen del nodo que las genera

6.2. Ideas generales.

- El problema se resuelve con la técnica de vuelta atrás.
- Al generar las combinaciones no debe tenerse en cuenta el orden en que aparecen los elementos. Por ejemplo `abc` y `acb` son la misma combinación.
- Generaremos las combinaciones siguiendo el siguiente árbol de exploración.



- La solución se genera siguiendo el esquema dado por el árbol de exploración. Se utiliza un parámetro para indicar el valor por el que debe comenzar el bucle a probar las soluciones.
- El esquema general de vuelta atrás para encontrar todas las soluciones a un problema es:

```
void vueltaAtras (TDatos const& datos, int k, Tupla & sol, tipo primero) {
   for(auto i = primero; i < ultHijoNivel(k); i = sigHijoNivel(k)){
      sol[k] = i;
      if (esValida(sol, k)){
         if (esSolucion(sol, k))
            tratarSolucion(sol);
         else
            vueltaAtras(datos, k + 1, sol, sigHijoNivel(k));
      }
   }
}
```

- Los parámetros indican:
  - `TDatos const & datos`, son los datos de entrada al problema que se necesiten para resolverlo.
  - `int k`, etapa del árbol en que se encuentra la ejecución.
  - `Tupla & sol`, vector en el que se va construyendo la solución.
  - `tipo primero`, valor por el que debe empezar el bucle. Es del tipo de la variable de control del bucle.
- En este problema:
  - `primerHijoNivel(k)` es el carácter `'a'`.
  - `ultHijoNivel(k)` es el carácter `'c'`.
  - `sigHijoNivel(k)` debe sumar uno al carácter.
  - `esValida(sol, k)` es siempre cierto, ya que no hay ninguna restricción para construir la palabra, por lo tanto no es necesario hacer esta comprobación.
  - `esSolucion(sol, k)` es cierto cuando la etapa `k` coincide con el tamaño de la palabra que se quiere construir.

6.3. Modificaciones al problema.

- Generar las combinaciones con repetición de m elementos tomados de n en n. El problema es semejante al anterior, pero deben generarse palabras con letras repetidas. Para ello se modificará la inicialización del bucle.
- Generar las combinaciones que cumplen una serie de propiedades.
- Generar la combinación mejor según algún criterio. En este caso se realizarán podas y estimaciones.

6.4. Implementación.

```
// Escribe una solucion
void escribirSolucion(std::vector<char> const& v) {
    for (char c : v) std::cout << c;
    std::cout << '\n';
}

// Calcula de forma recursiva las combinaciones de los elementos
void combinaciones (int m, int n, int k, std::vector<char> & sol, char primero) {
    for (char i = primero; i < 'a' + m; ++i){
        sol[k] = i;
        if (k == n-1){ // Es solucion
            escribirSolucion(sol);
        }
        else { // Sigue completando la solucion
            combinaciones(m,n,k+1,sol, i+1);
        }
    }
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuracion, y escribiendo la respuesta
bool resuelveCaso() {
    // Lectura de los datos de entrada
    int m,n; std::cin >> m >> n;
    if (!std::cin) return false;
    // Generar las soluciones
    std::vector<char> sol(n);
    combinaciones(m,n,0,sol, 'a');
    return true;
}


int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso()) {} //Resolvemos todos los casos

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```