# Lab Report

## ECPE 170 – Computer Systems and Networks – Spring 2017

**Name:** STEVE GUERRER0

**Lab Topic:** PERFORMANCE MEASUREMENT (LAB #: 05)

(1) Create a table that shows the real, user, and system times measured for the bubble and tree sort algorithms.

| TIME TYPES | BUBBLE SORT TIME (100,000) elements | TREE SORT TIME (100,000) elements |
|---|---|---|
| REAL | 34.088 seconds | 0.065 seconds |
| USER | 34.020 seconds | 0.052 seconds |
| SYSTEM | 0.008 seconds | 0.012 seconds |

(2) In the sorting program, what actions take user time?
The processing time that the cpu took to complete the sorting algorithm. The disk retrieving and writing data.

(3) In the sorting program, what actions take kernel time?
Operating system services requested by the process.

(4) Which algorithm is the fastest? *(It's so obvious, you don't even need a timer)*
The binary tree sort is much faster than bubble sort.

(5) Create a table that shows the total Instruction Read (IR) counts for the bubble and tree sort routines. (In the text output format, IR is the default unit used. In the GUI program, un-check the "% Relative" button to have it display absolute counts of instruction reads).
ARRAY SIZE 100000

| | Bubble Sort | Tree Sort |
|---|---|---|
| Total IR | 164,783,180,820 | 98,717,805 |

(6) Create a table that shows the top 3 most active functions for the bubble and tree sort programs by IR count (excluding `main()`) - Sort by the "self" column if you're using kcachegrind, so that you see the IR counts for *just* that function, and not include the IR count for functions that it calls.

| Bubble Sort (Active Functions) | Tree Sort (Active Functions) |
|---|---|
| 1) bubbleSort (164,755,225,641) | 1) insert_element (49 373 069) |
| 2) Random_r(2,596,775) | 2) _int_malloc (14 899 996) |
| 3) VerifySort(1,900,001) | 3) _int_free (7 400 000) |

(7) Create a table that shows, for the bubble and tree sort programs, the most CPU intensive line that is part of the most CPU intensive function. (i.e. First, take the most active function for a program. Second, find the annotated source code for that function. Third, look inside the whole function code - what is the most active line? If by some chance it happens to be another function, drill down one more level and repeat.)

| Algorithm | Line # | Activity | Code |
|---|---|---|---|
| Bubble Sort | 40 | 74 999 250 000 | if (array_start[j-1] > array_start[j]); |
| Tree Sort | 115 | 9 476 760 | if ( element < ( *sr ) -> element ) |

(8) Show the Valgrind output file for the merge sort with the intentional memory leak. Clearly highlight the line where Valgrind identified where the block was originally allocated.

```
==10085== Memcheck, a memory error detector
==10085== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==10085== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==10085== Command: ./sorting_program merge
==10085== Parent PID: 8817
==10085==
==10085==
==10085== HEAP SUMMARY:
==10085==     in use at exit: 400,000 bytes in 1 blocks
==10085==   total heap usage: 2 allocs, 1 frees, 401,024 bytes allocated
==10085==
==10085== 400,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==10085==    at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10085==    by 0x40085C: main (sorting.c:42)
==10085==
==10085== LEAK SUMMARY:
==10085==    definitely lost: 400,000 bytes in 1 blocks
==10085==    indirectly lost: 0 bytes in 0 blocks
==10085==      possibly lost: 0 bytes in 0 blocks
==10085==    still reachable: 0 bytes in 0 blocks
==10085==         suppressed: 0 bytes in 0 blocks
==10085==
==10085== For counts of detected and suppressed errors, rerun with: -v
==10085== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

(9) How many bytes were leaked in the buggy program?
400,000 bytes were leaked in the buggy program.

(10) Show the Valgrind output file for the merge sort after you fixed the intentional leak.

```
==10151== Memcheck, a memory error detector
==10151== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==10151== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==10151== Command: ./sorting_program merge
==10151== Parent PID: 8817
==10151==
==10151==
==10151== HEAP SUMMARY:
==10151==     in use at exit: 0 bytes in 0 blocks
==10151==   total heap usage: 2 allocs, 2 frees, 401,024 bytes allocated
==10151==
==10151== All heap blocks were freed -- no leaks are possible
==10151==
==10151== For counts of detected and suppressed errors, rerun with: -v
==10151== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(11) Show the Valgrind output file for your tree sort.

==10183== Memcheck, a memory error detector
==10183== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==10183== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==10183== Command: ./sorting_program tree
==10183== Parent PID: 8817
==10183==
==10183==
==10183== HEAP SUMMARY:
==10183== in use at exit: 0 bytes in 0 blocks
==10183== total heap usage: 100,001 allocs, 100,001 frees, 2,401,024 bytes allocated
==10183==
==10183== All heap blocks were freed -- no leaks are possible
==10183==
==10183== For counts of detected and suppressed errors, rerun with: -v
==10183== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

(12) How many seconds of real, user, and system time were required to complete the amplify program execution with Lenna image (Lenna_org_1024.pgm)? Document the command used to measure this.

| TIME TYPES | Time to amplify Lenna_org_1024.pgm image |
|---|---|
| REAL | 1.394 seconds |
| USER | 0.856 seconds |
| SYSTEM | 0.528 seconds |

(13) Research and answer: why does real time != (user time + system time)?

Since the user time is the actual processing your hardware components do. The system or 'kernel' time does the managing of those calls to read and write to a disk or put the code into the cpu between thoses processes. So both get summed up to make up real time.

(14) In your report, put the output image for the Lenna image titled output<somenumber>.pgm.

Output2048.pgm

(15) Excluding **main()** and everything before it, what are the top three functions run by amplify executable when you **do** count sub-functions in the total? Document the commands used to measure this. Tip: Sorting by the "Incl" (Inclusive) column in kcachegrind should be what you want.

Commands:

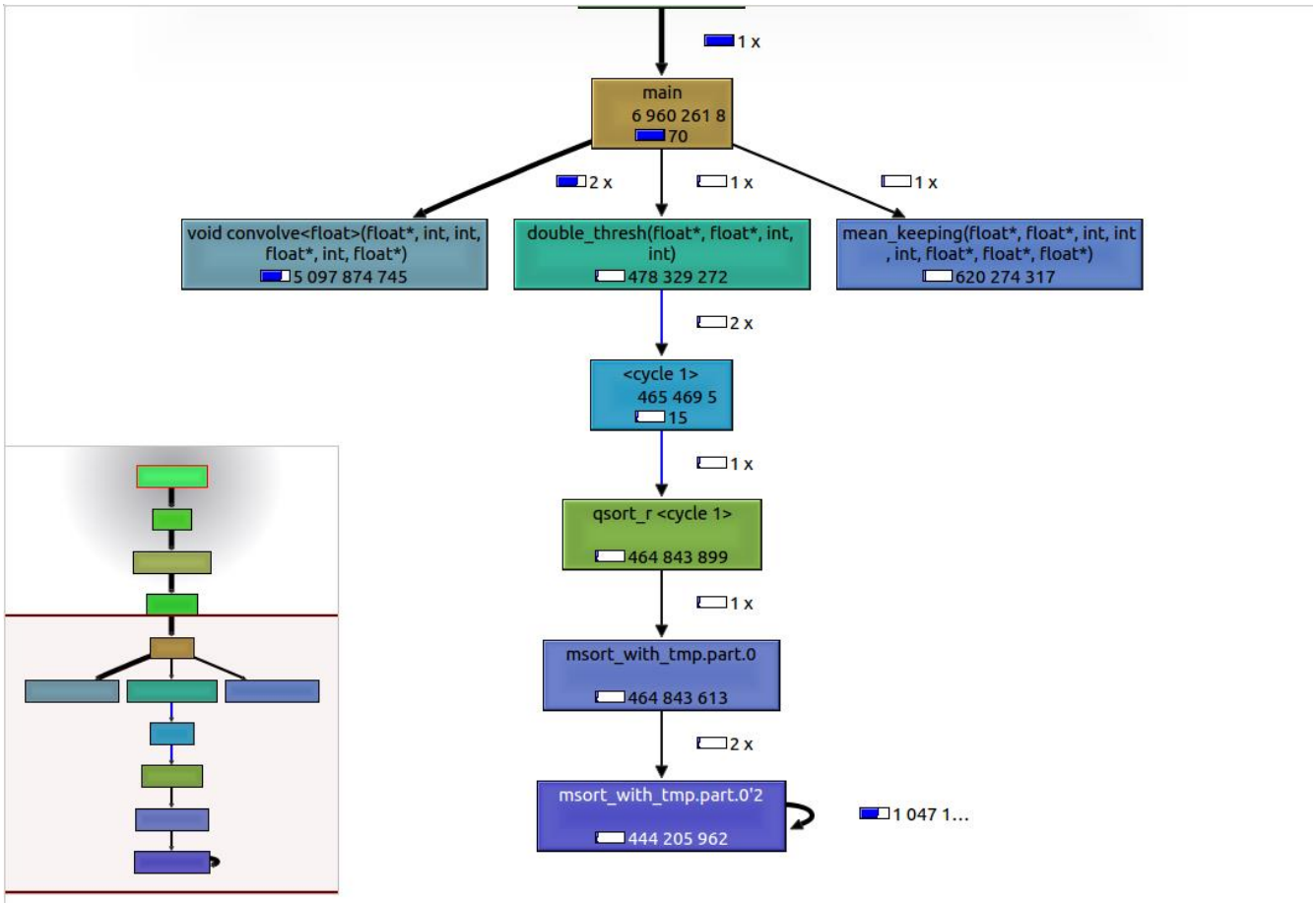
valgrind --tool=callgrind --dump-instr=yes --callgrind-out-file=callgrind.out ./amplify IMAGES/Lenna_org_1024.pgm 11 1.1 2

callgrind_annotate --auto=yes --threshold=100 callgrind.out > callgrind_results.txt

kcachegrind callgrind.out &

| Amplify Executable (Active Functions) |
|---|
| 1) Void convolve (5 097 922 722) |
| 2) Mean_keeping(620 280 155) |
| 3) Double_thresh (478 333 774) |

(16) Include a screen capture that shows the kcachegrind utility displaying the **callgraph** for the amplify executable, starting at main(), and going down for several levels.



(17) Which function dominates the execution time? What fraction of the total execution time does this function occupy?

Line 53: if(((i+k) < img_height) && ((j+l) < img_width) && ((i+k)>= 0) && ((j+l)>=0))

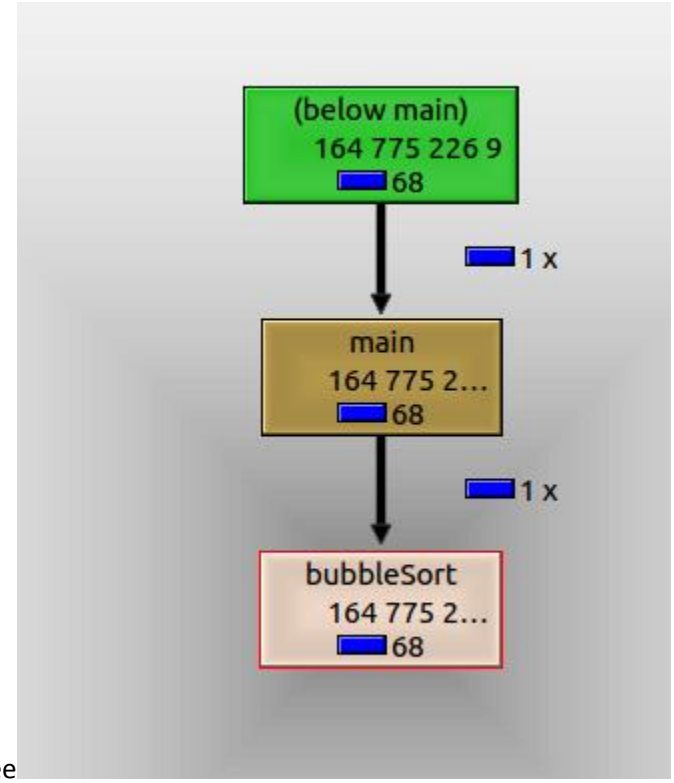
Fraction of total time:

(2 348 521 224) / (6 962 390 673) = .3373

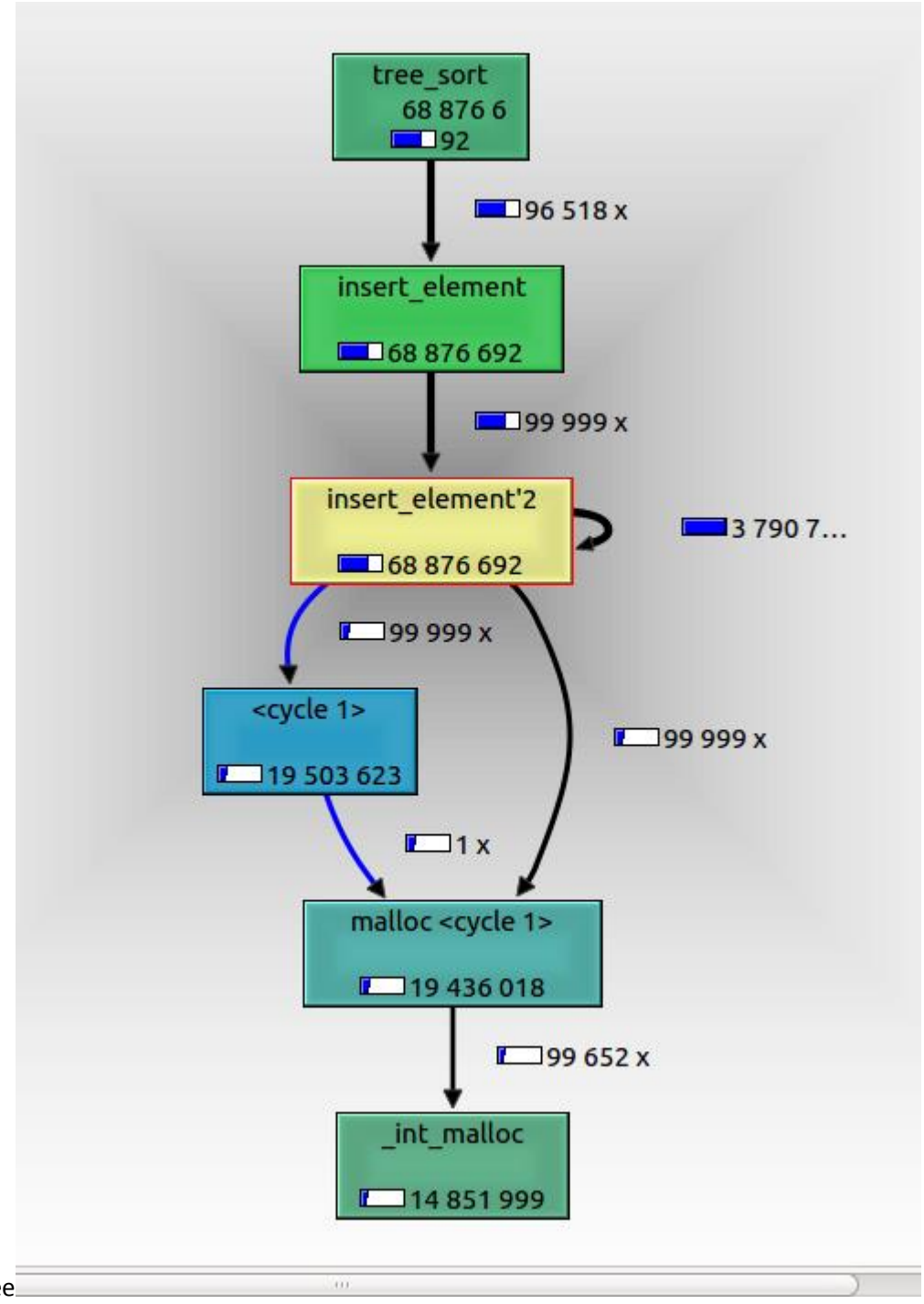
(18) Does this program show any signs of memory leaks? **Optional:** Fix it for 10 points extra credit.

Yes, there are huge memory leaks. Memcheck shows 25,166,844 bytes in 7 blocks definitely lost.

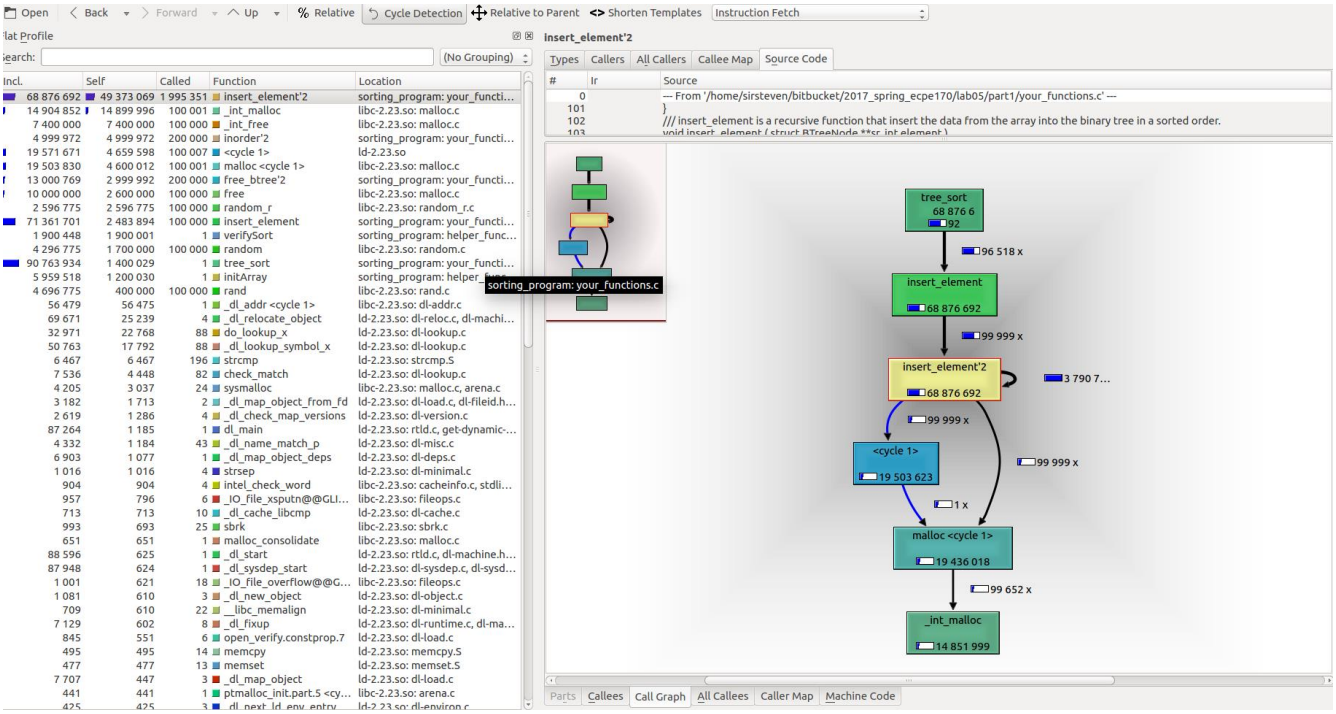
Appendix:

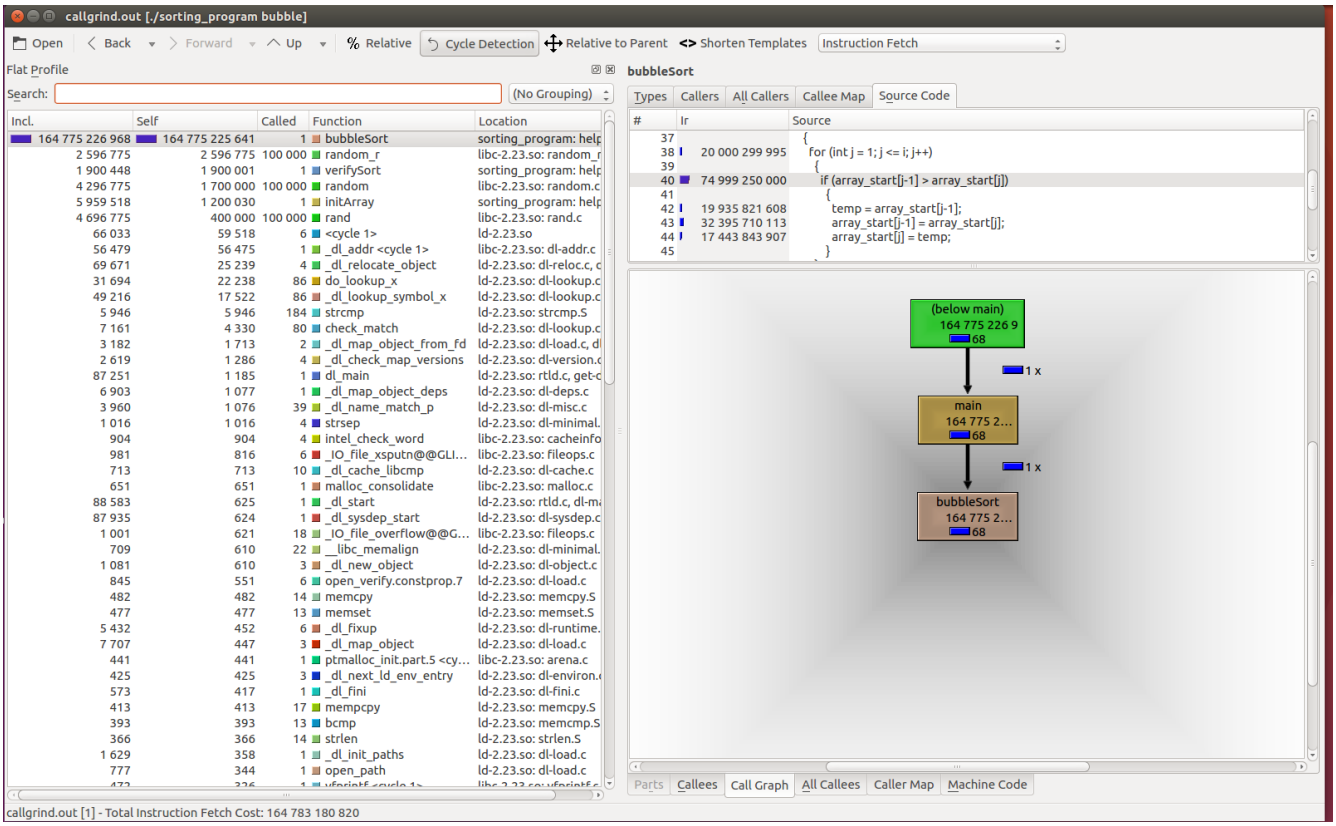



BubbleSort CallTree




TreeSort CallTree


Open
Back
Forward
Up
Relative
Cycle Detection
Relative to Parent
Shorten Templates
Instruction Fetch
Flat Profile
Search:
(No Grouping)
Incl.
Self
Called
Function
Location
68 876 692
49 373 069
1 995 351
insert_element'2
sorting_program: your_functi...
14 904 852
14 899 996
100 001
_int_malloc
libc-2.23.so: malloc.c
7 400 000
7 400 000
100 000
_int_free
libc-2.23.so: malloc.c
4 999 972
4 999 972
200 000
inorder'2
sorting_program: your_functi...
19 571 671
4 659 598
100 007
<cycle 1>
ld-2.23.so
19 503 830
4 600 012
100 001
malloc <cycle 1>
libc-2.23.so: malloc.c
13 000 769
2 999 992
200 000
free_btree'2
sorting_program: your_functi...
10 000 000
2 600 000
100 000
free
libc-2.23.so: malloc.c
2 596 775
2 596 775
100 000
random_r
libc-2.23.so: random_r.c
71 361 701
2 483 894
100 000
insert_element
sorting_program: your_functi...
1 900 448
1 900 001
1
verifySort
sorting_program: helper_func...
4 296 775
1 700 000
100 000
random
libc-2.23.so: random.c
90 763 934
1 400 029
1
tree_sort
sorting_program: your_functi...
5 959 518
1 200 030
1
initArray
sorting_program: helper_
4 696 775
400 000
100 000
rand
libc-2.23.so: rand.c
56 479
56 475
1
_dl_addr <cycle 1>
ld-2.23.so: dl-addr.c
69 671
25 239
4
_dl_relocate_object
ld-2.23.so: dl-reloc.c, dl-machi...
32 971
22 768
88
do_lookup_x
ld-2.23.so: dl-lookup.c
50 763
17 792
88
_dl_lookup_symbol_x
ld-2.23.so: dl-lookup.c
6 467
6 467
196
strcmp
ld-2.23.so: strcmp.S
7 536
4 448
82
check_match
ld-2.23.so: dl-lookup.c
4 205
3 037
24
sysmalloc
libc-2.23.so: malloc.c, arena.c
3 182
1 713
2
_dl_map_object_from_fd
ld-2.23.so: dl-load.c, dl-fileid.h...
2 619
1 286
4
_dl_check_map_versions
ld-2.23.so: dl-version.c
87 264
1 185
1
dl_main
ld-2.23.so: rtld.c, get-dynamic-...
4 332
1 184
43
_dl_name_match_p
ld-2.23.so: dl-misc.c
6 903
1 077
1
_dl_map_object_deps
ld-2.23.so: dl-deps.c
1 016
1 016
4
strsep
ld-2.23.so: dl-minimal.c
904
904
4
intel_check_word
libc-2.23.so: cacheinfo.c, stdli...
957
796
6
_IO_file_xsputn@@GLI...
libc-2.23.so: fileops.c
713
713
10
_dl_cache_libcmp
ld-2.23.so: dl-cache.c
993
693
25
sbrk
libc-2.23.so: sbrk.c
651
651
1
malloc_consolidate
libc-2.23.so: malloc.c
88 596
625
1
_dl_start
ld-2.23.so: rtld.c, dl-machine.h...
87 948
624
1
_dl_sysdep_start
ld-2.23.so: dl-sysdep.c, dl-sysd...
1 001
621
18
_IO_file_overflow@@G...
libc-2.23.so: fileops.c
1 081
610
3
_dl_new_object
ld-2.23.so: dl-object.c
709
610
22
__libc_memalign
ld-2.23.so: dl-minimal.c
7 129
602
8
_dl_fixup
ld-2.23.so: dl-runtime.c, dl-ma...
845
551
6
open_verify.constprop.7
ld-2.23.so: dl-load.c
495
495
14
memcpy
ld-2.23.so: memcpy.S
477
477
13
memset
ld-2.23.so: memset.S
7 707
447
3
_dl_map_object
ld-2.23.so: dl-load.c
441
441
1
ptmalloc_init.part.5 <cy...
libc-2.23.so: arena.c
425
425
3
_dl_next_ld_env_entry
ld-2.23.so: dl-environ.c
insert_element'2
Types
Callers
All Callers
Callee Map
Source Code
#
Ir
Source
0
--- From '/home/sirsteven/bitbucket/2017_spring_ecpe170/lab05/part1/your_functions.c' ---
101
}
102
/// insert_element is a recursive function that insert the data from the array into the binary tree in a sorted order.
sorting_program: your_functions.c
tree_sort
68 876 6
92
96 518 x
insert_element
68 876 692
99 999 x
insert_element'2
68 876 692
3 790 7...
99 999 x
<cycle 1>
19 503 623
99 999 x
1 x
malloc <cycle 1>
19 436 018
99 652 x
_int_malloc
14 851 999
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code


callgrind.out [./sorting_program bubble]
Open
Back
Forward
Up
Relative
Cycle Detection
Relative to Parent
Shorten Templates
Instruction Fetch
Flat Profile
Search:
(No Grouping)
Incl.
Self
Called
Function
Location
164 775 226 968
164 775 225 641
1
bubbleSort
sorting_program: help
2 596 775
2 596 775
100 000
random_r
libc-2.23.so: random_r
1 900 448
1 900 001
1
verifySort
sorting_program: help
4 296 775
1 700 000
100 000
random
libc-2.23.so: random.c
5 959 518
1 200 030
1
initArray
sorting_program: help
4 696 775
400 000
100 000
rand
libc-2.23.so: rand.c
66 033
59 518
6
<cycle 1>
ld-2.23.so
56 479
56 475
1
_dl_addr <cycle 1>
libc-2.23.so: dl-addr.c
69 671
25 239
4
_dl_relocate_object
ld-2.23.so: dl-reloc.c, d
31 694
22 238
86
do_lookup_x
ld-2.23.so: dl-lookup.c
49 216
17 522
86
_dl_lookup_symbol_x
ld-2.23.so: dl-lookup.c
5 946
5 946
184
strcmp
ld-2.23.so: strcmp.S
7 161
4 330
80
check_match
ld-2.23.so: dl-lookup.c
3 182
1 713
2
_dl_map_object_from_fd
ld-2.23.so: dl-load.c, dl
2 619
1 286
4
_dl_check_map_versions
ld-2.23.so: dl-version.c
87 251
1 185
1
dl_main
ld-2.23.so: rtld.c, get-c
6 903
1 077
1
_dl_map_object_deps
ld-2.23.so: dl-deps.c
3 960
1 076
39
_dl_name_match_p
ld-2.23.so: dl-misc.c
1 016
1 016
4
strsep
ld-2.23.so: dl-minimal.
904
904
4
intel_check_word
libc-2.23.so: cacheinfo
981
816
6
_IO_file_xsputn@@GLI...
libc-2.23.so: fileops.c
713
713
10
_dl_cache_libcmp
ld-2.23.so: dl-cache.c
651
651
1
malloc_consolidate
libc-2.23.so: malloc.c
88 583
625
1
_dl_start
ld-2.23.so: rtld.c, dl-m
87 935
624
1
_dl_sysdep_start
ld-2.23.so: dl-sysdep.c
1 001
621
18
_IO_file_overflow@@G...
libc-2.23.so: fileops.c
709
610
22
__libc_memalign
ld-2.23.so: dl-minimal.
1 081
610
3
_dl_new_object
ld-2.23.so: dl-object.c
845
551
6
open_verify.constprop.7
ld-2.23.so: dl-load.c
482
482
14
memcpy
ld-2.23.so: memcpy.S
477
477
13
memset
ld-2.23.so: memset.S
5 432
452
6
_dl_fixup
ld-2.23.so: dl-runtime.
7 707
447
3
_dl_map_object
ld-2.23.so: dl-load.c
441
441
1
ptmalloc_init.part.5 <cy...
libc-2.23.so: arena.c
425
425
3
_dl_next_ld_env_entry
ld-2.23.so: dl-environ.
573
417
1
_dl_fini
ld-2.23.so: dl-fini.c
413
413
17
mempcpy
ld-2.23.so: memcpy.S
393
393
13
bcmp
ld-2.23.so: memcmp.S
366
366
14
strlen
ld-2.23.so: strlen.S
1 629
358
1
_dl_init_paths
ld-2.23.so: dl-load.c
777
344
1
open_path
ld-2.23.so: dl-load.c
bubbleSort
Types
Callers
All Callers
Callee Map
Source Code
#
Ir
Source
37
{
38
20 000 299 995
for (int j = 1; j <= i; j++)
39
{
40
74 999 250 000
if (array_start[j-1] > array_start[j])
41
{
42
19 935 821 608
temp = array_start[j-1];
43
32 395 710 113
array_start[j-1] = array_start[j];
44
17 443 843 907
array_start[j] = temp;
45
}
(below main)
164 775 226 9
68
1 x
main
164 775 2...
68
1 x
bubbleSort
164 775 2...
68
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code
callgrind.out [1] - Total Instruction Fetch Cost: 164 783 180 820