Push_Swap: un algoritmo eficiente de ordenación posicional

Por Mia Combeau
En 42 proyectos escolares 24 de junio de 2022
24 Min leer
Añadir comentario P

Push_swap es un proyecto molesto para muchos estudiantes de 42 años. El objetivo es crear un programa en C que imprima la serie de instrucciones más optimizada para ordenar una lista aleatoria de enteros. Para este proyecto, necesitaremos implementar un algoritmo de ordenación eficiente que respete las restricciones que la asignatura le impone.

El algoritmo que construiremos aquí es una de las muchas soluciones a este problema. Se basa en un sistema de indexación y posicionamiento que lo hace muy eficaz y accesible.

No se trata de un tutorial paso a paso y no habrá soluciones prefabricadas. Se trata de una guía para explorar los problemas y conceptos que subyacen al tema, un posible enfoque que podríamos adoptar y algunos consejos para probar nuestro código.

Índice

Las reglas de Push_Swap

El tema de Push_swap establece reglas bastante estrictas. Nuestro programa debe recibir una serie de enteros aleatorios positivos y negativos como argumentos. Debe imprimir "error" si hay algún número duplicado o si uno de ellos excede los límites de un entero.

Si la serie de números no tiene errores, nuestro programa debe imprimir una serie de instrucciones, cada una en una nueva línea, para ordenar los números en orden ascendente.

Se nos permiten dos pilas a efectos de clasificación, la pila A y la pila B:

- La pila A se llena inicialmente con todos los números a ordenar. El primer número de la pila es el superior.
- La pila B empieza vacía.

Lista de posibles acciones de Push_Swap

Tendremos que hacer malabarismos entre estas dos pilas A y B para ordenar los números de la pila A en orden ascendente, es decir, del más pequeño arriba al más grande abajo. Para ello, tenemos un número limitado de acciones posibles: empujar, intercambiar, rotar y rotar a la inversa.

Empuje

`pa` (empujar A): Toma el primer elemento de la parte superior de la pila B y lo coloca en la parte superior de la pila
A. No hacer nada si B está vacío. Por ejemplo:

```
A : 1 3 4
B : 5 9
pa
A : 5 1 3 4
B : 9
```

Lenguaje del código: plaintext (texto sin formato)

`pb` (empujar B): Toma el primer elemento de la parte superior de A y lo coloca en la parte superior de B. No hace nada si A está vacío. Por ejemplo:

```
A : 0 9 2
B : 1 4
pb
A : 9 2
B : 0 1 4
```

Lenguaje del código: plaintext (texto sin formato)

Intercambiar

`sa` (intercambiar A): Intercambia los 2 primeros elementos en la parte superior de la pila A. No hace nada si sólo hay uno o ningún elemento. Por ejemplo:

```
A : 8 3 9
sa
A : 3 8 9
```

Lenguaje del código: plaintext (texto sin formato)

`sb` (intercambiar B): Intercambia los 2 primeros elementos de la parte superior de la pila B. No hace nada si sólo hay uno o ningún elemento. Por ejemplo:

```
B : 6 5 7
sb
B : 5 6 7
```

Lenguaje del código: plaintext (texto sin formato)

`ss`: `sa` y `sb` al mismo tiempo. Por ejemplo:

```
A : 2 1 3
B : 5 0
ss
A : 1 2 3
B : 0 5
```

Lenguaje del código: plaintext (texto sin formato)

Gire

`ra` (girar A): Desplaza todos los elementos de la pila A hacia arriba en 1. El primer elemento pasa a ser el último. Por ejemplo:

```
A : 9 2 5 8
ra
A : 2 5 8 9
```

Lenguaje del código: plaintext (texto sin formato)

`rb` (girar B): Desplaza todos los elementos de la pila B hacia arriba en 1. El primer elemento pasa a ser el último. Por ejemplo:

```
B : 7 3 4 6
rb
B : 3 4 6 7
```

Lenguaje del código: plaintext (texto sin formato)

`rr`: `ra` y `rb` al mismo tiempo. Por ejemplo:

```
A : 8 0 1 2 3
B : 9 5 6
rr
A : 0 1 2 3 8
B : 5 6 9
```

Lenguaje del código: plaintext (texto sin formato)

Giro inverso

`rra` (rotación inversa A): Desplaza todos los elementos de la pila A hacia abajo en 1. El último elemento se convierte en el primero. Por ejemplo:

```
A : 1 2 4 9 0
rra
A : 0 1 2 4 9
```

Lenguaje del código: plaintext (texto sin formato)

`rrb` (rotación inversa B): Desplaza todos los elementos de la pila B hacia abajo en 1. El último elemento se convierte en el primero. Por ejemplo:

```
B : 8 3 4 6 1
rrb
B : 1 8 3 4 6
```

Lenguaje del código: plaintext (texto sin formato)

`rrr` : `rra` y `rrb` `al mismo tiempo`. Por ejemplo:

```
A : 7 6 4 9 2
B : 8 5 3 1
rrr
A : 2 7 6 4 9
B : 1 8 5 3
```

Lenguaje del código: plaintext (texto sin formato)

Sistema de clasificación de Push_Swap

Durante la evaluación del proyecto, nuestra nota depende de lo eficientes que sean las acciones que genera nuestro programa push_swap para ordenar los números. En el momento de escribir esto, esta es la escala de calificación:

- Ordenar 3 valores: no más de 3 acciones.
- Clasificación de 5 valores: no más de 12 acciones.
- Clasificación de 100 valores: puntuación de 1 a 5 puntos en función del número de acciones:
    - 5 puntos por menos de 700 acciones
    - 4 puntos por menos de 900 acciones
    - 3 puntos por menos de 1.100 acciones
    - 2 puntos por menos de 1.300 acciones
    - 1 punto por menos de 1.500 acciones
- Clasificación de 500 valores: puntuación de 1 a 5 puntos en función del número de acciones:
    - 5 puntos por menos de 5.500 acciones
    - 4 puntos por menos de 7.000 acciones
    - 3 puntos por menos de 8.500 acciones
    - 2 puntos por menos de 10.000 acciones
    - 1 punto por menos de 11.500 acciones

Para validar este proyecto, necesitamos una nota de al menos el 80%.

Varios métodos posibles para Push_Swap

Lo primero que hay que tener en cuenta con push_swap es que el programa en sí no necesita estar extremadamente optimizado, sólo su salida. Esto significa que bien podríamos ordenar nuestra pila de números múltiples veces entre bastidores para elegir la acción más eficiente a imprimir.

A pesar de las restricciones impuestas en el tema push_swap, existen multitud de formas de diseñar un algoritmo de ordenación. Uno de los métodos más populares es la ordenación radix, que utiliza operadores bitshifting y bitwise. Leo Fu lo describe aquí. Otro método, descrito aquí por Jamie Dawson, consiste en dividir los números en trozos.

En este artículo, probaremos un algoritmo diferente muy optimizado que ordena por índice y posición.

Configuración de un algoritmo de ordenación posicional para Push_Swap

Este algoritmo es muy eficiente: sus resultados están muy por debajo de los límites que impone la evaluación push_swap. Implementado correctamente, se garantiza una nota del 100%. Además, es relativamente fácil de entender e implementar.

Configuración de listas enlazadas

Utilizaremos listas enlazadas para implementar este algoritmo. Cada elemento de una lista contendrá varias variables:

```
typedef struct s_stack
{
	int			valor;
	int
```

```
                            índice
    ;
    int                     pos;
    int
                            posición_ob
    jetivo;
    int
                            coste_a
    ;
    int
                            coste_b
    ;
    struct s_stack *next;
}   t_stack;
```

Lenguaje de código: C++ (cpp)

- valor: el entero que debemos ordenar,
- índice: su índice en la lista de todos los valores que deben ordenarse,
- pos: su posición actual en la pila,
- target_pos: para los elementos de la pila B, la posición de destino en la pila A donde debería estar,
- coste_a: cuántas acciones costaría rotar la pila A para que el elemento en la posición objetivo llegue a la parte superior de la pila A,
- cost_b: cuántas acciones costaría girar la pila B para que este elemento llegue a la parte superior de la pila B,
- next: puntero al siguiente elemento de la lista.

La importancia de todas estas variables se irá aclarando a lo largo de la descripción del algoritmo.

Asignación de índices

Una vez que hemos llenado nuestra pila A con los valores a ordenar, tenemos que asignar a cada elemento un índice, del menor al mayor. Por razones prácticas, el índice empezará en 1 y terminará en el número total de valores a ordenar. Por ejemplo, supongamos que tenemos un
lista de 10 números para ordenar, asignaremos un índice a cada uno así:

Valor	índice
1900	9

42	8
18	7
-146	2
-30	3
2 147 483 647	10
3	6
0	5
-2 147 483 648	1
-2	4

Atribuyendo los índices de esta manera será mucho más fácil saber el orden en el que deben estar los números aleatorios.

Elegir un algoritmo de ordenación en función del número de valores a ordenar

La mayoría de los demás algoritmos push_swap tienen métodos de ordenación distintos para 5, 100 y 500 números. Este no es el caso aquí. Lo que funciona para 500 números también funciona para 5.
Sin embargo, todavía necesitamos un algoritmo diferente para una ordenación de 3 números.

Pero antes de elegir entre dos métodos de ordenación, es una buena idea comprobar si la pila A ya está ordenada, por si acaso. Si lo está, el trabajo de push_swap termina aquí.

Sin embargo, si la pila A no está ordenada, tendremos que proceder de forma diferente en función del número de valores que tengamos que ordenar:

- 2 valores: todo lo que tenemos que hacer es `sa`.
- 3 valores: saltar al algoritmo de 3 números
- > 3 valores: saltar al algoritmo de más de tres números

Algoritmo de ordenación para 3 números

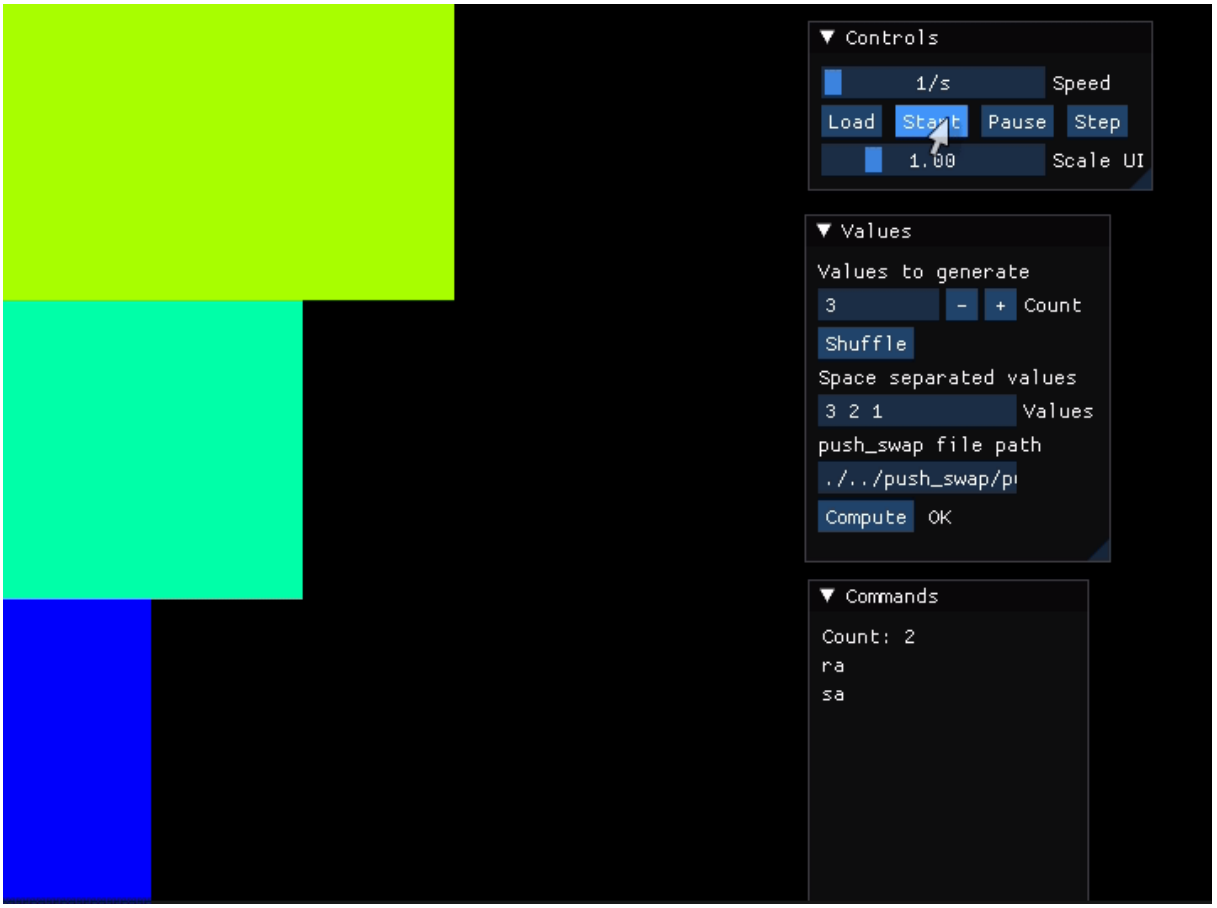
Según la escala de clasificación, deberíamos ser capaces de clasificar 3 números con 3 o menos acciones. Aquí, nunca usaremos más de dos. Para 3 valores, hay seis casos posibles:

```
Caso      acciones
1 2 3=>         ninguna acción
1 3             2-> rra -> 2 1 3 -> sa->  1 2 3 => 2 acciones
2 1             3->    sa-> 1 2 3 => 1 acción
2 3             1-> rra -> 1 2 3 => 1 acción
3 1             2->    ra-> 1 2 3 => 1 acción
3 2             1->    ra-> 2 1 3 -> sa->     1 2 3 => 2 acciones
```

Lenguaje del código: plaintext (texto sin formato)

Nuestro pequeño algoritmo de ordenación de 3 números sólo necesita elegir una de tres acciones: `ra`, `rra` y `sa`, dependiendo de la posición del número mayor. Podemos resumir el sencillo algoritmo de esta manera:

- Si el índice del primer número es mayor, haz `ra`,
- En caso contrario, si el índice del segundo número es mayor, haz `rra`,
- Entonces, si el índice del primer número es mayor que el índice del segundo número, haz `sa`.



Visualización de una ordenación de tres números con el visualizador de o-reo.

Aquí no usamos la pila B en absoluto: ¡un `pb` para empujar un número de la pila A a la pila B, seguido de un `pa` para enviarlo de vuelta a la pila A sería un desperdicio de acciones!

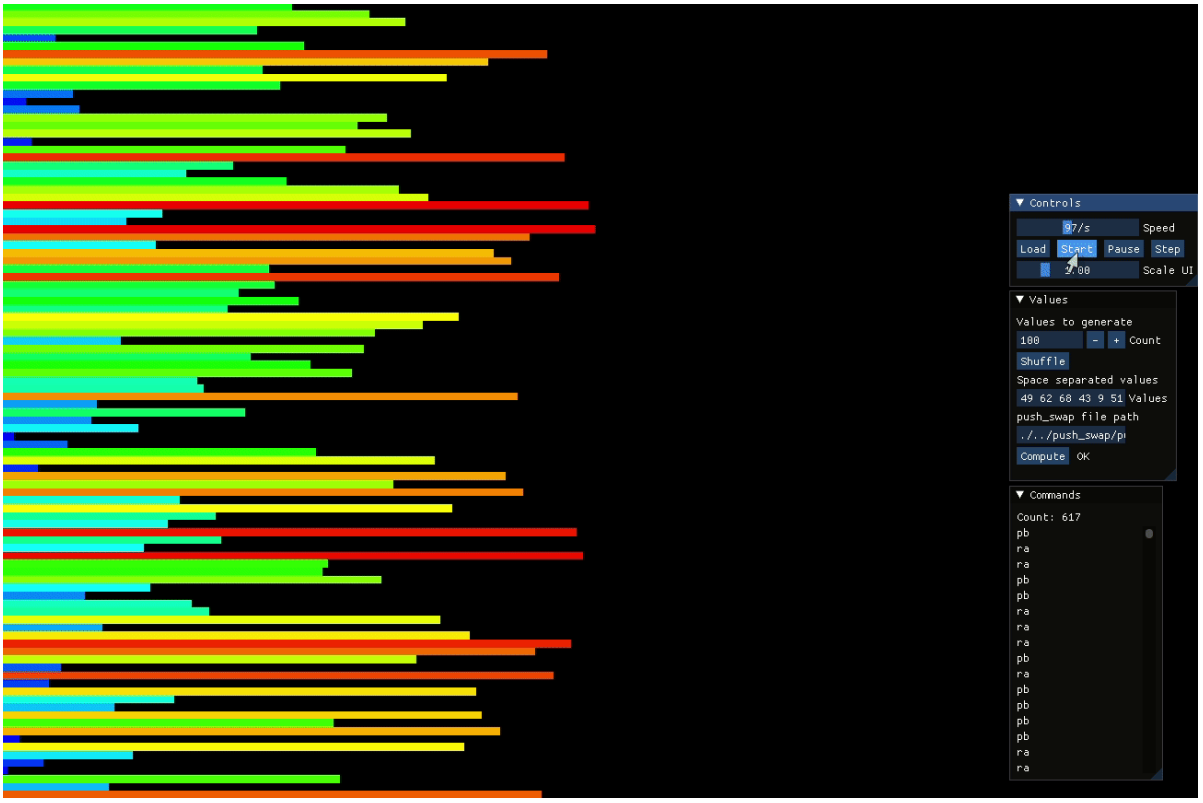
Algoritmo de ordenación para más de 3 números

Para ordenar una serie de números más grande, necesitaremos calcular más acciones. Para decidir qué movimiento hacer a continuación para ordenar cada valor, necesitamos calcular la posición de cada elemento en su pila. Luego podemos comparar el número de acciones que harían falta para colocar cada valor en la parte superior de su pila. Sólo entonces podremos elegir la serie de acciones más barata.

Nuestro algoritmo de ordenación tendrá 8 pasos, siempre, claro está, que la pila A no esté ya ordenada:

1. `pb` todos los elementos de la pila A, excepto tres.
2. Ordena los 3 números que quedan en la pila A.
3. Bucle mientras haya elementos en la pila B:
    1. Encuentra la posición actual de cada elemento en la pila A y B.
    2. Calcular la posición objetivo en la pila A donde debe estar cada elemento de la pila B.
    3. Calcula el número de acciones (el coste) para colocar cada elemento de la pila B en su posición de destino en la pila A y elige el elemento más barato de mover.
    4. Ejecuta la secuencia de acciones necesarias para mover el elemento de la pila B a la pila A.
4. Si la pila A no está ordenada, elige entre `ra` y `rra` para girarla en sentido ascendente

    orden.



Visualización de la ordenación de 100 números con el visualizador de o-reo.
En esta visualización, podemos ver la pila A a la izquierda y la pila B a la derecha. Los bloques de color representan los números aleatorios que debemos ordenar. Empezamos empujando todos los valores menos tres a la pila B. A medida que push_swap da sus instrucciones, podemos ver los valores de la pila B siendo insertados correctamente de nuevo en la pila A. Al final, la pila A debe rotar para estar en orden ascendente.

Paso 1: Enviar todo a la pila B

El primer paso es enviar todos los elementos de la pila A a la pila B, excepto 3. Mantenemos tres elementos en la pila A para evitar acciones adicionales. Después de todo, ya tenemos un algoritmo sencillo que podemos utilizar para ordenar 3 números directamente en la pila A.

Sin embargo, empujar todos nuestros elementos a la pila B no significa que no podamos hacerlo de una forma algo ordenada. De hecho, gracias a nuestro sistema de indexación, ya podemos hacer una burda ordenación en 2 pasos.

En primer lugar, podemos empujar todos los valores más pequeños. Así, si un elemento tiene un índice menor que el índice medio de todos los elementos (el número total de valores a ordenar dividido

por 2), podemos empujarlo a la pila B. De lo contrario, giramos A. Después de eso, podemos libremente `pb`
el resto de los elementos excepto los tres últimos que permanecerán en la pila A.

De esta forma, la pila B ya estará vagamente ordenada, lo que reducirá el número de acciones que tendremos que hacer más tarde.

Paso 2: Ordenar los 3 números que quedan en la pila A

Lo más probable es que los tres elementos que guardamos en la pila A necesiten ser ordenados. Afortunadamente, tenemos nuestro pequeño algoritmo de ordenación de 3 números para ayudarnos.

Con estos tres valores en orden, todo lo que tenemos que hacer es insertar cada elemento de la pila B en la posición correcta de la pila A. Pero encontrar la secuencia de acciones más eficiente es un poco más complicado que eso. Veamos cómo podemos conseguirlo.

Paso 3: Cálculo de posiciones

Un paso clave en nuestro algoritmo push_swap es encontrar la posición de cada elemento en su pila, así como la posición de destino en la pila A a la que debe ir un elemento de la pila B. Con esta información posicional, podremos calcular la secuencia de acciones más barata y elegir qué elemento mover primero.

Encontrar la posición de cada elemento

La posición de un elemento es simplemente una representación de dónde se encuentra actualmente en su pila. Tomemos una lista aleatoria de cuatro valores:

```
valor:           3    0    9    1
índice:        [3]  [1]  [4]  [2]
posición: <0> <1> <2> <3>
```

Lenguaje del código: plaintext (texto sin formato)

Todo lo que tenemos que hacer para encontrar la posición de cada elemento es escanear la pila de arriba a abajo, y asignar a cada elemento una posición que incrementamos en cada iteración. Debemos encontrar las posiciones de cada elemento en ambas pilas de esta manera. Por supuesto

tendremos que actualizar estas posiciones a menudo, cada vez que queramos calcular la secuencia de acciones más eficiente.

Encontrar la posición de destino de cada elemento de la pila B

Para cada elemento de la pila B, necesitamos calcular una posición objetivo. La posición objetivo es donde un elemento de la pila B tiene que estar en la pila A. Más precisamente, es la posición del elemento de la pila A que debe estar en la parte superior de la pila A cuando empujamos este elemento de B a A.

Podemos localizar esta posición objetivo simplemente escaneando la pila A en busca del índice superior más cercano al del elemento de la pila B. La posición del elemento de la pila A con ese índice será la posición objetivo de nuestro elemento de la pila B.

Ejemplo de búsqueda de puestos objetivo

```
La pila A contiene
    valor:           8    0    1    3
    índice:         [6] [1] [2] [4]
    posición: <0> <1> <2> <3>

La pila B contiene
    valor:           2    6    9
    índice:         [3] [5] [7]
    posición: <0> <1> <2>
    pos. objetivo: (3) (0) (1)
```

Lenguaje del código: plaintext (texto sin formato)

Aquí, el elemento en la parte superior de la pila B tiene un valor de 2 y un índice de 3. Esto significa que es el tercer valor en orden ascendente de todos los números que necesitan ordenación (0, 1, 2...). Como este elemento está en la parte superior de la pila B, podríamos empujarlo a la parte superior de la pila A. Pero no estaría en el lugar correcto. Este elemento debe estar entre índices 2 y 4. Si el elemento con índice 4, actualmente en la tercera posición de la pila A estuviera en la parte superior de su pila, podríamos empujar con seguridad nuestro elemento de la pila B encima de él. Así que la posición de destino de este primer elemento de la pila B es la posición del elemento con índice 4 de la pila A, que es 3.

Si un elemento de la pila B tiene un índice superior a todos los elementos de la pila

A, como es el caso del último elemento de la pila B en nuestro ejemplo, tenemos que utilizar la posición del índice más pequeño como posición de destino. Esto se debe a que queremos que el índice más pequeño

de la pila A esté en la parte superior antes de que introduzcamos en ella nuestro índice más alto. Esto asegura que el valor mayor estará en la parte inferior de la pila A después de rotarla.

¿Por qué calcular así las posiciones?

Con estos valores posicionales, podremos determinar cuántas acciones serán necesarias para colocar cada pila en la posición correcta para cada elemento. Entonces podremos averiguar qué elemento de la pila B es más barato mover primero.

Obviamente, sólo necesitamos calcular las posiciones objetivo de los elementos de la pila B.
Los elementos de la pila A no necesitan una posición de destino puesto que (con suerte) ya están en la posición correcta.

Por supuesto, varios elementos de B tendrán la misma posición de destino, especialmente cuando la pila B tenga más elementos que la pila A. Su posición en B será el factor determinante, desde el punto de vista del coste.

Paso 4: Calcular el coste de la acción más barata

Una vez que hayamos determinado las posiciones y las posiciones objetivo de cada elemento en ambas pilas, podremos comparar sus costes en términos de número de acciones necesarias para colocar cada elemento en la parte superior de B y en el lugar correcto de la pila A.

Cálculo de los costes de las pilas A y B

Para cada elemento de la pila B, tenemos que calcular dos costes. El primero es el coste de llevar el elemento a la parte superior de B. El segundo cálculo de coste es el mismo, pero para la pila A, con el fin de determinar cuántas acciones necesitamos para mover el elemento de la posición objetivo hasta la parte superior de la pila A.

En ambos casos, debemos distinguir entre rotar (`ra` o `rb`) y rotar a la inversa (`rra` o `rrb`) para calcular correctamente los costes. Como recordatorio, la rotación mueve el elemento superior a la parte inferior de la pila, mientras que la rotación inversa hace lo contrario, llevando el elemento inferior a la parte superior

de la pila.

Distinción entre rotación y rotación inversa

Para determinar si necesitamos invertir la rotación en lugar de simplemente rotar, podemos medir el tamaño de la pila y dividirlo por la mitad. De esta forma, es fácil comprobar si la posición de nuestro elemento se encuentra en la mitad superior o inferior de la pila. Si la posición del elemento es menor que la posición media de la pila, se encuentra en la mitad superior, lo que significa que tenemos que hacer `rb`, de lo contrario, tendremos que ir con `rrb`.

Sería muy útil que el coste de la rotación inversa fuera negativo. De esta forma, durante la ejecución de la secuencia de acciones, podremos comprobar el signo del coste para saber si rotar o invertir la rotación. Y si los costes de A y B son del mismo signo, podemos ahorrar aún más en acciones eligiendo hacer `rr` o `rrr`, que rotarán o invertirán la rotación de ambas pilas simultáneamente.

Por último, podemos comparar el coste de mover cada elemento de la pila B a la pila A para elegir el más barato. Esta elección es sencilla, todo lo que tenemos que hacer es sumar los costes de la pila A y de la pila B para cada elemento de la pila B. Más bien, tenemos que sumar los valores absolutos de los costes para que la suma siga siendo cierta incluso con los valores de coste negativos rotativos inversos.

¿Por qué no usar Swap?

Este algoritmo se basa en calcular las posiciones de cada elemento de una pila para elegir cuál mover primero. Esto implica utilizar principalmente acciones de rotación y rotación inversa para colocar cada pila en su posición. No utilizaremos intercambios (`sa`, `sb` o `ss`); estas acciones no están optimizadas porque no cambian las posiciones de todos los elementos de la pila. Después de un sb `+ pa`, `por` ejemplo, estamos de vuelta en el mismo elemento que antes en la parte superior de nuestra pila, que no mueve nuestro push_swap hacia adelante.

Ejemplos de cálculo de costes

Tomemos nuestro ejemplo anterior de los cálculos de posición y modifiquémoslo un poco para ilustrar el cálculo de costes.

```
La pila A contiene
     valor:          8    0    1    3    4
```

```
    índice:         [7]  [1]  [2]  [4]  [5]
    posición: <0>  <1>  <2>  <3>  <4>

La pila B contiene
    valor:           2    6    9
    índice:         [3] [6] [8]
    posición: <0> <1> <2>
    pos. objetivo: (3) (0) (1)
```

Lenguaje del código: plaintext (texto sin formato)

El primer elemento de la pila B quiere colocarse antes que el elemento en 3ª posición de la pila A. Como este elemento ya está en la parte superior de B, su coste para B será
0. Sin embargo, habrá un coste para mover su objetivo desde la 3ª posición a la parte superior de la pila A. Como el elemento objetivo está en la mitad inferior de la pila A, necesitamos dos `rra` para llevarlo a la parte superior. Esto significa que su coste para A es -2. El coste total de mover el primer elemento de B es `(abs)0 + (abs)-2 = 2`.

El segundo elemento de la pila B tiene que estar en la posición 0 de la pila A. Tenemos que hacer un `rb` para que esté en la parte superior de la pila B, y la pila A ya está en la posición correcta. El coste de mover este elemento en B es 1 y el coste de mover la pila A a su posición es 0. El coste total de llevar este elemento al lugar correcto es `(abs)1 + (abs)0 = 1`.

Finalmente, tenemos un último elemento en la pila B. Como está en la mitad inferior de su pila, necesitamos un `rrb para llevarlo a la parte superior`. Así que su coste B será -1. Su posición objetivo es el segundo elemento de la pila A. Necesitaremos hacer un `ra para llevarlo a la parte superior` de la pila A,
lo que significa que el coste A es 1. El coste total de mover este último elemento B es `(abs)-1 + (abs)1 = 2`.

En este caso, está claro que la mejor opción que podría tomar ahora mismo nuestro algoritmo push_swap es mover el segundo elemento de la pila B.

Paso 5: Ejecutar la secuencia de acciones elegida

Una vez que hemos encontrado el elemento más barato de la pila B, tenemos que

ejecutar la secuencia de acción que lo empujará a la parte superior de la pila A.

Como hemos guardado los costes A y B de este elemento en dos variables separadas, podemos usarlas para calcular cuántas acciones necesitamos ejecutar en cada pila. Además, hemos hecho que el coste de rotación inversa sea negativo, a diferencia del coste de rotación. Así podemos distinguir fácilmente entre estas dos acciones para ejecutarlas perfectamente. También podemos ahorrar en acciones haciendo una `rrr` en lugar de una `rra` y una `rrb` por separado, por ejemplo.

Tomemos un elemento con un coste A de 2 y un coste B de 3, por ejemplo. Ambos son positivos, así que sabemos que tenemos que rotar y no invertir la rotación. Además, podemos ver claramente que podemos hacer simplemente dos `rr` y una `rb`. Para un elemento con un coste A de -4 y un coste B de 1, tendremos que hacer cuatro `rra` y un `rb`. Sin embargo, si el coste A es -3 y el coste B es -1, podemos hacer una `rrr` y dos `rra`.

Con los elementos de la pila A y la pila B en posición, podemos finalmente `pa`. Y, mientras siga habiendo elementos en B, volvemos al paso 3 para reevaluar todas las posiciones y todos los costes para la siguiente secuencia de acciones.

Paso 6: Girar la pila A a la posición correcta

¡Cuando no hay más elementos en la pila B, eso no significa que la pila A esté totalmente en orden ! Eso depende del último elemento que se haya colocado de la pila B. No podemos olvidar ajustar la pila A con `ra` o `rra` hasta que el elemento con el índice más pequeño esté arriba y el elemento con el índice más grande esté abajo.

Ya está. La pila A debería estar ahora correctamente ordenada en orden ascendente, y nuestro programa push_swap debería haber impreso el menor número de instrucciones para que así sea.

Consejos para probar Push_Swap

Antes incluso de implementar el algoritmo, es importante comprobar que la lista enlazada y que las funciones para cada acción (`pa`, `pb`, `sa`, `sb`, `ss`, `ra`, `rb`, `rr`, `rra`, `rrb` y `rrr`) funcionan correctamente. Para ello, necesitamos imprimir los valores o índices de todos los elementos de una pila en un bucle para comprobar los resultados.

También debemos comprobar que nuestro programa push_swap detecta correctamente las entradas erróneas. En particular, debemos estar atentos a los números duplicados, como 0, +0, -0, 00000 o incluso 1, +1, 00001 y 1.0. También debemos tener cuidado de comprobar que ningún número excede INT_MAX o INT_MIN.

Prueba de Push_Swap con el comprobador

42 nos proporciona un comprobador para ayudarnos a probar nuestro push_swap. Para hacer uso de él, tenemos que ejecutarlo con una lista de números en una variable shell, así:

```
ARG="1 -147 2 89 23 30"; ./push_swap $ARG | ./checker_linux $ARG
```

Lenguaje de código: Sesión Shell (shell)

El comprobador sólo imprime `OK` o `KO`, dependiendo de si la pila se ha ordenado correctamente o no. Para ver cuántas acciones se necesitaron para realizar la ordenación, podemos utilizar el comando wc:

```
ARG="1 -147 2 89 23 30"; ./push_swap $ARG | wc -l
```

Lenguaje de código: Sesión Shell (shell)

Prueba de Push_Swap con el comando Shuf

Eso era útil para pequeñas series de números, ¡pero está lejos de ser ideal para 100 o 500 números! Para probarlos, podemos utilizar el comando shuf, que baraja una serie de
enteros de forma aleatoria. Notemos, sin embargo, que cuando generamos una lista de números aleatorios con shuf, nunca habrá números negativos. Para generar 500 números entre 0 y 1000 con el comando shuf, podemos hacer esto:

```
ARG=$(shuf -i 0-1000 -n 500); ./push_swap $ARG | ./checker_linux $ARG
```

Lenguaje de código: PHP (php)

Gracias a este comando, podemos incluso añadir algunas reglas Makefile para realizar pruebas rápidas:

```
test3:          $(NOMBRE)
          $(eval ARG = $(shell shuf -i 0-50 -n 3))
          ./push_swap $(ARG) | ./checker_linux $(ARG)
          @echo -n "Instrucciones: "
```

```
		@./push_swap $(ARG) | wc -l
```

```
test5:          $(NOMBRE)
          $(eval ARG = $(shell shuf -i 0-50 -n 5))
          ./push_swap $(ARG) | ./checker_linux $(ARG)
          @echo -n "Instrucciones:
          "
          @./push_swap $(ARG) | wc -l

test100: $(NOMBRE)
          $(eval ARG = $(shell shuf -i 0-1000 -n 100))
          ./push_swap $(ARG) | ./checker_linux $(ARG)
          @echo -n "Instrucciones: "
          @./push_swap $(ARG) | wc -l

test500: $(NOMBRE)
          $(eval ARG = $(shell shuf -i 0-2000 -n 500))
          ./push_swap $(ARG) | ./checker_linux $(ARG)
          @echo -n "Instrucciones: "
          @./push_swap $(ARG) | wc -l
```

Lenguaje de código: Makefile (makefile)

Entonces todo lo que tenemos que hacer para ejecutar la prueba es:

```
hacer la prueba500
```

Lenguaje de código: Sesión Shell (shell)

Pruebas de Push_Swap con visualizadores y probadores

En este proyecto, es importante poder visualizar el funcionamiento interno de nuestro programa push_swap. Los estudiantes han creado visualizadores que no sólo son bonitos de ver, sino también muy informativos durante el proceso de depuración. El utilizado en este artículo, como ya se ha mencionado, es el de o-reo.

Como último recurso, podemos utilizar uno de los varios probadores creados por otros 42 estudiantes de la escuela. Una simple búsqueda en Google revela varios.

Código completo de Push_Swap

Mi código completo de push_swap está disponible aquí, en GitHub. Si eres un estudiante de 42, te animo a que uses los elementos anteriores para construir tu propio código a tu manera, antes de echar un vistazo al mío.

Deja un comentario si has encontrado otra solución u otros consejos para probar push_swap.

Muchas gracias a los ingeniosos estudiantes del 42 que me han echado una mano en este proyecto.