PUSH SWAP

**El "programa definitivo"**

Crear el "programa definitivo" es un reto, pero puedes acercarte usando una combinación de estrategias. Por ejemplo, podrías usar Radix Sort para listas grandes y otra estrategia (como la división y conquista) para listas más pequeñas. Determinar qué estrategia usar podría basarse en benchmarks para diferentes tamaños y disposiciones de listas.

Además, podrías usar técnicas de optimización para reducir el número de operaciones. Por ejemplo, si detectas que una serie de operaciones puede ser reemplazada por una operación más corta que tiene el mismo efecto, podrías hacer ese reemplazo.

**ESTRUCTURA**

1. **Estructura de Datos**: Usaría listas enlazadas porque permiten inserciones y eliminaciones rápidas en ambos extremos, que son operaciones cruciales para el problema push\_swap. Además, las listas enlazadas no tienen un tamaño fijo, lo que las hace flexibles.
2. **Algoritmo**:
   * Para listas pequeñas (hasta 5 elementos): Implementaría una serie de comprobaciones y operaciones manuales para manejar todos los posibles ordenamientos. Es eficiente y directo.
   * Para listas medianas (hasta ~100 elementos): Podría usar un algoritmo como la ordenación por inserción, adaptado a las operaciones disponibles de **push\_swap**.
   * Para listas más grandes: Consideraría un enfoque más sofisticado, como un algoritmo de división y conquista, similar a la ordenación por fusión, pero adaptado para trabajar con las operaciones de **push** y **swap**.
3. **Optimizaciones**:
   * **Precálculo**: Antes de comenzar con las operaciones reales, podría analizar la lista para obtener información útil, como encontrar el valor medio o determinar segmentos ya ordenados.
   * **Reducir el número de operaciones**: Una vez que tenga una solución que funcione, podría buscar formas de reducir el número de operaciones. Por ejemplo, si detecto que estoy realizando un **ra** seguido de un **rb**, podría consolidarlo en un solo comando **rr**.
   * **Algoritmos adaptativos**: Podría detectar si la lista ya está ordenada o inversamente ordenada y elegir un algoritmo específico para esas situaciones, como se discutió anteriormente.
4. **Modularidad**: Separaría mi solución en funciones distintas basadas en la funcionalidad. Por ejemplo, una función para manejar listas de tamaño 3, otra para listas de tamaño 5, y así sucesivamente. Esto facilita la prueba y depuración.
5. **Pruebas**: Asegurarme de probar la solución en una variedad de casos, desde listas ya ordenadas hasta inversamente ordenadas y todo lo demás.
6. **Errores y Límites**: Debido a que **push\_swap** a menudo tiene restricciones sobre el número máximo de operaciones permitidas para ciertos tamaños de lista, sería crucial implementar lógica para manejar estos límites y asegurarse de que el programa se mantenga dentro de ellos.
7. **Debugging**: Implementaría algún tipo de modo de depuración o de registro para que pueda ver qué operaciones se están realizando en cada paso. Esto sería esencial para la depuración y la optimización.

**¿Qué es la Complejidad Temporal?**

La complejidad temporal describe cuánto aumenta el tiempo de ejecución de un algoritmo a medida que aumenta el tamaño del input (por ejemplo, el número de elementos en una lista).

**Notación Big O (O grande)**

La "Notación Big O" es una forma de describir el peor caso de la complejidad temporal de un algoritmo. Indica cómo crece el tiempo de ejecución en relación al tamaño del input, pero en términos generales, sin entrar en constantes específicas.

Algunos ejemplos comunes:

1. **O(1)**: Tiempo constante. Sin importar cuán grande sea el input, el algoritmo tomará un tiempo constante. Un ejemplo es acceder a un elemento de un array por su índice.
2. **O(n)**: Tiempo lineal. El tiempo de ejecución crece linealmente con el tamaño del input. Por ejemplo, buscar un elemento en una lista no ordenada requiere, en el peor de los casos, mirar todos los elementos.
3. **O(n^2)**: Tiempo cuadrático. El tiempo de ejecución crece proporcionalmente al cuadrado del tamaño del input. Un buen ejemplo de esto es el algoritmo de ordenamiento Bubble Sort.
4. **O(log n)**: Tiempo logarítmico. El tiempo de ejecución crece logarítmicamente con el tamaño del input. Un ejemplo típico es una búsqueda binaria en una lista ordenada.
5. **O(n log n)**: Algunos algoritmos, como el MergeSort y el QuickSort, tienen esta complejidad. Es más eficiente que O(n^2) pero menos que O(n).

* [Complejidad computacional](https://es.wikipedia.org/wiki/Complejidad_computacional) (peor caso, caso promedio y mejor caso) en términos de *n*, el tamaño de la lista o arreglo. Para esto se usa el concepto de *orden* de una función y se usa la notación [O](https://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica)(*n*). El mejor comportamiento para ordenar (si no se aprovecha la estructura de las claves) es O(*n* log *n*). Los algoritmos más simples son cuadráticos, es decir O(*n*²). Los algoritmos que aprovechan la estructura de las claves de ordenamiento (p. ej. *bucket sort*) pueden ordenar en O(*kn*) donde *k* es el tamaño del espacio de claves. Como dicho tamaño es conocido *a priori*, se puede decir que estos algoritmos tienen un desempeño lineal, es decir O(*n*).

**PARA 5 RAMDOM NUMBERS:**

Para 5 números, la estrategia de mover los números más pequeños primero puede ser eficiente. Vamos a analizarlo:

Supón que tienes los números: **4 2 5 1 3**

1. Identificas el número más pequeño (**1**) y el segundo más pequeño (**2**). Los mueves a la pila B en ese orden, por lo que:
   * Pila A: **4 5 3**
   * Pila B: **2 1**
2. Ahora, ordenas la Pila A. Con los números restantes, una simple rotación o swap podría ser suficiente:
   * Pila A (después de ordenar): **3 4 5**
   * Pila B: **2 1**
3. Ahora mueves **1** y **2** de vuelta a la Pila A:
   * Pila A: **1 2 3 4 5**
   * Pila B: vacía

**100 RANDOM NUMBERS**

¡Claro! Te voy a resumir el proceso que has descrito:

1. **Objetivo Principal**: Ordenar 100 números aleatorios utilizando dos pilas, Stack A y Stack B.
2. **Método Inicial**: Usar el algoritmo de Inserción:
   * Buscar el número más pequeño en Stack A.
   * Mover ese número a la parte superior de Stack A.
   * Pasar ese número a Stack B.
   * Repetir hasta que Stack A esté vacío.
   * Finalmente, pasar todos los números de Stack B a Stack A para que estén ordenados del más pequeño al más grande.
3. **Optimización**: En lugar de ordenar 100 números a la vez, divide los números en 5 grupos (chunks) de 20 números cada uno.
4. **Proceso de Optimización**:
   * **Paso 1**: Busca si un número del Chunk 1 está en Stack A. Llámalo **hold\_first**.
   * **Paso 2**: Busca otro número diferente del Chunk 1 desde el final de Stack A. Llámalo **hold\_second**.
   * **Paso 3**: Determina cuál de los dos números, **hold\_first** o **hold\_second**, está más cerca de la parte superior de Stack A usando comandos **ra** y **rra**. Usando una lógica matemática basada en la mitad de la longitud de la lista, se decide cuál comando usar.
   * **Paso 4**: Asegúrate de que el número que vas a mover a Stack B se coloque en la posición correcta en comparación con los números ya presentes en Stack B.
   * **Paso 5**: Repite los pasos 1-4 hasta que todos los números del Chunk 1 ya no estén en Stack A.
   * **Paso 6**: Haz lo mismo para los demás chunks.
5. **Finalización**: Una vez que Stack A está vacío, toma el número más grande de Stack B, muévelo a la parte superior y pásalo a Stack A. Usa lógica similar al Paso 3 para determinar qué comando usar (**rb** o **rrb**). Repite hasta que Stack B esté vacío.

Al final, Stack A tendrá todos los números ordenados de menor a mayor.

**En código**

ft\_sort\_hundred:

Para hacerlo más visual, supongamos que la barra de chocolate tiene un total de 100 unidades de largo y que quieres dividirlo en 5 partes iguales. Cada segmento, o "chunk", tendría entonces un tamaño de **100 / 5 = 20** unidades.

Entonces, **chunk\_size\_value** sería 20 en este caso.

Si asignamos un índice **i** a cada segmento:

* Para el primer segmento (**i=0**):
  + El inicio del segmento sería **0 \* 20 = 0**.
  + El final sería **0 + 20 = 20**.
  + Así que el primer segmento va de 0 a 20.
* Para el segundo segmento (**i=1**):
  + El inicio sería **1 \* 20 = 20**.
  + El final sería **20 + 20 = 40**.
  + Así que el segundo segmento va de 20 a 40.
* Para el tercer segmento (**i=2**):
  + El inicio sería **2 \* 20 = 40**.
  + El final sería **40 + 20 = 60**.
  + Así que el tercer segmento va de 40 a 60.

Y así sucesivamente.

El valor **i \* chunk\_size\_value** es simplemente una manera de calcular dónde comienza cada segmento basado en su índice. Si multiplicas el índice del segmento (i) por el tamaño del segmento (**chunk\_size\_value**), obtienes el punto de inicio de ese segmento.

**Según gemartin:**

SIZE3 ///////////////////////////////

La función `size3` parece ser una función que maneja el caso específico en el que tienes 2 o 3 números en la lista. Se encarga de ordenar esta pequeña lista usando un conjunto mínimo de operaciones.

El algoritmo es directo. Dado que hay solo un pequeño número de permutaciones posibles con 2 o 3 elementos, el código simplemente verifica cada posible disposición y realiza las operaciones necesarias para ordenar la lista. Vamos a desglosarlo:

1. \*\*Si hay 2 números (d->argc == 2)\*\*:

- Si el primer número es mayor que el segundo, realiza un swap (`sa`).

2. \*\*Si hay 3 números\*\*:

- Revisa todas las permutaciones posibles y realiza las operaciones necesarias:

a) Si ya están en orden: No hace nada.

b) Si los primeros dos están en orden pero el tercero es menor que el segundo, realiza una rotación inversa (`rra`) seguida de un swap (`sa`).

c) Si el primer número es el mayor, realiza un swap (`sa`).

d) Si el primer número es el del medio y el tercero es el menor, realiza una rotación inversa (`rra`).

e) Si el primer número es el más pequeño y el tercero es el del medio, realiza una rotación (`ra`).

f) Si el primer número es el más grande y el tercero es el más pequeño, realiza una rotación (`ra`) seguida de un swap (`sa`).

Las funciones `sa`, `ra`, y `rra` son operaciones estándar en el problema `push\_swap`.

- `sa` hace un swap de los dos primeros elementos de la lista.

- `ra` rota todos los elementos de la lista hacia arriba (el primer elemento se convierte en el último).

- `rra` rota todos los elementos de la lista hacia abajo (el último elemento se convierte en el primero).

El enfoque de este algoritmo es muy eficiente para listas tan pequeñas porque explora todas las permutaciones posibles y actúa en consecuencia. No sería eficiente para listas más grandes debido a la explosión combinatoria de permutaciones posibles, pero para 2 o 3 elementos, es óptimo.

SIZE 5 ///////////////////////////

La estrategia aquí parece ser similar a la utilizada para **size3**, pero adaptada para manejar listas de 4 y 5 elementos. En lugar de manejar cada posible permutación de los números, el enfoque se basa en reconocer y gestionar ciertas condiciones o patrones que puedan surgir en una lista de 4 o 5 números.

Explicaré en términos generales cómo funciona:

1. **size4**: Esta función trata principalmente el caso de tener 4 números. Aquí, el algoritmo verifica dónde se encuentra el número más pequeño y realiza las operaciones necesarias para enviarlo al final de la lista **B** (**pb**). Una vez que ese número está en **B**, tiene una lista de 3 números en **A**, y puede aplicar la función **size3** para ordenarlos. Al final, regresa el número de **B** a **A** con **pa**.
2. **size5\_2**: Este es un ayudante para la función **size5** que maneja algunos patrones específicos. La estrategia sigue siendo similar a **size4**: identificar la ubicación del número más pequeño y aplicar las operaciones necesarias para enviarlo a **B**.
3. **size5**: Aquí, el algoritmo es nuevamente un juego de identificar patrones y aplicar las operaciones necesarias. Si hay 5 números, el algoritmo se centra en reconocer dónde está el número más pequeño y moverlo a **B**. Una vez que se ha manejado, se tiene una lista de 4 números, lo que permite usar la función **size4**. Después de que **size4** haya ordenado esos 4 números, el último paso es traer de vuelta el número que se movió a **B**.

SIZE100 /////////////////////////

La función **size100** es para manejar una lista de tamaño cercano a 100. Aquí, el enfoque es un poco diferente al de las listas más pequeñas. A diferencia de las listas de 3, 4 y 5 elementos, no es práctico o eficiente lidiar con cada posible permutación de 100 números. Por lo tanto, para listas más grandes, a menudo es mejor dividirlas en bloques o segmentos más pequeños que se pueden manejar por separado.

A continuación, se detalla una descomposición básica de la función:

1. **Inicialización de Variables**:
   * **lsp**: Probablemente la última posición en la lista.
   * **control**: Una variable de control (su uso específico no se especifica en este fragmento).
   * **percentage**: Parece estar relacionado con el progreso de la ordenación.
   * **cont**: Esta variable se usa para calcular el tamaño del bloque.
   * **size\_block**: Calcula el tamaño del bloque dividiendo el número total de elementos por **cont** y luego dividiendo por 2.
   * **argmax**: El máximo número de argumentos o tamaño de la lista.
2. **Funciones**:
   * **sortnum(d)**: Aunque el cuerpo de esta función no está incluido, el nombre sugiere que se encarga de ordenar números de alguna manera.
   * **changenum(d)**: De nuevo, el cuerpo de esta función no está incluido, pero probablemente se encargue de cambiar números basándose en ciertas condiciones.
   * **ordernums(d)** y **ordernums\_peq(d)**: Estas funciones probablemente se encargan de ordenar los números. La existencia de dos funciones diferentes sugiere que hay dos diferentes estrategias de ordenación dependiendo del tamaño de la lista, una para listas más grandes y otra para listas más pequeñas.