DSA Homework 1 - Outline

1. Dynamic k-th integer problem
2. Concepts of makefile
    - Variable
    - Pattern
    - Foreach
3. How to test your program

Dynamic K-th Integer

- Given a sequence of N integers, for each input, your program should find out that what's the K-th smallest number in the recent M integers
- A generalized version of dynamic median number problem
- You can apply data structure `std::deque` and binary search algorithm to solve this problem

Dynamic K-th Integer - Input

- Line#1: Three space-separated integers N, M, K
  - N is the total length for the input sequence. $(1 \leq N \leq 200000)$
  - The recent M integers are we care about $(1 \leq M \leq 100000, M \leq N)$, and you are asked to find $Kth (1 \leq K \leq M)$ smallest integer in these integers.
- Line#2: A sequence consists of N integers $a_1, a_2, \ldots, a_n (1 \leq a_i \leq 2147483647)$

Dynamic K-th Integer - Output

- For each input, when there are at least M integers, output the Kth smallest integer in one line.
  - i.e. Output **N-M+1** lines

Dynamic K-th Integer - Example

- For example:
  - ```
    9 5 3
    3 4 2 1 5 5 1 1 5
    ```
- Outputs:
  - ```
    3
    4
    2
    1
    5
    ```

Dynamic K-th Integer - Example

3	4	2	1	5	5	1	1	5	1th	2th	3th	4th	5th	
3	4	2	1	5					1	2	3	4	5	ans=3
	4	2	1	5	5				1	2	4	5	5	ans=4
		2	1	5	5	1			1	1	2	5	5	ans=2
			1	5	5	1	1		1	1	1	5	5	ans=1
				5	5	1	1	5	1	1	5	5	5	ans=5

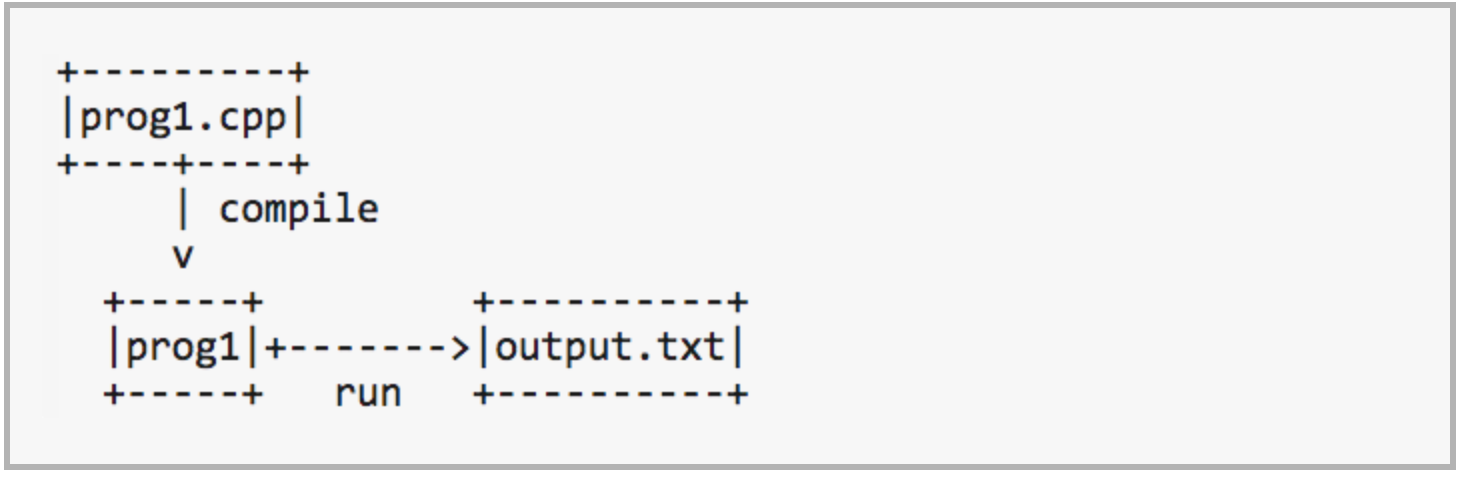
Deque Approach

1. Sort the array when it comes to m-th input
    - Since the array is sorted, `ary[K-1]` is the k-th integer
2. Maintain an ordered array dynamically
    - Need random access and insert element in middle
    - Use the STL container `std::deque<int>`

Deque Approach

3. Implement with deque:
    - `ary.insert(iterator, new_int)` for new integer
    - `ary.erase(iterator)` for delete oldest integer
        - Record the recent M integers so you can find the oldest integer
4. Since the array is sorted, you can implement **binary search** to find the index to insert/erase

Makefile - Example 1



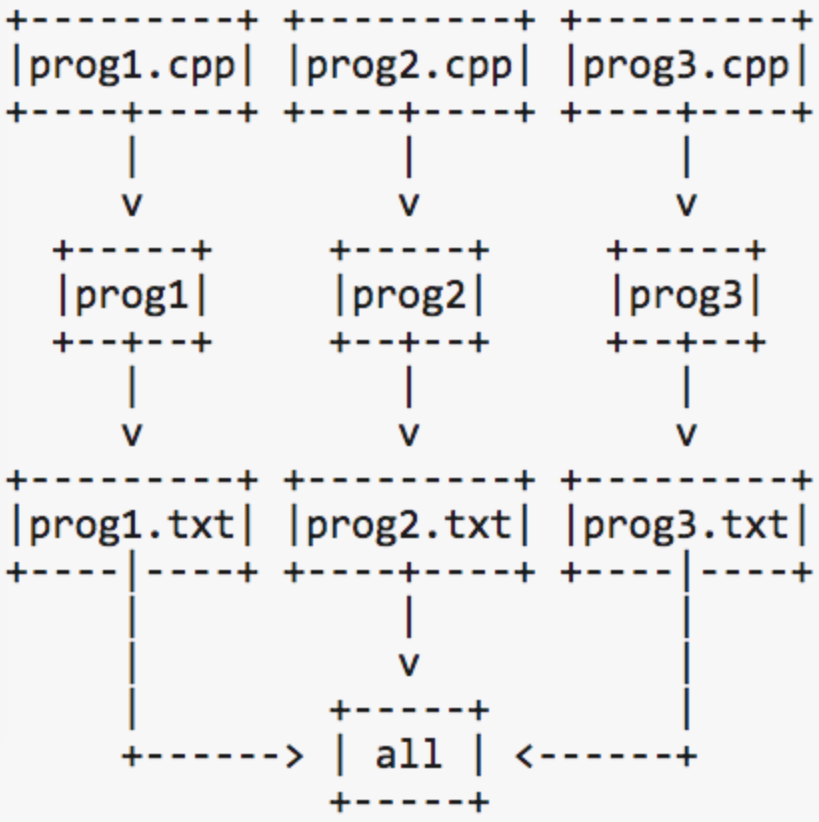
Makefile - Example 1

```
1  prog1: prog1.cpp
2      g++ -std=c++11 -O2 prog1.cpp -o prog1
3
4  output1.txt: prog1
5      ./prog1 > output1.txt
```

Makefile - How does it work?

1. `make` ensures "target" file is the newest
    - Determined by last modification time
2. Commands below target (called "recipe") can be something like `cp`, `echo`, `python`, `make`, ...etc, not only `g++`

Makefile - Example 2



Makefile - Example 2

```
all: output1.txt output2.txt output3.txt
	cat output1.txt output2.txt output3.txt

prog1: prog1.cpp
	g++ -std=c++11 -O2 prog1.cpp -o prog1
output1.txt: prog1
	./prog1 > output1.txt

prog2: prog2.cpp
	g++ -std=c++11 -O2 prog2.cpp -o prog2
output2.txt: prog2
	./prog2 > output2.txt

prog3: prog3.cpp
	g++ -std=c++11 -O2 prog3.cpp -o prog3
output3.txt: prog3
	./prog3 > output3.txt
```

Makefile - Example 2

```
➜ make_demo git:(master) make
g++ -std=c++11 -O2 prog1.cpp -o prog1
./prog1 > output1.txt
g++ -std=c++11 -O2 prog2.cpp -o prog2
./prog2 > output2.txt
g++ -std=c++11 -O2 prog3.cpp -o prog3
./prog3 > output3.txt
cat output1.txt output2.txt output3.txt
hello prog1
hello prog2
hello prog3
```

Makefile - Notes

1. Run `make` along will lead to the `.DEFAULT_GOAL`, by default it is the first target in the makefile
   - `make` runs `make all` in this case
2. By specifying `make all -j4`, it runs at most 4 jobs parallelly (if possible)

Makefile - Example 2 - Variables

```
all: output1.txt output2.txt output3.txt
	cat output1.txt output2.txt output3.txt

CXX = g++
CXXFLAGS = -std=c++11 -O2

prog1: prog1.cpp
	$(CXX) $(CXXFLAGS) prog1.cpp -o prog1
output1.txt: prog1
	./prog1 > output1.txt

prog2: prog2.cpp
	$(CXX) $(CXXFLAGS) prog2.cpp -o prog2
output2.txt: prog2
	./prog2 > output2.txt

prog3: prog3.cpp
	$(CXX) $(CXXFLAGS) prog3.cpp -o prog3
output3.txt: prog3
	./prog3 > output3.txt
```

Makefile - Example 2 - Variables

```
all: output1.txt output2.txt output3.txt
	cat $^

CXX = g++
CXXFLAGS = -std=c++11 -O2

prog1: prog1.cpp
	$(CXX) $(CXXFLAGS) $< -o $@
output1.txt: prog1
	./$< > $@

prog2: prog2.cpp
	$(CXX) $(CXXFLAGS) $< -o $@
output2.txt: prog2
	./$< > $@

prog3: prog3.cpp
	$(CXX) $(CXXFLAGS) $< -o $@
output3.txt: prog3
	./$< > $@
```

Makefile - Variables

Before & after substitution:

- `$(CXX) $(CXXFLAGS) prog1.cpp -o prog1`
- `g++ -std=c++11 -O2 prog1.cpp -o prog1`

Automatic Variables

According to GNU make manual:

1. `$@` means the target
2. `$<` means the first prerequisite
3. `$^` means all prerequisites, space-separated

Makefile - Example 2 - Pattern

```
1  all: output1.txt output2.txt output3.txt
2      cat $^
3
4  CXX = g++
5  CXXFLAGS = -std=c++11 -O2
6
7  prog1 prog2 prog3: %: %.cpp
8      $(CXX) $(CXXFLAGS) $< -o $@
9  output1.txt output2.txt output3.txt: output%.txt: prog%
10     ./$< > $@
```

Makefile - Example 2 - Pattern

```
➜  demo git:(master) ✗ make
g++ -std=c++11 -O2 prog1.cpp -o prog1
./prog1 > output1.txt
g++ -std=c++11 -O2 prog2.cpp -o prog2
./prog2 > output2.txt
g++ -std=c++11 -O2 prog3.cpp -o prog3
./prog3 > output3.txt
cat output1.txt output2.txt output3.txt
hello prog1
hello prog2
hello prog3
```

Makefile - Pattern

According to static pattern rules, one can define:

- `targets …: target-pattern: prereq-patterns …`
- All the targets in list (space-separated string) applies the pattern rules to find its prerequisites

Makefile - Example 2 - Var & Pattern

```
1  PROG_LIST = prog1 prog2 prog3
2  OUTPUT_LIST = output1.txt output2.txt output3.txt
3  all: $(OUTPUT_LIST)
4      cat $^
5
6  CXX = g++
7  CXXFLAGS = -std=c++11 -O2
8
9  $(PROG_LIST): %: %.cpp
10     $(CXX) $(CXXFLAGS) $< -o $@
11 $(OUTPUT_LIST): output%.txt: prog%
12     ./$< > $@
```

Makefile - Example 2 - Foreach

```
1  ID_LIST := $(shell seq 1 3)
2  PROG_LIST := $(foreach X,$(ID_LIST),prog$X)
3  OUTPUT_LIST := $(foreach X,$(ID_LIST),output$X.txt)
4  all: $(OUTPUT_LIST)
5      cat $^
6
7  CXX = g++
8  CXXFLAGS = -std=c++11 -O2
9
10 $(PROG_LIST): %: %.cpp
11     $(CXX) $(CXXFLAGS) $< -o $@
12 $(OUTPUT_LIST): output%.txt: prog%
13     ./$< > $@
```

How to test your program

There are some useful code in attachment file `homework1`:

```
➜ DSA-HW1 git:(master) tree ./homework1
./homework1
├── Makefile
├── generator.cpp
├── prog1.cpp
└── prog2.cpp
```

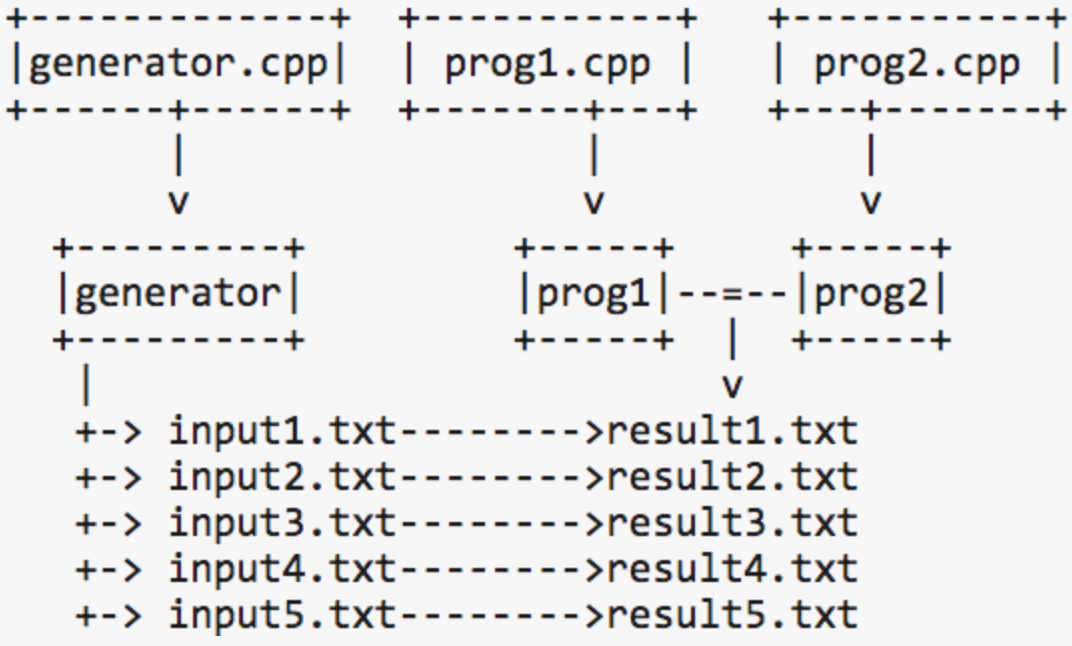
How to test your program

Let's say `prog1.cpp` is a brute force solution, to check if your solution `prog2.cpp` is correct, it needs some testcases

1. Compile `generator.cpp` and run to generate input file into stdout
2. Compile and run `prog1`&`prog2`
3. Use shell command `diff` to check if outputs are same

You can simply modify `Makefile` to do things above

How to test your program



How to test your program

To use a testdata generator:

```
➜ homework1 git:(master) ✗ make generator
g++ -std=c++11 -O2 generator.cpp -o generator
➜ homework1 git:(master) ✗ ./generator
usage: generator N M k [data_max] [seed]
➜ homework1 git:(master) ✗ ./generator 5 2 1 30
5 2 1
15 1 22 19 22
```

How to test your program

```
input1.txt: generator Makefile
	./generator 8 3 2 19 1337 > input1.txt
input2.txt: generator Makefile
	./generator 20 5 5 49 1337 > input2.txt
input3.txt: generator Makefile
	./generator 600 300 149 2147483647 1337 > input3.txt
input4.txt: generator Makefile
	./generator 10000 5000 2499 2147483647 1337 > input4.txt
input5.txt: generator Makefile
	./generator 200000 100000 49999 2147483647 1337 > input5.txt
```

How to test your program

```
➜ homework1 git:(master) ✗ make result1.txt
g++ -std=c++11 -O2 generator.cpp -o generator
./generator 8 3 2 19 1337 > input1.txt
g++ -std=c++11 -O2 prog1.cpp -o prog1
g++ -std=c++11 -O2 prog2.cpp -o prog2
echo "testcase: input1.txt" > result1.txt
echo "prog1:" >> result1.txt
./prog1 < input1.txt > output1.prog1.txt
cat output1.prog1.txt >> result1.txt
echo "prog2:" >> result1.txt
./prog2 < input1.txt > output1.prog2.txt
cat output1.prog2.txt >> result1.txt
diff output1.prog1.txt output1.prog2.txt || echo "wrong answer" >> result1.txt
```

```
testcase: input1.txt
prog1:
7
12
12
6
4
4
prog2:
12
19
6
4
4
17
wrong answer

```

Happy Coding :)