## 1. Memcheck

a. Invalid write of size 4

```
==7844== Invalid write of size 4
==7844==    at 0x1091C0: main (memleak.c:49)
==7844==  Address 0x4a7d068 is 0 bytes after a block of size 40 alloc'd
==7844==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==7844==    by 0x10919E: main (memleak.c:46)
==7844==
```

**Explanation:**

The program allocates 40 bytes of memory using the malloc() function, which is enough space to store 10 integers (i.e., malloc(10 * sizeof(int)) gives access to arr[0] - arr[9] in the heap). However, it writes an additional 4 bytes (the size of an int) just after the allocated memory block. Accessing arr[10], which is one element beyond the valid range of arr[0] to arr[9]. This kind of error results in a heap buffer overflow, where memory outside the allocated region is being incorrectly modified. So, occur invalid write of size 4.

---

The possible loop used in this test:

```
int *arr = (int *)malloc(10 * sizeof(int));
for (int i = 0; i <= 10; i++) {
    arr[i] = 0;  // Error when i = 10 → out-of-bounds write
}
```

---

To slove the error we need to adjust our loop like below:

```
int *arr = (int *)malloc(10 * sizeof(int));
for (int i = 0; i < 10; i++) {
          arr[i] = 0;
}
```

---

b. Invalid read of size 4

```
==7844== Invalid read of size 4
==7844==    at 0x1091ED: main (memleak.c:54)
==7844==  Address 0x4a7d068 is 0 bytes after a block of size 40 alloc'd
==7844==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==7844==    by 0x10919E: main (memleak.c:46)
==7844==
```

**Explanation:**

This error occurs when the program attempts to read 4 bytes (the size of an int) from memory just beyond the allocated buffer. Just like the invalid write error, this happens because the program accesses an out-of-bounds index, like arr[10], when only arr[0] through arr[9] are valid. This time, the program is trying to read from that invalid memory location. So, occur invalid read of size 4.

---

The possible loop used in this test:

```
int sum = 0;
for (int i = 0; i <= 10; i++) {
    sum += arr[i];  // Error when i = 10 → out-of-bounds read
}
```

---

To slove the error we need to adjust our loop like below:

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += arr[i];
}
```

---

c. Conditional jump or move depends on uninitialized value(s) / Use of uninitialized value of size 8

```
==7844== Conditional jump or move depends on uninitialised value(s)
==7844==    at 0x48D20CB: __printf_buffer (vfprintf-process-arg.c:58)
==7844==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==7844==    by 0x48C81B2: printf (printf.c:33)
==7844==    by 0x109214: main (memleak.c:57)
==7844==
==7844== Use of uninitialised value of size 8
==7844==    at 0x48C70BB: _itoa_word (_itoa.c:183)
==7844==    by 0x48D1C9B: __printf_buffer (vfprintf-process-arg.c:155)
==7844==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==7844==    by 0x48C81B2: printf (printf.c:33)
==7844==    by 0x109214: main (memleak.c:57)
==7844==
==7844== Conditional jump or move depends on uninitialised value(s)
==7844==    at 0x48C70CC: _itoa_word (_itoa.c:183)
==7844==    by 0x48D1C9B: __printf_buffer (vfprintf-process-arg.c:155)
==7844==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==7844==    by 0x48C81B2: printf (printf.c:33)
==7844==    by 0x109214: main (memleak.c:57)
==7844==
==7844== Conditional jump or move depends on uninitialised value(s)
==7844==    at 0x48D1D85: __printf_buffer (vfprintf-process-arg.c:186)
==7844==    by 0x48D373A: __vfprintf_internal (vfprintf-internal.c:1544)
==7844==    by 0x48C81B2: printf (printf.c:33)
==7844==    by 0x109214: main (memleak.c:57)
==7844==
```

**Explanation:**

This error means that the program is using a function such as printf based on a value that was never initialized. Typically, this happens when a variable (such as an integer or pointer) is declared but not assigned any value before being used in output or computation. In C, variables are not automatically initialized to zero. Instead, they contain whatever random data is already present at that memory location. This is known as a "garbage value" and using it can cause unpredictable behavior, like:

- Use of uninitialized value of size 8:
  - A value is used (e.g., printed) before being set.
- Conditional jump or move depends on uninitialized value(s):
  - The program makes a decision (e.g., an if-condition or printf's formatting logic) using that garbage value.

---

The possible program used in this test:

```
int sum;                // Declared but not initialized
printf("%d\n", sum);  // Using it before assigning a value
```

---

To slove the error we need to adjust our program like below:

```
int sum = 0;            // Initialize to a known value
printf("%d\n", sum);  // Safe: prints 0
```

---

d. Argument 'size' of malloc has a fishy (possibly negative) value

```
==7844== Argument 'size' of function malloc has a fishy (possibly negative) value: -40
==7844==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==7844==    by 0x109220: main (memleak.c:61)
```

**Explanation:**

This error means the program is calling malloc() with an invalid size, specifically a negative number (e.g., -40). In C, the argument passed to malloc() must be a non-negative value of type size_t. However, when a negative integer is implicitly converted to size_t, it wraps around into a very large positive number, causing unexpected behavior or allocation failures. This usually happens when the size passed to malloc is incorrect values, such as:

---

The possible program used in this test:

```
int size = -10;

int *arr = (int *)malloc(size * sizeof(int));  // results in malloc(-40)
```

---

To slove the error we need to adjust our program like below:

```
int size = 10;

int *arr = (int *)malloc(size * sizeof(int));  // malloc(40), OK
```

---

e. Invalid free() / delete / delete[] / realloc()

```
==7844== Invalid free() / delete / delete[] / realloc()
==7844==    at 0x48490C4: free (vg_replace_malloc.c:884)
==7844==    by 0x10924A: main (memleak.c:65)
==7844==  Address 0x4a7d4f0 is 0 bytes inside a block of size 40 free'd
==7844==    at 0x48490C4: free (vg_replace_malloc.c:884)
==7844==    by 0x10923E: main (memleak.c:64)
==7844==  Block was alloc'd at
==7844==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==7844==    by 0x10922E: main (memleak.c:63)
```

**Explanation:**

This error means the program attempted to free a memory block that was already freed earlier. In C, calling free() on the same pointer more than once leads to undefined behavior, such as crashing or corrupting the heap. In the log, we can see `Address 0x4a7d4f0 is 0 bytes inside a block of size 40`, This indicates that the program is trying to free the exact same block of memory that was already released. The first call to free() was valid, but the second call to free() on the same pointer caused the error. So, occur invalid free() / delete / delete[] / realloc()

---

The possible program used in this test:

```
int *arr = malloc(40);

free(arr);

free(arr);  // Error: double free
```

---

To slove the error we need to adjust our program like below:

```
int *arr = malloc(40);

free(arr);

arr = NULL;  // Clears the pointer

free(arr);  // Safe: free(NULL) does nothing
```

---

f. Memory leak

```
==7844== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==7844==    at 0x484684F: malloc (vg_replace_malloc.c:393)
==7844==    by 0x10919E: main (memleak.c:46)
==7844==
==7844== LEAK SUMMARY:
==7844==    definitely lost: 40 bytes in 1 blocks
==7844==    indirectly lost: 0 bytes in 0 blocks
==7844==      possibly lost: 0 bytes in 0 blocks
==7844==    still reachable: 0 bytes in 0 blocks
==7844==         suppressed: 0 bytes in 0 blocks
```

**Explanation:**

This error means the program allocated memory on the heap using malloc(), but never released it using free(). More specifically, the pointer to the allocated memory was either lost, went out of scope, or was simply never freed, making it impossible to reclaim that memory during the program's execution. We can see in log, `40 bytes in 1 blocks are definitely lost`. This means 40 bytes of memory were allocated but no corresponding free() was found. The tag "definitely lost" mean that no pointer still points to that memory, it is permanently unreachable. So, occur memory leaked.

---

The possible program used in this test:

```
int *arr = (int *)malloc(40);

// ... some code

// forgot to call free(arr); before program ends
```

---

To slove the error we need to adjust our program like below:

```
int *arr = (int *)malloc(40);

// use arr ...

free(arr);  // explicitly free the allocated memory
```

---

## 2. Cachegrind

```
ben@ben-server:~/Desktop$ cat 312551002_good_log
==15714== Cachegrind, a cache and branch-prediction profiler
==15714== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==15714== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==15714== Command: ./good
==15714== Parent PID: 7540
==15714==
--15714-- warning: L3 cache found, using its data for the LL simulation.
==15714==
==15714== I   refs:      30,158,770
==15714== I1  misses:         1,063
==15714== LLi misses:         1,042
==15714== I1  miss rate:       0.00%
==15714== LLi miss rate:       0.00%
==15714==
==15714== D   refs:      14,052,656  (12,039,322 rd   + 2,013,334 wr)
==15714== D1  misses:       126,516  (    63,684 rd   +    62,832 wr)
==15714== LLd misses:        63,811  (     1,000 rd   +    62,811 wr)
==15714== D1  miss rate:        0.9% (       0.5%     +       3.1%  )
==15714== LLd miss rate:        0.5% (       0.0%     +       3.1%  )
==15714==
==15714== LL refs:          127,579  (    64,747 rd   +    62,832 wr)
==15714== LL misses:         64,853  (     2,042 rd   +    62,811 wr)
==15714== LL miss rate:         0.1% (       0.0%     +       3.1%  )
```

```
ben@ben-server:~/Desktop$ cat 312551002_bad_log
==15715== Cachegrind, a cache and branch-prediction profiler
==15715== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==15715== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==15715== Command: ./bad
==15715== Parent PID: 7540
==15715==
--15715-- warning: L3 cache found, using its data for the LL simulation.
==15715==
==15715== I   refs:      30,158,770
==15715== I1  misses:         1,065
==15715== LLi misses:         1,044
==15715== I1  miss rate:       0.00%
==15715== LLi miss rate:       0.00%
==15715==
==15715== D   refs:      14,052,656  (12,039,322 rd   + 2,013,334 wr)
==15715== D1  misses:       203,632  (   102,995 rd   +   100,637 wr)
==15715== LLd misses:        63,810  (       999 rd   +    62,811 wr)
==15715== D1  miss rate:        1.4% (       0.9%     +       5.0%  )
==15715== LLd miss rate:        0.5% (       0.0%     +       3.1%  )
==15715==
==15715== LL refs:          204,697  (   104,060 rd   +   100,637 wr)
==15715== LL misses:         64,854  (     2,043 rd   +    62,811 wr)
==15715== LL miss rate:         0.1% (       0.0%     +       3.1%  )
```

**Explanation:**

| Metric | ./good | ./bad | Explanation |
|---|---|---|---|
| D1 misses (data cache) | 126,516 | 203,632 | bad has 77,116 more cache misses |
| D1 miss rate | 0.9% (0.5% + 3.1%) | 1.4% (0.9% + 5.0%) | bad shows much higher write miss rate |
| LL references (LL refs) | 127,579 | 204,697 | More frequent access to last-level cache in bad |

Although both programs execute the same number of instructions and data accesses, their cache behaviors are very different. This difference is caused by how the programs access memory in loops. In C, arrays are stored in row-major order, meaning that elements in the same row are stored next to each other in memory. `good` accesses adjacent memory addresses, so better cache hit rate, `bad` jumps between distant memory addresses, so more cache misses.

This explains:

- D1 (data cache) miss rate is higher in ./bad (1.4%) than in ./good (0.9%)
- Write miss rate in ./bad is 5.0%, compared to ./good's 3.1%
- LL (last-level cache) accesses are higher in ./bad due to more frequent cache evictions

---

The possible program used in good test:

```
for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
        matrix[i][j]++; // good locality
```

---

The possible program used in bad test:

```
for (int j = 0; j < cols; j++)
    for (int i = 0; i < rows; i++)
        matrix[i][j]++; // poor locality
```

---

The performance gap between `good` and `bad` is caused by inefficient memory access in `bad`. Although both programs do the same amount of work, `bad` accesses memory in a way that leads to poor cache utilization (likely column-wise), while `good` uses a more cache-friendly (row-wise) access pattern. This demonstrates how data locality can have a significant impact on performance.

## 3. Massif

a. Please observe the relationship between time and memory allocation throughout the **entire** program execution, and provide **one** screenshot of the output file containing relevant information. (You must clearly display the total number of **snapshots** each time the system records the information **in intervals**).

```
ben@ben-server:~/Desktop$ ms_print massif.out.20416
--------------------------------------------------------------------------------
Command:            ./heap
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.20416
--------------------------------------------------------------------------------


    KB
234.0^                                                                     #
     |                                                                  :::#:@
     |                                                               @@::: #:@
     |                                                           ::::@ ::: #:@
     |                                                       ::::::  @ ::: #:@
     |                                                    ::::: :::  @ ::: #:@
     |                                                   @:  :: :::  @ ::: #:@
     |                                             ::::::@:  :: :::  @ ::: #:@
     |                                         ::::::    @:  :: :::  @ ::: #:@
     |                                      :::::::::    @:  :: :::  @ ::: #:@
     |                                   ::@:: ::::::    @:  :: :::  @ ::: #:@
     |                               ::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |                            :::::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |                        ::@:: :::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |                      ::::@:: :::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |                  ::::::::@:: :::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |              ::::::: ::::@:: :::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |           :@::   ::: ::::@:: :::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |       :::::@::   ::: ::::@:: :::::::@:: ::::::    @:  :: :::  @ ::: #:@
     |   ::::: :::@::   ::: ::::@:: :::::::@:: ::::::    @:  :: :::  @ ::: #:@
   0 +----------------------------------------------------------------------->KB
     0                                                                   243.9

Number of snapshots: 76
 Detailed snapshots: [9, 19, 29, 39, 49, 59, 65 (peak), 75]
```

**Explanation:**

The output reveals a steadily increasing trend in heap memory usage throughout the program's execution. This pattern reflects continuous memory allocation without corresponding deallocation, due to the absence of free() calls in the source code.

No significant memory release is observed, suggesting that all allocated memory is retained until the program terminates. A total of 76 snapshots were recorded by Massif, 75 at regular intervals and 1 final snapshot at the program's end. The massif graph clearly illustrates a gradual and consistent growth in heap memory, driven by nested function calls and persistent allocations.

b. Then, point out how many bytes are allocated and used **at peak** respectively.

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 60        225,568          225,568          225,000           568            0
 61        229,576          229,576          229,000           576            0
 62        231,584          231,584          231,000           584            0
 63        235,592          235,592          235,000           592            0
 64        239,600          239,600          239,000           600            0
 65        239,600          239,600          239,000           600            0
99.75% (239,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->48.41% (116,000B) 0x10919A: a (in /home/ben/Desktop/heap)
| ->26.71% (64,000B) 0x1091B4: b (in /home/ben/Desktop/heap)
| | ->10.02% (24,000B) 0x1091CE: c (in /home/ben/Desktop/heap)
| | | ->03.34% (8,000B) 0x1091F7: d (in /home/ben/Desktop/heap)
| | | | ->01.67% (4,000B) 0x109234: e (in /home/ben/Desktop/heap)
| | | | | ->01.67% (4,000B) 0x1092AF: main (in /home/ben/Desktop/heap)
| | | | |
| | | | ->01.67% (4,000B) 0x1092C3: main (in /home/ben/Desktop/heap)
| | | |
```

**Explanation:**

**- Overall Trend**: Heap memory usage continuously increases, indicating sustained allocation without deallocation.

**- Peak Usage**: The peak heap memory usage occurs at snapshot 65, reaching 239600 bytes, of which 239000 bytes are useful heap and 600 bytes are extra heap (heap management overhead).

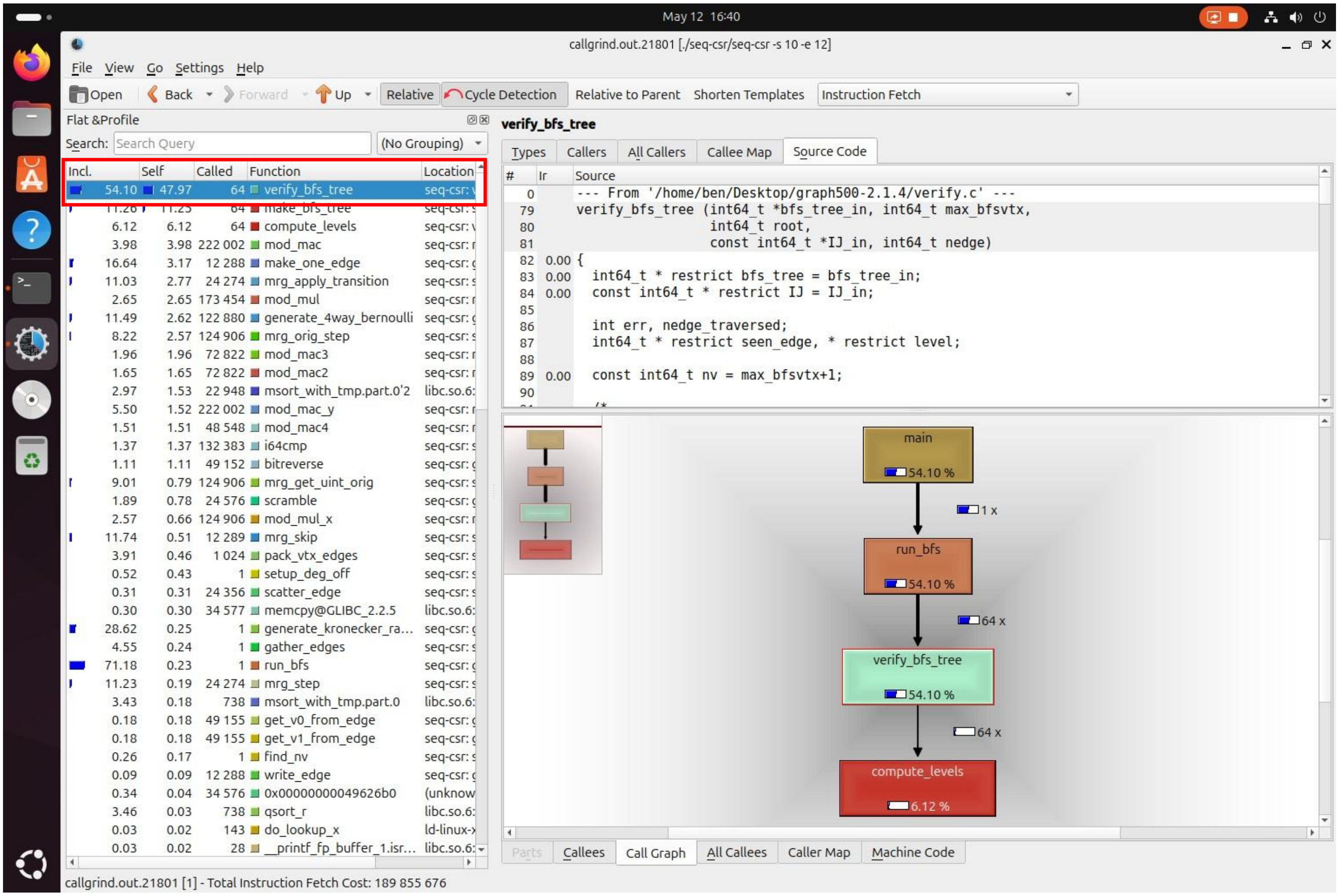
**- Primary Allocation Sources:**

- `a()` is the top contributor, allocating approximately **116,000 bytes** (**48.41%** of total heap).
- `a()` is repeatedly called via `b()→c()→d()→e()`, forming a recursive call chain that leads to layered memory allocations.
- All function calls ultimately originate from `main()`, making it the root cause of heap memory growth.

**- Potential Memory Leak**: Since no `free()` operations are observed, the memory is never deallocated during execution. This behavior strongly suggests a memory leak, where memory is allocated repeatedly but not released, resulting in a cumulative increase in usage.

## 4. Callgrind

a. Please use kcachegrind GUI to indicate which function is most expensive in terms of time (**excluding the time of their callee functions**). Please include a screenshot of the call graph.



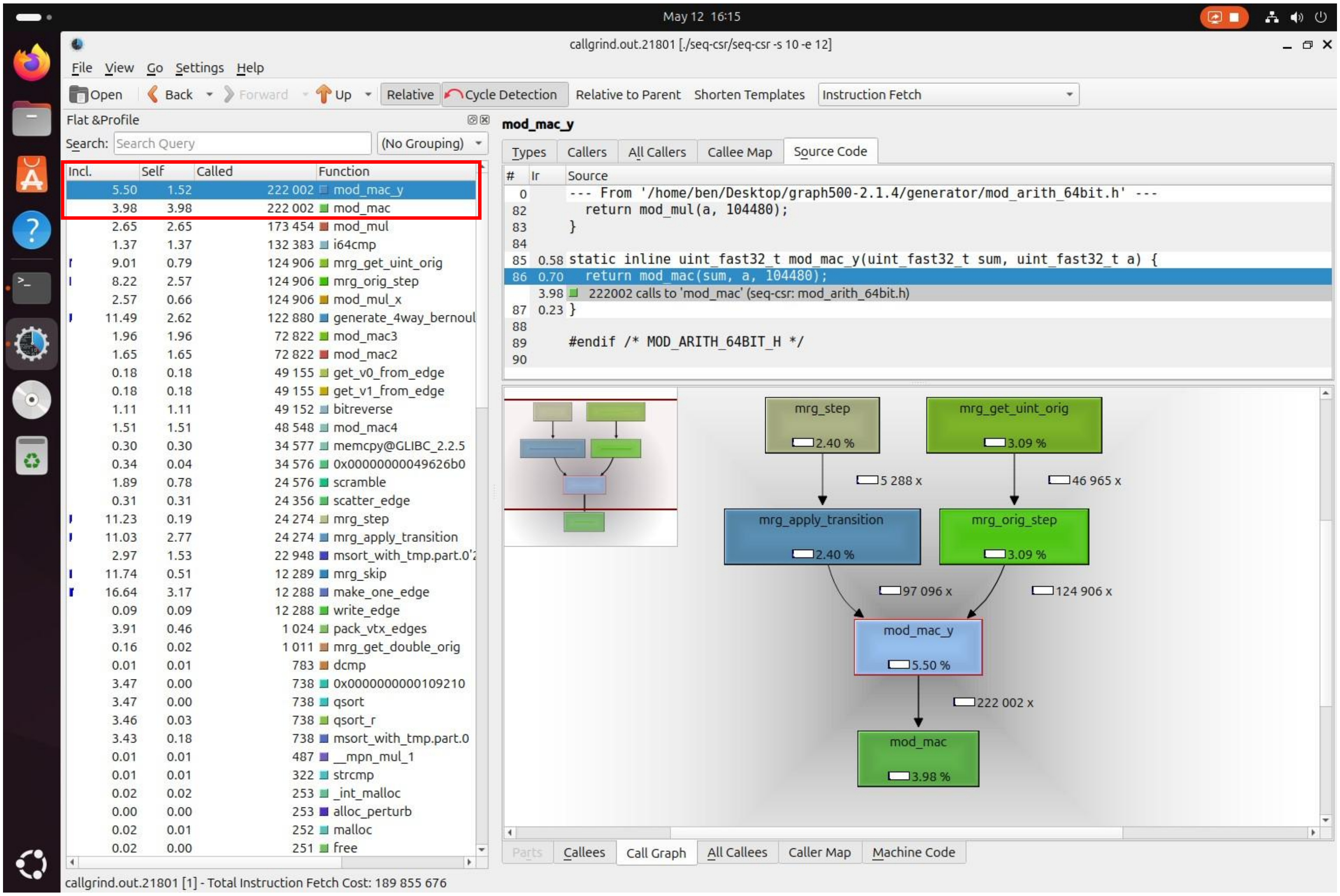
**Explanation:**

To determine which function is most expensive in terms of execution time (excluding time spent in its callees), we refer to the "Self" column in kcachegrind GUI. This value reflects the number of instructions executed directly within the function itself.

According to the kcachegrind GUI, the function verify_bfs_tree has the highest Self cost of 47.97%, making it the most time-consuming function in terms of its own operations. This indicates that nearly half of all instruction fetches during execution occurred within this function alone. It was invoked 64 times, each from the run_bfs() function.

The call graph shows the hierarchy: main() → run_bfs() → verify_bfs_tree() → compute_levels().

b. Point out which function is called most frequently, and identify its caller as well. Please include a screenshot of the call graph.




**Explanation:**

The most frequently called function is mod_mac_y, which was called 222002 times directly by its callers. Although mod_mac is also called 222002 times, it is only called internally within mod_mac_y.

From the call graph, mod_mac_y is called 222002 time by:

- mrg_apply_transition: 97,096 times
- mrg_orig_step: 124,906 times

## 5. Pytorch profiler

a. Please follow the tutorial to get the columns shown in the picture below. Please provide a screenshot of the analysis result, ensuring that the username and machinename are visible in the first line. Then, identify the top three functions in terms of CPU time excluding the time of their callee functions, **except for the model label** (e.g. model_inference in tutorials).

```
(pytorch_env) ben@ben-server:~$ python test2.py
---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
                       Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg       CPU Mem  Self CPU Mem    # of Calls
---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
               aten::addmm        52.10%       8.510ms        56.10%       9.163ms     186.993us       3.66 Mb       3.66 Mb            49
          aten::bernoulli_        10.01%       1.635ms        10.01%       1.635ms      96.154us           0 b           0 b            17
           model_inference         9.36%       1.529ms       100.00%      16.333ms      16.333ms       6.09 Mb      -5.63 Mb             1
               aten::copy_         7.06%       1.154ms         7.06%       1.154ms       8.807us           0 b           0 b           131
                 aten::bmm         2.05%     335.458us         2.07%     337.504us      18.750us     673.88 Kb     673.88 Kb            18
           aten::clamp_min         1.85%     302.395us         1.85%     302.395us      50.399us       1.27 Mb       1.27 Mb             6
                 aten::mul         1.42%     232.024us         1.42%     232.024us      13.648us     900.00 Kb     900.00 Kb            17
                 aten::add         1.34%     218.978us         1.34%     218.978us      12.881us     900.00 Kb     900.00 Kb            17
              aten::einsum         1.19%     194.623us         7.93%       1.294ms      71.912us       2.10 Mb           0 b            18
    aten::native_layer_norm         1.12%     182.561us         1.34%     219.010us      14.601us     798.19 Kb         768 b            15
            aten::_softmax         1.05%     171.399us         1.05%     171.399us      19.044us     205.88 Kb     205.88 Kb             9
                aten::view         0.90%     146.264us         0.90%     146.264us       0.710us           0 b           0 b           206
                aten::tril         0.86%     140.302us         0.86%     140.302us     140.302us         576 b         576 b             1
               aten::empty         0.77%     125.341us         0.77%     125.341us       1.266us       2.88 Mb       2.88 Mb            99
                aten::div_         0.73%     119.502us         1.14%     185.556us      10.915us           8 b         -60 b            17
             aten::reshape         0.61%      99.454us         5.24%     856.204us       6.850us       1.90 Mb           0 b           125
              aten::linear         0.60%      97.645us        58.06%       9.483ms     193.521us       3.66 Mb           0 b            49
             aten::permute         0.55%      90.178us         0.67%     110.028us       1.223us           0 b           0 b            90
           aten::unsqueeze         0.52%      84.698us         0.59%      96.300us       2.534us           0 b           0 b            38
        aten::index_select         0.46%      74.819us         0.55%      90.630us      22.658us     216.00 Kb     216.00 Kb             4
---------------------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------  ------------
Self CPU time total: 16.333ms
```

**Explanation:**

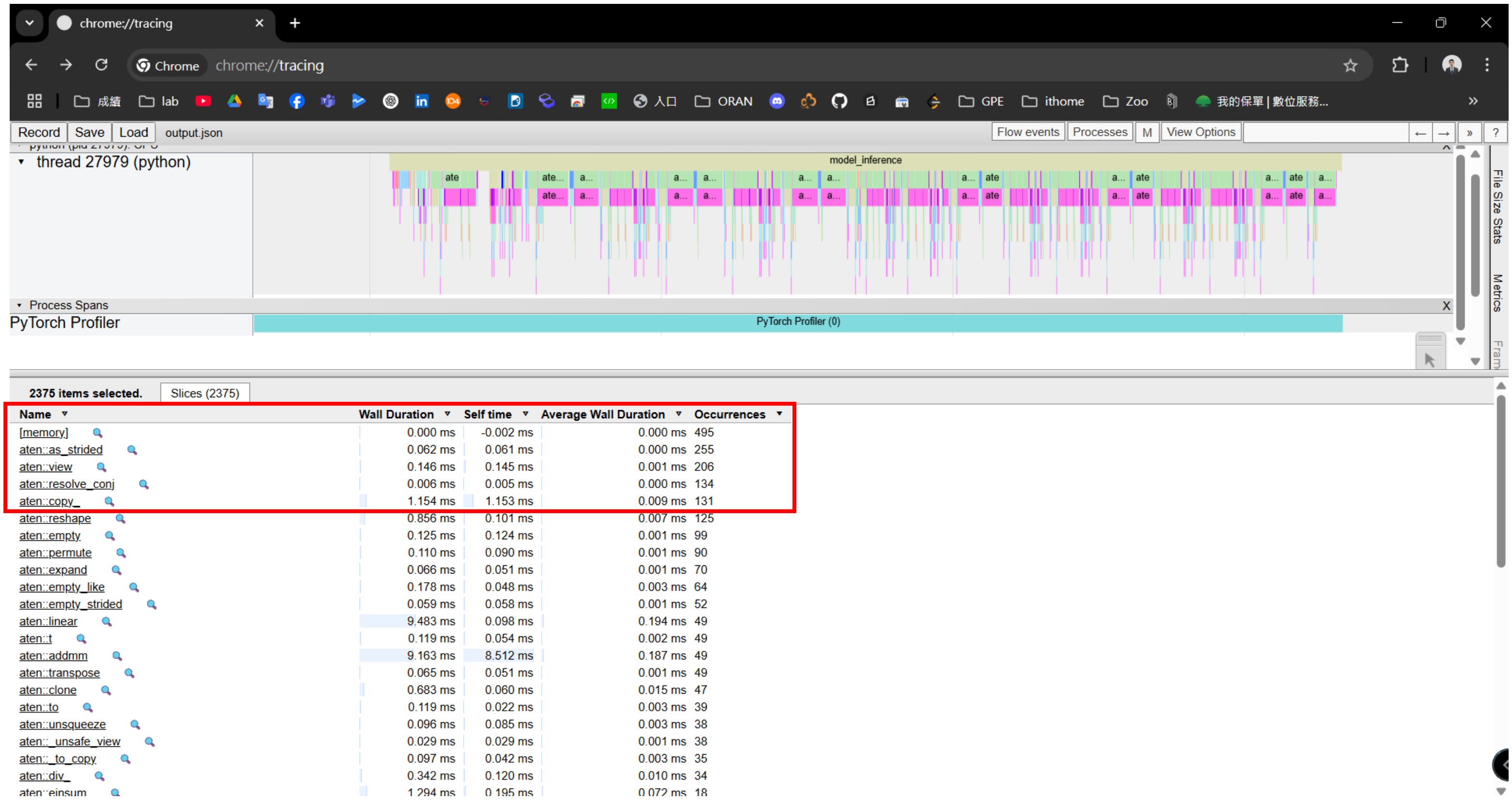
The top three functions can be find based on the "Self CPU %" or "Self CPU" columns.

1. aten::addmm (Self CPU %: 52.10%, Self CPU total: 8.510ms)

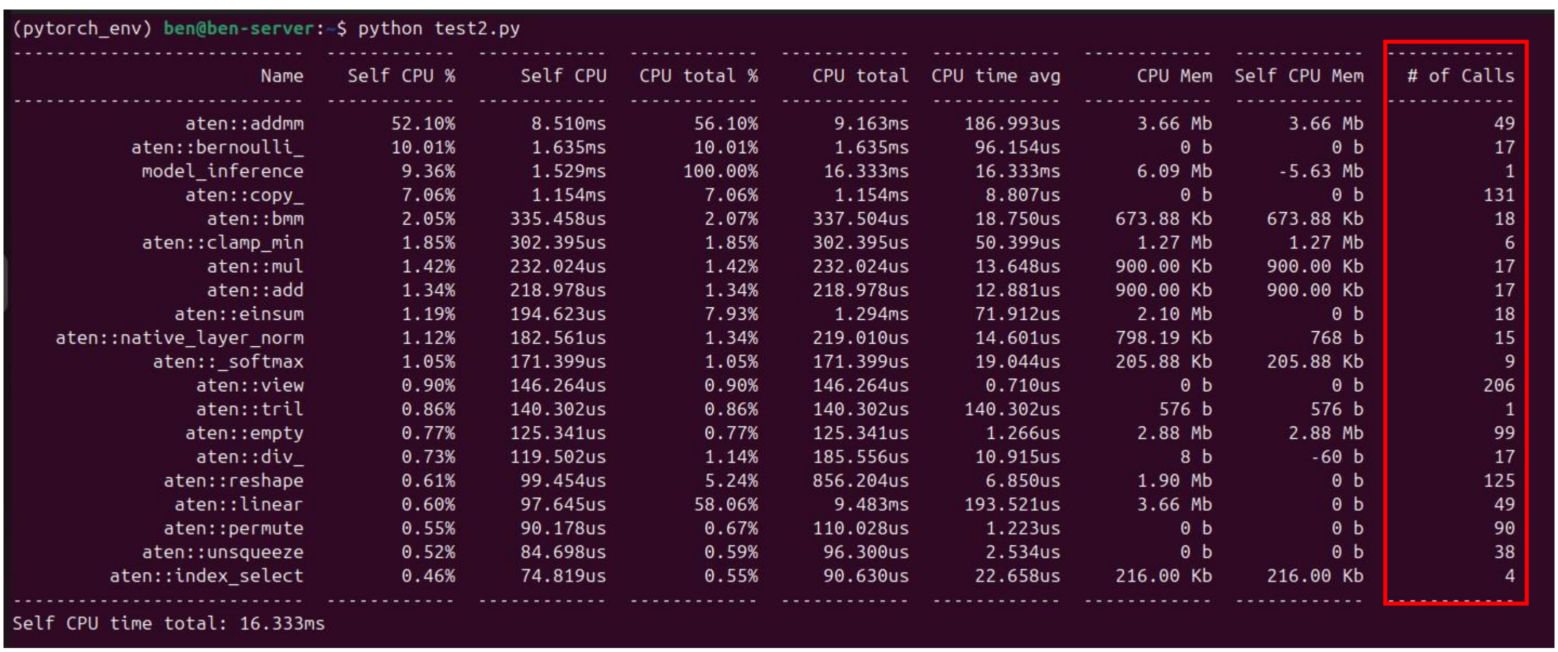
2. aten::bernoulli_ (Self CPU %: 10.01%, Self CPU total: 1.635ms)

3. aten::copy_ (Self CPU %: 7.06%, Self CPU total: 1.154ms)

b. Output the profiling results to <output>.json and analyze in Chrome trace viewer. Take a screenshot of the visualization and point out **which two functions (colors)** appear the most (in terms of time), **except for the model label** (e.g. model_inference in tutorials).




| Name | Wall Duration | Self time | Average Wall Duration | Occurrences |
|---|---|---|---|---|
| [memory] | 0.000 ms | -0.002 ms | 0.000 ms | 495 |
| aten::as_strided | 0.062 ms | 0.061 ms | 0.000 ms | 255 |
| aten::view | 0.146 ms | 0.145 ms | 0.001 ms | 206 |
| aten::resolve_conj | 0.006 ms | 0.005 ms | 0.000 ms | 134 |
| aten::copy_ | 1.154 ms | 1.153 ms | 0.009 ms | 131 |
| aten::reshape | 0.856 ms | 0.101 ms | 0.007 ms | 125 |
| aten::empty | 0.125 ms | 0.124 ms | 0.001 ms | 99 |
| aten::permute | 0.110 ms | 0.090 ms | 0.001 ms | 90 |
| aten::expand | 0.066 ms | 0.051 ms | 0.001 ms | 70 |
| aten::empty_like | 0.178 ms | 0.048 ms | 0.003 ms | 64 |
| aten::empty_strided | 0.059 ms | 0.058 ms | 0.001 ms | 52 |
| aten::linear | 9.483 ms | 0.098 ms | 0.194 ms | 49 |
| aten::t | 0.119 ms | 0.054 ms | 0.002 ms | 49 |
| aten::addmm | 9.163 ms | 8.512 ms | 0.187 ms | 49 |
| aten::transpose | 0.065 ms | 0.051 ms | 0.001 ms | 49 |
| aten::clone | 0.683 ms | 0.060 ms | 0.015 ms | 47 |
| aten::to | 0.119 ms | 0.022 ms | 0.003 ms | 39 |
| aten::unsqueeze | 0.096 ms | 0.085 ms | 0.003 ms | 38 |
| aten::_unsafe_view | 0.029 ms | 0.029 ms | 0.001 ms | 38 |
| aten::_to_copy | 0.097 ms | 0.042 ms | 0.003 ms | 35 |
| aten::div_ | 0.342 ms | 0.120 ms | 0.010 ms | 34 |
| aten::einsum | 1.294 ms | 0.195 ms | 0.072 ms | 18 |



(pytorch_env) ben@ben-server:~$ python test2.py

| Name | Self CPU % | Self CPU | CPU total % | CPU total | CPU time avg | CPU Mem | Self CPU Mem | # of Calls |
|---|---|---|---|---|---|---|---|---|
| aten::addmm | 52.10% | 8.510ms | 56.10% | 9.163ms | 186.993us | 3.66 Mb | 3.66 Mb | 49 |
| aten::bernoulli_ | 10.01% | 1.635ms | 10.01% | 1.635ms | 96.154us | 0 b | 0 b | 17 |
| model_inference | 9.36% | 1.529ms | 100.00% | 16.333ms | 16.333ms | 6.09 Mb | -5.63 Mb | 1 |
| aten::copy_ | 7.06% | 1.154ms | 7.06% | 1.154ms | 8.807us | 0 b | 0 b | 131 |
| aten::bmm | 2.05% | 335.458us | 2.07% | 337.504us | 18.750us | 673.88 Kb | 673.88 Kb | 18 |
| aten::clamp_min | 1.85% | 302.395us | 1.85% | 302.395us | 50.399us | 1.27 Mb | 1.27 Mb | 6 |
| aten::mul | 1.42% | 232.024us | 1.42% | 232.024us | 13.648us | 900.00 Kb | 900.00 Kb | 17 |
| aten::add | 1.34% | 218.978us | 1.34% | 218.978us | 12.881us | 900.00 Kb | 900.00 Kb | 17 |
| aten::einsum | 1.19% | 194.623us | 7.93% | 1.294ms | 71.912us | 2.10 Mb | 0 b | 18 |
| aten::native_layer_norm | 1.12% | 182.561us | 1.34% | 219.010us | 14.601us | 798.19 Kb | 768 b | 15 |
| aten::_softmax | 1.05% | 171.399us | 1.05% | 171.399us | 19.044us | 205.88 Kb | 205.88 Kb | 9 |
| aten::view | 0.90% | 146.264us | 0.90% | 146.264us | 0.710us | 0 b | 0 b | 206 |
| aten::tril | 0.86% | 140.302us | 0.86% | 140.302us | 140.302us | 576 b | 576 b | 1 |
| aten::empty | 0.77% | 125.341us | 0.77% | 125.341us | 1.266us | 2.88 Mb | 2.88 Mb | 99 |
| aten::div_ | 0.73% | 119.502us | 1.14% | 185.556us | 10.915us | 8 b | -60 b | 17 |
| aten::reshape | 0.61% | 99.454us | 5.24% | 856.204us | 6.850us | 1.90 Mb | 0 b | 125 |
| aten::linear | 0.60% | 97.645us | 58.06% | 9.483ms | 193.521us | 3.66 Mb | 0 b | 49 |
| aten::permute | 0.55% | 90.178us | 0.67% | 110.028us | 1.223us | 0 b | 0 b | 90 |
| aten::unsqueeze | 0.52% | 84.698us | 0.59% | 96.300us | 2.534us | 0 b | 0 b | 38 |
| aten::index_select | 0.46% | 74.819us | 0.55% | 90.630us | 22.658us | 216.00 Kb | 216.00 Kb | 4 |

Self CPU time total: 16.333ms

**Explanation:**

Chrome Trace Viewer shows the two functions that appear the most, which we can see the "Occurrences" (corresponding to the "# of Calls" column in the 5-1 results). We ignore memory-related events and PyTorch functions.

1. aten::view (# of Calls: 206)

2. aten::copy_ (# of Calls: 131)