Métodos de Calculo para Combinatoria

Valentina Cartagena Henríquez — valentina.cartagena@usach.cl
Diego Velásquez Figueroa — diego.velasquez.f@usach.cl

Departamento de Matemática y Ciencia de la Computación
Universidad de Santiago de Chile

Segundo Semestre 2023

Abstract

Se realiza un análisis sobre tres de los diversos métodos para calcular operaciones de combinatoria, observando su desempeño en tiempo y alcance.

KeyWords

Triángulo de pascal, numero factorial.

1 Introducción

El siguiente articulo presenta tres metodologías que permiten el cálculo de una combinatoria, la primera hace alusión a la forma $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, el segundo método corresponde a la formula $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ $1 \leq k < n$ $\binom{n}{0} = \binom{n}{n} = 1$, $n \geq 0$, por ultimo el tercer método trabaja utilizando el triángulo de pascal.

2 Métodos

A continuación se señalara cada método sometido a investigación con su respectiva descripción ,análisis y complejidad respectivamente.

2.1 Método Fuerza Bruta

El algoritmo trabaja descomponiendo cada factorial en números primos, posteriormente se simplifica revisando los elementos que coinciden entre los listados de primos de los factoriales $(n-k)!$ y $k!$ con $n!$. Finalmente se multiplican los primos restantes de cada listado, y se realiza la operación $\frac{n!}{k!(n-k)!}$.

2.1.1 Descripción de Algoritmo

```
struct fp
{
    unsigned long long int primo;
    unsigned int apariciones;
    struct fp *sig;
};

struct fp *ingreso_lista(struct fp *top, unsigned long long int j)
{
    struct fp *q = NULL;
```

```
    if (!top || top->primo > j)
    {
        q = malloc(sizeof(struct fp));
        q->primo = j;
        q->apariciones = 1;
        q->sig = top;
        top = q;
    }
    else if (top->primo == j)
    {
        top->apariciones += 1;
    }
    else
    {
        struct fp *temporal = top;
        while (temporal->sig && temporal->sig->primo < j)
        {
            temporal = temporal->sig;
        }

        if (temporal->sig && temporal->sig->primo == j)
        {
            temporal->sig->apariciones += 1;
        }
        else
        {
            q = malloc(sizeof(struct fp));
            q->primo = j;
            q->apariciones = 1;
            q->sig = temporal->sig;
            temporal->sig = q;
        }
    }
    return top;
}


struct fp *factorizacion_primos(unsigned long long int numero)  //O(n^2)
{
    struct fp *top = NULL;
    unsigned long long int n;

    for (unsigned long long int i = 1; i <= numero; i++)
    {
        n = i;
        unsigned long long int j = 2;

        while (j * j <= n)
        {
            if (n % j == 0)
```

```
            {
                top = ingreso_lista(top, j);
                n /= j;
            }
            else
            {
                j += 1;
            }
        }
        if (n > 1)
            top = ingreso_lista(top, n);
    }
    return top;
}

struct fp *fp_algoritmo(unsigned long long int numero)  //O(n^3)
{
    struct fp *factores_primos;

    for (unsigned long long int i = 1; i <= numero; i++) //O(n)
    {
        factores_primos = factorizacion_primos(i);  // O(n^3)
    }
    return factores_primos;
}


void factorizacion(struct fp **primera, struct fp **segunda)
{
    struct fp *temp1 = *primera;
    struct fp *temp2 = *segunda;
    struct fp *prev1 = NULL, *prev2 = NULL;
    struct fp *aux;


    prev1 = temp1;
    prev2 = temp2;

    while (temp1 != NULL && temp2 != NULL)
    {
        if (temp1->primo < temp2->primo)
        {
            prev1 = temp1;
            temp1 = temp1->sig;
        }
        else if (temp1->primo > temp2->primo)
        {
            prev2 = temp2;
            temp2 = temp2->sig;
        }
        else
```

```
        {
            if (temp1->apariciones > temp2->apariciones)
            {
                temp1->apariciones -= temp2->apariciones;
                temp2->apariciones = 0;
                prev2->sig = temp2->sig;
                aux = temp2;
                temp2 = temp2->sig;
            }
            else if (temp1->apariciones < temp2->apariciones)
            {
                temp2->apariciones -= temp1->apariciones;
                temp1->apariciones = 0;
                prev1->sig = temp1->sig;
                aux = temp1;
                temp1 = temp1->sig;
            }
            else
            {
                prev2->sig = temp2->sig;
                aux = temp2;
                temp2 = prev2->sig;

                prev1->sig = temp1->sig;
                aux = temp1;
                temp1 = prev1->sig;
            }
        }
    }
}


void metodo_1(unsigned int n, unsigned int k) //O(n^3)
{
    struct fp *n_p, *k_p, *n_k_p, *aux;
    unsigned long long int rn = 1, rk = 1, rnk = 1, resultado;

    n_p = fp_algoritmo(n);
    k_p = fp_algoritmo(k);
    n_k_p = fp_algoritmo(n-k);

    factorizacion(&n_p, &k_p);
    factorizacion(&n_p, &n_k_p);

    while(n_p)  //3n
    {
        rn *= pow2(n_p->primo, n_p->apariciones);
        aux = n_p;
        n_p = n_p->sig;
        free(aux);
    }
```

```
    while(k_p)
    {
        rk *= pow2(k_p->primo, k_p->apariciones);
        aux = k_p;
        k_p = k_p->sig;
        free(aux);
    }

    while(n_k_p)
    {
        rnk *= pow2(n_k_p->primo, n_k_p->apariciones);
        aux = n_k_p;
        n_k_p = n_k_p->sig;
        free(aux);
    }

    resultado = rn /(rk*rnk);

    printf("resultado (%u %u): %llu\n", n, k, resultado);

    return;
}
```

2.1.2 Análisis de Complejidad

Este algoritmo, debido a los llamados de funciones que realiza alcanza una complejidad de $O(n^3)$

2.2 Método Recursivo

Se ve basado en la principal formulación de que la combinatoria es resoluble mediante la descomposición de casos más simples del desarrollo, en donde se conoce un caso base del cual se puede alimentar una iteración de datos según el número y las cantidad de formas de elegir k elementos de un conjunto de n elementos, siendo una descomposición mediante un conjunto de sumatorias.

Dentro de los aspectos que se han de considerar en este método es que la utilización de este implica una simplicidad que cuesta en ámbitos de consumo de memoria y tiempo debido a las múltiples llamadas recursivas que se almacenan mediante programación dinámica, dentro de lo cual también éxito un desbordamiento en torno a las magnitud de números que se pueden operar mediante y el costo implicado.

Para este algoritmo la recursividad fue simulada haciendo uso de dos ciclos y una matriz.

2.2.1 Descripción de Algoritmo

```
int min(int a, int b) {
    if(a < b)
        return a;
    else
        return b;
}

void metodo_2(unsigned int n, unsigned int k) //O(n)
```

```
{
    unsigned long long int resultado[n+1][k+1];
    unsigned int i, j;

    for (i = 0; i <= n; i++) {
        for (j = 0; j <= min(i, k); j++) {
            if (j == 0 || j == i)
                resultado[i][j] = 1;
            else
                resultado[i][j] = resultado[i-1][j-1] + resultado[i-1][j];
        }
    }

    printf("resultado (%u %u): %llu\n", n, k, resultado[n][k]);
    return;
}
```

2.2.2 Análisis de Complejidad

La complejidad de este algoritmo es de $O(n^2)$

2.3 Método Mediante triangulo de Pascal

Este método esta formulado bajo la idea matemática de un triángulo de Pascal, donde se comprende la disposición geométrica acorde a los coeficientes binominales, entendiendo el calculo para descubrir la cantidad de formas donde k elementos de un conjunto de n elementos, siendo este la combinatoria de n en k, relacionándose de manera en la que el elemento existente en la fila n y la columna k es equivalente al coeficiente binomial, donde se ubica el elemento correspondiente en el triángulo, sumando los anteriores.

En el algoritmo implementado se realizo una optimización del mismo, de forma que se avanzan los espacios de la primera mitad del triángulo de pascal, puesto que la segunda mitad realiza un efecto de espejo.

2.3.1 Descripción de Algoritmo

```
void metodo_3(unsigned int n, unsigned int k)
{
    unsigned long long int resultado = 1;
    unsigned int i, j;

    for (i = 0; i <= n; i++) {
        for (j = 0; j <= i; j++) {
            if (j == 0 || j == i) {
                resultado = 1;
            } else {
                resultado = resultado * (n - j + 1) / j;
            }
            if (i == n && j == k) {
                printf("resultado (%u %u): %llu\n", n, k, resultado);
                return;
            }
        }
```

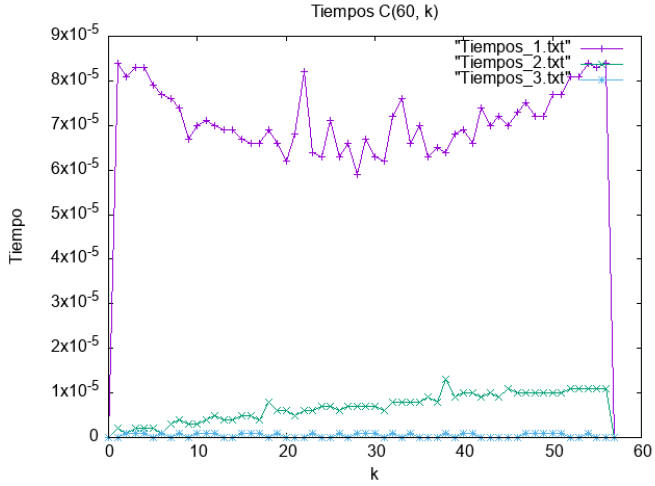
```
    }
}
```

2.3.2 Análisis de Complejidad

La complejidad de este algoritmo es de $O(n)$

3 Resultados

Se logro observar que los tres métodos se ven limitados a trabajar hasta aproximadamente $\binom{60}{30}$, y de estos, como revela su complejidad, el más veloz es el método tres, mientras que el más tardío seria el método uno por fuerza bruta. En el gráfico número 1 es posible identificar lo antes mencionado.



4 Conclusiones

Por medio de lo trabajado, se pudo observar que el método del triángulo de pascal es probablemente el mas óptimo para trabajar con combinatoria, se podría realizar un cambio de base de forma que pueda trabajar con combinatorias mas grandes.

5 Bibliografia y referencias

1. http://disfrutalasmatematicas.com/triangulo-pascal.html
2. https://upcommons.upc.edu/bitstream/handle/2117/186922/matd-combinatoria-recurrencias-4760.pdf