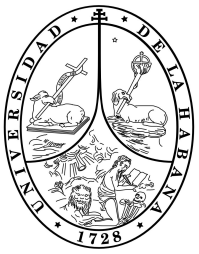


Proyecto de Diseño y Análisis de Algoritmos
"Sliding window"

Est. **Ariel Plasencia Díaz** a.plasencia@estudiantes.matcom.uh.cu

C-312

Índice

©Diseño y Análisis de Algoritmos. 2018-2019 .

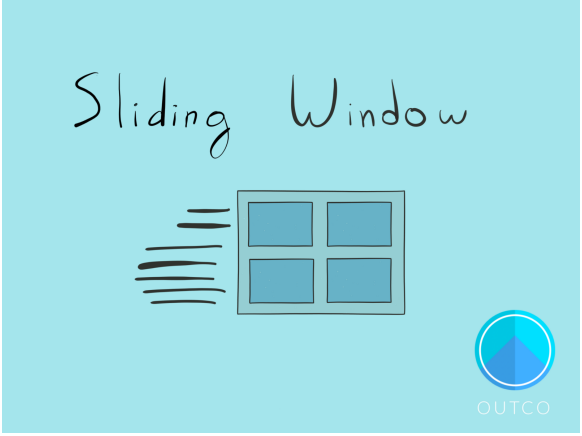
1 Introducción

Resolver problemas sobre arrays (cadenas) en busca de operaciones sobre subarrays (subcadenas) puede en algunos casos ser ineficiente. Como solución a lo anterior mostraremos la técnica conocida como **sliding window**. A modo de curiosidad, diremos que la traducción a **sliding window** es ventana corrediza, sin embargo a lo largo de este trabajo nos referiremos al algoritmo como **sliding window**.

En este trabajo pretendemos explicar y desarrollar dicho algoritmo. El análisis teórico del algoritmo será acompañado por la resolución de problemas cuya solución utiliza esta idea y, en muchos casos, será guiado por ella. Con los problemas se mostrarán además sus respectivos análisis e implementaciones.

©Diseño y Análisis de Algoritmos. 2018-2019 .

2 Presentación del algoritmo



A continuación veremos algunos conceptos fundamentales que nos serán de gran utilidad a lo largo de nuestro trabajo.

2.1 Conceptos fundamentales

2.1.1 Definición

Deque: Es una estructura de datos que soporta las operaciones de inserción y eliminación tanto por delante como por detrás, todas ellas en $O(1)$.

2.1.2 Definición

Multiset: Es una estructura de datos que permite elementos repetidos, además soporta las operaciones de inserción, eliminación y búsqueda en $O(N \cdot log(N))$, donde N es la cantidad de elementos en la estructura.

En qué consiste el algoritmo **sliding window**?
El algoritmo **sliding window** consiste en un subarray o ventana de tamaño constante que se mueve de izquierda a derecha sobre un array y en la cual guardamos información importante como el máximo de la ventana así como el mínimo, la suma, etc. Todo lo anterior usando diferentes estructuras de datos.

Cuándo estamos en presencia del algoritmo **sliding window**?

1. Cuando el problema involucra una estructura de datos en la cual es posible iterar (array o string).
2. Se busca un subarray o substring sobre alguna lista de elementos.
3. Existe una solución por fuerza bruta en $O(N^2)$ o en otra complejidad superior.

Una de las ventajas de este algoritmo es que baja el costo temporal a $O(N)$ usando la técnica de **sliding window**. Veamos un primer ejemplo para un mejor entendimiento del algoritmo.

2.2 Primer problema

2.2.1 Maximum of all subarrays of size k

Dado un array de números de tamaño N y un entero K ($K \leq N$), calcular el máximo de todos los substrings de longitud K.

2.2.2 Análisis y solución

Como primera solución al problema se nos ocurre la siguiente fuerza bruta. Solo se muestra el pseudocódigo del metodo principal.

```
void solution(arr, N, K)
    for i: 0 to N - K + 1
        max_sum <-- arr[i]
        for j: 1 to K
            if arr[i + j] > max_sum
                max_sum <-- arr[i + j]
    print max_sum
```

Este algoritmo realiza una iteración sobre el array y para cada posición calcula la suma del substring de longitud K, la cual va comparando para guardar la mayor suma. Esta idea es $O(N \cdot K)$, que es $O(N^2)$ en el caso peor, cuando $N = K$.

Usando el algoritmo **sliding window**, mejoramos considerablemente el problema anterior. Nosotros creamos un deque, con capacidad K, que guarda sólo elementos útiles de la ventana actual. Un elemento es útil si está en la ventana actual y es mayor que todos los otros elementos de su izquierda que están en la ventana. Nosotros procesamos todos los elementos de la serie uno por uno y mantenemos en el deque los elementos útiles de manera ordenada. El elemento al frente del deque es el más grande y el último elemento el más pequeño de dicha ventana actual.

2.2.3 Complejidad temporal

Si nos fijamos con detenimiento nos percatamos que se realiza un ciclo hasta N y que cada elemento se inserta y se elimina a lo sumo una vez de nuestra estructura, donde sus operaciones son $O(1)$. Por tanto, nuestra complejidad temporal es $O(N)$ amortizado.

2.2.4 Implementación

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

void solution(int arr[], int n, int k) {

    deque<int> Q(k);

    int i;
    for(i = 0; i < k; i++) {
        while((!Q.empty()) && arr[i] >= arr[Q.back()]) {
            Q.pop_back();
        }
        Q.push_back(i);
    }

    for( ; i < n; i++) {
        cout << arr[Q.front()] << " ";
        while((!Q.empty()) && Q.front() <= i - k) {
            Q.pop_front();
        }
        while((!Q.empty()) && arr[i] >= arr[Q.back()]) {
            Q.pop_back();
        }
        Q.push_back(i);
    }
    cout << arr[Q.front()] << endl;;
```

```
}

int main() {
    int N, K, a;
    cin >> N >> K;

    int arr[N];

    for(int i = 0; i < N; i++) {
        cin >> a;
        arr[i] = a;
    }

    solution(arr, N, K);
}
```

3 Ejercicios

En esta sección se utilizará el algoritmo de **sliding window** para la resolución de problemas un poco más interesantes, mostrando la potencia de esta idea. En algunos casos mostraremos la solución en varios lenguajes, pero siempre haciendo referencia a C++.

3.1 Ejercicio 1

3.1.1 Problema

Dado dos cadenas X y Y ($1 \leq |X|, |Y| \leq 10^5$). Encuentra todos los anagramas de Y que son subcadenas de X.
ENTRADA: La primera y única línea son las cadenas X y Y en ese orden.
SALIDA: La salida del programa de la forma .ªnagram i present at index j"(sin comillas), donde i es el anagrama y j el índice donde empieza.

3.1.2 Análsis

En el problema se nos pide imprimir todas las subcadenas de X que existen tal que forman una permutación de Y. Para ello utilizaremos dos *multiset* $< char >$, teniendo en cuenta que se pueden repetir caracteres. En el primero guardaremos todas las letras de la cadena Y, por el contrario, en el segundo *multiset* agregamos las primeras $|Y|$ letras, pero en este caso de la cadena X. Este último nos servirá de ventana.

A partir de la posición $|Y| + 1$, iteramos por la cadena X de tal manera que en la i ésima iteración comparamos los dos *multiset*, en caso de que sean iguales obtuvimos una respuesta (permutación), y en caso contrario corremos la ventana, o sea sacamos al primero y metemos al i ésimo. El costo temporal es $O(N)$ amortizado.

3.1.3 Implementación

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

// function to find all substrings of string X that are permutations of string Y
void find_all_anagrams(string X, string Y) {

    // m and n stores size of string Y and X respectively
```

```
    int m = Y.length(), n = X.length();

    // invalid input
    if (m > n)
        return;

    // maintain count of characters in second string
    multiset<char> set;
    // maintain count of characters in current window
    multiset<char> window;

    // insert all characters of string Y into set
    for(int i = 0; i < m; i++) {
        set.insert(Y[i]);
    }
    // note that multiset can maintain duplicate elements unlike set

    // maintain a sliding window of size m with adjacent characters of string X
    for(int i = 0; i < n; i++) {
        // add first m characters of string X into current window
        if(i < m)
            window.insert(X[i]);
        else {
            // if all characters in current window matches that of string Y, we found
            // an anagram
            if(window == set) {
                string s = X.substr(i - m, m);
                int index = i - m;
                cout << Anagram << s << present at index << index << endl;
            }

            // consider next substring of X by removing leftmost element of the sliding
            // window and add next character of string X to it

            // delete only one occurrence of leftmost element of current window
            auto iter = window.find(X[i - m]);
            if(iter != window.end())
                window.erase(iter);

            // insert next character of string X in current window
            window.insert(X[i]);
        }
    }

    // if last m characters of string X matches that of string Y, we found an anagram
    if(window == set)
        cout << Anagram << X.substr(n - m, m) << present at index << n - m << endl;
}

int main() {
    string X, Y;
    cin >> X >> Y;

    find_all_anagrams(X, Y);
}
```

3.2 Ejercicio 2

3.2.1 Problema

Dado un array A de N ($1 \leq N \leq 10^5$) elementos, donde $-10^5 \leq A_i \leq 10^5$ ($1 \leq i \leq N$), y un entero K ($1 \leq K \leq 10^4$). Encuentra un subarray tal que su suma sea igual a K. Aseguramos que siempre existe solución.

3.2.2 Análsis

Para la resolución del problema anterior utilizaremos un *deque* y una variable $window_{sum}$ como variables locales importantes. Recorremos el array e insertamos por atrás en nuestra estructura mientras $window_{sum}$ no exceda K. Si en algún momento $window_{sum} = K$ entonces encontramos una solución y el subarray se encuentra en *deque* y si $window_{sum} > K$ entonces eliminamos por delante en la estructura e insertamos por detrás el elemento i ésimo.

De esta manera verificamos todos los posibles subarrays de una manera inteligente haciendo una pasada por el array y donde sus elementos se insertan y se remueven a lo sumo una vez del *deque*, por lo que la complejidad temporal es $O(N)$.

3.2.3 Implementación

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

void print_deque(deque<int> wind, int arr[]) {
    while(!wind.empty()) {
        int current = wind.front();
        cout << arr[current] << endl;
        wind.pop_front();
    }
}

// function to print sub-array having given sum using sliding window technique
deque<int> find_subarray(int arr[], int n, int sum) {
    // maintain sum of current window
    int window_sum = 0;
    deque<int> window;

    // maintain a window [low, high - 1]
    int low = 0, high = 0;

    // consider every sub-array starting from low index
    for(low = 0; low < n; low++) {
        // if current window sum is less than the given sum, then add elements
        // to current window from right
        while(window_sum < sum && high < n) {
            window_sum += arr[high];
            window.push_back(high);
            high++;
        }

        // if current window sum is equal to the given sum
        if(window_sum == sum) {
            cout << Subarray found [ << low << - << high - 1 << ] << endl;
            return window;
        }
```

```
        // At this point the current window sum is more than the given sum remove
        // current element (leftmost element) from the window
        window_sum -= arr[low];
        window.pop_front();
    }
}

int main() {
    int N, SUM, a;
    cin >> N >> SUM;

    int arr[N];
    for(int i = 0; i < N; i++) {
        cin >> a;
        arr[i] = a;
    }

    deque<int> window = find_subarray(arr, N, SUM);
    print_deque(window, arr);
}
```

3.3 Ejercicio 3

3.3.1 Problema

Given a two positive integer N, K ($K \leq N$) and array, find the first negative integer for each and every window(contiguous subarray) of size K. If a window does not contain a negative interger, then print 0 for that window.

Input: $N = 5, K = 2, arr[] = \{-8, 2, 3, -6, 10\}$
Output: $-8\ 0\ -6\ -6$
Explanation First negative integer for each window of size k:
$\{-8, 2\} = -8$
$\{2, 3\} = 0$ (does not contain a negative integer)
$\{3, -6\} = -6$
$\{-6, 10\} = -6$

Input: $N = 8, K = 3, arr[] = \{12, -1, -7, 8, -15, 30, 16, 28\}$
Output: $-1\ -1\ -7\ -15\ -15\ 0$

3.3.2 Análsis

En este problema nos piden buscar el primer elemento negativo en un subarray de tamaño K. Es una variación del problema *SlidingWindowMaximum* (visto en el epígrafe 2,2). Creamos un dequeue, window de capacidad K que guarda sólo elementos útiles de la ventana actual. Un elemento es útil si está en la ventana actual y es un entero negativo. Luego, procesamos todos los elementos de la secuencia uno por uno y mantenemos en el frente de nuestra estructura el primer entero negativo para esa ventana. Notar que si la ventana está vacía devolveremos 0, que significa que todos los elementos para esa ventana son positivos, es decir que no hay números negativos.

3.3.3 Implementación

Solución en C++

```
#include <bits/stdc++.h>
#define endl "\n"

using namespace std;

// function to find the first negative integer in every window of size k
void print_first_negative_integer(int arr[], int n, int k) {
```

```
    deque<int> window;

    int i;
    for(i = 0; i < k; i++) {
        if(arr[i] < 0)
            window.push_back(i);
    }

    for( ; i < n; i++) {
        if(!window.empty())
            cout << arr[window.front()] << ;
        else
            cout << 0 << ;

        while((!window.empty()) && window.front() < (i - k + 1))
            window.pop_front();

        if(arr[i] < 0)
            window.push_back(i);
    }

    if(!window.empty())
        cout << arr[window.front()] << endl;
    else
        cout << 0 << endl;
}

int main() {

    int N, K, a;
    cin >> N >> K;

    int arr[N];
    for(int i = 0 ; i < N; i++) {
        cin >> a;
        arr[i] = a;
    }

    print_first_negative_integer(arr, N, K);
}
```

3.4 Ejercicio 4

3.4.1 Hotels along the croatian coast

Ver problema

There are N hotels along the beautiful Adriatic coast. Each hotel has its value in Euros. Sroljo has won M Euros on the lottery. Now he wants to buy a sequence of consecutive hotels, such that the sum of the values of these consecutive hotels is as great as possible, but not greater than M. You are to calculate this greatest possible total value.

Input: In the first line of the input there are integers N and M ($1 \leq N \leq 300000, 1 \leq M \leq 2^{31} - 1$). In the next line there are N natural numbers less than 10^6, representing the hotel values in the order they lie along the coast.

Output: Print the required number (it will be greater than 0 in all of the test data).

3.4.2 Análsis

La resolución de este problema es análoga al problema 2.

3.4.3 Implementación

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

int find_subarray(int arr[], int n, int m) {

    int sol = INT_MIN;
    int window_sum = 0;
    deque<int> window;
    int low = 0, high = 0;

    for(low = 0; low < n; low++) {
        while(window_sum < m && high < n) {
            sol = max(sol, window_sum);
            window_sum += arr[high];
            window.push_back(high);
            high++;
        }

        if(window_sum == m) {
            sol = m;
            break;
        }

        window_sum -= arr[low];
        window.pop_front();
    }

    return sol;
}

int main() {

    int N, M, a;
    cin >> N >> M;

    int arr[N];
    for(int i = 0; i < N; i++) {
        cin >> a;
        arr[i] = a;
    }

    int sol = find_subarray(arr, N, M);
    cout << sol << endl;
}
```

3.5 Ejercicio 5

3.5.1 Problema

Dado una cadena S ($1 \leq |S| \leq 10^5$). Encuentra la mayor subcadena de S tal que esta contenga a la mayor cantidad de caracteres de S distintos posibles.

3.5.2 Implementación

Solución en C++
Solución en csharp

```
#include<bits/stdc++.h>
#define endl '\n'
#define CHAR_RANGE 128

using namespace std;

string longest_substr(string str, int n) {

vector<bool> window(CHAR_RANGE);
   int begin = 0, end = 0;

   for(int low = 0, high = 0; high < n; high++) {
      if(window[str[high]]) {
         while(str[low] != str[high])
            window[str[low++]] = false;
         low++;
      }
      else {
         window[str[high]] = true;
         if(end - begin < high - low) {
            begin = low;
            end = high;
         }
      }
   }

   return str.substr(begin, end - begin + 1);
}

int main() {
   string S;
   cin >> S;

   int N = S.length();

   cout << longest_substr(S, N) << endl;
}
```

3.6 Ejercicio 6

3.6.1 Problema

Dado una array de N ($1 \leq N \leq 10^5$) números enteros tal que $N_i \in [0, 1]$ ($1 \leq i \leq N$). Encuentra la logitud del mayor subarray donde cambiamos un 0 por un 1 y se forma un subarray tal que todos sus elementos son todos iguales a 1.

©Diseño y Análisis de Algoritmos. 2018-2019 .

3.6.2 Implementación

Solución en C++
Solución en csharp

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

int find_index_of_zero(int arr[], int n) {
    int left = 0;
    int count = 0;
    int max_count = 0;
    int max_index = -1;
    int prev_zero_index;

    for(int i = 0; i < n; i++) {
        if(arr[i] == 0) {
            prev_zero_index = i;
            count++;
        }
        if(count == 2) {
            while(arr[left])
                left++;
            left++;
            count = 1;
        }
        if((i - left + 1) > max_count) {
            max_count = i - left + 1;
            max_index = prev_zero_index;
        }
    }

    return max_index;
}

int main() {
    int N, a;
    cin >> N;

    int arr[N];
    for(int i = 0; i < N; i++) {
        cin >> a;
        arr[i] = a;
    }

    int index = find_index_of_zero(arr, N);
    cout << index << endl;
}
```