

DM865 – Spring 2019  
Heuristics and Approximation Algorithms

## Working Environment

Marco Chiarandini

Department of Mathematics & Computer Science  
University of Southern Denmark

# Outline

1. Working Environment

2. Random Numbers

# Outline

1. Working Environment

2. Random Numbers

# Building a Working Environment

What will you need during the project? How will you organize it? How will you make things work together?

- `src/` `code` that implements the algorithm (likely, several versions)
- `bin/` place where to put your executables
- `data/` `input`: Instances for the solver, parameters to guide the solver
- `scripts/` `code` that runs batches of experiments or parses files
- `log/` other log files produced by the run of the algorithm
- `res/` `output`: The result, the performance measurements
- `r/` `analysis tools`: statistics, data analysis, visualization
- `doc/` or `tex/` `journal/report`: A record of your experiments and findings, together with description of the algorithms.
- `Makefile` compiles the sources in `src` and puts the executables in `bin`.
- `README` explains how to compile, test and run the program. Eventually, it explains differences among versions.

↪ organize everything like if you had to reproduce the same results in a few years from now.

## Input controls on command line

```
xyz --main::instance ins1.txt --main::output-file log.txt --main::seed 12 > data.log
```

## Output on stdout, self-describing

```
#stat instance.in 30 90  
seed: 9897868  
Parameter1: 30  
Parameter2: A  
Read instance. Time: 0.016001  
begin try 1  
best 0 col 22 time 0.004000 iter 0 par_iter 0  
best 3 col 21 time 0.004000 iter 0 par_iter 0  
best 1 col 21 time 0.004000 iter 0 par_iter 0  
best 0 col 21 time 0.004000 iter 1 par_iter 1  
best 6 col 20 time 0.004000 iter 3 par_iter 1  
best 4 col 20 time 0.004000 iter 4 par_iter 2  
best 2 col 20 time 0.004000 iter 6 par_iter 4  
exit iter 7 time 1.000062  
end try 1
```

If a single program that implements many heuristics

- proceed for new versions but take old versions with a journal in archive.  
Consider using a version control system (eg, git, svn).
- use command line parameters to choose among the heuristics
- Python: `getopt`, `optparse`  
C: `getopt`, `getopt_long`, `opag` (option parser generator)  
Java: package `org.apache.commons.cli`  
EasyLocal: boost libraries
- use identifying labels in naming file outputs  
Example:  
`c0010.i0002.t0001.s02010.log`

# Example

- You will need:  
multiple runs, multiple instances, multiple classes and multiple algorithms.  
Arrange this outside of your program: ➡ unix scripts (eg, bash one line program)

- Parse outputfiles:  
Example

```
grep #stat * | cut -f 2 -d " "
```

See <http://www.gnu.org/software/coreutils/manual/> for shell tools.

- Data in form of matrix or data frame goes directly into R imported by `read.table()`, untouched by human hands!

```
alg instance      run sol time
RDS 1e450_15a.col 3 21 0.00267
RDS 1e450_15b.col 3 21 0
RDS 1e450_15d.col 3 31 0.00267
RLF 1e450_15a.col 3 17 0.00533
RLF 1e450_15b.col 3 16 0.008
...
```

- vim (Vi IMproved)
- emacs, atom, spyder3, visual code, jupyter-lab, eclipse
- Integrated development environment. Choose your favourite:  
[http://en.wikipedia.org/wiki/Integrated\\_development\\_environment](http://en.wikipedia.org/wiki/Integrated_development_environment)



Visualization helps understanding

- Problem visualization (matplotlib, networkx, graphviz)
- Algorithm animation
- Results visualization: recommended R (more on this later)

## Manual testing

- Print at run time
- Debug in `spyder3` or other IDE
- Plot the development of
  - best visited solution quality
  - current solution qualityover time and compare with other features of the algorithm.
- Take it well: it is like a detective job

# Be Suspicious – Unit Testing Development

## Automatic testing

- Check the correctness of your solutions at output everytime you do [refactoring](#) or new versions.
- [edge cases](#): test of arguments or inputs that lead to undefined cases
- useful as silent documentation on the use of the code
- coverage: did things work before a change or was a certain aspect never tested?

In Python, `assert` throws `AssertionError`

In Python, module `unittest`: tests as methods of a `Test` class.

automates the calls to the test functions

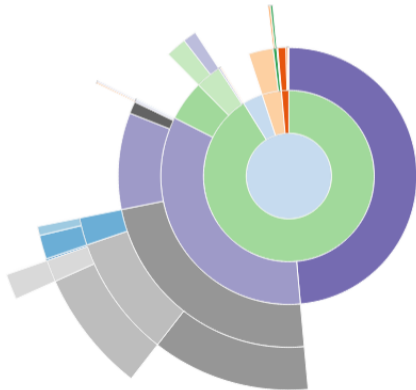
- Profile time consumption per program components
  - under Linux and OSX: gprof
    1. add flag `-pg` in compilation
    2. run the program
    3. `gprof gmon.out > a.txt`
  - under OSX:
    - Instruments
  - Java VM profilers (plugin for eclipse)  
<http://visualvm.java.net/>

# Program Profiling

cProfile

<https://github.com/ymichael/cprofilev>

<https://jiffyclub.github.io/snakeviz/>



Execution time can be extremely short:

- put the statement to measure in a loop
- execute several times

Take minimum measured time to make sure other processes running on the computer do not influence the measured result too much.

In Python, module `timeit`

The timer object has a function `repeat` that takes two parameters

`repeat` number of overall repetitions

`number` number of iterations in the loop

# Measuring Memory Consumption

- In Unix (and MacOS) `top`
- In Python, `python -m memory_profiler myscript`  
and `memprow` (for time dependent analysis)  
[https://pypi.python.org/pypi/memory\\_profiler](https://pypi.python.org/pypi/memory_profiler)

### Planning

Release planning creates the schedule • Make frequent small releases • The project is divided into iterations • Publish early, revise often

### Designing

Simplicity • No functionality is added early • Refactor: eliminate unused functionality and redundancy

### Coding

Code must be written to agreed standards • Code the unit test first • All production code is pair programmed • Leave optimization till last • No overtime • Pair programming

### Testing

All code must have unit tests • All code must pass all unit tests before it can be released • When a bug is found tests are created



# Development of Heuristics

- Model
- implement
- experiment
- fail
- think
- try again!

# Outline

1. Working Environment

2. Random Numbers

Carachtersitics of a good pseudo-random generator  
(from stochastic simulation)

- long period
- uniform unbiased distribution
- uncorrelated (time series analysis)
- efficient

Suggested: MRG32k3a by L'Ecuyer <http://www.iro.umontreal.ca/~lecuyer/>

# Ideal Random Shuffle

Let's consider a sequence of  $n$  elements:  $\{e_1, e_2, \dots, e_n\}$ .

The **ideal random shuffle** is a permutation chosen uniformly at random from the set of all possible  $n!$  permutations.

- $\pi_1$  is uniformly randomly chosen among  $\{e_1, e_2, \dots, e_n\}$ .
- $\pi_2$  is uniformly randomly chosen among  $\{e_1, e_2, \dots, e_n\} - \{\pi_1\}$ .
- $\pi_3$  is uniformly randomly chosen among  $\{e_1, e_2, \dots, e_n\} - \{\pi_1, \pi_2\}$
- ...

Joint probability of  $(\pi_1, \pi_2 \dots \pi_n)$  is  $\frac{1}{n} \cdot \frac{1}{n-1} \cdot \dots \cdot 1 = \frac{1}{n!}$

```
long int* Random::generate_random_array(const int& size) {
    long int i, j, help;
    long int *v = new long int[size];
    for ( i = 0 ; i < size; i++ )
        v[i] = i;
    for ( i = 0 ; i < size-1 ; i++) {
        j = (long int) ( ranU01( ) * (size - i));
        help = v[i];
        v[i] = v[i+j];
        v[i+j] = help;
    }
    return v; }
```

# Reservoir Sampling

How to select an element at random from a sequence of  $n$  elements where  $n$  is unknown?

- Keep the first item in memory.
- When the  $i$ -th item arrives (for  $i > 1$ ):
  - with probability  $1/i$ , keep the new item (discard the old one)
  - with probability  $1 - 1/i$  keep the old item (ignore the new one)

Probability of an element to be selected:

$$\Pr(\text{being selected once}) \cdot \Pr(\text{not being swapped later}) = 1 \cdot \prod_{i=2}^n \left(1 - \frac{1}{i}\right)$$

Extension:

How to select a sample of  $k$  elements from a sequence  $S$  of unknown length?

- Keep the first  $k$  items in memory.
- When the  $i$ th item arrives (for  $i > k$ ):
  - with probability  $k/i$ , keep the new item (discard an old one, selecting which to replace at random, each with chance  $1/k$ )
  - with probability  $1 - k/i$ , keep the old items (ignore the new one)

For the first element of the sequence the probability of being selected is:

$$\Pr(\text{being selected first time}) \cdot \prod_{i=k+1}^{|S|} \Pr(\text{not being swapped}) = 1 \cdot \prod_{i=k+1}^{|S|} \left(1 - \frac{k}{i} \cdot \frac{1}{k}\right)$$

where

$$\Pr(\text{not being swapped}) = 1 - \Pr(\text{swapping exactly that one}) = 1 - \frac{k}{i} \frac{1}{k}$$

$$\Pr(\text{not being swapped}) = \Pr(\text{not swapping}) + p(\text{swapping one of the others}) = \left(1 - \frac{k}{i}\right) + \left(\frac{k}{i} \frac{k-1}{k}\right)$$

## Implementation:

```
// S has items to sample, R will contain the result
ReservoirSample(S[1..n], R[1..k])
  // fill the reservoir array
  for i = 1 to k
    R[i] := S[i]

  // replace elements with gradually decreasing probability
  for i = k+1 to n
    j := random(1, i) // important: inclusive range
    if j <= k
      R[j] := S[i]
```