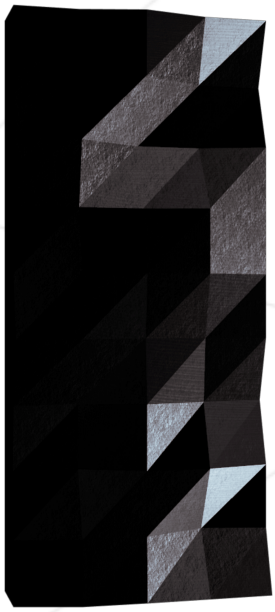

Push_swap

Porque Swap_push no suena tan natural

Resumen:
En este proyecto se deberán ordenar datos en un stack, con un conjunto limitado de instrucciones y utilizando el menor número posible de acciones. Para conseguirlo, se deben utilizar varios algoritmos y elegir la solución más apropiada (de entre muchas posibles) para conseguir ordenar los datos de la forma más optimizada.
Esto es un proyecto en grupo y debe ser realizado por dos personas.

Versión: 1.0

Índice general

Capítulo I

Prólogo

Érase una vez, en las misteriosas tierras de la **Informática**, un tal Donald Knuth[1] popularizó la notación *Big-O* para ayudarnos a hablar de cómo escalan los algoritmos. Big-O es básicamente una forma educada de decir: *"Tu código o será rápido para siempre... o se volverá tan lento que podrás hacerte un café, bebértelo y envejecer antes de que termine."*

Cuenta la leyenda que en los viejos tiempos los programadores no tenían Big-O. Simplemente ejecutaban su código y, si tardaba demasiado, le echaban la culpa al *hardware*. Entonces llegó Big-O y arruinó la fiesta demostrando matemáticamente que, a veces, tu algoritmo es simplemente **malo** — da igual cuántos hámsters metas en la rueda de la CPU.

¿Y por qué importa esto para `push_swap`? Porque aquí estás tú, con dos pilas y un conjunto de movimientos absurdamente limitados, intentando ordenar números más rápido que un insertion sort con sueño. Big-O te ayudará a enfrentar una realidad dura: una estrategia brillante de $\mathcal{O}(n \log n)$ siempre sobrevivirá a tu torpe $\mathcal{O}(n^2)$ cuando el tamaño de la entrada crece... a menos, claro, que decidas ignorar las matemáticas y ver cómo tu contador de operaciones despega hacia el infinito.

Así que cuando diseñes tus algoritmos, recuerda: Big-O no está para asustarte — está para evitar que te conviertas en la persona que escribe:

```
pb; pa; pb; pa; pb; pa; ...
```

10.000 veces seguidas y luego se pregunta por qué el checker le tiene tirria. **Ordena con cabeza, no con fuerza bruta.**

Big-O no es un tamaño de palomitas... pero si lo fuera, $\mathcal{O}(n \log n)$ seguiría siendo el punto ideal.

[1] Sí, el hombre de la barba legendaria y la paciencia infinita.

Capítulo II

Instrucciones generales

- El proyecto deberá estar escrito en C.
- El proyecto debe estar escrito siguiendo la Norma. Si tienes archivos o funciones adicionales, estas deberán estar incluidas en la verificación de la Norma y tendrás un `0` si hay algún error de norma en cualquiera de ellos.
- Las funciones no deben terminar de forma inesperada (segfault, bus error, double free, etc), excepto en el caso de comportamientos indefinidos. Si esto sucede, el proyecto será considerado no funcional y recibirás un `0` durante la evaluación.
- Toda la memoria asignada en la pila (heap) deberá liberarse adecuadamente cuando sea necesario. No se permitirán leaks de memoria.
- Si el enunciado lo requiere, se deberá entregar un `Makefile` que compilará tus archivos fuente a la salida requerida con las flags `-Wall`, `-Werror` y `-Wextra`. También se deberá utilizar cc y, por supuesto, el `Makefile` no debe hacer relink.
- El `Makefile` entregado debe contener al menos las normas `$(NAME)`, `all`, `clean`, `fclean` y `re`.
- Para entregar los bonus del proyecto se deberá incluir una regla `bonus` en el `Makefile`, en la que se añadirán todos los headers, librerías o funciones que estén prohibidas en la parte principal del proyecto. Los bonus deben estar en archivos distintos × `_bonus.{c/h}`. La parte obligatoria y los bonus se evalúan por separado.
- Si el proyecto permite el uso de la `libft`, se deberá copiar su fuente y sus `Makefile` asociados en un directorio `libft` con su correspondiente `Makefile`. El `Makefile` del proyecto debe compilar primero la librería utilizando su `Makefile`, y después compilar el proyecto.
- Es recomendable crear programas de prueba para el proyecto, aunque este trabajo **no será entregado ni evaluado**. Esto ofrece la oportunidad de verificar que el programa funciona correctamente durante las evaluaciones. Y sí, está permitido utilizar estas pruebas durante cualquier evaluación.
- Entrega el trabajo en el repositorio `Git` asignado. Solo el trabajo de tu repositorio `Git` será evaluado. Si el proyecto tiene que ser evaluado por Deepthought, la evaluación se realizará después de las evaluaciones personales. Si, durante la evaluación de Deepthought, se encuentra un error se, se interrumpirá su evaluación.

Capítulo III

Instrucciones sobre la IA

Contexto

Durante tu proceso de aprendizaje, la IA puede ayudarte con muchas tareas diferentes. Tómate el tiempo necesario para explorar las diversas capacidades de las herramientas de IA y cómo pueden apoyarte con tu trabajo. Sin embargo, siempre debes abordarlas con precaución y evaluar de forma crítica los resultados. Ya sea código, documentación, ideas o explicaciones técnicas, nunca podrás saber con total certeza si tu pregunta está bien formulada o si el contenido generado es el adecuado. Las personas que te rodean son tu recurso más valioso para ayudarte a evitar errores y puntos ciegos.

Mensaje principal:

- Utiliza la IA para reducir las tareas repetitivas o tediosas.
- Desarrolla habilidades de prompting, ya sea para programacion o para otros temas, que beneficiarán tu futura carrera.
- Aprende cómo funcionan los sistemas de IA para anticipar de forma eficiente y evitar los riesgos comunes, sesgos y problemas éticos.
- Sigue trabajando con tus compañeros para desarrollar tanto habilidades técnicas como habilidades transversales.
- Utiliza únicamente contenido generado por IA que entiendas completamente y del cual puedas responsabilizarte.

Reglas para estudiantes:

- Debes tomarte el tiempo necesario para explorar las herramientas de IA y comprender cómo funcionan, para poder utilizarlas de manera ética y reducir los sesgos potenciales.
- Debes reflexionar sobre tu problema antes de dar instrucciones a la IA. Esto te ayuda a escribir preguntas, instrucciones o conjuntos de datos más claros, detalladas y relevantes utilizando un vocabulario preciso.

- Debes desarrollar el hábito de revisar, cuestionar y probar sistemáticamente cualquier contenido generado por la IA.
- Debes buscar siempre la revisión de otras personas, no te limites a confiar en tu propia validación.

Resultados de esta etapa:

- Desarrollar habilidades de prompting tanto generales como de ámbito especifico.
- Aumentar tu productividad con un uso eficaz de las herramientas de IA.
- Seguir fortaleciendo el pensamiento computacional, la resolución de problemas, la adaptabilidad y la colaboración.

Comentarios y ejemplos:

- Ten en cuenta que la IA puede no tener la respuesta correcta porque esa respuesta no esté ni siquiera en Internet. Además, si te da soluciones incorrectas, intenta no insistir y busca ayuda entre las personas que te rodean. Vas a ahorrarte tiempo y vas a sumar en compresión.
- Vas a enfretarte con frecuencia a situaciones (como exámenes o evaluaciones) donde debes demostrar una comprensión real. Prepárate, sigue construyendo tanto tus habilidades técnicas como transversales.
- Explicar tu razonamiento y debatir con otras personas suele revelar lagunas en tu comprensión de un concepto. Prioriza el aprendizaje entre pares.
- Lo normal es que la herramienta de IA que utilices no conozca tu contexto específico (a menos que se lo indiques), así que te dará respuestas genéricas. Si buscas información más adecuada y más precisa en relación a tu entorno cercano, confía en el resto de estudiantes.
- Donde la IA tiende a generar la respuesta más probable, el resto de estudiantes puede proporcionar perspectivas alternativas y matices valiosos. Confía en la comunidad de 42 como un punto de control de calidad.

✓ Buenas prácticas:

> Le pregunto a la IA: "¿Cómo pruebo una función de ordenación?"Me da algunas ideas. Las pruebo y reviso los resultados con otra persona. Refinamos el enfoque de manera conjunta.

✗ Mala práctica:

> Le pido a la IA que escriba una función completa, la copio y la pego en mi proyecto. Durante la evaluación entre pares, no puedo explicar qué hace ni por qué. Pierdo credibilidad. Suspendo mi proyecto.

✓ Buenas prácticas:

Utilizo la IA para ayudarme a diseñar un parser. Luego, reviso la lógica con otra persona. Encontramos dos errores y lo reescribimos juntos: mejor, más limpio y comprendiendo al 100

✗ Mala práctica:

Dejo que Copilot genere mi código para una parte clave de mi proyecto. Compila, pero no puedo explicar cómo maneja los pipes. Durante la evaluación, no puedo justificarlo y suspendo mi proyecto.

Capítulo IV

Introducción

El proyecto **Push_Swap** es un proyecto simple, pero con un reto algorítmico muy concreto: tienes que ordenar datos.

Para esto, tendrás a tu disposición una lista de valores enteros, 2 stacks y un conjunto de instrucciones que usarás para manipular ambos stacks.

Tu objetivo en este proyecto es escribir un programa llamado `push_swap` que calcule y muestre por pantalla la secuencia de instrucciones más corta que necesitarías para ordenar la lista de enteros recibida.

Parece fácil, ¿a que sí?

Eso habrá que verlo...

Capítulo V

Objetivos

El objetivo de este proyecto es que descubras la complejidad algorítmica de una manera muy concreta.

Ordenar números es fácil; ordenarlos *rápido* usando solo dos stacks y con un puñado de movimientos permitidos es otra historia. Ordenar una lista completamente aleatoria y ordenar una lista prácticamente ordenada, también son dos cosas completamente distintas.

Vas a darte cuenta rápidamente de que la elección entre varios algoritmos distintos marca la diferencia entre una victoria rápida y una lista interminable de operaciones.

Capítulo VI

Parte obligatoria

VI.1. Requisitos de proyectos en grupo

- **Este proyecto debe ser completado por dos estudiantes trabajando de manera conjunta.**
- Ambos estudiantes deben contribuir significativamente al proyecto y entender correctamente todos los algoritmos implementados.
- El repositorio de entrega debe indicar claramente en el archivo README.md las contribuciones de cada estudiante.
- Durante la evaluación, ambos estudiantes deben ser capaces de explicar cualquier parte del código.
- El proyecto entregado debe incluir los "logins" de ambas personas estudiantes en el repositorio.

VI.2. Las reglas

- Tenéis 2 stacks, llamados `a` y `b`.
- Para empezar:
  - El stack `a` contiene una cantidad aleatoria de números positivos y/o negativos.
  - El stack `b` está vacío.
- El objetivo es ordenar los números del stack `a` en orden ascendente. Para hacerlo están disponibles las siguientes operaciones:

sa (`swap a`): Intercambia los dos primeros elementos del stack `a`.
No hace nada si hay solo uno o ningún elemento.

sb (`swap b`): Intercambia los dos primeros elementos del stack `b`.
No hace nada si hay solo uno o ningún elemento.

ss : `sa` y `sb` a la vez.

pa (`push a`): Toma el primer elemento del stack `b` y lo coloca el primero en el stack `a`.
No hace nada si `b` está vacío.

pb (`push b`): Toma el primer elemento del stack `a` y lo coloca el primero en el stack `b`.
No hace nada si `a` está vacío.

ra (`rotate a`): Desplaza hacia arriba todos los elementos del stack `a` una posición, convirtiendo el primer elemento en el último.

rb (`rotate b`): Desplaza hacia arriba todos los elementos del stack `b` una posición, convirtiendo el primer elemento en el último.

rr : `ra` y `rb` a la vez.

rra (`reverse rotate a`): Desplaza hacia abajo todos los elementos del stack `a` una posición, convirtiendo el último elemento en el primero.

rrb (`reverse rotate b`): Desplaza hacia abajo todos los elementos del stack `b` una posición, convirtiendo el último elemento en el primero

rrr : `rra` y `rrb` a la vez.

VI.3. Requisitos del algoritmo

Para asegurar un buen entendimiento de la complejidad algorítmica (*tiempo* y *espacio*), **se deben implementar cuatro estrategias de ordenación distintas** e integrarlas todas en el programa `push_swap`. Además, el programa debe ser capaz de seleccionar una estrategia u otra durante la ejecución, en función de la configuración de entrada.

VI.3.1. Modelo de complejidad y restricciones

Todas las estrategias deben ser implementadas en C y deben generar secuencias de operaciones de Push_swap para ordenar los elementos. Esto significa que:

- Los algoritmos en C analizan la entrada y generan la secuencia adecuada de operaciones para ordenarla: `sa`, `sb`, `ss`, `pa`, `pb`, `ra`, `rb`, `rr`, `rra`, `rrb`, `rrr`.
- La salida por consola de esta estrategia deberá ser la secuencia de operaciones necesarias para ordenar el stack.
- Al expresar la complejidad, debe medirse en función del **número de operaciones de Push_swap que el programa produce**, y no en base a la complejidad teórica de un algoritmo tradicional sobre arrays.

VI.3.2. Índice de desorden (obligatorio)

En este proyecto, el **desorden** corresponde a un número entre `0` y `1` que refleja lo lejos que el stack `a` se encuentra de estar ordenado al comienzo del programa.

Si todos los números están en orden, el índice de desorden será `0`. Si los números están lo más desordenados posibles, el índice de desorden será `1`. Todo lo que haya entre medias

significará que el stack se encuentra parcialmente ordenado, pero sigue teniendo desorden.

Para calcular el índice de desorden, imagina que observas todos los pares posibles de números en el stack. Cada vez que un número mayor aparece antes que uno menor, consideramos que hay un *error* en el orden. Cuantos más errores haya, mayor será el índice de desorden, acercándose al valor `1`, que refleja el desorden absoluto.

```
function compute_disorder(stack a):
    mistakes = 0
    total_pairs = 0
    for i from 0 to size(a)-1:
        for j from i+1 to size(a)-1:
            total_pairs += 1
            if a[i] > a[j]:
                mistakes += 1
    return mistakes / total_pairs
```

El índice de desorden se calculará **antes** de hacer ningún movimiento.

VI.3.3. Estrategias requeridas

1. **Algoritmo simple ($\mathbf{O(n^2)}$)**:
   Implementa al menos **un** algoritmo base perteneciente a la clase de complejidad $O(n^2)$. Por ejemplo:
   - Adaptación del orden por inserción
   - Adaptación del orden por selección
   - Adaptación del orden burbuja
   - Métodos simples de extracción del mínimo/máximo

2. **Algoritmo intermedio ($\mathbf{O(n\sqrt{n})}$)**:
   Implementa al menos **un** algoritmo perteneciente a la clase de complejidad $O(n\sqrt{n})$. Por ejemplo:
   - Orden basado en chunks (dividiendo en $\sqrt{n}$ chunks)
   - Métodos de partición basados en bloques
   - Adaptaciones del orden por buckets con $\sqrt{n}$ buckets
   - Estrategias de orden basadas en rangos

3. **Algoritmo complejo ($\mathbf{O(n \log n)}$)**:
   Implementa al menos **un** algoritmo perteneciente a la clase de complejidad $O(n \log n)$. Por ejemplo:
   - Adaptación del orden radix (LSD o MSD)
   - Adaptación del orden por fusión utilizando dos stacks
   - Adaptación del orden rápido con partición por stacks
   - Adaptación del orden por montículos
   - Algoritmos de árbol binario indexado

4. **Algoritmo adaptativo personalizado (diseñado por el estudiantado)**: Debe diseñarse una estrategia **adaptativa** que seleccione distintos métodos internos según el `índice de desorden`. No existe **ninguna** limitación en cuanto a algoritmos específicos, las técnicas internas quedan totalmente a elección vuestra. Sin embargo, el diseño debe respetar los siguientes **objetivos de complejidad** para cada régimen (según el modelo de operaciones de Push_swap):

   Índice de desorden bajo: Si desorden $< 0{,}2$, el método elegido debe ejecutarse en $O(n)$.

   Índice de desorden medio: Si $0{,}2 \leq$ desorden $< 0{,}5$, el método elegido debe ejecutarse en $O(n\sqrt{n})$.

   Índice de desorden alto: Si desorden $\geq 0{,}5$, el método elegido debe ejecutarse en $O(n \log n)$.

Deberá quedar documentado en el repositorio (por ejemplo, en el archivo `README.md`) la justificación de los valores límite (umbrales) que definen cada nivel de desorden, una descripción de las técnicas internas empleadas en cada caso y un breve análisis de complejidad -indicando las cotas superiores de espacio y tiempo- dentro del modelo de Push_swap.

VI.4. Ejemplos

Para entender mejor los efectos de algunos de los movimientos, vamos a ver cómo se ordenaría una lista de números aleatoria. En este ejemplo, consideraremos que ambos stacks crecen desde la derecha.

```
---------------------------------------------------------------------------------------------------------
Init a and b:
2
1
3
6
5
8

- -
a b
---------------------------------------------------------------------------------------------------------
Exec sa:
1
2
3
6
5
8

- -
a b
---------------------------------------------------------------------------------------------------------
Exec pb pb pb:
6 3
5 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------
Exec ra rb (equiv. to rr):
5 2
8 1
6 3

- -
a b
---------------------------------------------------------------------------------------------------------
Exec rra rrb (equiv. to rrr):
6 3
5 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------
Exec sa:
5 3
6 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------
Exec pa pa pa:
1
2
3
5
6
8

- -
a b
---------------------------------------------------------------------------------------------------------
```

Los enteros del stack `a` se han ordenado en 12 movimientos. ¿Creéis que se puede hacer mejor?

VI.5. El programa push_swap

Nombre de programa	`push_swap`
Archivos a entregar	`Makefile, *.h, *.c`
Makefile	`NAME, all, clean, fclean, re`
Argumentos	`stack a: una lista de enteros`
Funciones autorizadas	• `read, write, malloc, free, exit` • `ft_printf y cualquier función equivalente que se haya creado`
Se permite usar libft	`Si`
Descripción	`Ordena stacks`

Tu programa deberá cumplir las siguientes normas:

- Se debe entregar un `Makefile` que compile los archivos fuente. No debe hacer relink.
- Las variables globales están prohibidas.
- Se debe crear un programa llamado `push_swap` que recibirá por argumentos:
  - El stack `a` con el formato de una lista de enteros (el primer argumento debe ser el que esté encima del stack).
  - Un **selector de estrategia** opcional:

    `--simple` fuerza el uso de el algoritmo O(n^2) seleccionado.

    `--medium` fuerza el uso de el algoritmo O($n\sqrt{n}$) seleccionado.

    `--complex` fuerza el uso de el algoritmo O($n \log n$) seleccionado.

    `--adaptive` fuerza el uso de el algoritmo adaptativo basado en `desorden` seleccionado. Este será el comportamiento por defecto si no se indica un selector.
- El programa debe mostrar la lista de operaciones `Push_swap` más corta posible necesaria para ordenar el stack `a`, dejando el número más pequeño en la cima del stack.
- Las operaciones deben estar separadas por un `\n` y nada más.
- La clase de complejidad que se declare para cada algoritmo debe ser `válida dentro de este modelo`.

- El selector de estrategia debe funcionar para **todos los inputs válidos**. Cualquier flag de selección debe de funcionar correctamente, independientemente del tamaño de la entrada o del índice de desorden.
- Si no se especifican parámetros, el programa no deberá mostrar nada y deberá devolver el control al usuario.
- En caso de error, deberá mostrar `"Error"` seguido de un `\n` en la salida de error estándar. Algunos de los posibles errores son, por ejemplo: argumentos que no son enteros, argumentos superiores a un número entero, y/o números duplicados.
- El binario debe implementar las cuatro estrategias (Simple $O(n^2)$, Intermedia $O(n\sqrt{n})$, Compleja $O(n \log n)$ y Adaptativa). El nombre y la clase de complejidad de la estrategia seleccionada deberá estar disponible en el modo `--bench`.
- El modo **benchmark** (`--bench`) es opcional, y mostrará tras la ordenación lo siguiente:
  - El `índice de desorden` (% con dos decimales).
  - El nombre de la estrategia usada y su clase de complejidad teórica.
  - El número total de operaciones empleadas.
  - El número de operaciones de cada tipo (`sa`, `sb`, `ss`, `pa`, `pb`, `ra`, `rb`, `rr`, `rra`, `rrb`, `rrr`) empleadas durante la ordenación.

  La salida del modo **benchmark** debe enviarse a la salida `stderr` y solo se mostrará cuando la flag esté presente.

VI.5.1. Ejemplos de uso

Nota: Las lineas que comienzan con `[bench]` representan mensajes generados por el modo benchmark (a `stderr`). El flujo de operaciones se mantiene en `stdout`.

```
$>./push_swap 2 1 3 6 5 8
ra
pb
rra
pb
pb
ra
pb
ra
pb
pb
pa
pa
pa
pa
pa
pa
```

Selector de algoritmo por defecto (`--adaptive`) y contador de operaciones:

```
$> ARG="4 67 3 87 23"; ./push_swap --adaptive $ARG | wc -l
13
```

Forzado del uso de la estrategia simple (O(n^2)):

```
$> ARG="4 67 3 87 23"; ./push_swap --simple 5 4 3 2 1
rra
pb
rra
pb
rra
pb
ra
pb
pb
pa
pa
pa
pa
pa
```

Forzado del uso de la estrategia compleja (O(n log n)) y verificado con el checker:

```
$> ARG="4 67 3 87 23"; ./push_swap --complex $ARG | ./checker_linux $ARG
OK
```

push_swap con un input largo:

```
$> shuf -i 0-9999 -n 500 > args.txt ; ./push_swap $(cat args.txt) | wc -l
6784
$>
```

Ejecución del programa con el modo benchmark; ocultando las operaciones y mostrando solo las métricas:

```
$>shuf -i 0-9999 -n 500 > args.txt ; ./push_swap --bench $(cat
args.txt) 2> bench.txt | ./checker_linux $(cat args.txt)
OK
$> cat bench.txt
[bench] disorder:  49.93%
[bench] strategy:  Adaptive / O(n√n)
[bench] total_ops:  7997
[bench] sa:  0  sb:  0  ss:  0  pa:  500  pb:  500
[bench] ra:  4840  rb:  1098  rr:  0  rra:  0  rrb:  1059  rrr:  0
```

Redirección de las operaciones al checker y guardado del output del modo benchmark en un archivo:

```
$> ARG="4 67 3 87 23"; ./push_swap --bench --adaptive $ARG 2>
bench.txt | ./checker_linux $ARG
OK
$> cat bench.txt
[bench] disorder:  40.00%
[bench] strategy:  Adaptive / O(n√n)
[bench] total_ops:  13
[bench] sa:  0  sb:  0  ss:  0  pa:  5  pb:  5
[bench] ra:  2  rb:  1  rr:  0  rra:  0  rrb:  0  rrr:  0
```

Ejemplos del manejo de errores:

```
$> ./push_swap --adaptive 0 one 2 3
Error
$> ./push_swap --simple 3 2 3
Error
```

VI.6. Prueba de rendimiento

Para superar el proyecto, se deben cumplir ciertos objetivos en cuanto al número de operaciones empleadas, que reflejará el buen rendimiento del programa:

- Para **100 números aleatorios**, el programa deberá usar:
  - menos de **2000 operaciones** para superar el corte (requisito mínimo)
  - menos de **1500 operaciones** para un buen rendimiento
  - menos de **700 operaciones** para un rendimiento excelente
- Para **500 números aleatorios**, el programa deberá usar:
  - menos de **12000 operaciones** para superar el corte (requisito mínimo)
  - menos de **8000 operaciones** para un buen rendimiento
  - menos de **5500 operaciones** para un rendimiento excelente

Todo esto será comprobado durante la evaluación usando el programa “checker” que se incluye como recurso del proyecto.

Capítulo VII

Requisitos del Readme

Debe incluirse un archivo `README.md` en la raíz del repositorio Git. Su propósito es permitir que cualquier persona que no esté familiarizada con el proyecto (pares, personal, responsables de selección, etc.) pueda entender rápidamente de qué trata el proyecto, cómo ejecutarlo y dónde encontrar más información sobre el tema.
El `README.md` debe incluir, como mínimo:

- La primera línea debe estar en cursiva y decir: *Este proyecto ha sido creado como parte del currículo de 42 por <login1>[, <login2>[, <login3>[...]]].*
- Una sección de "**Descripción**"que presente claramente el proyecto, incluyendo su objetivo y una breve visión general.
- Una sección de "**Instrucciones**"que contenga cualquier información relevante sobre compilación, instalación y/o ejecución.
- Una sección de "**Recursos**"que enumere referencias clásicas relacionadas con el tema (documentación, artículos, tutoriales, etc.), así como una descripción del uso de IA, especificando para qué tareas y en qué partes del proyecto se ha utilizado.

➠ **Podrían requerirse secciones adicionales dependiendo del proyecto** (por ejemplo, ejemplos de uso, lista de características, decisiones técnicas, etc.).

Cualquier contenido extra requerida se listará explícitamente a continuación.

- Se deberá incluir una explicación detallada y una justificación de los algoritmos seleccionados.

La elección del idioma queda sujeto a criterio del grupo de trabajo.
Se recomienda usar el inglés, pero no es obligatorio.

Capítulo VIII

Parte bonus

Este proyecto deja muy poco margen para añadir funcionalidades extra debido a su simplicidad. Sin embargo, ¿qué tal sería crear un checker propio?

Gracias al programa checker, podrás probar si la lista de instrucciones generadas por el programa push_swap realmente ordena el stack de forma correcta.

La parte bonus no será evaluada si la parte obligatoria no está perfecta. Perfecta quiere decir que se ha completado la parte obligatoria y que funciona perfectamente, sin errores. En este proyecto, esto implica validar todas las pruebas de rendimiento sin excepción. Si no has pasado todas las pruebas obligatorias, tu parte bonus no será evaluada.

VIII.1. El programa checker

Nombre de programa	`checker`
Archivos a entregar	`*.h, *.c`
Makefile	`bonus`
Argumentos	`stack a: una lista de números enteros`
Funciones autorizadas	• `read, write, malloc, free, exit` • `ft_printf y cualquier función equivalente que se haya creado`
Se permite usar libft	`Sí`
Descripción	`Ejecuta las instrucciones de ordenación`

- Se deberá escribir un programa llamado `checker`, que tome como argumento el stack `a` en forma de una lista de enteros. El primer argumento debe estar encima del stack (cuidado con el orden). Si no se da argumento, `checker` termina y no muestra nada.
- Durante la ejecución de `checker` se esperará y leerá una lista de instrucciones, separadas utilizando '\n'. Cuando todas las instrucciones se hayan leído, `checker` las ejecutará utilizando el stack recibido como argumento.
- Si tras ejecutar todas las instrucciones, el stack `a` está ordenado y el stack `b` vacío, entonces el programa `checker` mostrará `"OK"` seguido de un '\n' en la **'stdout'**.
- En cualquier otro caso, deberá mostrar `"KO"` seguido de un '\n' en la **'stdout'**.
- En caso de error, se deberá mostrar `Error` seguido de un '\n' en la **'stderr'**. Los errores incluyen, por ejemplo, algunos o todos los argumentos no sean enteros, algunos o todos los argumentos sean más grandes que un número entero, que haya duplicados, una instrucción no exista y/o no tenga el formato correcto.

```
$>./checker 3 2 1 0
rra
pb
sa
rra
pa
OK
$>./checker 3 2 1 0
sa
rra
pb
```

```
KO
$>./checker 3 2 one 0
Error
$>./checker "" 1
Error
$>
```

No hay que replicar el comportamiento exacto del binario que se entrega. Es obligatorio gestionar errores, pero es decisión de cada grupo cómo gestionar los argumentos.

Capítulo IX

Entrega y evaluación

El trabajo deberá entregarse en el repositorio `git` como de costumbre. Solo el trabajo en el repositorio será evaluado. Se deberá comprobar el nombre de los archivos para asegurar que son correctos.

Requisitos para la entrega de proyectos en grupo:

- Todas las personas que compongan el grupo deberán estar mencionadas como colaboradores del proyecto en el repositorio.
- Deberá quedar reflejado en el archivo README.md la contribución de cada persona al proyecto.
- Todas las personas del grupo deberán estar presentes durante la evaluación del proyecto.
- Cada persona del grupo debería ser capaz de explicar cualquier parte del código.

Durante la evaluación, es posible que se solicite una ligera **modificación del proyecto**. Esto puede consistir en ajustar ligeramente el comportamiento, modificar unas cuantas líneas de código o incorporar una característica fácil de implementar.

Puede que este paso **no sea necesario en todos los proyectos**, pero hay que tenerlo en cuenta si así se especifica en la hoja de evaluación.

Este paso sirve para verificar la comprensión real de una parte específica del proyecto. La modificación se puede realizar en cualquier entorno de desarrollo que se elija (por ejemplo, la configuración habitual), y debería ser factible en unos pocos minutos, a menos que se defina un plazo específico como parte de la evaluación.
Por ejemplo, se puede pedir hacer una pequeña actualización en una función o *script*, modificar lo que se vería en pantalla o ajustar una estructura de datos para almacenar nueva información, etc.

Los detalles (alcance, objetivo, etc.) se especificarán cada **hoja de evaluación** y pueden

variar de una evaluación a otra para el mismo proyecto.

file.bfe:VABB7yO9xm7xWXROeASsmsgnYOoOsDMJev7zFHhwQS8mvM8V5xQQp
Lc6cDCFXDWTiFzZ2H9skYkiJ/DpQtnM/uZO