

Prova Finale

Algoritmi e Strutture Dati

AA 2017/2018

Tutors

COGNOMI da P a R



Niccolo' Raspa

niccolo.raspa@mail.polimi.it

COGNOMI da S a Z



Mattia Salvini

mattia.salvini@mail.polimi.it

- Email con oggetto: **[ProvaFinaleAPI]**
- Usate la mail ufficiale / specificate la vostra matricola

Tutors

- Usate il form nel sito:

[Overview](#)

Communication

TASK_PROVA

[Statement](#)

[Submissions](#)

TUTORIAL

[Statement](#)

[Submissions](#)

[Documentation](#)

[Testing](#)

Questions

Subject

Text

Ask question

Reset

Agenda

- Informazioni pratiche sui tutorati
- Struttura della prova e valutazione
- Demo tool valutazione
- Consigli pratici implementazione
- Tool di supporto (gdb - valgrind - callgrind - kcachegrind)

Logistica Tutorati

- Indicativamente 1-2 a Luglio e 1-2 a Settembre
- Modalità:
 - ▶ Risposte a domande individuali
 - ▶ Guardando il vostro lavoro ci permette di essere più chiari
 - ▶ Possiamo rispondere a **solo a domande tecniche**

Deadlines

12 SETTEMBRE ore 24:00

- Per i laureandi di LUGLIO:

11 Luglio ore 24:00

- NO ESTENSIONI, NO RECUPERI

Struttura della prova

Implementazione in linguaggio C standard un interprete di Macchine di Turing non-deterministiche, nella variante a nastro singolo e solo accettori.



Un unico file sorgente contenente la vostra implementazione

- ▶ NO librerie esterne eccetto standard (stdio, stdlib, string, math, ecc...)
- ▶ Ma posso usare questa libreria per... ? **NO**

Input



- ▶ Unico file di testo fornito tramite lo standard input
- ▶ Contiene:
 - ▶ funzione di transizione
 - ▶ stati accettazione
 - ▶ numero max di passi per una singola computazione
 - ▶ serie di stringhe da far leggere alla macchina

Input

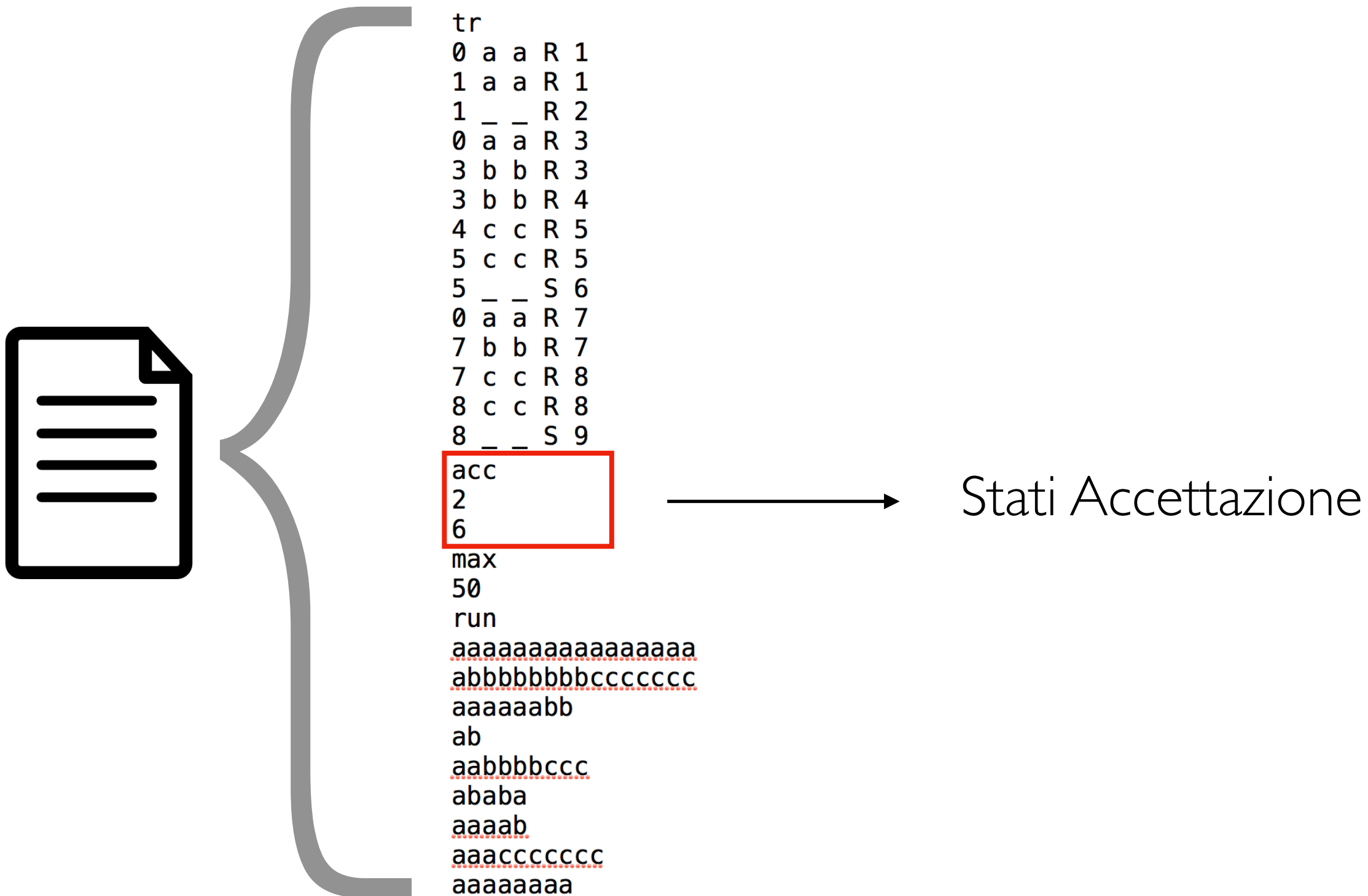


```
tr
0 a a R 1
1 a a R 1
1 _ _ R 2
0 a a R 3
3 b b R 3
3 b b R 4
4 c c R 5
5 c c R 5
5 _ _ S 6
0 a a R 7
7 b b R 7
7 c c R 8
8 c c R 8
8 _ _ S 9
acc
2
6
max
50
run
aaaaaaaaaaaaaaaaa
bbbbbbbbbccccccc
aaaaabb
ab
aabbbbbbccc
ababa
aaaab
aaaccccccc
aaaaaaaa
```

Descrizione TM

- ▶ simboli di nastro sono dei *char*
- ▶ stati sono *int*
- ▶ "_" indica il simbolo "blank".
- ▶ I caratteri "L", "R", "S" indicano movimento della testina.
- ▶ La macchina parte sempre:
 - ▶ stato 0
 - ▶ primo carattere della stringa

Input



Input



tr
0 a a R 1
1 a a R 1
1 _ _ R 2
0 a a R 3
3 b b R 3
3 b b R 4
4 c c R 5
5 c c R 5
5 _ _ S 6
0 a a R 7
7 b b R 7
7 c c R 8
8 c c R 8
8 _ _ S 9

acc
2
6

max
50

run

aaaaaaaaaaaaaaaa
bbbbbbbbbcccccc
aaaaabb
ab
aabbbbccc
ababa
aaaab
aaaccccccc
aaaaaaa

Numero Max Passi

Input



tr
0 a a R 1
1 a a R 1
1 _ _ R 2
0 a a R 3
3 b b R 3
3 b b R 4
4 c c R 5
5 c c R 5
5 _ _ S 6
0 a a R 7
7 b b R 7
7 c c R 8
8 c c R 8
8 _ _ S 9

acc
2
6
max
50

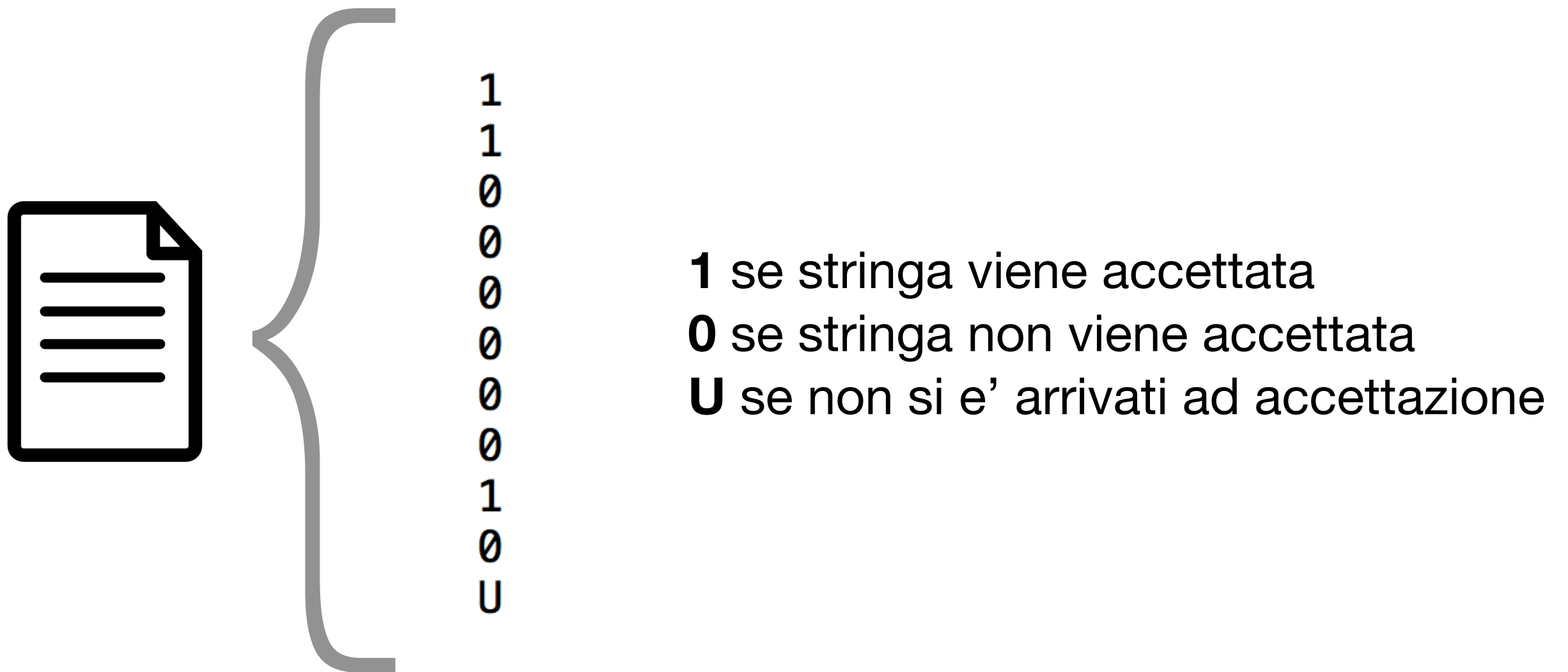
run
aaaaaaaaaaaaaaaa
bbbbbbbbbcccccc
aaaaabb
ab
aabbbbccc
ababa
aaaab
aaaccccccc
aaaaaaa



Stringhe

Output

- Unico file di testo fornito tramite lo standard output



Valutazione della prova

- Upload del codice sorgente
- Compilazione e valutazione automatica
- Valutazione su casi di test pubblici
- Valutazione su casi di test privati
- Esito

Valutazione della prova

TASK PUBLIC

▼ Subtask 1 (0 / 0)				
#	Outcome	Details	Execution time	Memory used
1	Correct	Output is correct	0.000 s	128 KiB

Non valutato

TASK PRIVATE

▼ Subtask 2 (3 / 3)				
#	Outcome	Details	Execution time	Memory used
2	Correct	Output is correct	2.199 s	512 KiB

▼ Subtask 3 (1 / 1)				
#	Outcome	Details	Execution time	Memory used
3	Correct	Output is correct	5.349 s	640 KiB

Valutazione
3 + 1 + 1

▼ Subtask 4 (1 / 1)				
#	Outcome	Details	Execution time	Memory used
4	Correct	Output is correct	18.648 s	896 KiB

Valutazione della prova

▼ Subtask 1			(0 / 0)	
#	Outcome	Details	Execution time	Memory used
1	Correct	Output is correct	0.000 s	128 KiB

- ▶ Implementazione deve essere **CORRETTA & EFFICIENTE**
 - ▶ tempo di esecuzione
 - ▶ memoria occupata

Casi di Test Privati

- ▶ Test pubblici da intendersi di base
- ▶ Effettuati su centinaia di stringhe
- ▶ Stringhe di lunghezza arbitraria
- ▶ Verifica della correttezza dell'output
- ▶ Vincoli di tempo e memoria incrementali

Assumptions

- ▶ Potete assumere che i file di input siano sintatticamente corretti e coerenti con le specifiche
- ▶ La funzione di transizione può non essere ordinata per numero di stato
- ▶ Non ci saranno archi uscenti da uno stato di accettazione
- ▶ Se esiste lo stato N esistono anche gli stani $N-1, N-2, \dots, 0$
- ▶ Non ci sono vincoli riguardo alla lunghezza del file di input e delle stringhe di input
- ▶ Il parametro U in caso di macchina non-deterministica si riferisce al singolo percorso non-deterministico

Be aware...



We're watching you

NON COPIATE

- ▶ Verranno eseguiti controlli sui sorgenti
- ▶ **Tutti** i progetti coinvolti verranno annullati
- ▶ Non condividere il proprio sorgente
 - ▶ NB: caricare il proprio sorgente su GitHub = condividere



Demo Verificatore

<http://dum-e.deib.polimi.it>

Implementazione



Un unico file sorgente contenente la vostra implementazione

- ▶ Non iniziare subito a scrivere codice
- ▶ Iniziare ad impostare la soluzione (prima di settembre)
- ▶ Pensare alle strutture dati per soddisfare le specifiche sia funzionali che di complessità
- ▶ Il codice deve essere leggibile e ben commentato
- ▶ Sfruttare il paradigma procedurale (divide et impera)
- ▶ Esecuzione sequenziale (no multithreading)

Compilazione

Some details

Type	Batch	
Time limit	1 second	
Memory limit	256 MiB	
Compilation commands	C11 / gcc	<pre>/usr/bin/gcc -DEVAL -std=c11 -O2 -pipe -static -s -o Tutorial Tutorial.c -lm</pre>
Tokens	You have an infinite number of tokens for this task.	

Attachments



input_000.txt

11 B

plain text document

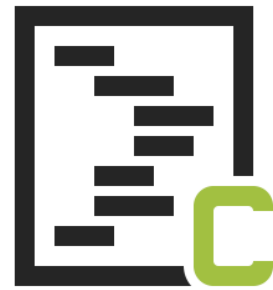


output_000.txt

6 B

plain text document

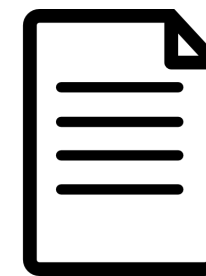
Compilazione e Test in Locale



main.c



input_000.txt



output_000.txt

- Testare sempre in locale prima di caricare il codice (no brute force)

COMPILAZIONE

```
gcc -g -std=c11 -Wall -Werror -o main main.c
```

ESECUZIONE

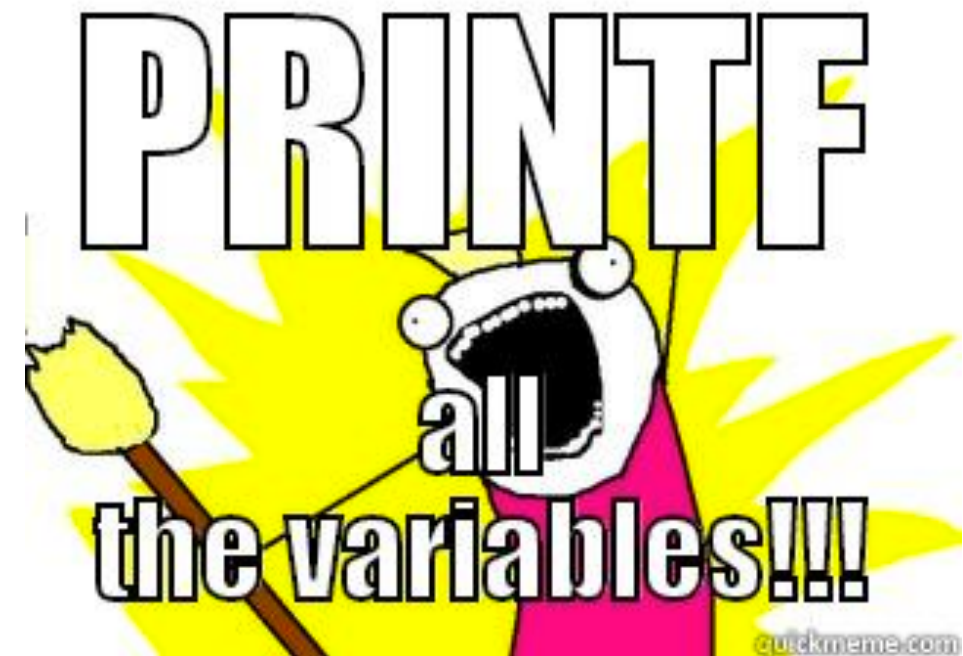
```
cat input_000.txt | ./main > my_output.txt
```

CONTROLLO

```
diff output_000.txt my_output.txt
```


Debugging with GDB

- ▶ “GNU Debugger”
- ▶ It allows you to inspect what the program is doing at a certain point during execution.
- ▶ Useful for logical errors
- ▶ <https://www.gnu.org/software/gdb/documentation/>



GDB
The GNU Project
Debugger

GDB - Comandi Utili

- ▶ **start** - begin executing your program
- ▶ **list** - examine your source code from within the debugger
- ▶ **step** - execute the next line of your program
- ▶ **next** - execute the next line of your program, but if it's a subroutine call, treat the entire subroutine as a single line
- ▶ **print** - examine the contents of a variable
- ▶ **x** - examine memory directly
- ▶ **watch, rwatch** - set a watch for when a variable is written or read: return to the debugger once this happens
- ▶ **break** - set a breakpoint: return to the debugger when this line of code is about to be executed
- ▶ **info watch** - show info on watchpoints
- ▶ **info break** - show info on breakpoints
- ▶ **delete #** - delete watchpoint or breakpoint “#”
- ▶ **continue** - continue from a breakpoint, watchpoint, step, next, etc.; basically begin running your program from where it left off
- ▶ **set var name=value** set the value of variable “name” to “value”
- ▶ **backtrace** - show the call frames for your program
- ▶ **frame #** - set the current frame to #. Variables you reference etc. will be those within that context.
- ▶ **quit** - leave the debugger

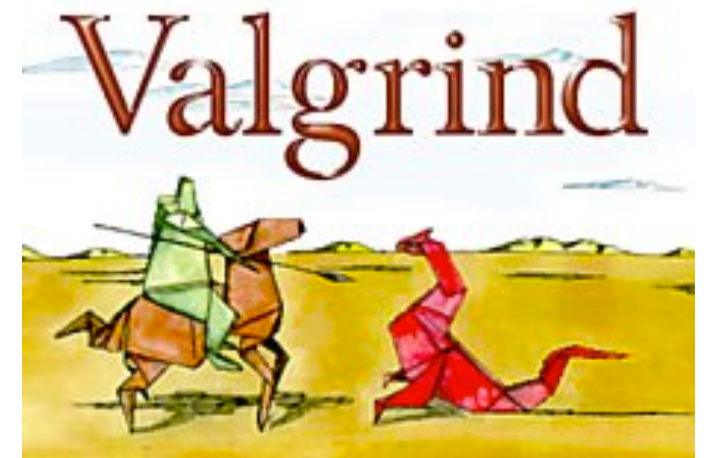
GDB in action

```
##### # # # # (gdb) break main
# # ## # # # Breakpoint 1 at 0x8048426: file hello10.c, line 6.
# # # # # (gdb) run
# ##### # # # # Starting program: /home/gary/hello10
# # # # #
# # # # # Breakpoint 1, main () at hello10.c:6
# # # # # 6 for(i=0;i<10;i++)
##### # # #####
(gdb) █

#####
# # ##### ##### # # ##### #####
# # # # # # # # # # # # # #
# # ##### ##### # # # # # #
# # # # # # # # # # # # #####
# # # # # # # # # # # # #####
# # # # # # # # # # # # # #
##### ##### ##### ##### ##### ##### ##### # #
```

Valgrind

- ▶ Valgrind is a multipurpose code profiling and memory debugging tool for Linux.
- ▶ It allows you to run your program in Valgrind's own environment that monitors memory usage such as calls to malloc and free
- ▶ Can detect many memory-related errors commonly found in C / C++
- ▶ <http://valgrind.org/>



valgrind --leak-check=yes eseguibile <args>

Understand Valgrind output

- ▶ Memory Leak = there is a memory chunk allocated but it's not accessible
 - ▶ **Still Reachable**
 - Still have pointer(s) to the start of the block
 - No problem. You can still free it. Non reported by default
 - ▶ **Definitely Lost**
 - No pointers to memory block can be found.
 - You can't free it.
 - ▶ **Indirectly Lost**
 - All pointers to the block are lost
 - e.g. if the root node of a binary tree is lost, all the children are indirectly lost

What can Valgrind detect?

► Memory Leaks

Remember to compile with -g flag!

```
#include <stdlib.h>
int main()
{
    char *x = malloc(100);
    return 0;
}
```

```
==2330== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2330==    at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==2330==    by 0x804840F: main (example1.c:5)
```

What can Valgrind detect?

► Invalid Pointers Use

```
#include <stdlib.h>

int main()
{
    char *x = malloc(10);
    x[10] = 'a';
    return 0;
}
```

```
==9814== Invalid write of size 1
==9814==    at 0x804841E: main (example2.c:6)
==9814== Address 0x1BA3607A is 0 bytes after a block of size 10 alloc'd
==9814==    at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
==9814==    by 0x804840F: main (example2.c:5)
```


What can Valgrind detect?

► Use of Uninitialized Variables

```
#include <stdio.h>

int main()
{
    int x;
    if(x == 0)
    {
        printf("X is zero");
    }
    return 0;
}
```

```
==17943== Conditional jump or move depends on uninitialised value(s)
==17943==      at 0x804840A: main (example3.c:6)
```


What can Valgrind detect?

► Use of Uninitialized Variables (cont.)

```
#include <stdio.h>

int foo(int x)
{
    if(x < 10)
    {
        printf("x is less than 10\n");
    }
}

int main()
{
    int y;
    foo(y);
}
```

```
==4827== Conditional jump or move depends on uninitialised value(s)
==4827==      at 0x8048366: foo (example4.c:5)
==4827==      by 0x8048394: main (example4.c:14)
```

What **can't** Valgrind detect?

- ▶ Valgrind doesn't perform bounds checking on static arrays (allocated on the stack).

```
int main()
{
    char x[10];
    x[11] = 'a';
}
```

**Valgrind doesn't detect static
buffer overflow**

Callgrind

- ▶ Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph.
- ▶ Callgrind can also be used to find performance problems
 - ▶ What code lines eat up most instructions
- ▶ Basic Usage:

```
valgrind --tool=callgrind [callgrind options] program [program options]
```

- ▶ While the simulation is running, you can observe execution with:

```
callgrind_control -b
```

Callgrind

- ▶ The profile data is written out to a file at program termination.
 - ▶ `callgrind.out.XXX`
- ▶ The data file contains information about the calls made in the program among the functions executed, with **Instruction Read** (Ir) event counts.
- ▶ To generate a function-by-function summary from the profile data file:

`callgrind_annotate --auto=yes callgrind.out.XXX`
- ▶ More info: <http://valgrind.org/docs/manual/cl-manual.html>

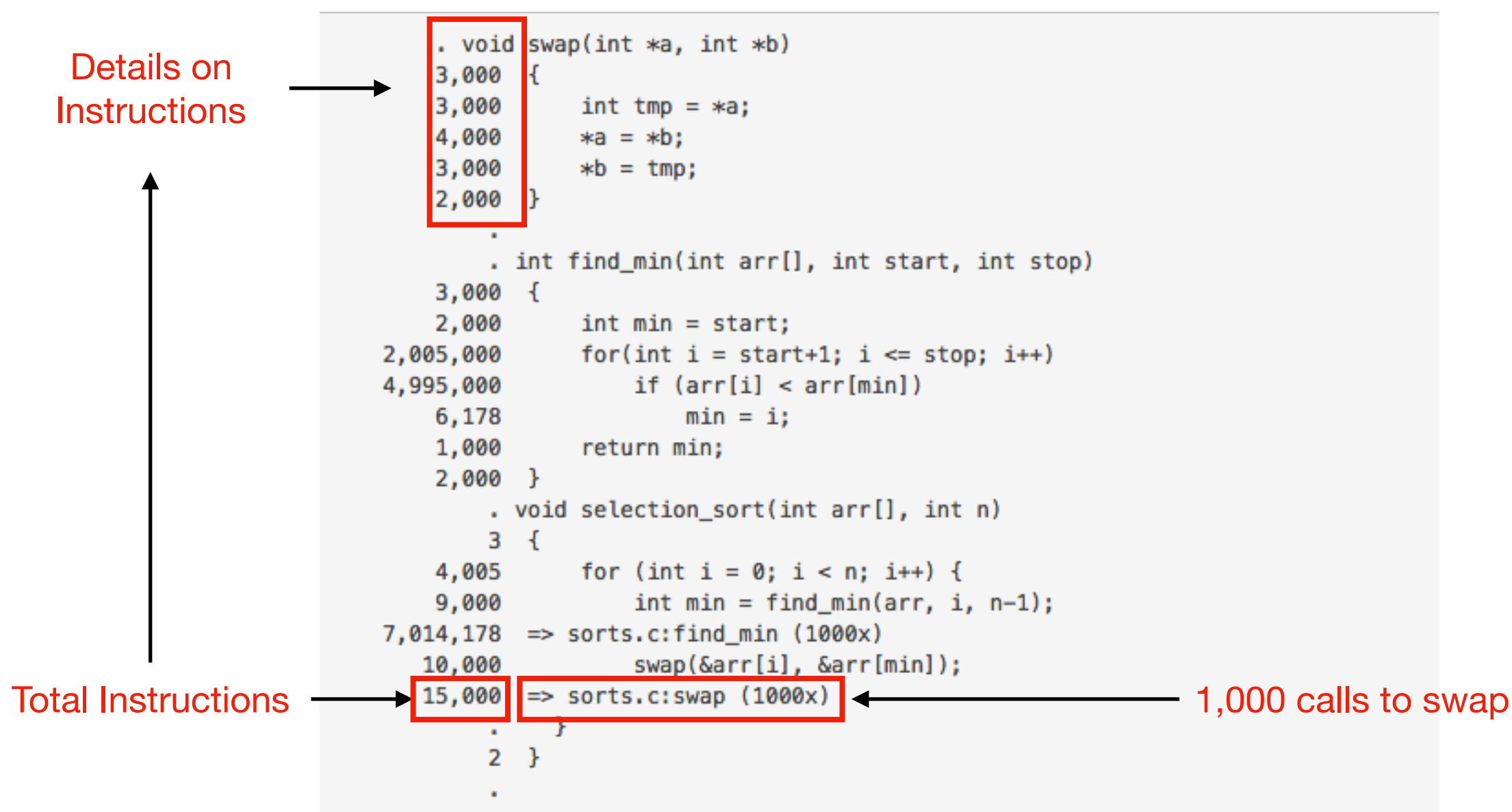
Interpreting the result

- ▶ The IR counts are basically the count of assembly instructions executed.
- ▶ A single C statement can translate to 1, 2, or several assembly instructions.

```
. void swap(int *a, int *b)
3,000 {
3,000     int tmp = *a;
4,000     *a = *b;
3,000     *b = tmp;
2,000 }
.
. int find_min(int arr[], int start, int stop)
3,000 {
2,000     int min = start;
2,005,000     for(int i = start+1; i <= stop; i++)
4,995,000         if (arr[i] < arr[min])
6,178             min = i;
1,000     return min;
2,000 }
. void selection_sort(int arr[], int n)
3 {
4,005     for (int i = 0; i < n; i++) {
9,000         int min = find_min(arr, i, n-1);
7,014,178 => sorts.c:find_min (1000x)
10,000         swap(&arr[i], &arr[min]);
15,000 => sorts.c:swap (1000x)
.     }
2 }
.
```

Interpreting the result

- ▶ The IR counts are basically the count of assembly instructions executed.
- ▶ A single C statement can translate to 1, 2, or several assembly instructions.



Function Summary

- ▶ The `callgrind_annotate` includes a function call summary, sorted in order of decreasing count:

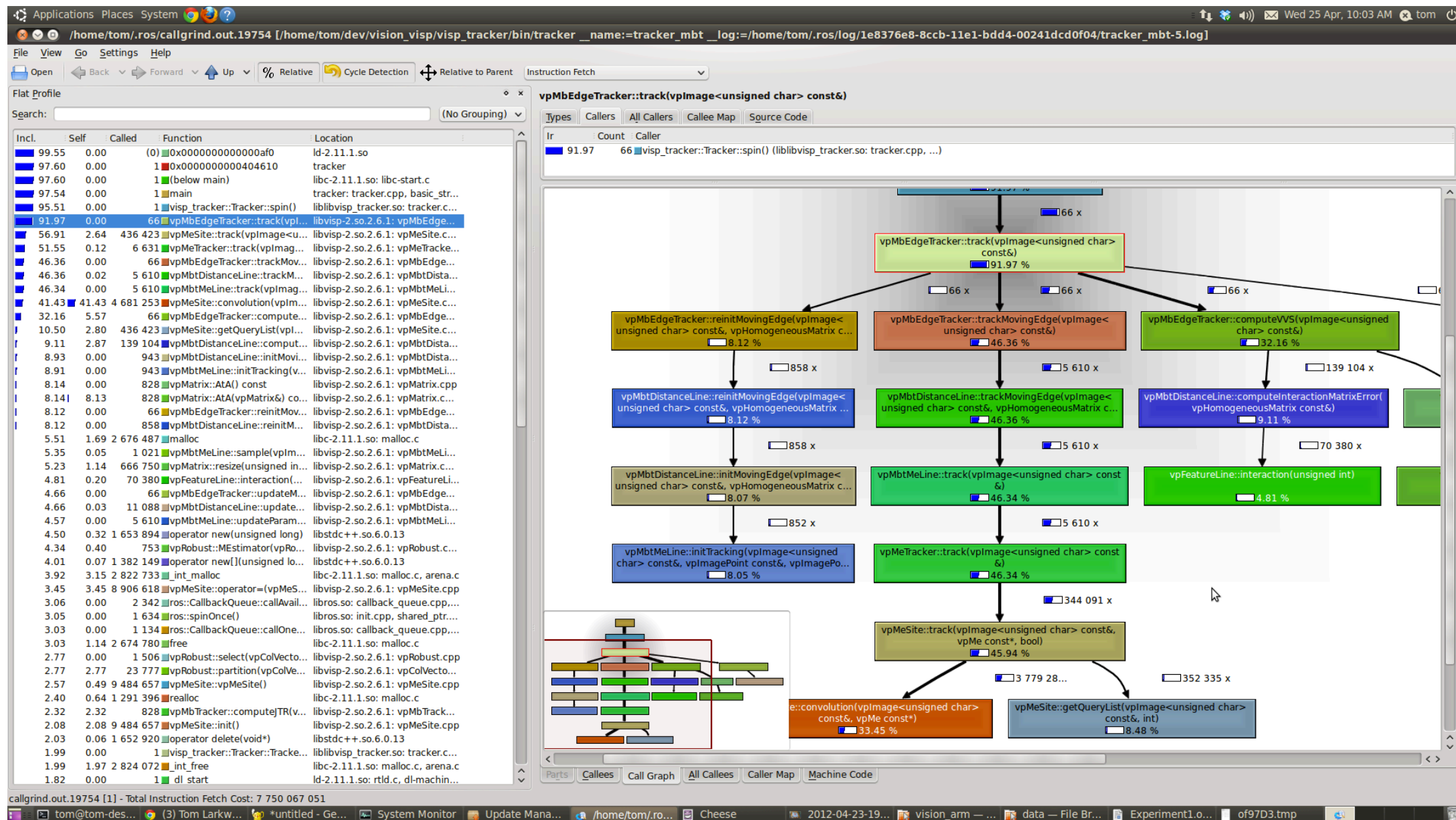
```
7,014,178  sorts.c:find_min [sorts]
 25,059    ???:_do_lookup_x [/lib/ld-2.5.so]
 23,010    sorts.c:selection_sort [sorts]
 20,984    ???:_dl_lookup_symbol_x [/lib/ld-2.5.so]
 15,000    sorts.c:swap [sorts]
```

- ▶ Counts are *exclusive*, they include only the time spent in that function and not in the functions that it calls.
- ▶ Exclusive counts is great way to highlight your bottlenecks
- ▶ `--inclusive=yes` to `callgrind_annotate` to make it *inclusive*

Kcachegrind

- ▶ Visualization tool for Callgrind output
- ▶ <https://kcachegrind.github.io/>

kcachegrind callgrind.out.XXX



Conclusioni

- ▶ Fate **sempre riferimento** alle pagine dei professori o al sito della prova per le comunicazioni ufficiali e le specifiche del progetto
- ▶ *Ma nelle slide dei tutor c'era scritto che... **non vale** come giustificazione*
- ▶ Slide Available: tinyurl.com/slide_api_18

COGNOMI da P a R



Niccolo' Raspa

niccolo.raspa@mail.polimi.it

COGNOMI da S a Z



Mattia Salvini

mattia.salvini@mail.polimi.it

Credits & More Details

- ▶ **Gdb:**

- ▶ <https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>
- ▶ <https://www.cs.cmu.edu/~gilpin/tutorial/>

- ▶ **Valgrind:**

- ▶ <https://www.cprogramming.com/debugging/valgrind.html>
- ▶ <http://valgrind.org/docs/manual/quick-start.html#quick-start.intro>

- ▶ **Callgrind:**

- ▶ <https://web.stanford.edu/class/cs107/guide/callgrind.html>
- ▶ <http://valgrind.org/docs/manual/cl-manual.html>

- ▶ **KCachegrind:**

- ▶ <https://kcachegrind.github.io/html/Home.html>