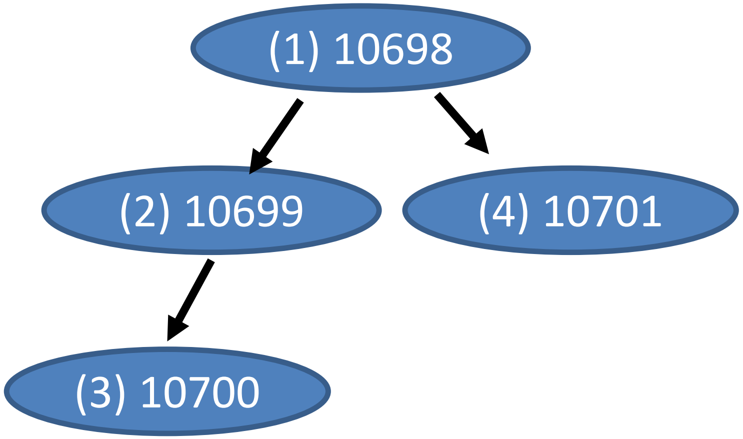
Part 2 Exercise for Process

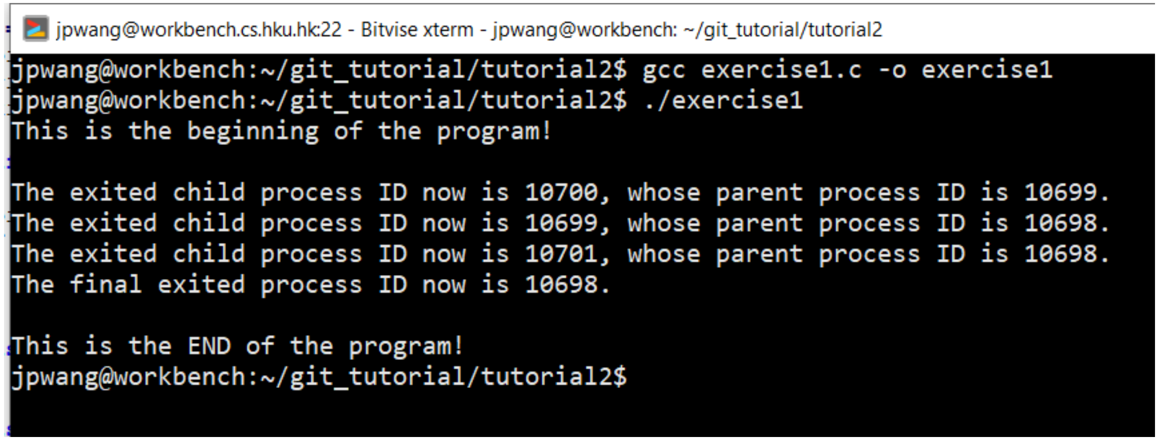
□ Exercise 1 (40%)

Write a program using fork() and wait():
Create 3 child processes by **fork(),** and the relationship between processes is shown in the process tree below. Design your program using fork() and wait() to guarantee that the creation order of 4 processes (including parent process) is **1-2-3-4**, and the exited order should be **3-2-4-1**.

- **Process tree**



- **Output example**


jpwang@workbench.cs.hku.hk:22 - Bitvise xterm - jpwang@workbench: ~/git_tutorial/tutorial2

```
jpwang@workbench:~/git_tutorial/tutorial2$ gcc exercise1.c -o exercise1
jpwang@workbench:~/git_tutorial/tutorial2$ ./exercise1
This is the beginning of the program!

The exited child process ID now is 10700, whose parent process ID is 10699.
The exited child process ID now is 10699, whose parent process ID is 10698.
The exited child process ID now is 10701, whose parent process ID is 10698.
The final exited process ID now is 10698.

This is the END of the program!
jpwang@workbench:~/git_tutorial/tutorial2$
```

Part 2 Exercise for Process

□ Exercise 1

Write a program using fork() and wait():

Create 3 child processes by **fork().** If the order of creation (including parent process) is **1-2-3-4**, the exited order should be **3-2-4-1**.

➢ Requirements

1. Firstly, print "This is the beginning of the program!”;

2. Create 3 child processes and control the creation and exit order using fork() and wait().

   - Each child process should print:

     *“The exited child process ID now is **, whose parent process ID is **.”*

   - The final exited parent process should print:

     *“The final exited process ID now is *.”*

3. Finally, print "This is the END of the program!”;

Note that:

(1) **sleep() is not allowed here.**

(2) If the creation and exited order is wrong, no points will be given.

(3) If the output format is not consistent with the requirements, it will be penalized.

Part 2 Exercise for Process

□ Exercise 2 (30%)

Process address space:

According to the provided program, analyze the memory layout of the program.

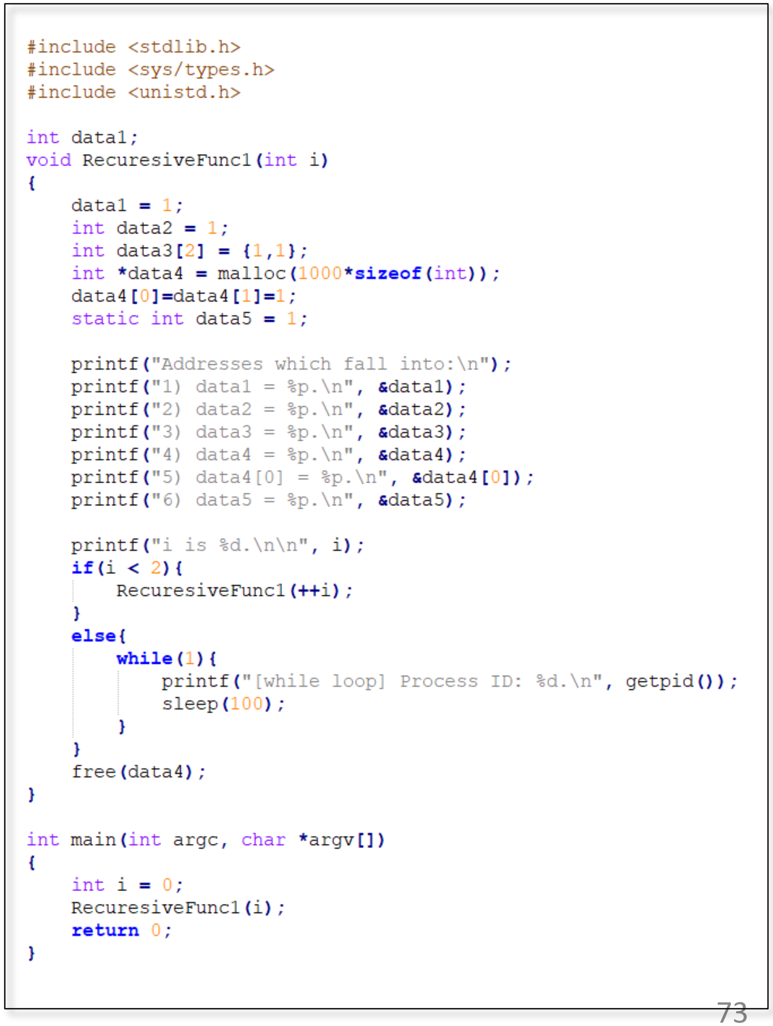
For the variables *data1, data2, data3, data4, data4[0], data5*:

(1) Where are these variables stored? Give you reasons.

A. Code segment
B. Data segment
C. BSS
D. Heap
E. Stack segment

(2) Estimate the stack size of *__RecursiveFunc1__* and give your reasons.

➢ **Code:** exercise2.c

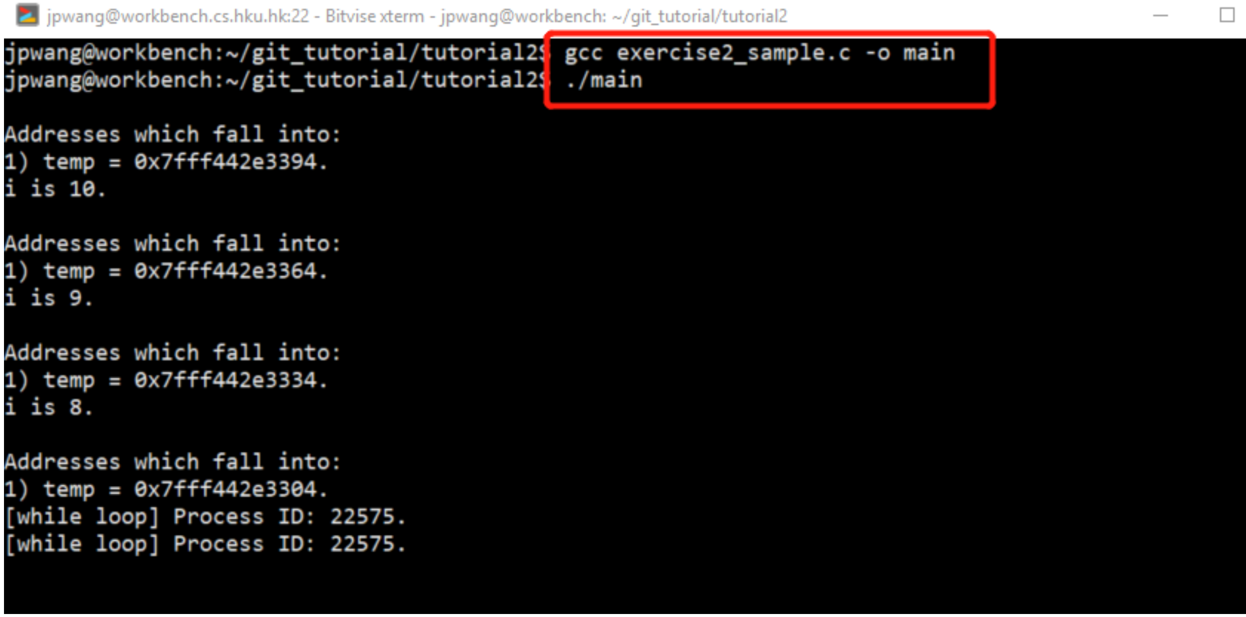


```c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int data1;
void RecuresiveFunc1(int i)
{
    data1 = 1;
    int data2 = 1;
    int data3[2] = {1,1};
    int *data4 = malloc(1000*sizeof(int));
    data4[0]=data4[1]=1;
    static int data5 = 1;

    printf("Addresses which fall into:\n");
    printf("1) data1 = %p.\n", &data1);
    printf("2) data2 = %p.\n", &data2);
    printf("3) data3 = %p.\n", &data3);
    printf("4) data4 = %p.\n", &data4);
    printf("5) data4[0] = %p.\n", &data4[0]);
    printf("6) data5 = %p.\n", &data5);

    printf("i is %d.\n\n", i);
    if(i < 2){
        RecuresiveFunc1(++i);
    }
    else{
        while(1){
            printf("[while loop] Process ID: %d.\n", getpid());
            sleep(100);
        }
    }
    free(data4);
}

int main(int argc, char *argv[])
{
    int i = 0;
    RecuresiveFunc1(i);
    return 0;
}
```

□ Exercise 2

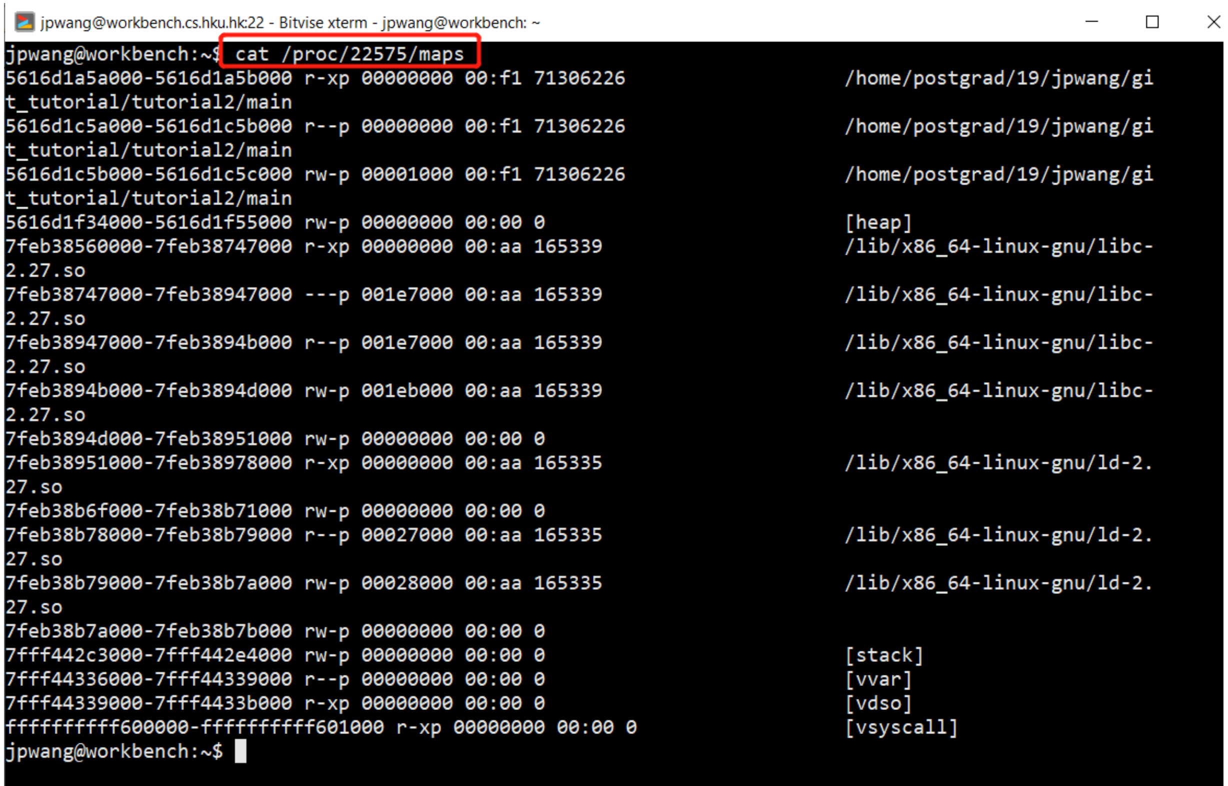
Sample output

Output of **exercise2_sample.c**


jpwang@workbench.cs.hku.hk:22 - Bitvise xterm - jpwang@workbench: ~/git_tutorial/tutorial2

```
jpwang@workbench:~/git_tutorial/tutorial2$ gcc exercise2_sample.c -o main
jpwang@workbench:~/git_tutorial/tutorial2$ ./main

Addresses which fall into:
1) temp = 0x7fff442e3394.
i is 10.

Addresses which fall into:
1) temp = 0x7fff442e3364.
i is 9.

Addresses which fall into:
1) temp = 0x7fff442e3334.
i is 8.

Addresses which fall into:
1) temp = 0x7fff442e3304.
[while loop] Process ID: 22575.
[while loop] Process ID: 22575.
```

Run the command:
cat /proc/22575/maps


jpwang@workbench.cs.hku.hk:22 - Bitvise xterm - jpwang@workbench: ~

```
jpwang@workbench:~$ cat /proc/22575/maps
5616d1a5a000-5616d1a5b000 r-xp 00000000 00:f1 71306226                   /home/postgrad/19/jpwang/gi
t_tutorial/tutorial2/main
5616d1c5a000-5616d1c5b000 r--p 00000000 00:f1 71306226                   /home/postgrad/19/jpwang/gi
t_tutorial/tutorial2/main
5616d1c5b000-5616d1c5c000 rw-p 00001000 00:f1 71306226                   /home/postgrad/19/jpwang/gi
t_tutorial/tutorial2/main
5616d1f34000-5616d1f55000 rw-p 00000000 00:00 0                          [heap]
7feb38560000-7feb38747000 r-xp 00000000 00:aa 165339                     /lib/x86_64-linux-gnu/libc-
2.27.so
7feb38747000-7feb38947000 ---p 001e7000 00:aa 165339                     /lib/x86_64-linux-gnu/libc-
2.27.so
7feb38947000-7feb3894b000 r--p 001e7000 00:aa 165339                     /lib/x86_64-linux-gnu/libc-
2.27.so
7feb3894b000-7feb3894d000 rw-p 001eb000 00:aa 165339                     /lib/x86_64-linux-gnu/libc-
2.27.so
7feb3894d000-7feb38951000 rw-p 00000000 00:00 0
7feb38951000-7feb38978000 r-xp 00000000 00:aa 165335                     /lib/x86_64-linux-gnu/ld-2.
27.so
7feb38b6f000-7feb38b71000 rw-p 00000000 00:00 0
7feb38b78000-7feb38b79000 r--p 00027000 00:aa 165335                     /lib/x86_64-linux-gnu/ld-2.
27.so
7feb38b79000-7feb38b7a000 rw-p 00028000 00:aa 165335                     /lib/x86_64-linux-gnu/ld-2.
27.so
7feb38b7a000-7feb38b7b000 rw-p 00000000 00:00 0
7fff442c3000-7fff442e4000 rw-p 00000000 00:00 0                          [stack]
7fff44336000-7fff44339000 r--p 00000000 00:00 0                          [vvar]
7fff44339000-7fff4433b000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
jpwang@workbench:~$ █
```

Part 2 Exercise for Process

□ Exercise 2

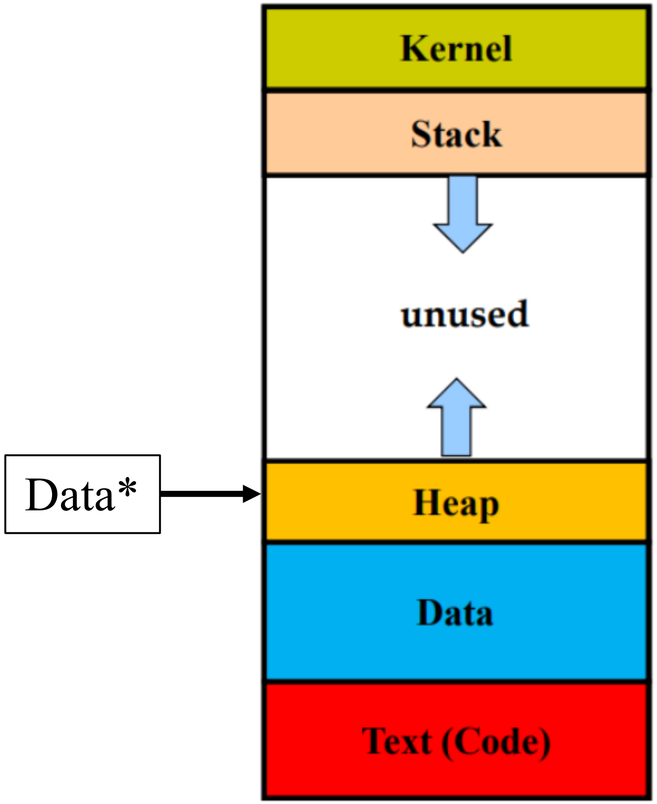
➢ **Requirments**

Submit your answers in **one PDF** file:

(1) **Draw a diagram** to show the locations of variables and give your reasons.

(2) **Analyze the stack size** and give your reasons;

Include some screenshots of running results if needed instead of only providing theoretical analysis.



sample diagram

Part 2 Exercise for Process

□ Exercise 3 (30%)

Write a program using Shared Memory which receives three positive integers from the command line. In the program:

1. Check whether the number of command line arguments is 3. If not, print the error information and return;

   *"Error: The number of input integers now is *. Please input 3 integers."*

2. Use fork() to create a child process:

   1) In the child process, sort these 3 integers in increasing order, and print the sorting results;

      *"Child process ID: **; Sorting results: **, **, **."*

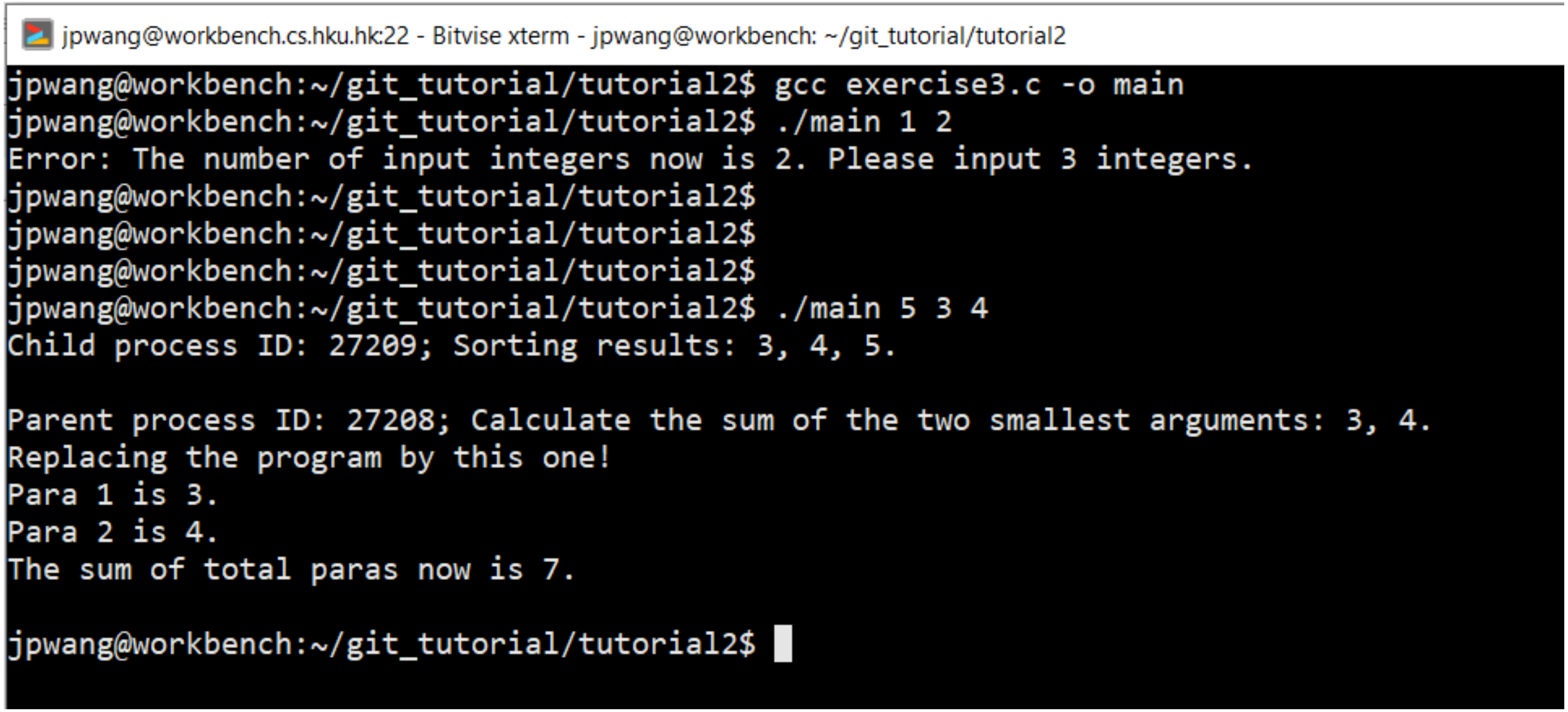
   2) According to the sorting results above, use execv() to call para_sum.c to calculate the sum of the two smallest arguments in the parent process;

      *"Parent process ID: **; Calculate the sum of the two smallest arguments: **, **.\n"*

Part 2 Exercise for Process

□ Exercise 3 (30%)

Sample output



jpwang@workbench.cs.hku.hk:22 - Bitvise xterm - jpwang@workbench: ~/git_tutorial/tutorial2

```
jpwang@workbench:~/git_tutorial/tutorial2$ gcc exercise3.c -o main
jpwang@workbench:~/git_tutorial/tutorial2$ ./main 1 2
Error: The number of input integers now is 2. Please input 3 integers.
jpwang@workbench:~/git_tutorial/tutorial2$
jpwang@workbench:~/git_tutorial/tutorial2$
jpwang@workbench:~/git_tutorial/tutorial2$
jpwang@workbench:~/git_tutorial/tutorial2$ ./main 5 3 4
Child process ID: 27209; Sorting results: 3, 4, 5.

Parent process ID: 27208; Calculate the sum of the two smallest arguments: 3, 4.
Replacing the program by this one!
Para 1 is 3.
Para 2 is 4.
The sum of total paras now is 7.

jpwang@workbench:~/git_tutorial/tutorial2$ █
```