Bloque 1: arreglos y matrices

Programación: concepto, ejemplo y problema de juez

Carlos A Delgado S

Pontificia Universidad Javeriana Cali
Sesiones de refuerzo de algoritmia

Los temas

Cómo va cada tema

Tres partes en el mismo orden: el concepto con lo que cuesta, un ejemplo sobre un arreglo de cinco casillas con su traza en una tabla, y el problema de UVa donde se usa, resuelto entre todos y escrito en Python y en C++.

Método	Problema
El acumulado que se arrastra	UVa 11078, *Open Credit System*
El mejor tramo	UVa 507, *Jill Rides Again*
Acumulados en dos dimensiones	UVa 108, *Maximum Sum*

Compilar y correr

C++

```
g++ -std=c++17 -Wall -Wextra solucion.cpp -o solucion
./solucion < 1.in > salida.out
diff salida.out 1.out
```

Python

```
python3 solucion.py < 1.in > salida.out
diff salida.out 1.out
```

Qué hace cada cosa

Los dos signos y el `diff`

< entrega el archivo como si alguien lo estuviera tecleando: el programa no abre ningún archivo, sigue leyendo de la entrada estándar. > manda a un archivo lo que imprime. `diff` no dice nada cuando las dos salidas son iguales. Y `-Wall -Wextra` atrapa en el computador lo que el juez devuelve como respuesta equivocada: una variable sin inicializar o una comparación entre con signo y sin signo.

Contenido

1 La entrada y lo que cuesta cada operación

2 Un recorrido en vez de dos

3 El mejor tramo de un arreglo

4 Rectángulos en una matriz

5 Tarea del bloque

Contenido

1 La entrada y lo que cuesta cada operación

2 Un recorrido en vez de dos

3 El mejor tramo de un arreglo

4 Rectángulos en una matriz

5 Tarea del bloque

¿Qué trae esta entrada?

Archivo A

```
5
3 -1 4 1 5
```

Archivo B

```
5
3 -1
4 1
5
```

Pregunta

Un programa que lee los valores de la segunda línea con `input().split()`, ¿lee también el archivo B?

¿Qué trae esta entrada?

Respuesta

No: con B, la segunda línea trae solo `3 -1`. Se lee por palabras:

- Python: `datos = sys.stdin.read().split()` trae toda la entrada partida en espacios y saltos de línea por igual; un índice `pos` avanza sobre esa lista.
- C++: `scanf("%d", &x)` salta todo espacio en blanco antes del número, saltos de línea incluidos.

Leer hasta fin de archivo

Pregunta

La entrada trae pares de enteros y nadie dice cuántos. ¿Cuándo para el ciclo de lectura?

Python: `hasta_eof.py`

```python
import sys

for linea in sys.stdin:
    partes = linea.split()
    if len(partes) == 2:
        a = int(partes[0])
        b = int(partes[1])
        print(a, b, a + b)
```

C++: `hasta_eof.cpp`

```cpp
#include <cstdio>

int main() {
    int a, b;
    while (scanf("%d %d", &a, &b) ==
    ↪  2) {
        printf("%d %d %d\n", a, b, a
        ↪  + b);
    }
    return 0;
}
```

Leer hasta fin de archivo

Respuesta

Cuando se acaba la entrada. En Python, `for linea in sys.stdin` termina solo; en C++, `scanf` devuelve cuántos valores leyó y al final ya no devuelve 2. UVa 100, de los problemas del bloque, se lee así.

Lo que cuesta cada operación



Operación	Python (`list`)	C++ (`vector`)	Costo
Acceder por posición	`a[i]`	`a[i]`	$O(1)$
Tamaño	`len(a)`	`a.size()`	$O(1)$
Agregar al final	`a.append(x)`	`a.push_back(x)`	$O(1)$ en promedio
Quitar el último	`a.pop()`	`a.pop_back()`	$O(1)$
Insertar en p	`a.insert(p, x)`	`a.insert(a.begin() + p, x)`	$O(n-p)$
Borrar en p	`del a[p]`	`a.erase(a.begin() + p)`	$O(n-p)$
Buscar un valor	`x in a`	recorrido con `for`	$O(n)$
Máximo	`max(a)`	recorrido con `for`	$O(n)$

El costo que no se ve

`x in a` y `max(a)` caben en una línea, pero recorren la lista. Dentro de un ciclo de n vueltas, el programa cuesta $O(n^2)$. Lo mismo pasa con `a.insert(0, x)`: la `list` de Python es un arreglo, no una lista enlazada.

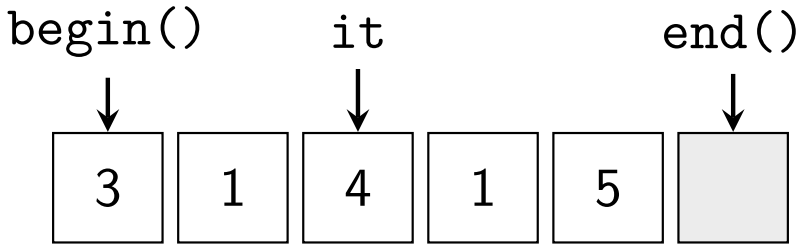
Las estructuras del curso y cómo están hechas

Estructura	C++	Python	Cómo está hecha
Arreglo dinámico	`vector` 	`list`	casillas seguidas en memoria; al llenarse se copia a un bloque del doble
Lista enlazada	`list` 	no viene	nodos sueltos, cada uno con apuntador al anterior y al siguiente
Doble extremo	`deque`	`collections.deque`	bloques de tamaño fijo enlazados, que crecen por los dos lados
Pila	`stack`	`list` con `append` y `pop`	un deque por dentro que solo deja tocar el final
Cola	`queue`	`deque` con `append` y `popleft`	un deque: se entra por el final y se sale por el frente

Las estructuras del curso y cómo están hechas

Estructura	C++	Python	Cómo está hecha
Direccionamiento directo	arreglo de tamaño U	`list` de tamaño U	la clave es la posición en el arreglo
Tabla hash	`unordered_map`, `unordered_set`	`dict`, `set`	una función lleva la clave a una casilla; las que chocan se encadenan o se reubican
Árbol balanceado	`map`, `set`	no viene	árbol rojo-negro: se rota al insertar para que la altura quede en $O(\log n)$
Árbol binario de búsqueda	a mano, con nodos	a mano, con nodos	a la izquierda los menores, a la derecha los mayores, sin rebalancear
Montículo	`priority_queue`	`heapq` sobre una `list`	árbol completo guardado en un arreglo; los hijos de i están en $2i+1$ y $2i+2$

Punteros e iteradores



`*it` vale 4; `end()` queda una casilla después del último

El concepto

Un apuntador guarda la dirección de un dato. Un iterador generaliza esa idea a cualquier estructura: `*it` lee el elemento, `++it` pasa al siguiente y `it != c.end()` pregunta si quedan. En un `vector` el iterador es casi un apuntador; en una `list` o un `map` sigue los apuntadores entre nodos. El mismo ciclo recorre las dos.

Punteros e iteradores

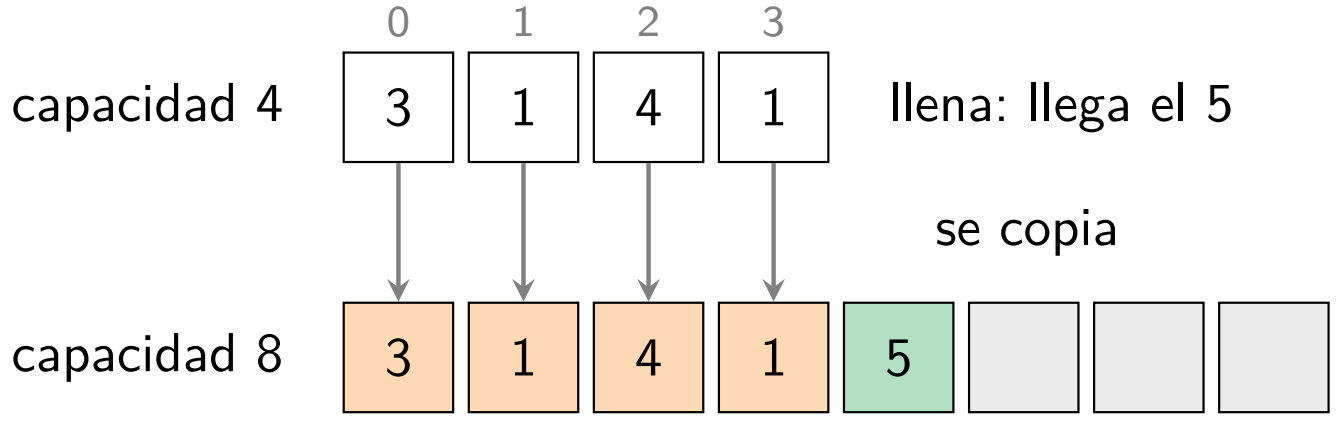
Iterador inválido

Un iterador, un apuntador o una referencia queda *inválido* cuando la estructura mueve o libera el elemento al que apuntaba. Usarlo después no da un error claro: lee basura, escribe sobre otro dato o termina el programa, según la suerte. Por eso cada estructura que sigue dice qué operaciones invalidan qué.

Dónde vive cada cosa

Las variables locales viven en la pila de llamadas, que es pequeña y se libera sola al salir de la función. Los contenedores guardan sus elementos en el *heap*, que es grande y se pide con `new`: el contenedor lo libera por su cuenta, y lo que se pide a mano con `new` se libera a mano con `delete`. Python maneja todo en el *heap* y cada variable es una referencia a un objeto.

Arreglo dinámico: `vector` y `list`



Cómo funciona

Las casillas están seguidas en memoria, así que `a[i]` se calcula con una suma: inicio más i por el tamaño de cada elemento. Cuando se llena se pide un bloque del doble y se copian todas. Esa copia cuesta $O(n)$, pero pasa cada vez que el tamaño se duplica, y repartida entre los `push_back` da $O(1)$ en promedio.

Arreglo dinámico: `vector` y `list`



Memoria, punteros e iteradores

El objeto `vector` guarda tres datos: un apuntador al bloque en el *heap*, el tamaño y la capacidad. Un iterador es casi un apuntador a una casilla: `*it` lee y `it + k` salta k casillas en $O(1)$. `v.reserve(n)` pide la capacidad de una vez y evita las copias.

Dónde se rompe

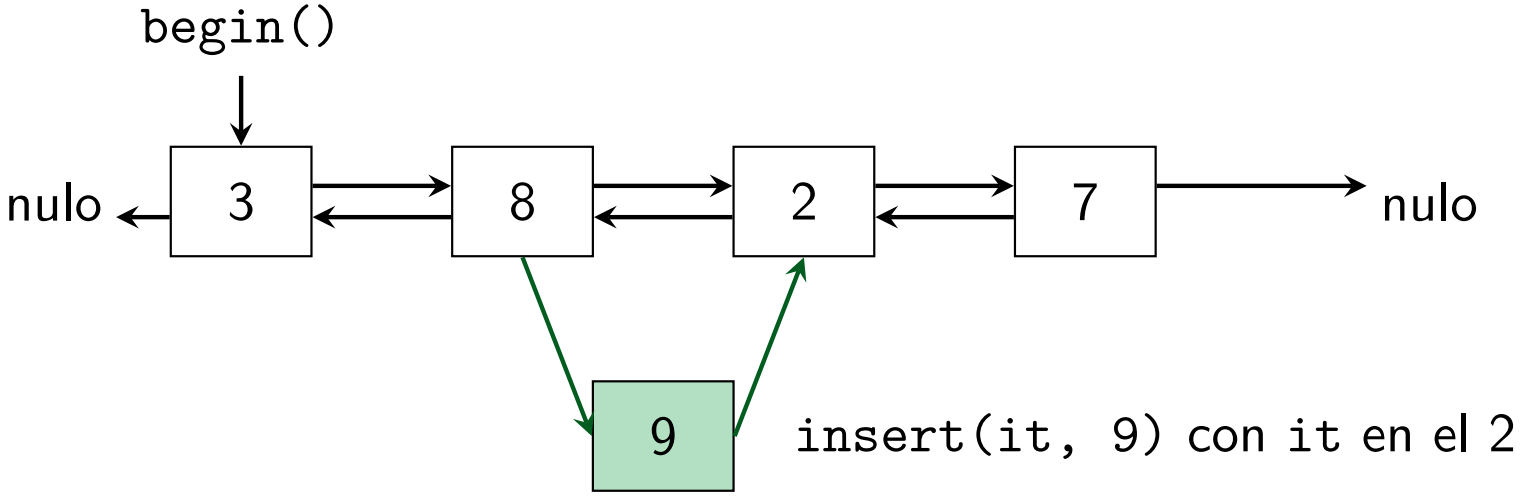
Si un `push_back` copia el bloque, todo apuntador, referencia o iterador guardado sigue apuntando al bloque viejo, que ya se liberó: leerlo es comportamiento indefinido, y el programa puede dar bien en el computador y fallar en el juez. `insert` y `erase` invalidan desde la posición tocada hasta el final.

Arreglo dinámico: `vector` y `list`

En Python

La `list` guarda referencias, no valores. `[[0] * m] * n` crea n referencias a la misma fila, y escribir en una cambia todas; la matriz se arma con `[[0] * m for i in range(n)]`.

Lista enlazada: `list` de C++



Cómo funciona

Cada nodo vive en un lugar distinto de la memoria y guarda el valor y dos apuntadores, al anterior y al siguiente. Insertar o borrar donde está el iterador cambia cuatro apuntadores y no mueve a nadie: $O(1)$. Llegar a la posición i obliga a seguir i apuntadores desde el comienzo: $O(i)$.

Lista enlazada: `list` de C++

Memoria, punteros e iteradores

Cada `insert` hace un `new` de un nodo, que en 64 bits gasta 16 bytes de apuntadores además del valor. El iterador envuelve el apuntador al nodo: `++it` sigue `sig` y `--it` sigue `ant`. No hay `it + 3`; avanzar k son k pasos.

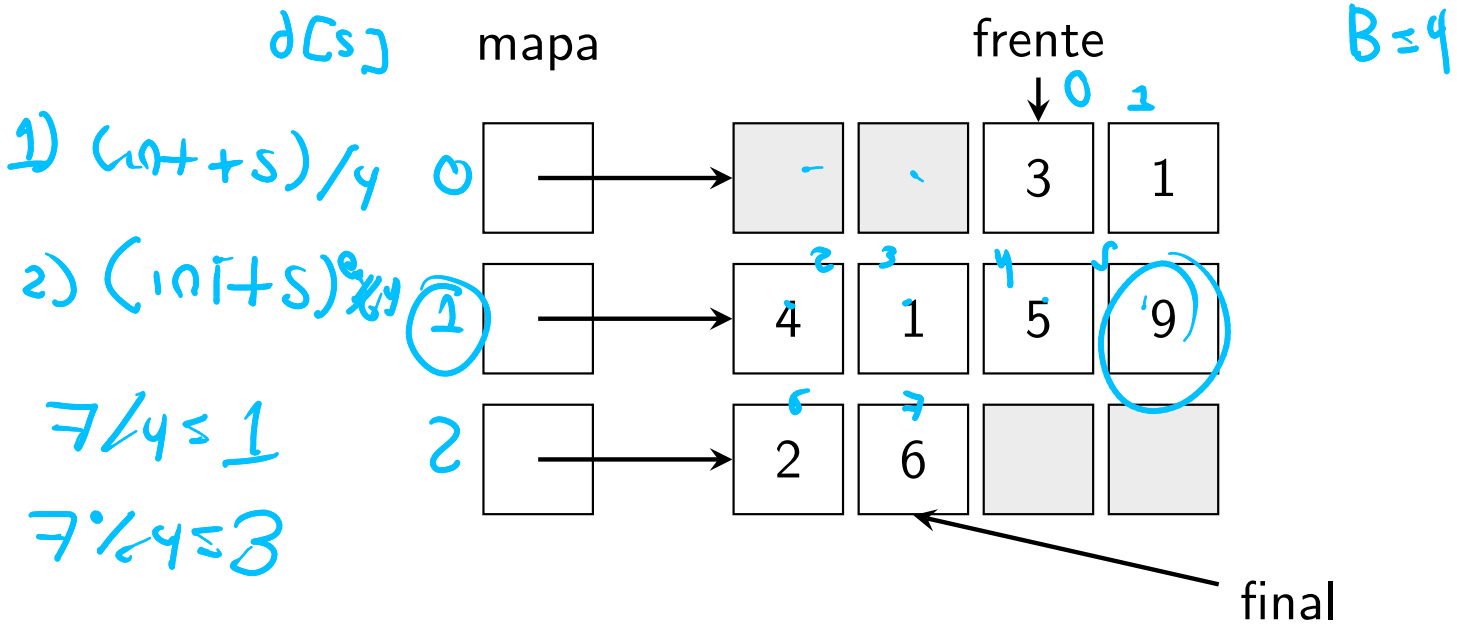
Dónde se rompe

Insertar no invalida ningún iterador, porque ningún nodo se mueve. Borrar invalida solo el del nodo borrado, así que dentro de un recorrido se escribe `it = l.erase(it)`: `erase` devuelve el siguiente. Hacer `l.erase(it)` y después `it++` avanza desde un nodo que ya no existe.

El costo que no sale en la tabla

Los nodos quedan dispersos en memoria y el procesador no los trae juntos a la caché. Recorrer una `list` es $O(n)$ como un `vector`, pero varias veces más lento en la práctica.

Doble extremo: deque



Cómo funciona

Los valores van en bloques de tamaño fijo y el mapa apunta a cada bloque. `push_front` y `push_back` escriben en las casillas libres de los extremos, y si el bloque se llena se pide otro sin copiar nada. En C++ `d[i]` es $O(1)$: una división dice el bloque y la casilla. El deque de Python enlaza los bloques sin mapa, y `d[i]` en la mitad cuesta $O(n)$.

Doble extremo: `deque`

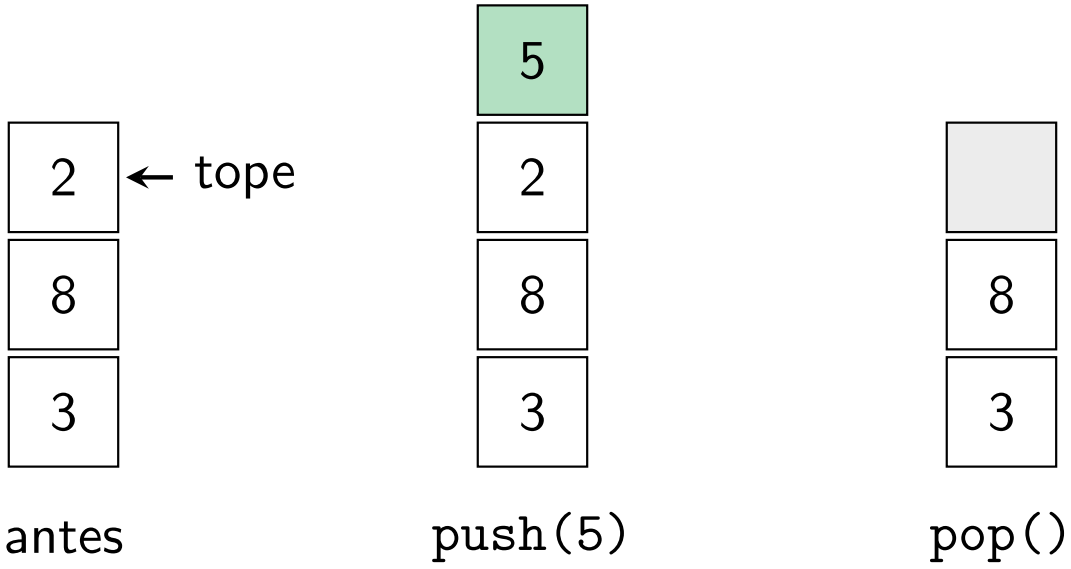
Memoria, punteros e iteradores

El iterador guarda dos cosas: en qué bloque está y en qué casilla del bloque. `it + k` se resuelve con la misma división que `d[i]`, en $O(1)$.

Dónde se rompe

`push_front` y `push_back` pueden agrandar el mapa e invalidan los iteradores, pero las referencias y los apuntadores a los elementos siguen sirviendo, porque los bloques no se mueven. Insertar o borrar en la mitad invalida todo.

Pila: `stack` y la `list` de Python



Cómo funciona

Un arreglo que solo se toca por el final: el tope es la última casilla ocupada. `stack` de C++ envuelve un `deque` y solo deja ver `push`, `pop` y `top`; en Python se usa la `list` con `append`, `pop()` y `a[-1]`. Las tres son $O(1)$ porque ninguna corre elementos.

Pila: `stack` y la `list` de Python

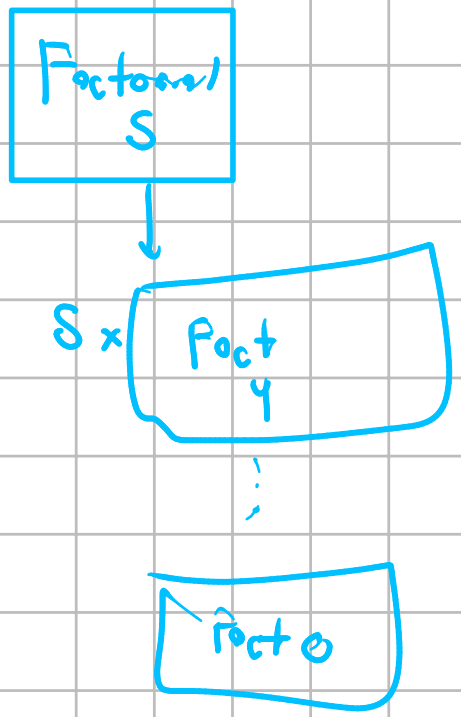
Memoria, punteros e iteradores

`stack` no tiene iteradores ni `begin()`: solo deja ver el tope. Para mirar lo que hay abajo se saca con `pop`, y lo sacado se pierde. Si el problema necesita recorrer la pila, se usa un `vector` con `push_back`, `pop_back` y `back`.

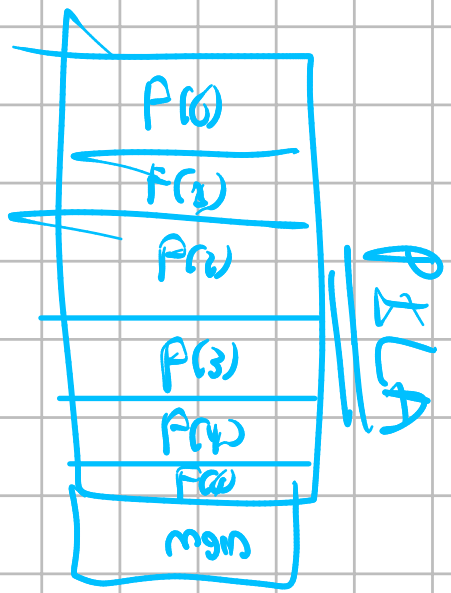
Dónde se rompe

`top()` y `pop()` sobre una pila vacía son comportamiento indefinido en C++, y `a.pop()` sobre una `list` vacía lanza `IndexError` en Python. Antes se pregunta `s.empty()` o `len(a) > 0`. La recursión usa una pila propia, la de llamadas, que en C++ tiene unos 8 MB: una recursión de 10^6 niveles la desborda.

```
long factorial(int n) {
        if (n == 0) {
            return 1;
        }
        else {
            return n*factorial(n-1);
        }
}
```

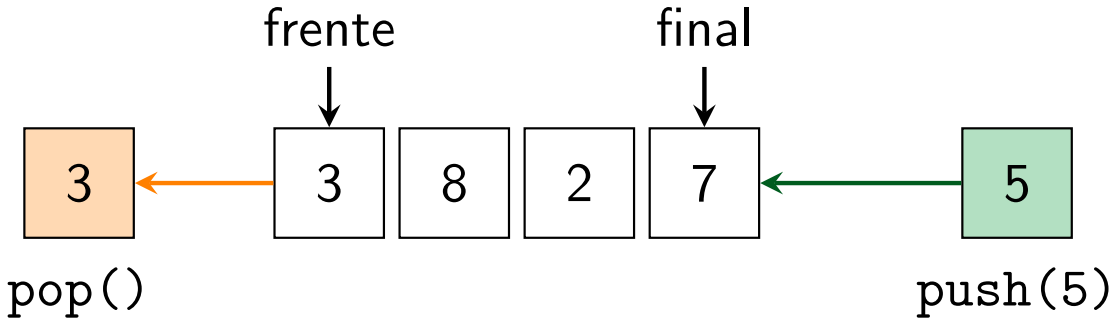


fact(5) = 5*fact(4)
fact(4) = 4*fact(3)
fact(3) = 3*fact(2)
fact(2) = 2*fact(1)
fact(1) = 1*fact(0)
fact(0) = 1



```
(gdb) back
#0  factorial (n=1) at recursion.cpp:3
#1  0x0000555555555165 in factorial (n=2) at recursion.cpp:6
#2  0x0000555555555165 in factorial (n=3) at recursion.cpp:6
#3  0x0000555555555165 in factorial (n=4) at recursion.cpp:6
#4  0x0000555555555165 in factorial (n=5) at recursion.cpp:6
#5  0x0000555555555188 in main () at recursion.cpp:12
```

Cola: queue y collections.deque



Cómo funciona

Se entra por el final y se sale por el frente. queue de C++ envuelve un deque; en Python se usa `collections.deque` con `append` y `popleft`. Las dos operaciones son $O(1)$ porque el frente avanza sin correr a los que quedan, que es justo lo que no hace `list.pop(0)`.

Cola: queue y collections.deque

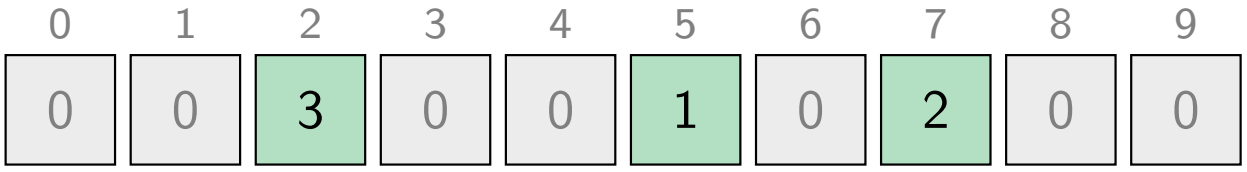
Memoria, punteros e iteradores

queue tampoco tiene iteradores: se ve el frente con `front()` y el final con `back()`. Por dentro es un deque, así que los bloques que se vacían por el frente se liberan y la memoria no crece con los que ya salieron.

Dónde se rompe

`front()` sobre una cola vacía es comportamiento indefinido. Y `q.front()` devuelve una referencia: si se guarda con `int &x = q.front()` y después se hace `pop()`, `x` apunta a memoria liberada. Se copia el valor, `int x = q.front()`, antes de sacarlo.

Direccionamiento directo



claves 2, 2, 2, 5, 7, 7: se cuenta cuántas veces sale cada una

Cómo funciona

La clave es la posición: para guardar o leer la clave k se va directo a `t[k]`, en $O(1)$ y sin comparar nada. Sirve cuando las claves son enteros pequeños, como las 26 letras o los números hasta 10^6. El arreglo mide U aunque se usen pocas casillas, y recorrerlo cuesta $O(U)$.

Direccionamiento directo

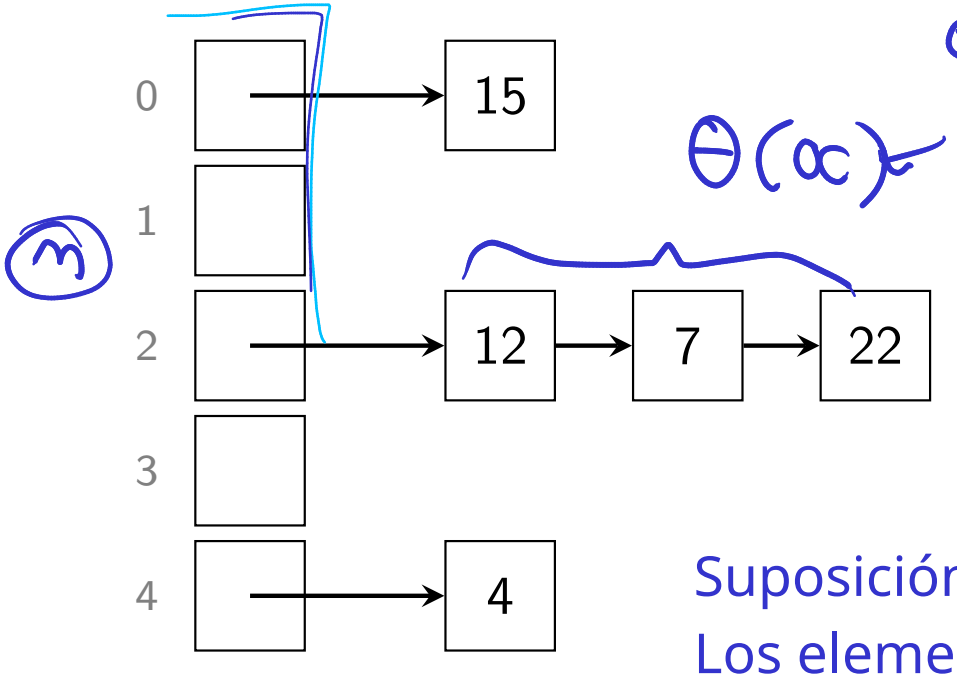
Memoria

El arreglo ocupa U casillas aunque se usen pocas: con $U = 10^6$ enteros son 4 MB, con $U = 10^9$ son 4 GB y no cabe. Antes de usarlo se multiplica U por el tamaño de cada casilla.

Dónde se rompe

Un arreglo grande declarado dentro de una función vive en la pila de llamadas, que tiene unos 8 MB, y el programa termina con error de ejecución sin decir nada más. Se declara global o se usa un `vector`, que guarda las casillas en el *heap*. Una clave fuera de $[0, U)$ escribe fuera del arreglo: C++ no lo revisa.

Tabla hash: `unordered_map` y `dict`



$h(k) = k \bmod 5$

12, 7 y 22 chocan en la casilla 2

Suposición de hashing uniforme

Los elementos están distribuidos uniforme

Cómo funciona

Una función h convierte la clave, número o texto, en una casilla del arreglo. Las claves que caen en la misma casilla forman una cadena: así lo hace `unordered_map`, y el `dict` de Python busca otra casilla libre. Buscar es calcular $h(k)$ y revisar una cadena corta: $O(1)$ en promedio. Cuando la tabla se llena se agranda y se vuelve a repartir todo.

Tabla hash: `unordered_map` y `dict`

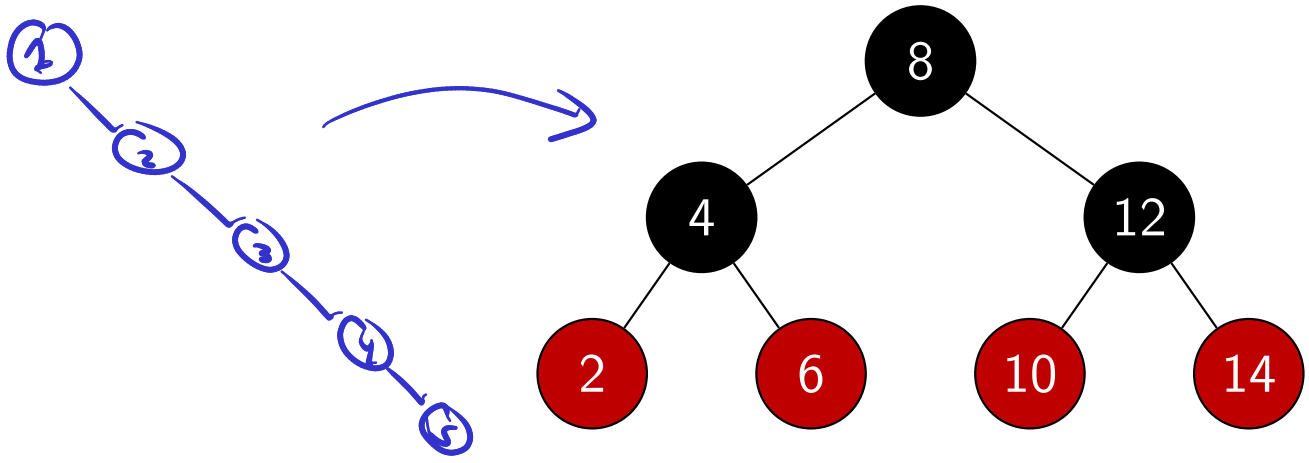
Memoria, punteros e iteradores

En `unordered_map` cada par clave y valor es un nodo aparte, enlazado desde su casilla. El iterador recorre las casillas en el orden del arreglo, que no tiene nada que ver con el orden de las claves.

Dónde se rompe

Cuando la tabla se agranda, las claves cambian de casilla: los iteradores quedan inválidos y el orden del recorrido cambia, aunque las referencias a los valores sigan sirviendo porque los nodos no se mueven. En Python, agregar o borrar claves de un `dict` mientras se recorre lanza `RuntimeError`; se recorre `list(m)`, una copia de las claves.

Árbol balanceado: `map` y `set`



Cómo funciona

Es un árbol binario de búsqueda rojo-negro: a la izquierda de cada nodo van los menores y a la derecha los mayores. Cada nodo es rojo o negro, un rojo no tiene hijos rojos y todos los caminos de la raíz a las hojas pasan por la misma cantidad de negros. Cuando una inserción rompe esas reglas, el árbol recolorea y rota, y la altura queda en $O(\log n)$. Leído de izquierda a derecha da 2, 4, 6, 8, 10, 12, 14: por eso `map` entrega las claves ordenadas.

Árbol balanceado: `map` y `set`

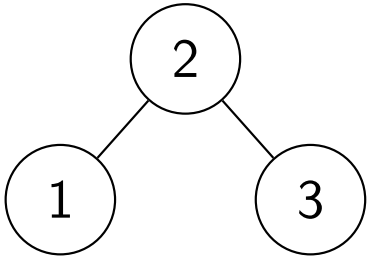
Memoria, punteros e iteradores

Cada nodo guarda la clave, el valor, el color y tres apuntadores: al padre y a los dos hijos. `++it` va al sucesor siguiendo esos apuntadores: baja al menor del subárbol derecho o sube hasta el primer padre que esté a la derecha. Recorrer todo el `map` así cuesta $O(n)$ en total.

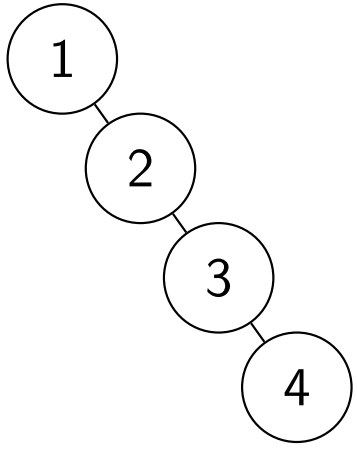
Dónde se rompe

Insertar no invalida iteradores y borrar invalida solo el del nodo borrado, igual que en la lista: `it = m.erase(it)`. Las claves no se pueden cambiar por medio del iterador, porque el nodo quedaría fuera de orden; se borra y se inserta la clave nueva. Y `m[k]` con una clave que no está crea un nodo: preguntar con `m[k]` dentro de un ciclo llena el árbol de ceros.

Árbol binario de búsqueda sin rebalancear



Llegan 2, 1, 3: altura 1



Llegan 1, 2, 3, 4: una cadena

Cómo funciona

Cada nodo guarda la clave y dos apuntadores; se busca bajando a la izquierda si la clave es menor y a la derecha si es mayor, así que todo cuesta $O(h)$. Si las claves llegan ordenadas, cada una cuelga de la anterior y el árbol es una lista: $h = n$. Eso es lo que corrige el rojo-negro.

Árbol binario de búsqueda sin rebalancear

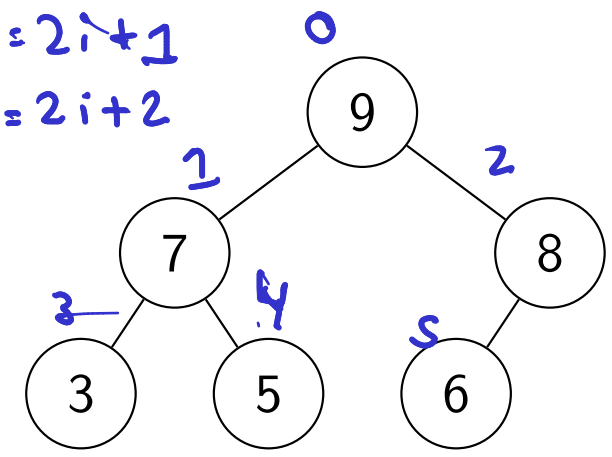
Memoria y punteros

En C++ el nodo es `struct Nodo { int clave; Nodo *izq; Nodo *der; };`, cada uno se crea con `new` y el árbol vacío es `nullptr`. La inserción recursiva recibe el apuntador por referencia, `Nodo *&r`, para poder colgar el nodo nuevo donde estaba el `nullptr`. En Python el nodo es un objeto y el vacío es `None`.

Dónde se rompe

Lo que se pide con `new` se libera con `delete`, recorriendo en postorden: primero los hijos y después el nodo. Liberar antes al padre deja a los hijos sin forma de alcanzarlos. Bajar por un `nullptr` sin preguntar antes termina el programa con violación de segmento.

Montículo: `priority_queue` y `heapq`

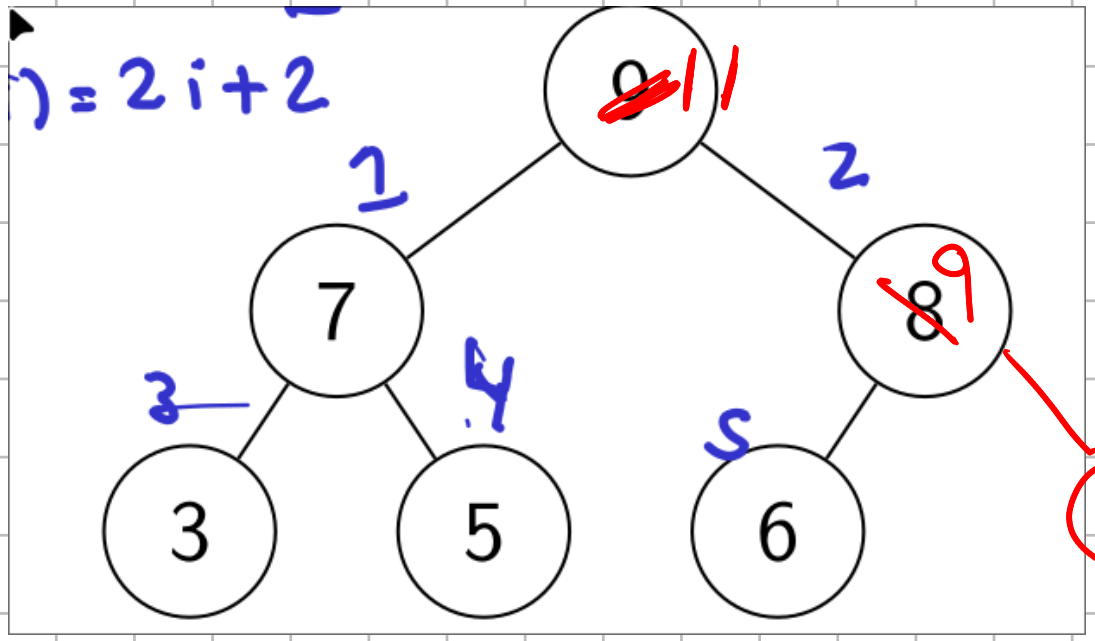


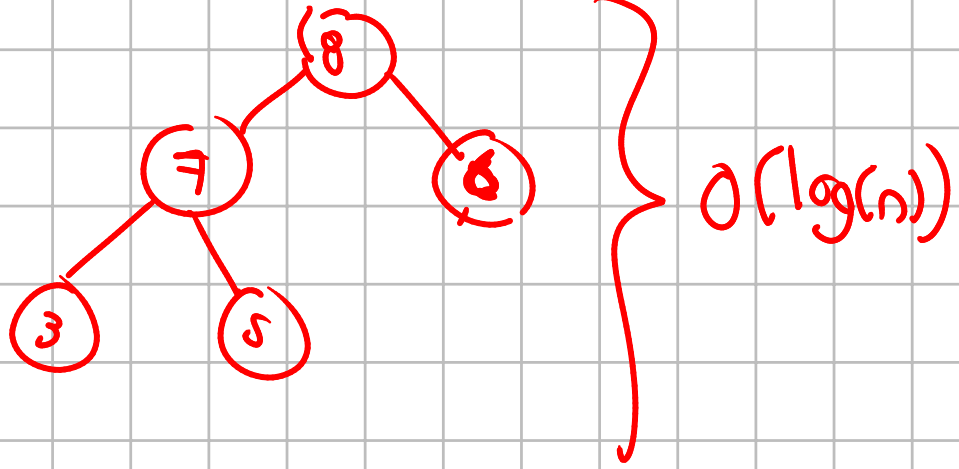
0	1	2	3	4	5
9	7	8	3	5	6

el mismo árbol en un arreglo

Cómo funciona

Es un árbol completo, lleno nivel por nivel, guardado en un arreglo sin apuntadores: los hijos de la casilla i están en $2i+1$ y $2i+2$, y su padre en $\lfloor (i-1)/2 \rfloor$. Cada padre es mayor o igual que sus hijos, así que el mayor está en la casilla 0. Insertar pone el valor al final y lo sube intercambiándolo con su padre; sacar el tope pone el último en la raíz y lo baja. Cada uno recorre una rama: $O(\log n)$.


)=2i+2
0
1
2
7
8
3
4
5
6
3
5
6
9
1
$\log_2(n)$


8
7
9
3
5
$O(\log(n))$

Montículo: `priority_queue` y `heapq`

Memoria, punteros e iteradores

`priority_queue` guarda el arreglo en un `vector` y no da iteradores: solo `top()`, que devuelve una referencia constante. En Python el montículo es la misma `list`: `heapq` solo reordena sus casillas.

Dónde se rompe

No hay forma de cambiar la prioridad de un elemento que ya entró. Se inserta otra vez con la prioridad nueva y, al sacarlo, se descarta la copia vieja si ya no vale; así lo hace el algoritmo de Dijkstra. En Python, escribir en la `list` sin `heappush` ni `heappop`, con `append` o `h[i] = x`, rompe el orden del montículo sin ningún error.

Lo que cuesta: las estructuras en fila

Estructura	Leer	Escribir	Buscar	Recorrer
Arreglo dinámico	$O(1)$ en i	$O(1)$ al final $O(n-p)$ en p	$O(n)$	$O(n)$
Lista enlazada	$O(i)$ en i	$O(1)$ en el iterador	$O(n)$	$O(n)$
Doble extremo	$O(1)$ en los extremos	$O(1)$ en los extremos	$O(n)$	$O(n)$
Pila	$O(1)$ el tope	$O(1)$ en el tope	no se busca	no se recorre
Cola	$O(1)$ el frente	$O(1)$ al final	no se busca	no se recorre

Las columnas

Leer es consultar un elemento sin quitarlo; *escribir*, insertarlo; *buscar*, preguntar si un valor está; *recorrer*, visitar todos una vez. Borrar cuesta lo mismo que escribir en el mismo sitio.

Lo que cuesta: las estructuras en fila

Buscar en un arreglo ordenado

Con el arreglo ordenado, la búsqueda binaria baja de $O(n)$ a $O(\log n)$.
Ordenar cuesta $O(n \log n)$ con `sort` y `a.sort()`, y se paga una sola vez.

Lo que cuesta: las estructuras por clave y por prioridad

Estructura	Leer	Escribir	Buscar	Recorrer
Direccionamiento directo	$O(1)$	$O(1)$	$O(1)$	$O(U)$
Tabla hash	$O(1)$ promedio	$O(1)$ promedio	$O(1)$ promedio	$O(n)$, sin orden
Árbol balanceado	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$, en orden
Árbol binario de búsqueda	$O(h)$	$O(h)$	$O(h)$	$O(n)$, en orden
Montículo	$O(1)$ el tope	$O(\log n)$	$O(n)$	no en orden

Las letras

U es la cantidad de claves posibles, como las 26 letras o los números entre 0 y 10^6. h es la altura del árbol: $\log_2 n$ si quedó parejo, n si las claves llegaron ordenadas y cada nodo cuelga del anterior.

Lo que cuesta: las estructuras por clave y por prioridad

Lo que da el árbol y no da la tabla

`map` y `set` recorren las claves de menor a mayor, y `lower_bound(x)` encuentra en $O(\log n)$ la primera que no es menor que x. La tabla hash solo responde si la clave está. Sacar el tope del montículo cuesta $O(\log n)$, y armarlo de una vez con n valores, $O(n)$.

Lo que la tabla no deja ver

La cola de Python con `list`

`a.pop(0)` corre todos los elementos una casilla: $O(n)$. Una cola de n elementos atendida así cuesta $O(n^2)$. La cola de Python es `collections.deque` con `popleft`, que es $O(1)$.

El tope del montículo

`priority_queue` de C++ deja arriba el *mayor*; `heapq` de Python, el *menor*. Para el menor en C++ se declara `priority_queue<int, vector<int>, greater<int>>`; para el mayor en Python se guardan los valores con el signo cambiado.

Lo que la tabla no deja ver

El promedio de la tabla hash

$O(1)$ es el caso promedio. Si muchas claves caen en la misma casilla, buscar se vuelve $O(n)$. Con `map` el costo es $O(\log n)$ siempre.

El índice en el `deque` de Python

`d[i]` es $O(1)$ cerca de los extremos y $O(n)$ en la mitad. Para leer por posición se usa `list`.

Leer, escribir y recorrer en C++

Estructura	Leer	Escribir	Recorrer
`vector`	`v[i]`	`v.push_back(x)`	`for` por índice
`list`	`*it`	`l.insert(it, x)`	con iterador, de `begin()` a `end()`
`stack`	`s.top()`	`s.push(x)`	se vacía con `s.pop()`
`queue`	`q.front()`	`q.push(x)`	se vacía con `q.pop()`
`map`	`m.find(k)`	`m[k] = v`	con iterador, en orden de clave
`set`	`s.count(x)`	`s.insert(x)`	con iterador, en orden
`priority_queue`	`pq.top()`	`pq.push(x)`	se vacía con `pq.pop()`

Leer una clave que no está

`m[k]` crea la clave con valor 0 si no existía. Para solo preguntar se usa `m.find(k) != m.end()` o `m.count(k)`.

Leer, escribir y recorrer en Python

Estructura	Leer	Escribir	Recorrer
`list`	`a[i]`	`a.append(x)`	`for x in a`
pila sobre `list`	`a[-1]`	`a.append(x)`	se vacía con `a.pop()`
`deque` como cola	`d[0]`	`d.append(x)`	se vacía con `d.popleft()`
`dict`	`m[k]`, `k in m`	`m[k] = v`	`for k in m`, en orden de llegada
`set`	`x in s`	`s.add(x)`	`for x in s`, sin orden
`heapq`	`h[0]`	`heappush(h, x)`	se vacía con `heappop(h)`

Leer una clave que no está

`m[k]` lanza `KeyError` si la clave no existe. Se pregunta antes con `k in m`, o se lee con `m.get(k, 0)`.

En orden

Para recorrer un `dict` o un `set` de menor a mayor se usa `sorted(m)`, que cuesta $O(n \log n)$.

Cuál usar según lo que hace el programa

El programa necesita	Estructura	Problema del refuerzo
Guardar la entrada y leerla por posición	arreglo dinámico	UVa 11078, UVa 507
Guardar la salida y escribirla al final	arreglo dinámico de líneas	UVa 10189
Deshacer: el último que entró sale primero	pila	UVa 839 (la pila de llamadas)
Atender en orden de llegada	cola	UVa 12100
Insertar en medio sin correr nada	lista enlazada	UVa 11988
Contar cuántas veces aparece cada clave	tabla hash o `map`	UVa 156, UVa 10226, UVa 11286
Saber si un valor ya apareció	conjunto	UVa 924 (los ya visitados)
Sacar siempre el menor o el mayor	montículo	HackerRank QHEAP1, UVa 1203
El vecino de un valor en orden	`set` o arreglo ordenado	UVa 10611
Visitar por capas desde un origen	cola	UVa 924

Cómo se escoge

Primero se escribe qué operación se repite dentro del ciclo principal; después se busca en las dos tablas la estructura donde esa operación es $O(1)$ u $O(\log n)$.

Contenido

1 La entrada y lo que cuesta cada operación

2 Un recorrido en vez de dos
- Concepto
- Ejemplo
- Ejercicio: UVa 11078

3 El mejor tramo de un arreglo

4 Rectángulos en una matriz

5 Tarea del bloque

El acumulado que se arrastra

El concepto

Si lo que se necesita de las casillas anteriores se puede actualizar con la que acaba de entrar, el segundo recorrido sobra: ese dato se guarda en una variable y se actualiza en la misma vuelta. Un recorrido, $\Theta(n)$, donde los dos ciclos anidados costaban del orden de n^2.

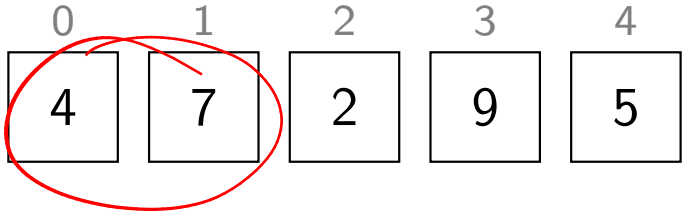
Qué se puede arrastrar

El máximo y el mínimo, la suma, el conteo de los que cumplen algo, el mejor resultado visto hasta ahora. Todos se actualizan en $O(1)$ con el valor nuevo.

Cómo se reconoce el caso

Dos ciclos anidados donde el de adentro vuelve a mirar lo que el de afuera ya recorrió.

Para cada posición, lo mejor de atrás

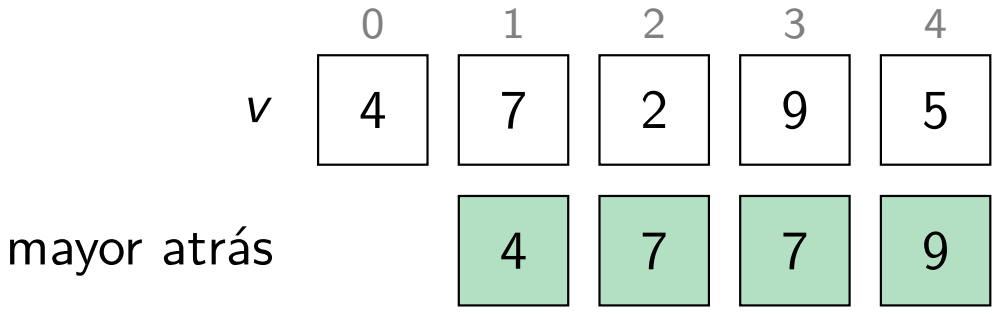


A mano

Para cada posición j, escriba el mayor valor que quedó atrás, entre $v[0]$ y $v[j-1]$.

j=0 v[0] y v[-1]
j=1 v[0] y v[0] = 4
j=2 v[0] y v[1] = 7
j=3 v[0] y v[3] = 9

Para cada posición, lo mejor de atrás



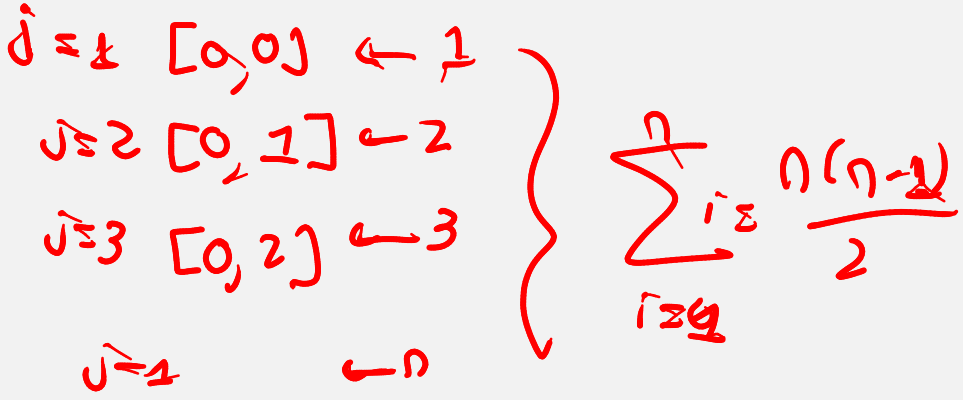
Lo que se ve al escribirlo

La casilla 0 no tiene nada atrás. De una posición a la siguiente el valor no baja nunca, y cambia solo cuando el que acaba de pasar es más grande que todos los anteriores.

La forma directa

Python: `prefijo_maximo.py`

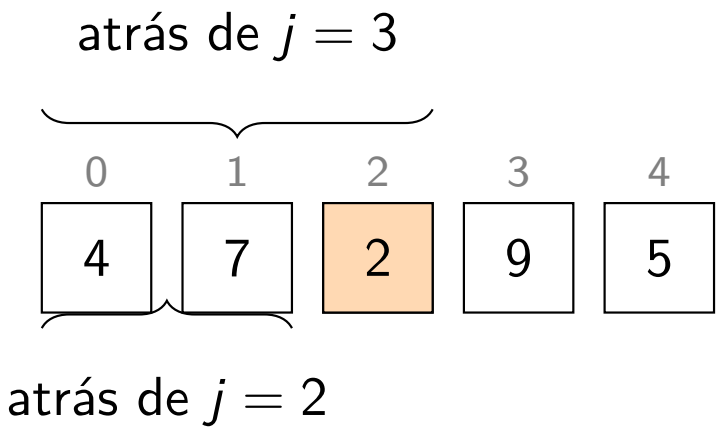
```
def maximos_dos_ciclos(v):
    res = [None] * len(v)
    for j in range(1, len(v)):
        mayor = v[0]
        for i in range(1, j):
            if v[i] > mayor:
                mayor = v[i]
        res[j] = mayor
    return res
```



¿Cuánto cuesta?

Por cada j se recorre lo de atrás: $n(n-1)/2$ comparaciones, del orden de n^2. Con $n = 100\,000$ son $5 \cdot 10^9$, y ningún límite de tiempo aguanta eso.

¿Qué cambia de una posición a la siguiente?



Pregunta

Los dos conjuntos se diferencian en una sola casilla, la que acaba de entrar. ¿Hace falta volver a recorrer el resto para saber cuál es el mayor?

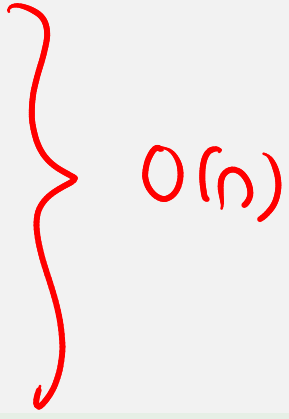
¿Qué cambia de una posición a la siguiente?

Respuesta

No. El mayor de atrás de $j+1$ es el mayor entre dos cosas: el mayor de atrás de j, que ya se tenía, y $v[j]$, el que acaba de entrar. Una comparación por posición, un recorrido, $\Theta(n)$.

Python: `prefijo_maximo.py`

```
def maximos_un_recorrido(v):
    res = [None] * len(v)
    mayor = v[0]
    for j in range(1, len(v)):
        res[j] = mayor
        if v[j] > mayor:
            mayor = v[j]
    return res
```



La traza, paso a paso

Estado

`mayor` guarda el máximo de lo ya recorrido. Con $v = [4, 7, 2, 9, 5]$ arranca en $v[0] = 4$.

j	$v[j]$	`mayor` al entrar	`mayor` al salir
1	7	4	7
2	2	7	?
3	9	?	?
4	5	?	?

La traza, paso a paso

j	$v[j]$	`mayor` al entrar	`mayor` al salir
1	7	4	7
2	2	7	7
3	9	7	9
4	5	9	9

El orden dentro de la vuelta

Primero se usa `mayor` y después entra $v[j]$. Al revés, en $j = 3$ se leería 9 como "lo de atrás" y la casilla se estaría comparando consigo misma.

UVa 11078: Open Credit System

Enunciado, en corto

Un curso tiene n estudiantes, numerados por antigüedad: si $i < j$, el estudiante i es más antiguo que el j. Cada uno presentó una prueba. El profesor quiere la mayor diferencia que un antiguo le saca a uno más nuevo: el máximo de $a[i] - a[j]$ con $i < j$. Sin valor absoluto.

Límites

Menos de 20 casos; en cada uno, $2 \leq n \leq 100\,000$, un puntaje por línea, $|a[i]| < 150\,000$.

Direcciones

Enunciado: `https://onlinejudge.org/external/110/11078.pdf`

Envío: `https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=25&page=submit_problem&problemid=2019`

¿Entendimos el problema?

Muestra

```
3
2
100
20
4
4 3 2 1
4
1 2 3 4
```

En el archivo, un número por línea.

Entradas y salidas

Entra T y, por caso, n y los n puntajes. Sale un número por caso.

A mano

- $[100, 20]$: una sola pareja, $100 - 20 = 80$.
- $[4, 3, 2, 1]$: la mejor es $4 - 1 = 3$.
- $[1, 2, 3, 4]$: todas las parejas dan negativo; la mejor es $1 - 2 = -1$.

Pregunta

En el tercer caso, ¿por qué la respuesta no es 3, que es $4 - 1$?

¿Entendimos el problema?

Respuesta

Porque el 4 es el último: es el más nuevo y tiene que ir a la derecha en la resta. Con $i < j$, las parejas del tercer caso son $1-2$, $1-3$, $1-4$, $2-3$, $2-4$ y $3-4$, y la mayor vale -1.

Lo que eso dice de la salida

La respuesta puede ser negativa. Un programa que imprime 0 cuando ningún antiguo supera a uno nuevo falla el tercer caso de la muestra.

Primera idea: todas las parejas

Python: `credito_ingenua.py`

```python
def mayor_diferencia(a):
    mejor = a[0] - a[1]
    for i in range(len(a)):
        for j in range(i + 1, len(a)):
            if a[i] - a[j] > mejor:
                mejor = a[i] - a[j]
    return mejor
```

¿Cuánto cuesta?

El ciclo de adentro da $n - 1 - i$ vueltas: en total $n(n-1)/2$ parejas. Con $n = 100\,000$ son unos $5 \cdot 10^9$ por caso, y puede haber 19 casos.

Primera idea: todas las parejas

Medido (un caso; Python 3.14, Ryzen 7 7435HS)

n	dos ciclos	un recorrido
10 000	2,84 s	0,02 s
20 000	11,32 s	0,02 s

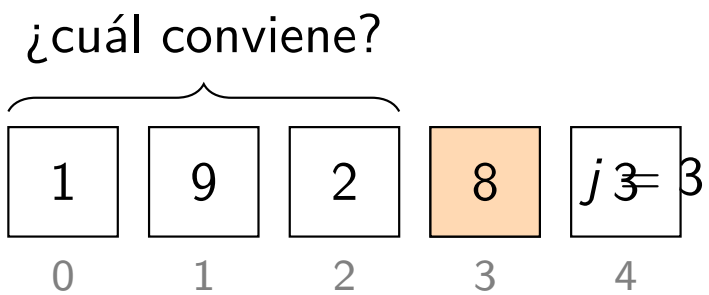
Contra los límites

Al duplicar n el tiempo de los dos ciclos se multiplica por 4. De 10 000 a 100 000 el factor es $10^2 = 100$: unos 280 segundos por caso. El límite del juez es de 3 segundos para todo el archivo.

Fijar al más nuevo

Pregunta

Fije el estudiante nuevo j. De todos los antiguos $i < j$, ¿cuál le saca la mayor diferencia?



Fijar al más nuevo

Respuesta

El de mayor puntaje entre $a[0], \ldots, a[j-1]$: restar siempre el mismo $a[j]$ deja la diferencia más grande cuando el primer término es el mayor. Para $j = 3$, el 9: $9 - 8 = 1$.

Y ese mayor no hay que buscarlo

Al pasar de j a $j+1$ entra un solo candidato nuevo, $a[j]$. El mayor de los antiguos se actualiza con una comparación, como el máximo de un recorrido. La respuesta es el mejor $\mathit{mayor} - a[j]$ que aparezca en el camino.

Costo

Un recorrido, dos comparaciones por vuelta: $\Theta(n)$ por caso.

Traza con cinco estudiantes

Estado

`mayor` es el máximo de $a[0..j-1]$ y `mejor` la mejor diferencia vista. Con $a = [1, 9, 2, 8, 3]$ arrancan en $a[0] = 1$ y en $a[0] - a[1] = -8$.

j	$a[j]$	$mayor - a[j]$	`mejor`	`mayor` al salir
inicio			-8	1
1	9	$1 - 9 = -8$	-8	9
2	2	?	?	?
3	8	?	?	?
4	3	?	?	?

Traza con cinco estudiantes

Estado

$a = [1, 9, 2, 8, 3]$.

j	$a[j]$	$mayor - a[j]$	`mejor`	`mayor` al salir
inicio			-8	1
1	9	$1 - 9 = -8$	-8	9
2	2	$9 - 2 = 7$	7	9
3	8	$9 - 8 = 1$	7	9
4	3	$9 - 3 = 6$	7	9

Respuesta: 7

El orden dentro de la vuelta importa: primero se compara $mayor - a[j]$, después $a[j]$ entra al máximo. Al revés, en $j = 1$ se calcularía $9 - 9 = 0$, una pareja del estudiante consigo mismo.

Python: open-credit-system.py

El recorrido

```
def mayor_diferencia(a):
    mayor = a[0]
    mejor = a[0] - a[1]
    for j in range(1, len(a)):
        if mayor - a[j] > mejor:
            mejor = mayor - a[j]
        if a[j] > mayor:
            mayor = a[j]
    return mejor
```

Python: open-credit-system.py

La lectura por palabras

```
def main():
    datos = sys.stdin.read().split()
    t = int(datos[0])
    pos = 1
    for caso in range(t):
        n = int(datos[pos])
        a = []
        for k in range(n):
            a.append(int(datos[pos + 1 + k]))
        pos = pos + 1 + n
        print(mayor_diferencia(a))


main()
```

C++: open-credit-system.cpp

El recorrido

```cpp
int mayor_diferencia(const vector<int>& a) {
    int mayor = a[0];
    int mejor = a[0] - a[1];
    for (int j = 1; j < (int) a.size(); j++) {
        if (mayor - a[j] > mejor) {
            mejor = mayor - a[j];
        }
        if (a[j] > mayor) {
            mayor = a[j];
        }
    }
    return mejor;
}
```

C++: open-credit-system.cpp

La lectura

```cpp
int main() {
    int t;
    scanf("%d", &t);
    for (int caso = 0; caso < t; caso++) {
        int n;
        scanf("%d", &n);
        vector<int> a(n);
        for (int k = 0; k < n; k++) {
            scanf("%d", &a[k]);
        }
        printf("%d\n", mayor_diferencia(a));
    }
    return 0;
}
```

¿Cabe en int?

$|a[i]| < 150\,000$, así que la diferencia está entre $-300\,000$ y $300\,000$.
Sobra espacio en 32 bits.

Trampas

`mejor = 0` al comienzo: pasa dos casos de la muestra y falla el tercero

j	$a[j]$	$mayor - a[j]$	`mejor`	`mayor` al salir
inicio			0	1
1	2	-1	0	2
2	3	-1	0	3
3	4	-1	0	4

Corrección

`mejor` arranca con una pareja que existe, $a[0] - a[1]$. Así el resultado siempre es la diferencia de dos estudiantes, aunque sea negativa.

`max(a[:j])` dentro del ciclo

Vuelve a dar $O(n^2)$: es el costo que no se ve de la tabla del repaso.

Variante para pensar

Con distancia mínima

El profesor solo compara estudiantes que estén al menos d puestos aparte: el máximo de $a[i] - a[j]$ con $j - i \geq d$. Con $a = [1, 9, 2, 8, 3]$ y $d = 2$, la respuesta es $9 - 3 = 6$. ¿Sigue siendo un solo recorrido?

Pista

Para el j actual, los antiguos permitidos son $a[0], \ldots, a[j - d]$. Cuando j avanza una posición, entra un solo candidato, $a[j - d]$.

Contenido

1 La entrada y lo que cuesta cada operación

2 Un recorrido en vez de dos

3 El mejor tramo de un arreglo
- Concepto
- Ejemplo
- Ejercicio: UVa 507

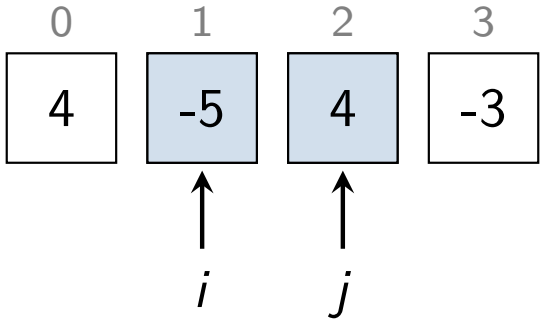
4 Rectángulos en una matriz

5 Tarea del bloque

¿Cuántos tramos tiene un arreglo?

Pregunta

Un tramo de v es un bloque de casillas seguidas, $v[i..j]$ con $i \leq j$.
¿Cuántos tramos tiene un arreglo de 4 casillas?



Respuesta

Diez: cuatro empiezan en 0, tres en 1, dos en 2 y uno en 3. Con n casillas son $n + (n-1) + \cdots + 1 = n(n+1)/2$, del orden de n^2. Un tramo queda dicho por dos índices, el primero y el último.

La suma de un tramo sin recorrerlo

Pregunta

Sumar $v[i..j]$ recorriéndolo cuesta $j - i + 1$ pasos. ¿Se puede en $O(1)$?

Respuesta: los acumulados

Con $acum[0] = 0$ y $acum[k+1] = acum[k] + v[k]$, la suma $v[i] + \cdots + v[j]$ es $acum[j+1] - acum[i]$. Prepararlos es un recorrido, $\Theta(n)$, una sola vez.

Con $v = [4, -5, 4, -3]$

k	0	1	2	3	4
$v[k]$	4	-5	4	-3	
$acum[k]$	0	4	-1	3	0

$$v[1.,2] = -5 + 4 = acum[3] - acum[1] = 3 - 4 = -1.$$

El mejor tramo que termina en cada casilla

El concepto

En lugar de mirar todos los tramos, se recorre una vez preguntando por el mejor tramo que termina en la casilla actual. Ese tramo es la casilla sola, o el mejor que terminaba en la casilla anterior más esta. Un recorrido, $\Theta(n)$, en vez de los $n(n+1)/2$ tramos.

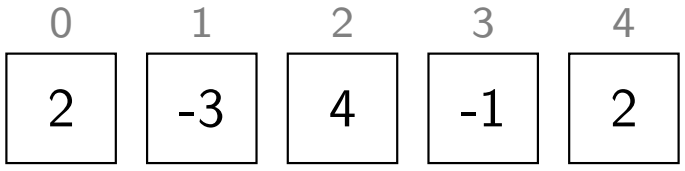
Por qué alcanza

Todo tramo termina en alguna casilla. Si para cada casilla se tiene el mejor que termina ahí, el mejor de todos está entre esos n valores.

Dónde más sirve

El mismo recorrido, cambiando la suma por otra cuenta: el tramo de más casillas que cumple algo, la mejor racha, la ganancia de comprar y vender una vez.

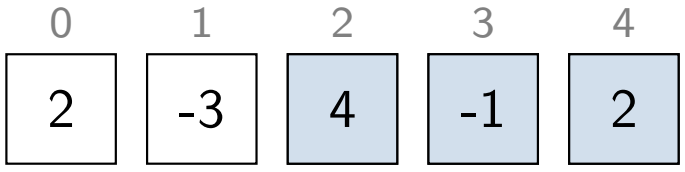
El tramo que más suma



A mano

Busque el tramo de casillas seguidas que más suma. Vale cualquier tramo, desde una casilla sola hasta el arreglo entero.

El tramo que más suma



Respuesta: 5

El tramo $v[2.,4]$, con $4 - 1 + 2 = 5$. Otros candidatos: $v[0.,0]$ suma 2, $v[0.,4]$ suma 4, $v[2.,2]$ suma 4. El -1 del medio se acepta porque lo que viene después lo paga.

La forma directa

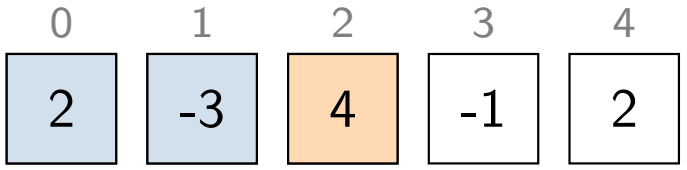
Python: `mejor_tramo.py`

```python
def mejor_tramo_acumulados(v):
    acum = [0] * (len(v) + 1)
    for k in range(len(v)):
        acum[k + 1] = acum[k] + v[k]
    mejor = v[0]
    for i in range(len(v)):
        for j in range(i, len(v)):
            suma = acum[j + 1] - acum[i]
            if suma > mejor:
                mejor = suma
    return mejor
```

¿Cuánto cuesta?

Los acumulados dan cada suma en $O(1)$, pero los tramos siguen siendo $n(n+1)/2$, del orden de n^2: con $n = 20\,000$, unos $2 \cdot 10^8$ por caso, y los casos vienen en tandas.

¿Conviene arrastrar lo que ya se lleva?



$v[0.,1] = -1$

Pregunta

El tramo que viene arrastrado desde la casilla 0 suma -1. Al llegar a la casilla 2, ¿conviene seguirlo o empezar un tramo nuevo ahí?

¿Conviene arrastrar lo que ya se lleva?

Respuesta

Empezar de nuevo. Un tramo que llega negativo solo resta: $-1 + 4$ da 3 y el que arranca en la casilla 2 da 4. La regla no depende de los números, solo del signo de lo que se lleva. Un recorrido, $\Theta(n)$, con dos variables: el tramo que termina en la casilla actual y el mejor visto.

Python: `mejor_tramo.py`

```
def mejor_tramo(v):
    mejor = v[0]
    actual = v[0]
    for k in range(1, len(v)):
        if actual < 0:
            actual = v[k]
        else:
            actual = actual + v[k]
        if actual > mejor:
            mejor = actual
    return mejor
```

La traza, paso a paso

Estado

`actual` es la mejor suma de un tramo que termina en k, y `mejor` la mejor de todas. Con $v = [2, -3, 4, -1, 2]$ las dos arrancan en $v[0] = 2$.

k	$v[k]$	¿arrastra?	`actual`	`mejor`
inicio			2	2
1	-3	sí, $2 \geq 0$	-1	2
2	4	?	?	?
3	-1	?	?	?
4	2	?	?	?

La traza, paso a paso

k	$v[k]$	¿arrastra?	`actual`	`mejor`
inicio			2	2
1	-3	sí, $2 \geq 0$	-1	2
2	4	no, $-1 < 0$	4	4
3	-1	sí, $4 \geq 0$	3	4
4	2	sí, $3 \geq 0$	5	5

Con todo negativo

En $[-4, -2, -7]$ el recorrido devuelve -2, la casilla menos mala. Arrancar `mejor` en 0 daría 0, que es la respuesta del tramo vacío: solo sirve si el enunciado acepta no tomar nada.

UVa 507: Jill Rides Again

Enunciado, en corto

Una ruta de bus tiene s paradas. Entre la parada i y la $i+1$ hay una calle con un valor n_i, distinto de cero, que dice cuánto le gusta a Jill. Ella baja en una parada, pedalea y vuelve a subir en otra. Hay b rutas, cada una con $2 \leq s \leq 20\,000$ paradas. Se pide el tramo de paradas $i < j$ con la mayor suma $n_i + \cdots + n_{j-1}$.

Desempates y salida

Si varios tramos tienen la suma máxima, gana el más largo (mayor $j - i$); si siguen empatados, el que empieza antes (menor i). Se imprime `The nicest part of route r is between stops i and j`, o `Route r has no nice parts` si la suma máxima no es positiva.

UVa 507: Jill Rides Again

Direcciones

Enunciado: `https://onlinejudge.org/external/5/507.pdf`

Envío: `https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=25&page=submit_problem&problemid=448`

¿Entendimos el problema?

Muestra

```
3
3
-1 6
10
4 -5 4 -3 4
4 -4 4 -5
4
-2 -3 -4
```

En el archivo, un número por línea.

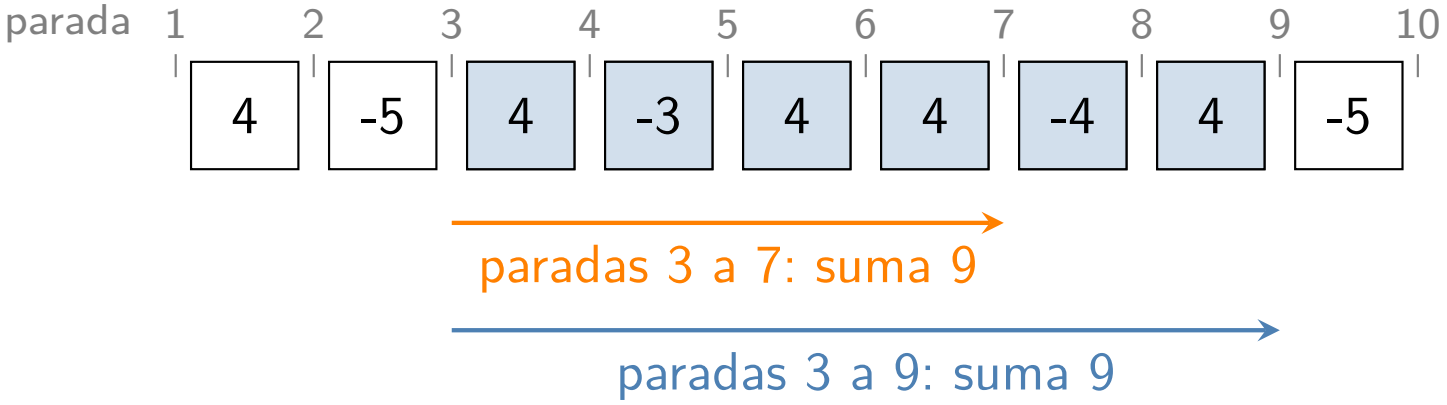
A mano

- Ruta 1: calles -1 y 6. Lo mejor es la segunda calle sola, de la parada 2 a la 3.
- Ruta 3: todas negativas. No hay tramo positivo: `Route 3 has no nice parts`.

Pregunta

En la ruta 2, el tramo de las paradas 3 a 7 suma $4 - 3 + 4 + 4 = 9$. ¿Por qué la respuesta es de la 3 a la 9?

¿Entendimos el problema?



Respuesta

Los dos suman 9 y el de 3 a 9 es más largo: las dos calles que agrega, -4 y 4, suman cero. Es el primer desempate.

Índices

`v[k]` une las paradas $k+1$ y $k+2$, así que el tramo de calles $v[i..j]$ va de la parada $i+1$ a la $j+2$.

Primera idea: todos los tramos

Python: `jill_acumulados.py`, con la suma de cada tramo en $O(1)$

```python
def mejor_tramo(v):
    acum = [0]
    for x in v:
        acum.append(acum[-1] + x)
    mejor = 0
    mi = 0
    mj = -1
    for i in range(len(v)):
        for j in range(i, len(v)):
            suma = acum[j + 1] - acum[i]
            if suma > mejor or (suma == mejor and j - i > mj - mi):
                mejor = suma
                mi = i
                mj = j
    return mejor, mi, mj
```

El desempate sale del orden: i crece afuera y la comparación de largo es estricta, así que entre dos tramos iguales en suma y en largo queda el de menor i.

Primera idea: todos los tramos

¿Cuánto cuesta?

Los tramos son $(s-1)s/2$, unos $2 \cdot 10^8$ con $s = 20\,000$. Sin acumulados, cada suma cuesta además hasta s pasos y el total pasa a $O(s^3)$.

Medido (una ruta; Python 3.14, Ryzen 7 7435HS)

s	todos los tramos	un recorrido
5 000	1,29 s	0,02 s
10 000	5,11 s	0,02 s
20 000	20,77 s	0,02 s

Contra los límites

Veinte segundos por una sola ruta de tamaño máximo, contra 3 segundos de límite para el archivo entero, que trae varias.

¿Sirve lo que se lleva sumado?

Pregunta

Se recorre la ruta llevando la suma del tramo que empezó en `inicio` y termina en la calle actual k. Si esa suma se vuelve negativa, ¿le conviene a algún tramo que siga después de k arrastrarla?

Con $v = [4, -5, 4, \ldots]$

Después de $4 - 5$ la suma es -1. Un tramo que empiece en la calle 2 vale lo mismo que con ese -1 delante, más uno.

¿Sirve lo que se lleva sumado?

Respuesta

No. Un prefijo negativo solo resta: la suma vuelve a 0 y el tramo empieza en $k + 1$. Es la idea de Kadane.

El recorrido, en palabras: $\Theta(s)$ por ruta

1. Sumar $v[k]$ a la suma actual.
2. Si la suma supera a la mejor, o la iguala con un tramo más largo, guardar el tramo $[inicio, k]$.
3. Si la suma quedó negativa, volver a 0 y poner $inicio = k + 1$.

¿Negativa o no positiva?

Se reinicia con $suma < 0$, no con $suma \leq 0$. Un prefijo que suma cero no resta, pero alarga el tramo, y el enunciado premia el tramo largo.

Traza con la ruta 2

k	$v[k]$	`suma`	`inicio`	`mejor`	tramo $[mi, mj]$
0	4	4	0	4	$[0, 0]$
1	-5	$-1 \to 0$	$0 \to 2$	4	$[0, 0]$
2	4	4	2	4	?
3	-3	?	?	?	?
4	4	?	?	?	?
5	4	?	?	?	?
6	-4	?	?	?	?
7	4	?	?	?	?
8	-5	?	?	?	?

Pregunta

En $k = 2$ la suma vuelve a 4, igual que la mejor. ¿Cambia el tramo guardado?

Traza con la ruta 2

k	$v[k]$	`suma`	`inicio`	`mejor`	tramo $[mi, mj]$
0	4	4	0	4	$[0, 0]$
1	-5	$-1 \to 0$	$0 \to 2$	4	$[0, 0]$
2	4	4	2	4	$[0, 0]$, mismo largo
3	-3	1	2	4	$[0, 0]$
4	4	5	2	5	$[2, 4]$
5	4	9	2	9	$[2, 5]$
6	-4	5	2	9	$[2, 5]$
7	4	9	2	9	$[2, 7]$, más largo
8	-5	4	2	9	$[2, 7]$

Respuesta

En $k = 2$ no: el tramo $[2, 2]$ tiene el mismo largo que $[0, 0]$ y empieza después.
En $k = 7$ sí: suma 9 otra vez, con más calles. Calles $[2, 7]$ son las paradas 3 a 9.

Python: jill-rides-again.py

El recorrido

```
def mejor_tramo(v):
    # v[k] es la calle entre las paradas k + 1 y k + 2
    suma = 0
    inicio = 0
    mejor = 0
    mi = 0
    mj = -1
    for k in range(len(v)):
        suma = suma + v[k]
        if suma > mejor or (suma == mejor and k - inicio > mj - mi):
            mejor = suma
            mi = inicio
            mj = k
        if suma < 0:
            suma = 0
            inicio = k + 1
    return mejor, mi, mj
```

Python: jill-rides-again.py

La lectura y la salida

```
def main():
    datos = sys.stdin.read().split()
    b = int(datos[0])
    pos = 1
    for r in range(1, b + 1):
        s = int(datos[pos])
        v = []
        for k in range(s - 1):
            v.append(int(datos[pos + 1 + k]))
        pos = pos + s
        mejor, mi, mj = mejor_tramo(v)
        if mejor > 0:
            print("The nicest part of route %d is between stops %d and %d"
                  % (r, mi + 1, mj + 2))
        else:
            print("Route %d has no nice parts" % r)


main()
```

C++: jill-rides-again.cpp

El recorrido

```cpp
long long mejor_tramo(const vector<int>& v, int& mi, int& mj) {
    long long suma = 0;
    int inicio = 0;
    long long mejor = 0;
    mi = 0;
    mj = -1;
    for (int k = 0; k < (int) v.size(); k++) {
        suma = suma + v[k];
        if (suma > mejor || (suma == mejor && k - inicio > mj - mi)) {
            mejor = suma;
            mi = inicio;
            mj = k;
        }
        if (suma < 0) {
            suma = 0;
            inicio = k + 1;
        }
    }
    return mejor;
}
```

C++: jill-rides-again.cpp

```cpp
int main() {
    int b;
    scanf("%d", &b);
    for (int r = 1; r <= b; r++) {
        int s;
        scanf("%d", &s);
        vector<int> v(s - 1);
        for (int k = 0; k < s - 1; k++) {
            scanf("%d", &v[k]);
        }
        int mi, mj;
        long long mejor = mejor_tramo(v, mi, mj);
        if (mejor > 0) {
            printf("The nicest part of route %d is between stops %d and %d\n",
                   r, mi + 1, mj + 2);
        } else {
            printf("Route %d has no nice parts\n", r);
        }
    }
    return 0;
}
```

Trampas

Reiniciar con $suma \leq 0$

Con $v = [1, -1, 5]$, los tramos $[0, 2]$ y $[2, 2]$ suman 5, y gana el largo: paradas 1 a 4. Si la suma se reinicia al llegar a cero, en $k = 1$ el tramo arranca de nuevo en la calle 2 y la respuesta sale de la parada 3 a la 4.

Comparar solo la suma

Con `suma > mejor` y nada más, la ruta 2 de la muestra da paradas 3 a 7.

La salida exacta

Paradas desde 1: la parada final es $mj + 2$. Cuando la mejor suma no es positiva va la otra frase, y la ruta se numera aunque no tenga tramo.

Variante para pensar

Ruta circular

Después de la última parada el bus vuelve a la primera, así que un tramo puede dar la vuelta: terminar en la calle 0 después de pasar por la última. ¿Cómo se halla la mejor suma con un recorrido?

Pista

Un tramo que da la vuelta deja por fuera un tramo del medio. Lo que queda es la suma total menos ese tramo: hay que quitar el de suma mínima, que también se halla con un recorrido.

Contenido

1 La entrada y lo que cuesta cada operación

2 Un recorrido en vez de dos

3 El mejor tramo de un arreglo

4 Rectángulos en una matriz
- Concepto
- Ejemplo
- Ejercicio: UVa 108

5 Tarea del bloque

¿Cómo se guarda una tabla?

Python: lista de listas

`m[i]` es la fila i; `m[i][j]`, la celda. Se construye fila por fila con `append`.

C++: `vector<vector<int>>`

`vector<vector<int>> m(n, vector<int>(n));` crea n filas de n ceros.

Lo que cuesta

`m[i][j]` es $O(1)$; recorrer la matriz de $f \times c$ es $\Theta(f \cdot c)$; sumar una fila es $\Theta(c)$ y una columna, $\Theta(f)$.

¿Cómo se guarda una tabla?

Pregunta: ¿qué imprime?

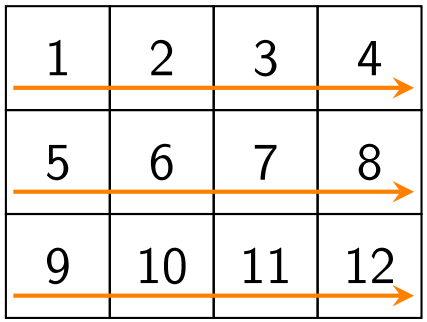
```
# Dos formas de crear una matriz de 3 x 3 con ceros.
mala = [[0] * 3] * 3
buena = [[0] * 3 for i in range(3)]
mala[0][0] = 7
buena[0][0] = 7
print("mala: ", mala)
print("buena:", buena)
```

Respuesta

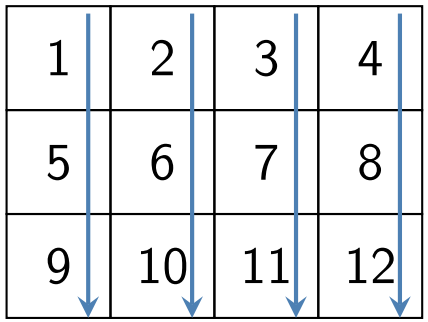
```
mala:  [[7, 0, 0], [7, 0, 0], [7, 0, 0]]
buena: [[7, 0, 0], [0, 0, 0], [0, 0, 0]]
```

`[fila] * 3` guarda tres referencias a la misma lista. Cada fila de una matriz se crea por separado.

Filas, columnas y vecinas



i afuera



j afuera

Qué ciclo va afuera

El de afuera se queda quieto mientras el de adentro corre. Para sumar una columna, *j* va afuera.

Pregunta

¿Cuántas vecinas tiene una celda, contando lados y esquinas? ¿Y una del borde?

Filas, columnas y vecinas

Respuesta

Ocho en el interior, cinco en un borde, tres en una esquina. Son $(i + d_i, j + d_j)$ con $d_i, d_j \in \{-1, 0, 1\}$, sin $d_i = d_j = 0$, y solo cuentan si $0 \leq i + d_i < f$ y $0 \leq j + d_j < c$. En Python `m[-1]` no da error, da la última fila: el borde se revisa antes del acceso.

Acumulados por columna

Pregunta

$col[c]$ es la suma de la columna c entre las filas `arriba` y `abajo`. Si `abajo` baja una fila, ¿hay que sumar todo otra vez?

Respuesta

No: a cada columna se le suma la celda nueva, $col[c] \mathrel{+}= m[abajo][c]$. Cuesta $\Theta(n)$ por fila.

Con la matriz de la muestra de UVa 108 y $arriba = 1$

`abajo`	$col[0]$	$col[1]$	$col[2]$	$col[3]$
1	9	2	-6	2
2	5	3	-10	3
3	4	11	-10	1

Preparar una vez, responder muchas

El concepto

En una tabla P se guarda, en $P[i][j]$, la suma de todo el rectángulo que va de la esquina $(0, 0)$ hasta la celda $(i - 1, j - 1)$. Llenarla es un recorrido de la matriz, $\Theta(N \cdot M)$ una sola vez, y después cada rectángulo sale con cuatro lecturas de P, sin importar su tamaño.

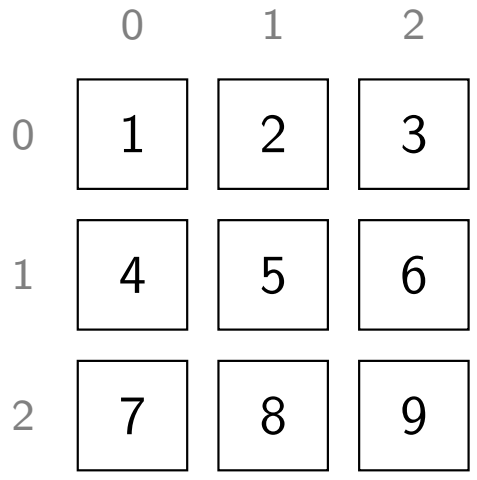
Cuándo conviene

Cuando la matriz no cambia y las consultas son muchas. Si la matriz se modifica entre consulta y consulta, la tabla hay que rehacerla y deja de salir a cuenta.

La otra salida en dos dimensiones

Fijar dos filas y mirar cada columna como una sola casilla con la suma de ese trozo: el problema vuelve a ser de una dimensión, el del mejor tramo.

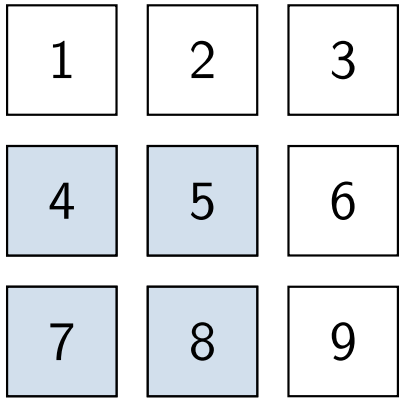
La suma de un rectángulo



A mano

¿Cuánto suma el rectángulo que va de la fila 1 a la 2 y de la columna 0 a la 1?

La suma de un rectángulo



Respuesta: 24

$4 + 5 + 7 + 8$. Recorrer las cuatro celdas es inmediato.

Pregunta

¿Y si hay que responder esa misma cuenta para miles de rectángulos de una matriz de 100×100?

La forma directa

Recorrer cada rectángulo

Dos ciclos anidados sobre las filas y las columnas del rectángulo: hasta $N \cdot M$ celdas por consulta.

Con muchas consultas

En una matriz de 100×100, un rectángulo grande cuesta 10^4 sumas. Mil consultas son 10^7, y cada consulta vuelve a sumar celdas que la anterior ya sumó.

Pregunta

En una dimensión, la suma de un tramo salía en $O(1)$ con los acumulados. ¿Qué sería el acumulado de una matriz?

La tabla de acumulados

Respuesta

La tabla P del concepto, con una fila y una columna más que la matriz: $P[i][j]$ acumula todo lo que queda arriba y a la izquierda de la celda (i, j). La fila 0 y la columna 0 quedan en ceros, que es el tramo vacío de una dimensión visto en dos.

Python: `acumulados2d.py`

```
def tabla_acumulados(m):
    p = []
    for i in range(len(m) + 1):
        p.append([0] * (len(m[0]) + 1))
    for i in range(len(m)):
        for j in range(len(m[0])):
            p[i + 1][j + 1] = m[i][j] + p[i][j + 1] + p[i + 1][j] - p[i][j]
    return p
```

Llenar la tabla

Estado

Con la matriz 1 a 9, cada celda es $m[i][j]$ más la de arriba y la de la izquierda en P, menos la de la diagonal.

P	0	1	2	3
0	0	0	0	0
1	0	1	3	6
2	0	5	?	?
3	0	?	?	?

Para la casilla $P[2][2]$

$m[1][1] + P[1][2] + P[2][1] - P[1][1] = 5 + 3 + 5 - 1$. El rectángulo de arriba y el de la izquierda comparten una parte, que entra dos veces y por eso se descuenta.

Llenar la tabla

P	0	1	2	3
0	0	0	0	0
1	0	1	3	6
2	0	5	12	21
3	0	12	27	45

La esquina de abajo a la derecha

$P[3][3] = 45$ es la suma de toda la matriz, de 1 a 9.

Cuatro lecturas por rectángulo

Python: `acumulados2d.py`

```python
def suma_rectangulo(p, f1, c1, f2, c2):
    total = p[f2 + 1][c2 + 1] - p[f1][c2 + 1] - p[f2 + 1][c1] + p[f1][c1]
    return total
```

El rectángulo de hace un momento

Filas 1 a 2, columnas 0 a 1:
$P[3][2] - P[1][2] - P[3][0] + P[1][0] = 27 - 3 - 0 + 0 = 24.$

Las dos trampas

Si se olvida sumar de vuelta la esquina, el resultado se queda corto por la parte que se restó dos veces. Y los índices van corridos en uno: la celda (i, j) de la matriz vive en $P[i + 1][j + 1]$.

UVa 108: Maximum Sum

Enunciado, en corto

Una matriz de $N \times N$ enteros, positivos y negativos. Se pide la mayor suma de un subrectángulo: un bloque de filas y columnas seguidas, de 1×1 en adelante.

Límites

$N \leq 100$; cada valor en $[-127, 127]$. Tras N vienen los N^2 números fila por fila, separados por espacios y saltos de línea en cualquier reparto.

Direcciones

Enunciado: `https://onlinejudge.org/external/1/108.pdf`

Envío: `https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=25&page=submit_problem&problemid=44`

¿Entendimos el problema?

Muestra, tal como viene

```
4
0 -2 -7 0 9 2 -6          2
-4 1 -4 1 -1
8 0 -2
```

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

A mano

Sale un número: el bloque marcado, filas 1 a 3 y columnas 0 y 1, suma $9 + 2 - 4 + 1 - 1 + 8 = 15$.

Pregunta

La segunda línea trae 8 números y la última 3. ¿Qué dice eso de cómo leer?

Respuesta

Que se leen N^2 números por palabras; la celda (i, j) es el número $i \cdot N + j$.

Primera idea: todos los rectángulos

¿Cuántos hay?

Un rectángulo es un par de filas $f_1 \leq f_2$ y un par de columnas $c_1 \leq c_2$: $(N(N+1)/2)^2$. Con $N = 100$, $5050^2 \approx 2{,}6 \cdot 10^7$.

Sumando cada uno celda por celda

Cuatro ciclos para elegirlo y dos más para sumarlo: $O(N^6)$. Contando celdas, son $(N(N+1)(N+2)/6)^2 \approx 2{,}9 \cdot 10^{10}$ sumas. No pasa.

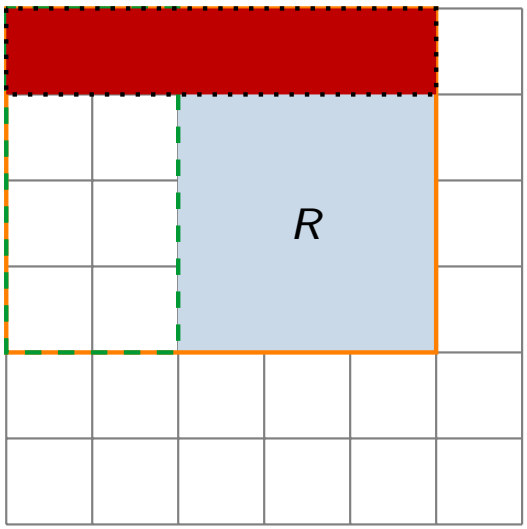
Pregunta

La suma de un tramo sale en $O(1)$ con acumulados. ¿Hay acumulados de un rectángulo?

Acumulados en dos dimensiones

Respuesta

Sí. $P[i][j]$ es la suma del rectángulo que va de la esquina $(0, 0)$ hasta la celda $(i-1, j-1)$, con P en ceros en la fila 0 y la columna 0. Un rectángulo cualquiera sale de cuatro esquinas:



$$R = P[f_2{+}1][c_2{+}1] - P[f_1][c_2{+}1] - P[f_2{+}1][c_1] + P[f_1][c_1]$$

La esquina de arriba a la izquierda se resta dos veces, por eso vuelve a sumarse.

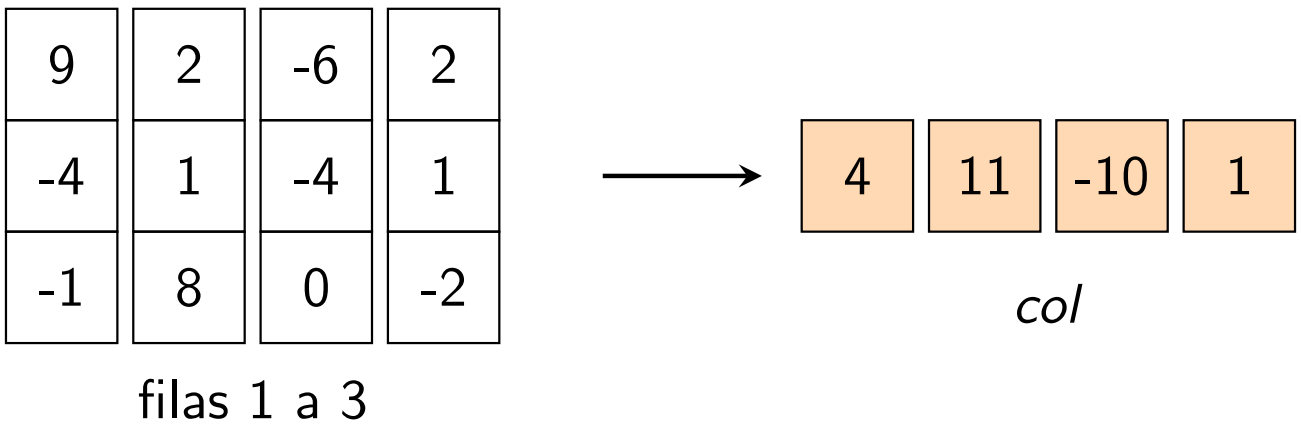
Costo

P se llena en $\Theta(N^2)$; cada rectángulo cuesta $O(1)$; todos, $O(N^4)$: unas $2{,}6 \cdot 10^7$ consultas, mil veces menos que antes.

¿Ya resolvimos esto?

Pregunta

Fije las filas `arriba` y `abajo`. Cada rectángulo entre ellas es un tramo de columnas. Si cada columna se cambia por su suma entre esas filas, ¿qué problema queda?



¿Ya resolvimos esto?

Respuesta

El del mejor tramo: el de suma máxima de *col*. Con $[4, 11, -10, 1]$ es $4 + 11 = 15$, el mismo rectángulo de la muestra.

El algoritmo

1. Para cada `arriba`, poner *col* en ceros.
2. Para cada `abajo` desde `arriba`: sumar la fila `abajo` a *col*, como en el repaso, y pasar *col* por el recorrido de Kadane.
3. Quedarse con el mayor resultado.

Costo

$N(N+1)/2$ pares de filas, y por cada uno $\Theta(N)$ para actualizar *col* y $\Theta(N)$ para el recorrido: $O(N^3)$, unos 10^6 pasos con $N = 100$.

Traza con la muestra, *arriba* $= 1$

Kadane sobre *col*, sin desempates

Aquí solo importa la suma, y puede ser negativa: `mejor` arranca en *col*[0], no en 0.

`abajo`	*col*	mejor tramo de *col*
1	$[9, 2, -6, 2]$	$9 + 2 = 11$
2	$[5, 3, -10, 3]$	?
3	$[4, 11, -10, 1]$	?

Traza con la muestra, *arriba* $= 1$

Kadane sobre *col*, sin desempates

Aquí solo importa la suma, y puede ser negativa: `mejor` arranca en *col*[0], no en 0.

`abajo`	*col*	mejor tramo de *col*
1	$[9, 2, -6, 2]$	$9 + 2 = 11$
2	$[5, 3, -10, 3]$	$5 + 3 = 8$
3	$[4, 11, -10, 1]$	$4 + 11 = 15$

Resultado

El mejor de los diez pares de filas es 15, con `arriba` $= 1$ y `abajo` $= 3$.

Python: maximum-sum.py

El tramo de suma máxima

```
def mejor_tramo(v):
    mejor = v[0]
    suma = 0
    for x in v:
        suma = suma + x
        if suma > mejor:
            mejor = suma
        if suma < 0:
            suma = 0
    return mejor
```

Python: `maximum-sum.py`

Fijar dos filas

```
def mayor_rectangulo(m):
    n = len(m)
    mejor = m[0][0]
    for arriba in range(n):
        col = [0] * n
        for abajo in range(arriba, n):
            for c in range(n):
                col[c] = col[c] + m[abajo][c]
            s = mejor_tramo(col)
            if s > mejor:
                mejor = s
    return mejor
```

Python: `maximum-sum.py`

La lectura por palabras

```
def main():
    datos = sys.stdin.read().split()
    n = int(datos[0])
    m = []
    for i in range(n):
        fila = []
        for j in range(n):
            fila.append(int(datos[1 + i * n + j]))
        m.append(fila)
    print(mayor_rectangulo(m))


main()
```

C++: maximum-sum.cpp

El tramo de suma máxima

```cpp
int mejor_tramo(const vector<int>& v) {
    int mejor = v[0];
    int suma = 0;
    for (int c = 0; c < (int) v.size(); c++) {
        suma = suma + v[c];
        if (suma > mejor) {
            mejor = suma;
        }
        if (suma < 0) {
            suma = 0;
        }
    }
    return mejor;
}
```

C++: maximum-sum.cpp

Fijar dos filas

```cpp
int mayor_rectangulo(const vector<vector<int>>& m) {
    int n = m.size();
    int mejor = m[0][0];
    for (int arriba = 0; arriba < n; arriba++) {
        vector<int> col(n, 0);
        for (int abajo = arriba; abajo < n; abajo++) {
            for (int c = 0; c < n; c++) {
                col[c] = col[c] + m[abajo][c];
            }
            int s = mejor_tramo(col);
            if (s > mejor) {
                mejor = s;
            }
        }
    }
    return mejor;
}
```

C++: maximum-sum.cpp

La lectura

```cpp
int main() {
    int n;
    scanf("%d", &n);
    vector<vector<int>> m(n, vector<int>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &m[i][j]);
        }
    }
    printf("%d\n", mayor_rectangulo(m));
    return 0;
}
```

¿Cabe en int?

La mayor suma posible es $127 \cdot 100^2 = 1\,270\,000$.

Trampas

Todo negativo

Con $N = 1$ y la celda -5, la respuesta es -5: el rectángulo tiene al menos una celda. Si `mejor` arranca en 0, sale 0.

Leer por líneas

La muestra misma reparte la matriz en líneas de 8, 5 y 3 números. Un programa que lee N números por línea falla con la muestra.

`col` sin reiniciar

col vuelve a ceros cada vez que cambia `arriba`; si no, arrastra filas de arriba que ya no están en el rectángulo.

Variante para pensar

Una matriz que no es cuadrada

La matriz tiene $f = 10$ filas y $c = 10\,000$ columnas. ¿Qué conviene fijar: dos filas o dos columnas?

Pista

Fijar dos filas cuesta $O(f^2 c)$ y fijar dos columnas, $O(c^2 f)$. Ponga los números.

Contenido

1 La entrada y lo que cuesta cada operación

2 Un recorrido en vez de dos

3 El mejor tramo de un arreglo

4 Rectángulos en una matriz

5 Tarea del bloque

Para el bloque 2

Cómo se trabajan

Los tres se resuelven, se envían al juez y se revisan al comienzo del bloque siguiente. Van en orden de dificultad.

Python en UVa

El juez corre una versión antigua de Python 3: la salida se arma con `print` y con `%`, como en las soluciones de hoy.

Base. UVa 100: The 3n + 1 problem

Direcciones

Enunciado: `https://onlinejudge.org/external/1/100.pdf`

Envío: `https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=25&page=submit_problem&problemid=36`

Cómo atacarlo

Un ciclo aplica la regla hasta llegar a 1 y cuenta los números que imprime; el recorrido del rango se queda con el máximo. La longitud cuenta el 1, así que la de 22 es 16. La entrada se lee hasta fin de archivo.

La trampa

i puede llegar mayor que j: el recorrido va del menor al mayor, pero la salida repite i y j en el orden en que llegaron. Los valores intermedios pasan de 10 000, así que no sirven como posición de un arreglo del tamaño de la entrada.

Intermedio. UVa 10189: Minesweeper

Direcciones

Enunciado: `https://onlinejudge.org/external/101/10189.pdf`

Envío: `https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=25&page=submit_problem&problemid=1130`

Cómo atacarlo

Por cada celda sin mina se cuentan las minas de sus ocho vecinas, con los dos desplazamientos recorriendo $-1, 0, 1$ y la comprobación de borde. El primer número son las filas y el segundo las columnas.

La trampa

Está en la salida: la línea en blanco va entre campos y no después del último, y las minas se conservan tal como llegaron.

Exigente. UVa 10920: Spiral Tap

Direcciones

Enunciado: `https://onlinejudge.org/external/109/10920.pdf`

Envío: `https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=25&page=submit_problem&problemid=1861`

Cómo atacarlo

Aquí la matriz no se puede construir: con un lado de hasta 100 000 son 10^{10} celdas. Se trabaja por anillos, cada uno con su rango de posiciones, y adentro del anillo la posición se convierte en fila y columna con aritmética.

La trampa

El tamaño: la posición llega hasta el lado al cuadrado y se pasa de 32 bits. La fila 1 es la de abajo y el centro de la espiral es la posición 1.

Para cerrar el bloque

El cuadrado de suma máxima

En una matriz de $N \times N$, encuentre el subcuadrado ($k \times k$, con cualquier k de 1 a N) de mayor suma. Escríbalo en Python y en C++ en $O(N^3)$, y pruébelo con la muestra de UVa 108.