XXXI Coloquio Víctor Neumann-Lara

Discreture

Una librería para construir objetos combinatorios

Miguel Raggi
mraggi@gmail.com

Escuela Nacional de Estudios Superiores
UNAM

2 de marzo

Índice:

Índice:

¿Qué es?

Definición

 *iscreture es una librería de fuente abierta escrita en C++ moderno, diseñada para ayudar a la investigación en combinatoria.*

¿Qué es?

Definición

 *iscreture es una librería de fuente abierta escrita en C++ moderno, diseñada para ayudar a la investigación en combinatoria.*

`https://github.com/mraggi/discreture`

¿Qué es?

Definición

 *iscreture es una librería de fuente abierta escrita en C++ moderno, diseñada para ayudar a la investigación en combinatoria.*

`https://github.com/mraggi/discreture`

- Se puede bajar el código completo ahí, compilar, usar en lo que quieras, etc.

¿Qué es?

Definición

 *iscreture es una librería de fuente abierta escrita en C++ moderno, diseñada para ayudar a la investigación en combinatoria.*

`https://github.com/mraggi/discreture`

- Se puede bajar el código completo ahí, compilar, usar en lo que quieras, etc.
- Trae instrucciones (en inglés).

Descripción larga

Características:

- Es un conjunto de archivos de código que puedes utilizar en tu propio código, que sirve para generar y trabajar con objetos combinatorios comunes, como combinaciones, permutaciones, particiones, etc.

Descripción larga

Características:

- Es un conjunto de archivos de código que puedes utilizar en tu propio código, que sirve para generar y trabajar con objetos combinatorios comunes, como combinaciones, permutaciones, particiones, etc.
- Además, se generan de uno por uno, sin necesidad de guardar toooodos en la memoria.

Descripción larga

Características:

- Es un conjunto de archivos de código que puedes utilizar en tu propio código, que sirve para generar y trabajar con objetos combinatorios comunes, como combinaciones, permutaciones, particiones, etc.
- Además, se generan de uno por uno, sin necesidad de guardar toooodos en la memoria.
- Es decir, cada que le pides al programa uno, te lo genera al momento y te lo da.

Descripción larga

Características:

- Es un conjunto de archivos de código que puedes utilizar en tu propio código, que sirve para generar y trabajar con objetos combinatorios comunes, como combinaciones, permutaciones, particiones, etc.
- Además, se generan de uno por uno, sin necesidad de guardar toooodos en la memoria.
- Es decir, cada que le pides al programa uno, te lo genera al momento y te lo da.
- Trae muchas funciones de utilidad para hacer investigación en combinatoria.

Descripción larga

Características:

- Es un conjunto de archivos de código que puedes utilizar en tu propio código, que sirve para generar y trabajar con objetos combinatorios comunes, como combinaciones, permutaciones, particiones, etc.
- Además, se generan de uno por uno, sin necesidad de guardar toooodos en la memoria.
- Es decir, cada que le pides al programa uno, te lo genera al momento y te lo da.
- Trae muchas funciones de utilidad para hacer investigación en combinatoria.
- Es más o menos fácil de utilizar y es muy eficiente.

Descripción larga

Características:

- Es un conjunto de archivos de código que puedes utilizar en tu propio código, que sirve para generar y trabajar con objetos combinatorios comunes, como combinaciones, permutaciones, particiones, etc.
- Además, se generan de uno por uno, sin necesidad de guardar toooodos en la memoria.
- Es decir, cada que le pides al programa uno, te lo genera al momento y te lo da.
- Trae muchas funciones de utilidad para hacer investigación en combinatoria.
- Es más o menos fácil de utilizar y es muy eficiente.
- Manuel Alejandro Romo de Vivar (Manolo) programó los caminos de dyck y parte de los de Motzkin.

Descripción larga

Características:

- Es un conjunto de archivos de código que puedes utilizar en tu propio código, que sirve para generar y trabajar con objetos combinatorios comunes, como combinaciones, permutaciones, particiones, etc.
- Además, se generan de uno por uno, sin necesidad de guardar toooodos en la memoria.
- Es decir, cada que le pides al programa uno, te lo genera al momento y te lo da.
- Trae muchas funciones de utilidad para hacer investigación en combinatoria.
- Es más o menos fácil de utilizar y es muy eficiente.
- Manuel Alejandro Romo de Vivar (Manolo) programó los caminos de dyck y parte de los de Motzkin.
- El nombre me fue sugerido por César Benjamín García Martínez.

Cuestiones legales

- La librería compila en Linux. Es altamente probable, pero no seguro, que compile en FreeBSD, OS X, Hurd, etc.

Cuestiones legales

- La librería compila en Linux. Es altamente probable, pero no seguro, que compile en FreeBSD, OS X, Hurd, etc.
- Tiene licencia “Apache 2.0”. Según lo [poco] que entiendo, puedes básicamente hacer lo que sea, salvo demandarme si el código se come tu computadora, te quita el novio, etc.

Cuestiones legales

- La librería compila en Linux. Es altamente probable, pero no seguro, que compile en FreeBSD, OS X, Hurd, etc.
- Tiene licencia “Apache 2.0”. Según lo [poco] que entiendo, puedes básicamente hacer lo que sea, salvo demandarme si el código se come tu computadora, te quita el novio, etc.
- Compilarlo e incluírlo en tu proyecto es muy fácil: Viene en las instrucciones.

Cuestiones legales

- La librería compila en Linux. Es altamente probable, pero no seguro, que compile en FreeBSD, OS X, Hurd, etc.
- Tiene licencia "Apache 2.0". Según lo [poco] que entiendo, puedes básicamente hacer lo que sea, salvo demandarme si el código se come tu computadora, te quita el novio, etc.
- Compilarlo e incluírlo en tu proyecto es muy fácil: Viene en las instrucciones.
- Si la utilizas, me gustaría que me avisaras, aunque no es necesario. Si la citas en un artículo, ¡mejor!

Cuestiones legales

- La librería compila en Linux. Es altamente probable, pero no seguro, que compile en FreeBSD, OS X, Hurd, etc.
- Tiene licencia "Apache 2.0". Según lo [poco] que entiendo, puedes básicamente hacer lo que sea, salvo demandarme si el código se come tu computadora, te quita el novio, etc.
- Compilarlo e incluírlo en tu proyecto es muy fácil: Viene en las instrucciones.
- Si la utilizas, me gustaría que me avisaras, aunque no es necesario. Si la citas en un artículo, ¡mejor!
- Además, con mucho gusto puedo ayudar a quien quiera.

Motivación

Motivación

- Supongamos que quieres, por ejemplo, hacer algo con todas las parejas de elementos un conjunto. ¿Cómo programas eso?

Motivación

- Supongamos que quieres, por ejemplo, hacer algo con todas las parejas de elementos un conjunto. ¿Cómo programas eso?
- Es muy fácil:

```
std::vector<Objeto> X;
// ...
for (int i = 0; i < X.size(); ++i)
{
  for (int j = i+1; j < X.size(); ++j)
  {
    // Hacer algo con X[i] y X[j]
  }
}
```

Motivación

- Ahora supón que quieres hacer algo con todas las tercias de elementos.

Motivación

- Ahora supón que quieres hacer algo con todas las tercias de elementos.
- Igual, sólo con un tercer for, con una tercera variable k que denote al tercero.

Motivación

- Ahora supón que quieres hacer algo con todas las tercias de elementos.
- Igual, sólo con un tercer for, con una tercera variable k que denote al tercero.
- ¿Y con cuartetas, quintetas, etc.?

¿Quintetas?

```
std::vector<Objeto> X;
// ...
for (int i = 0; i < X.size(); ++i)
{
  for (int j = i+1; j < X.size(); ++j)
  {
    for (int k = j+1; j < X.size(); ++k)
    {
      for (int l = k+1; l < X.size(); ++l)
      {
        for (int m = l+1; m < X.size(); ++m)
        {
          // Hacer algo con X[i],X[j],X[k],X[l],
        }
      }
    }
  }
```

Mejor solución

En discreture, el código se vería así:

```
std::vector<Objeto> X;
// ...
combinations C(X.size(),5);
for (auto& c : C) // c contiene los indices
{
  auto quinteta = compose(X,c);
  // O puedes trabajar con X[c[0]], X[c[1]], etc.
}
```

Sublibrerías

- Combinaciones. Subconjuntos (de índices) de un tamaño dado.
  - Ejemplo: [0,3,4], [0,1,5] $\in$ combinations(6,3)

Sublibrerías

- Combinaciones. Subconjuntos (de índices) de un tamaño dado.
  - Ejemplo: [0,3,4], [0,1,5] $\in$ combinations(6,3)
- Permutaciones. S_n
  - Ejemplo: [0,1,2], [2,0,1] $\in$ permutations(3)

Sublibrerías

- Combinaciones. Subconjuntos (de índices) de un tamaño dado.
  - Ejemplo: [0,3,4], [0,1,5] $\in$ combinations(6,3)
- Permutaciones. S_n
  - Ejemplo: [0,1,2], [2,0,1] $\in$ permutations(3)
- Particiones. Números que suman un número dado.
  - Ejemplo: [6,4,1], [3,3,3,1,1] $\in$ partitions(11)

Sublibrerías

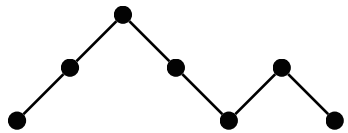
- Combinaciones. Subconjuntos (de índices) de un tamaño dado.
  - Ejemplo: [0,3,4], [0,1,5] $\in$ combinations(6,3)
- Permutaciones. S_n
  - Ejemplo: [0,1,2], [2,0,1] $\in$ permutations(3)
- Particiones. Números que suman un número dado.
  - Ejemplo: [6,4,1], [3,3,3,1,1] $\in$ partitions(11)
- Subconjuntos. Dados por su función característica
  - Ejemplo: [001101], [100111] $\in$ subsets(6)

Sublibrerías

- Combinaciones. Subconjuntos (de índices) de un tamaño dado.
  - Ejemplo: [0,3,4], [0,1,5] $\in$ combinations(6,3)
- Permutaciones. S_n
  - Ejemplo: [0,1,2], [2,0,1] $\in$ permutations(3)
- Particiones. Números que suman un número dado.
  - Ejemplo: [6,4,1], [3,3,3,1,1] $\in$ partitions(11)
- Subconjuntos. Dados por su función característica
  - Ejemplo: [001101], [100111] $\in$ subsets(6)
- Multiconjuntos. Todos los arreglos que son menores coordenada a coordenada que un arreglo fijo.
  - Ejemplo: [2,1,3], [0,1,1] $\in$ multisets([3,1,3])

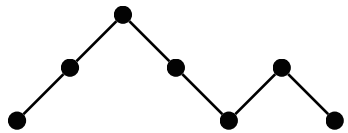
Sublibrerías

- Combinaciones. Subconjuntos (de índices) de un tamaño dado.
  - Ejemplo: [0,3,4], [0,1,5] $\in$ combinations(6,3)
- Permutaciones. S_n
  - Ejemplo: [0,1,2], [2,0,1] $\in$ permutations(3)
- Particiones. Números que suman un número dado.
  - Ejemplo: [6,4,1], [3,3,3,1,1] $\in$ partitions(11)
- Subconjuntos. Dados por su función característica
  - Ejemplo: [001101], [100111] $\in$ subsets(6)
- Multiconjuntos. Todos los arreglos que son menores coordenada a coordenada que un arreglo fijo.
  - Ejemplo: [2,1,3], [0,1,1] $\in$ multisets([3,1,3])
- Caminos de Dyck. De $(0,0)$ a $(2n,0)$ que y nunca es negativo y siempre van uno arriba o uno abajo.

 $\in$ dyck_paths(3)

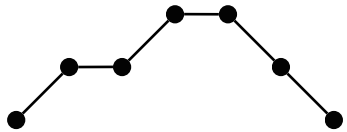
Sublibrerías

- Combinaciones. Subconjuntos (de índices) de un tamaño dado.
  - Ejemplo: [0,3,4], [0,1,5] $\in$ combinations(6,3)
- Permutaciones. S_n
  - Ejemplo: [0,1,2], [2,0,1] $\in$ permutations(3)
- Particiones. Números que suman un número dado.
  - Ejemplo: [6,4,1], [3,3,3,1,1] $\in$ partitions(11)
- Subconjuntos. Dados por su función característica
  - Ejemplo: [001101], [100111] $\in$ subsets(6)
- Multiconjuntos. Todos los arreglos que son menores coordenada a coordenada que un arreglo fijo.
  - Ejemplo: [2,1,3], [0,1,1] $\in$ multisets([3,1,3])
- Caminos de Dyck. De $(0,0)$ a $(2n,0)$ que y nunca es negativo y siempre van uno arriba o uno abajo.



$\in$ dyck_paths(3)

- Caminos de Motzkin. Igual, pero también se vale ir horizontal.



$\in$ motzkin_paths(6)

Índice:

Uso básico

Para iterar sobre un objeto combinatorio, primero se declara el objeto y después se utiliza la notación estándar de C++11:

Uso básico

Para iterar sobre un objeto combinatorio, primero se declara el objeto y después se utiliza la notación estándar de C++11:

```
combinations X(6,3);
for (auto& x : X)
  cout << x << "␣";
```

Uso básico

Para iterar sobre un objeto combinatorio, primero se declara el objeto y después se utiliza la notación estándar de C++11:

```
combinations X(6,3);
for (auto& x : X)
   cout << x << "␣";
```

Produce como resultado:

```
[ 0 1 2 ] [ 0 1 3 ] [ 0 2 3 ]  [ 1 2 3 ] [ 0 1 4 ]
[ 0 2 4 ] [ 1 2 4 ] [ 0 3 4 ] [ 1 3 4 ] [ 2 3 4 ]
```

Claro, `x[0]`,`x[1]`,... dan los números de la combinación.

Otro ejemplo

Por ejemplo, para imprimir todas las maneras de sumar 10 elementos:

```
partitions X(5);
for (auto& x : X)
  cout << x << endl;
```

Esto imprime todas las maneras de sumar 5 con enteros positivos:

```
[ 1 1 1 1 1 ]
[ 2 1 1 1 ]
[ 3 1 1 ]
[ 2 2 1 ]
[ 4 1 ]
[ 3 2 ]
[ 5 ]
```

Reverse Iterators

Además, algunos pueden recorrerse en reversa de la manera estándar en C++:

```
combinations X(5,3);
for (auto it = X.rbegin(); it != X.rend(); ++it)
{
  auto& x = *it;
  // Hacer cosas con x
}
```

No está completamente implementado esto para todos, pero pronto.

Random Access Iterators

Combinaciones, Permutaciones y Subsets pueden acceder a sus elementos:

```
permutations X(12);
cout << X[157122128] << endl;
```

Imprime `[ 3 11 2 10 9 0 4 1 6 7 5 8 ]`, que es la permutación número 157,122,128 en el órden de iteración (lexicográfico).

Random Access Iterators

Combinaciones, Permutaciones y Subsets pueden acceder a sus elementos:

```
permutations X(12);
cout << X[157122128] << endl;
```

Imprime `[ 3 11 2 10 9 0 4 1 6 7 5 8 ]`, que es la permutación número 157,122,128 en el órden de iteración (lexicográfico).

También se puede usar `get_index(objeto)` para que te diga en qué posición está una combinación (permutación, subconjunto) que hayas encontrado de alguna otra manera.

Dyck y Motzkin

Si quieremos generar, por ejemplo, todas las maneras de acomodar $2 \cdot 3 = 6$ paréntesis que tenga sentido, podemos hacer lo siguiente:

```
dyck_paths X(3);
for (auto& x : X)
  cout << dyck_paths::to_string(x, "()") << endl;
```

lo cual imprime: `((()))` `(()())` `()(())` `(())()` `()()()`

O también:

```
motzkin_paths X(6);
for (auto& x : X)
   cout<<motzkin_paths::to_string(x, "(*)")<<endl;
```

Lo cual imprime:

```
******    ()****    (*)***    *()***    (**)**
*(*)**    **()**    (***)*    *(**)*    **(*)*
***()*    (****)    *(***)    **(**)    ***(*)
****()    (())**    (()*)*    ((*))*    (*())*
*(())*    (()**)    ((*)*)    (*()*)    *(()*)
((**))    (*(*))    *((*))    (**())    *(*())
**(())    ()()**    ()(*)*    ()*()*    (*)()*
*()()*    ()(**)    ()*(*)    (*)(*)    *()(*)
()**()    (*)*()    *()*()    (**)()    *(*)()
**()()    ((()))    (()())    ()(())    (())()
                    ()()()
```

Algoritmos

Puedes utilizar algoritmos estándar de la STL de C++:

```
motzkin_paths X(10);
std::find_if(X.begin(), X.end(), condicion);
```

encuentra el primer camino de motzkin que satisface la condición (que puedes especificar).

Algoritmos Estándar

Incluso puedes hacer búsqueda binaria, cuando recorrer todos sería imposible:

```
// combination = vector<int>
using combination = combinations::combination;

// 8,233,430,727,600 combinaciones
combinations X(46,23);

auto it = std::partition_point(X.begin(), X.end(),
  [](const combination& x)
  {
    if (x.back() < 36)
      return true;
    return false;
  });
cout << *it << endl;
```

find_all

Combinations (y en un futuro cercano las demás) puede hacer esto:

find_all

Combinations (y en un futuro cercano las demás) puede hacer esto:

```
combinations X(10,5);
auto todos = X.find_all(
[](const vector<int>& comb) -> bool {
  for (int i = 0; i < comb.size()-1; ++i)
    if (comb[i]+1 == comb[i+1])
      return false;
  return true;
});
for (auto& v : todos)
  cout << v << endl;
```

Imprime [0 2 4 6 8] [0 2 4 6 9] [0 2 4 7 9] [0 2 5 7 9] [0 3 5 7 9] [1 3 5 7 9], que son las combinaciones que no tienen dos consecutivos. Lo interesante de esto es que no recorre todas, sino que va "podando".

Índice:

¿Cómo se compara con otros?

Por ejemplo, el código que genera combinaciones de la GNU scientific library se ve así:

```
gsl_combination * c = gsl_combination_calloc(6, 3
do
{
  // gsl_combination_get(c,i) para obtener el
  // i-esimo indice
} while (gsl_combination_next(c) == GSL_SUCCESS);
gsl_combination_free (c);
```

Más sencillo

```
combinations X(6,3);
for (auto& x : X)
{
  // x[i] para acceder al
  // i-esimo indice
}
```

Además de las otras ventajas.

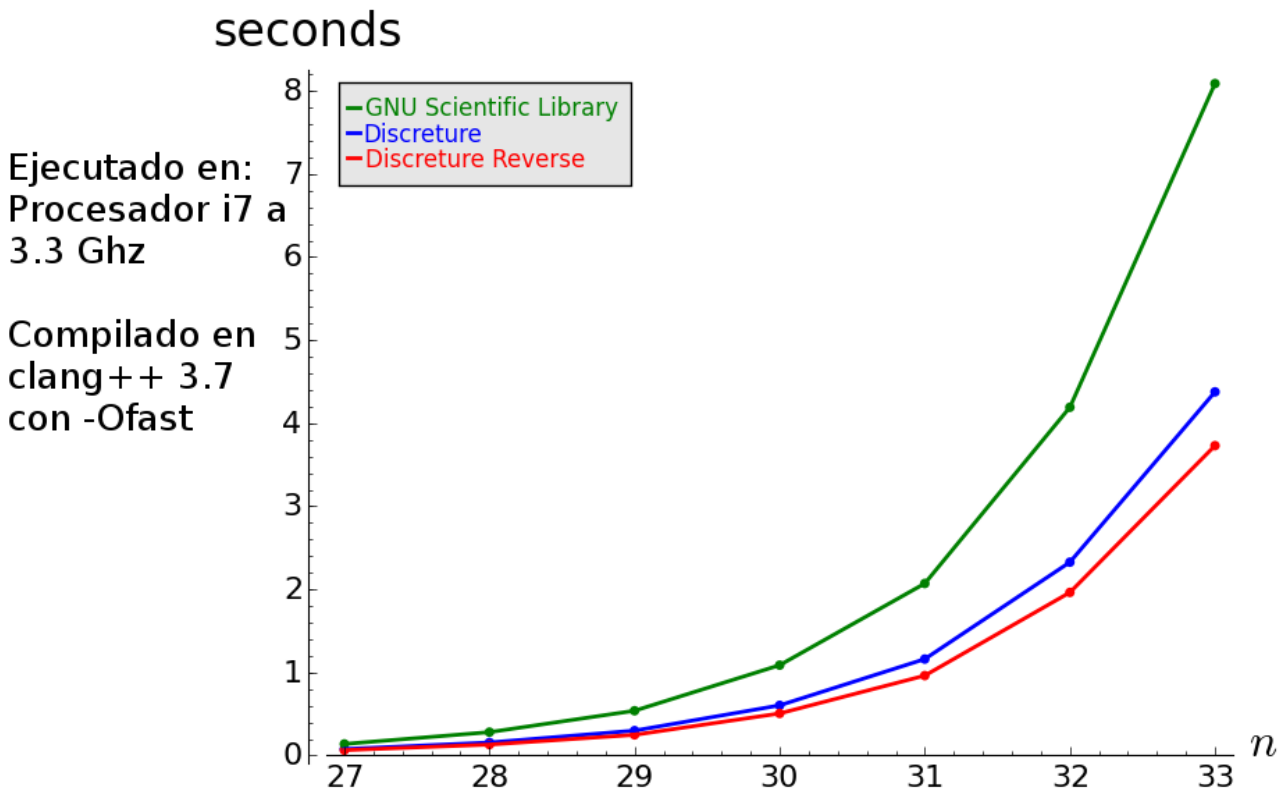
¿Y en velocidad?

¿Y en velocidad?

Tiempo para iterar sobre todas las combinaciones $\binom{n}{\lfloor n/2 \rfloor}$

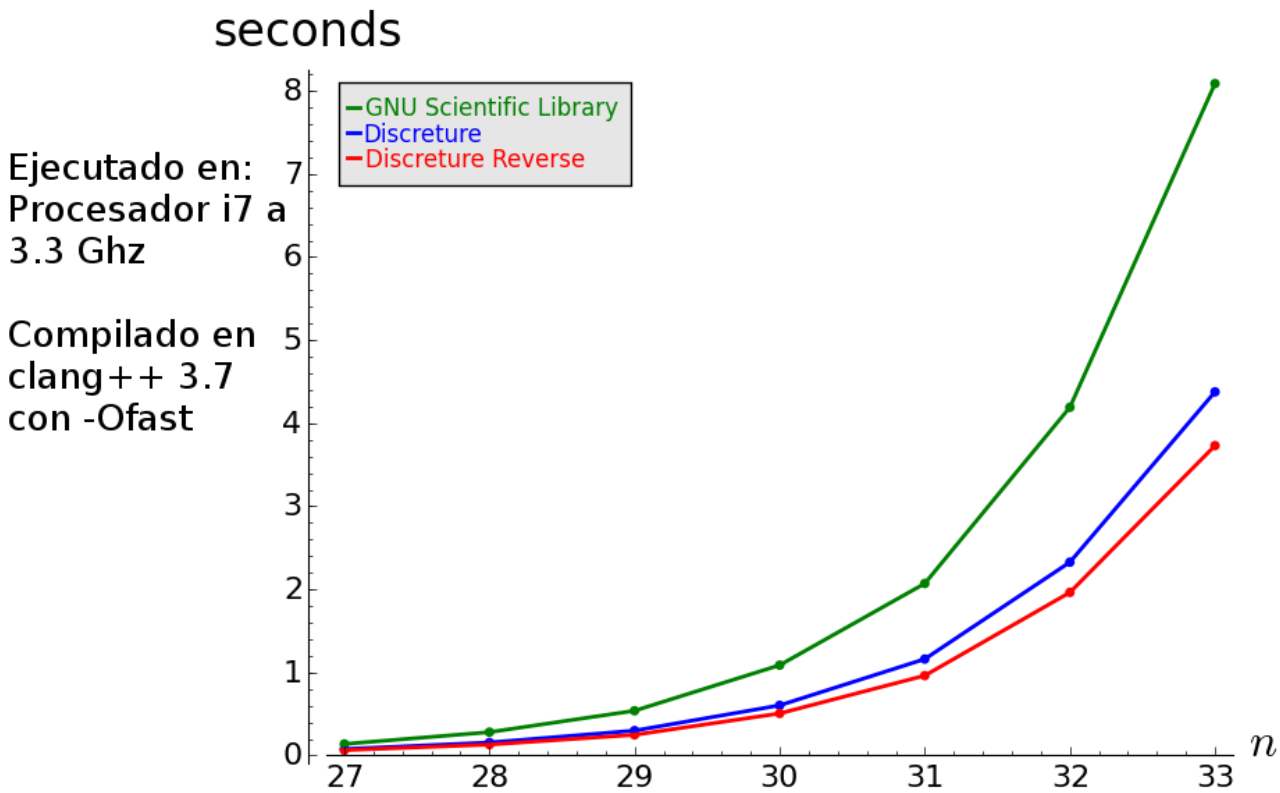
¿Y en velocidad?

Tiempo para iterar sobre todas las combinaciones $\binom{n}{\lfloor n/2 \rfloor}$



¿Y en velocidad?

Tiempo para iterar sobre todas las combinaciones $\binom{n}{\lfloor n/2 \rfloor}$



Para hacer las $\binom{33}{16} = 1,166,803,110$ combinaciones, GSL toma 8.1 segundos, discreture tarda 4.4 segundos y discreture en reversa tarda 3.7 segundos.

Otras comparaciones

- `sagemath` también tiene muchos de estos objetos.

Otras comparaciones

- `sagemath` también tiene muchos de estos objetos.
- En realidad no tiene sentido mostrarles gráficas de comparación:

Otras comparaciones

- `sagemath` también tiene muchos de estos objetos.
- En realidad no tiene sentido mostrarles gráficas de comparación:
- Por ejemplo, para recorrer las combinaciones de 24 en 12, sagemath tarde 12s.

Otras comparaciones

- `sagemath` también tiene muchos de estos objetos.
- En realidad no tiene sentido mostrarles gráficas de comparación:
- Por ejemplo, para recorrer las combinaciones de 24 en 12, sagemath tarde 12s.
- A diferencia de Discreture, que tarda $\approx$ 0s.

Capacidades

En mi laptop, estas son algunas pruebas de tiempo que tarda en recorrer:

Capacidades

En mi laptop, estas son algunas pruebas de tiempo que tarda en recorrer:

- $\binom{32}{16} = 601,080,390$ combinaciones: 2.47527s
- $\binom{32}{16} = 601,080,390$ combinaciones en orden reverso: 2.21083s
- $12! = 479,001,600$ permutaciones: 1.73026s
- $2^{29} = 536,870,912$ subconjuntos : 3.05884s
- $2^{29} = 536,870,912$ subconjuntos (modo rápido): 2.32129s
- $P_{90} = 56,634,173$ particiones de tamaño 90: 1.48486s
- 559,872,000 multiconjuntos de uno aleatorio: 2.34732s
- $C_{18} = \frac{1}{19}\binom{36}{18} = 477,638,700$ caminos de Dyck: 2.35329s
- $M_{20} = 50,852,019$ caminos de Motzkin: 1.23486s

Filosofía de diseño

- Esta librería pretende alcanzar un buen balance entre facilidad de uso y eficiencia.

Filosofía de diseño

- Esta librería pretende alcanzar un buen balance entre facilidad de uso y eficiencia.
- No es ni taaan fácil de usar como sagemath, ni taaan eficiente como usar 17 for's.

Filosofía de diseño

- Esta librería pretende alcanzar un buen balance entre facilidad de uso y eficiencia.
- No es ni taaan fácil de usar como sagemath, ni taaan eficiente como usar 17 for's.
- Mi esperanza es que le ayude a alguien a escribir pequeños programas sencillos que prueben todos los casos de alguna conjetura que tengan (para "n" chiquito), incluso de manera semi-inteligente.

Filosofía de diseño

- Esta librería pretende alcanzar un buen balance entre facilidad de uso y eficiencia.
- No es ni taaan fácil de usar como sagemath, ni taaan eficiente como usar 17 for's.
- Mi esperanza es que le ayude a alguien a escribir pequeños programas sencillos que prueben todos los casos de alguna conjetura que tengan (para "n" chiquito), incluso de manera semi-inteligente.
- Quizás no tenga nada innovador, pero creo que podría ser una herramienta útil para la comunidad combinatoria y por eso la presento aquí.

Índice:

¿Qué le falta?

¿Qué le falta?

Necesito ayuda para lo siguiente:

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.
- Hacer paquetes de instalación para diferentes distros de Linux, y posiblemente otros sistemas operativos.

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.
- Hacer paquetes de instalación para diferentes distros de Linux, y posiblemente otros sistemas operativos.
- Hacer Partitions, Dyck, Motzkin y Multisets random access conts.

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.
- Hacer paquetes de instalación para diferentes distros de Linux, y posiblemente otros sistemas operativos.
- Hacer Partitions, Dyck, Motzkin y Multisets random access conts.
  - Ni he pensado cómo, pero seguro se puede y no es demasiado difícil.

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.
- Hacer paquetes de instalación para diferentes distros de Linux, y posiblemente otros sistemas operativos.
- Hacer Partitions, Dyck, Motzkin y Multisets random access conts.
  - Ni he pensado cómo, pero seguro se puede y no es demasiado difícil.
  - Puedo dar ayuda técnica a alguien que se aviente a hacerlo.

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.
- Hacer paquetes de instalación para diferentes distros de Linux, y posiblemente otros sistemas operativos.
- Hacer Partitions, Dyck, Motzkin y Multisets random access conts.
  - Ni he pensado cómo, pero seguro se puede y no es demasiado difícil.
  - Puedo dar ayuda técnica a alguien que se aviente a hacerlo.
- Estudiar cómo hacerle cuando tenemos objetos combinatorios extremadamente grandes, en donde el *número de elementos* no cabe en un `long long unsigned int` (es decir, es más grande que 2^{64}).

¿Qué le falta?

Necesito ayuda para lo siguiente:

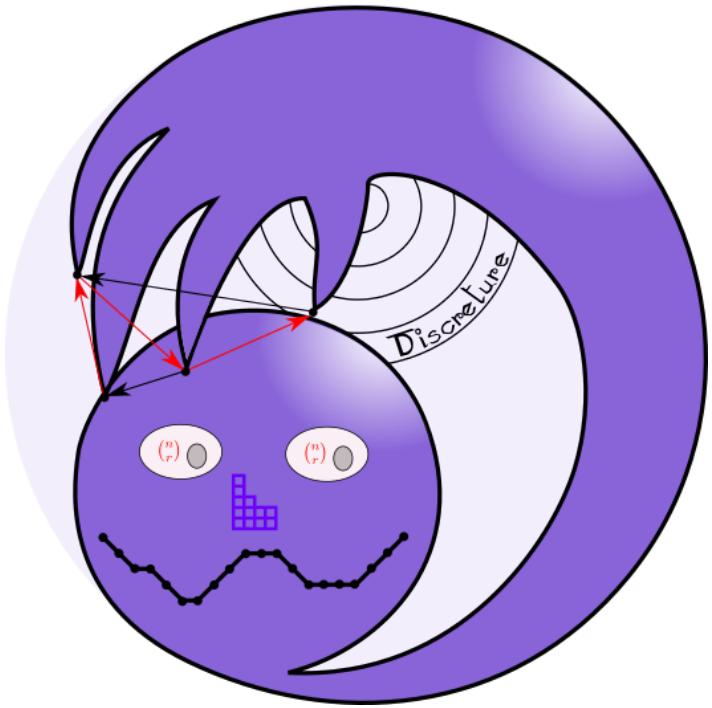
- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.
- Hacer paquetes de instalación para diferentes distros de Linux, y posiblemente otros sistemas operativos.
- Hacer Partitions, Dyck, Motzkin y Multisets random access conts.
  - Ni he pensado cómo, pero seguro se puede y no es demasiado difícil.
  - Puedo dar ayuda técnica a alguien que se aviente a hacerlo.
- Estudiar cómo hacerle cuando tenemos objetos combinatorios extremadamente grandes, en donde el *número de elementos* no cabe en un `long long unsigned int` (es decir, es más grande que 2^{64}).
  - Obviamente en este caso no puedes iterar por todos, pero poder hacer búsqueda binaria ahí no estaría mal.

¿Qué le falta?

Necesito ayuda para lo siguiente:

- Hacer pruebas y reportar errores. Seguro tiene muchos, a pesar de las pruebas que le he hecho.
- Documentar mejor, escribir tutoriales, etc.
- Hacer paquetes de instalación para diferentes distros de Linux, y posiblemente otros sistemas operativos.
- Hacer Partitions, Dyck, Motzkin y Multisets random access conts.
  - Ni he pensado cómo, pero seguro se puede y no es demasiado difícil.
  - Puedo dar ayuda técnica a alguien que se aviente a hacerlo.
- Estudiar cómo hacerle cuando tenemos objetos combinatorios extremadamente grandes, en donde el *número de elementos* no cabe en un `long long unsigned int` (es decir, es más grande que 2^{64}).
  - Obviamente en este caso no puedes iterar por todos, pero poder hacer búsqueda binaria ahí no estaría mal.
  - Benchmarks: comparar contra otros (*e.g.* haskell (¿David Flores?) Mathematica, etc).

¡Muchas Gracias!



github.com/mraggi/discreture