Preparing a Benchmark Driver

CS 165, Project in Algorithms and Data Structures
UC Irvine
Spring 2020

Presented by Rob Gevorkyan
What is a benchmark driver?

- An executable that runs a single function on a random input of a chosen size and outputs nothing.

- Example call:
  
  ```
  ./project1 merge_sort 32768
  ```
Why do we need one?

- Once we have a benchmark driver, we can run it repeatedly in a benchmark script and collect timing data for a variety of functions and input sizes, possibly including multiple of each size for averaging.

- Any function can be benchmarked independently of every other. If something goes wrong (exceptions, loss of power, etc.), you can resume benchmarking wherever you left off.
The basic idea

- Each of the functions being benchmarked is implemented in some external file(s).
- The driver is compiled with the function implementations and does the following in the main method:
  - generates a map from function names to the functions themselves. you can use std::map<std::string, fn_ptr>, where fn_ptr is a typedef for a function pointer with the signature matching your function (they should all be the same signature)
  - reads the command line arguments
  - converts any data as needed (e.g. strings to ints)
  - uses the name to function map to choose a function to execute
  - generates a random instance
  - run the mapped function on the random instance
Reading command line arguments

- To be able to use command line arguments, make your main method’s signature as shown below

- argc gives the number of elements in argv
- argc is always at least 1, since argv[0] is the name of the program itself
- for j > 1, argv[j] is the jth argument given to the program
- note: all the arguments are given as c strings, but these can easily be converted using the standard library to strings and integer values

```c
int main(int argc, char* argv[]) {
    ...
}
```
Creating the name to function map

- A convenient data structure for us to use is `std::map`
- A map stores (key, value) pairs. Feed it a key, and it returns a value.

```cpp
#include <map>

std::map<std::string, sort_fn> name_to_fn;
name_to_fn["insertion"] = insertion_sort;
name_to_fn["merge"] = merge_sort;
...
std::vector<int> nums = ...;
name_to_fn["insertion"](nums);
```

sorts nums with insertion sort
Function pointers

- C++ and other languages have support for references to functions in addition to data.
- A function pointer is like any other pointer, but it has a bizarre syntax. To make things more readable, we can use a typedef.

```cpp
typedef void (*sort_fn)(std::vector<int>& nums);
sort_fn insertion_sort_fn = insertion_sort;
std::vector<int> nums = ...;
insertion_sort_fn(nums);
```
Creating random instances

- Given an input size $n$, we can easily create a vector from 1 to $n$, in that order.

- To give each run some variety, we can do a random shuffle of this vector. To do this, we can use the random_shuffle method in the `<algorithm>` library. See `random_shuffle` for details and examples.

- You should have a dedicated function for creating an instance that returns a reference to the vector. You can use this in your main method.
Putting it all together

- With the name to function map created, the elements of argv in main determine how to create the random instance (by specifying size) and the function to use.

- The main method doesn’t care about the vector being sorted itself. It just runs the function on the vector. You should however test the logic separate from the driver.