Introduction to shell scripting Unix commands in bioinformatics

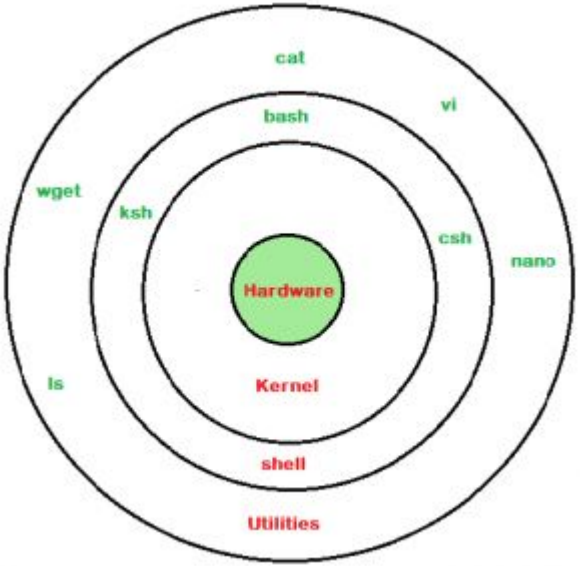
Lesson 02

Shell basics

- Shell is a **text-based interface** of an operating system
  - Users can enter command names, execute programs, and manipulate files, such as file handles, file permissions, and directories
  - Also, it offers a **scripting language**
- Use of the shell is fundamental to a wide range of advanced computing tasks, including high-performance computing and bioinformatics
- Popular shells
  - Unix / Linux: **bash**, zsh, ksh, etc.
  - Windows: Command Prompt (CMD), **Powershell**

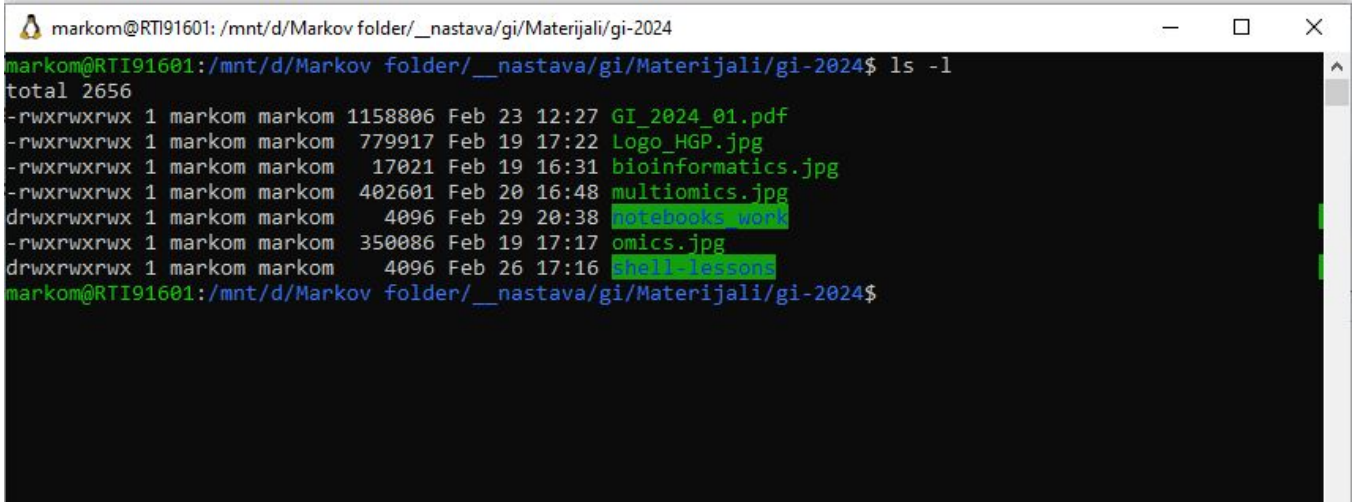
Shell basics

- Shell accepts **human-readable commands** from users and converts them into something which the kernel can understand
- It is a command language **interpreter** that executes commands read from input devices
  - Keyboard or files
- The shell gets started when the user logs in or starts the terminal



Terminal

- Users can access the shell through the **terminal window** or **console**
  - A window displaying a **command prompt** where users can enter commands
  - User enters commands and sees the output of those commands in a text-based interface



Shell scripting

- Shell script is a file that contains a sequences of **commands** to be executed together
- A **shell script** usually consists of the following elements:
  - Shell **keywords** – `if`, `else`, `break` etc.
  - Shell **commands** – `cd`, `ls`, `echo`, `pwd`, `touch` etc.
  - **Control flow** – `if..then..else`, `case` and shell loops etc.
  - **Functions**
- Allows combining existing tools into powerful **pipelines** and **workflows**
  - Handling large volumes of data automatically
  - Improving the reproducibility of workflows

Shell scripting in bioinformatics

- Majority of bioinformatics software is written for Unix (**Linux**)
  - Command line tools
- The easiest way to interact with **remote machines** and **supercomputers**
  - Cloud computing resources, remote servers
- Unix terminal commands could be powerful tool for some simpler analysis
  - Searching, counting, profiling textual formats
  - Automation, workflows and pipelines
- Usually **faster** than Python!

Typical use cases

- **Handling files** and directory structure / hierarchy
  - Navigate to a file/directory
  - Create, copy, move, rename, delete a file/directory
  - Inspect and set file/directory permissions
  - Check the length of a file
- **Chain commands** together
- **Retrieve** a set of files
  - Using wildcards and regular expressions
- **Iterate** over files
  - Inspection, searching, extracting data
- Run a **shell script** containing a pipeline

Running a terminal

- Unix / Linux
  - Gnome Terminal
  - KDE Konsole
  - **Xterm**
- Windows alternatives
  - **Windows Subsystem for Linux**
  - Cygwin
  - Git Bash
- If default shell is other than bash, type **bash** command

Some basic shell commands

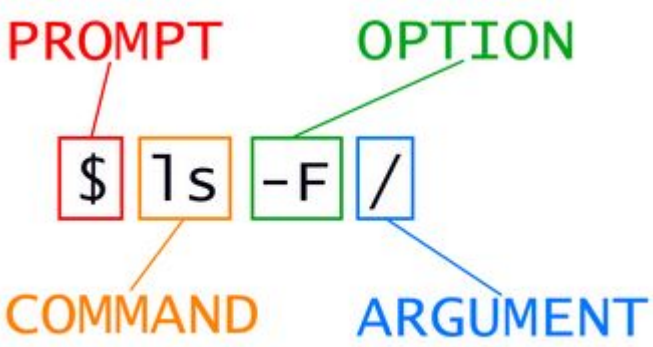
Command	What it does
ls	Lists the contents of the current directory
mkdir	Creates a new directory
mv	Moves or renames a file
cp	Copies a file
rm	Removes a file
cat	Print or concatenates files
less	Displays the contents of a file one page at a time
head	Displays the first ten lines of a file
tail	Displays the last ten lines of a file
cd	Changes current working directory
pwd	Prints working directory
find	Finds files matching an expression
grep	Searches a file for patterns
wc	Counts the lines, words, characters, and bytes in a file
history	Display previously executed commands

To get help use:
`man command`

For bash built-ins:
`help cmd`

General command syntax

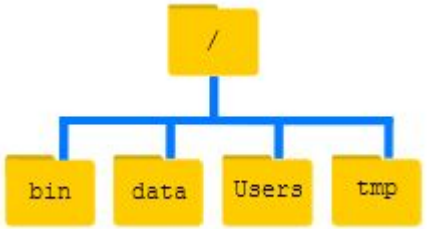
- Prompt
- Command
- Options
  - Change the behavior of a command
  - Short options (-)
  - Long options (--)
- Argument
  - Specifies what to operate on
  - Files or directories



Creating files

- Popular Unix / Linux **text editors**
  - `nano`, `joe`
  - `emacs`
  - **`vi`** / `vim`
  - `gvim`, `nedit` - need X windows terminal
- Creating an **empty** file
  - `touch`

Navigating files and directories



- Knowing **where you are**

  before running a command is important
  - **Absolute** and **relative** paths

    `/home/markom/gi-2024`, `gi-2024/notebooks/files`, `../data/bin`
  - **Root** directory
  - **Home** directory (e.g. `markom@RTI91601:~$`, `/home/markom`)
- Commands mostly read and write files in the **current working directory**
  - `pwd` command

    ```
    markom@RTI91601:/mnt/d/__nastava/gi/Materijali/gi-2024$ pwd
    /mnt/d/__nastava/gi/Materijali/gi-2024
    ```

Navigating files and directories

- **Listing** the content of the directory with `ls` and different options

```
markom@RTI91601:/home/markom/gi-2024$ ls
GI_2024_01.pdf  bioinformatics.jpg  multiomics.jpg  omics.jpg     terminal.png
Logo_HGP.jpg  filesystem.png     notebooks_work  shell-lessons

markom@RTI91601:/home/markom/gi-2024$ ls -lF
total 2684
-rwxrwxrwx 1 markom markom 1158806 Feb 23 12:27 GI_2024_01.pdf*
-rwxrwxrwx 1 markom markom  779917 Feb 19 17:22 Logo_HGP.jpg*
-rwxrwxrwx 1 markom markom   17021 Feb 19 16:31 bioinformatics.jpg*
-rwxrwxrwx 1 markom markom   3628 Mar  1 13:59 filesystem.png*
-rwxrwxrwx 1 markom markom  402601 Feb 20 16:48 multiomics.jpg*
drwxrwxrwx 1 markom markom   4096 Feb 29 20:38 notebooks_work/
-rwxrwxrwx 1 markom markom  350086 Feb 19 17:17 omics.jpg*
drwxrwxrwx 1 markom markom   4096 Feb 26 17:16 shell-lessons/
-rwxrwxrwx 1 markom markom   22738 Mar  1 12:29 terminal.png*
```

- **Navigating** through the file system with `cd`

```
markom@RTI91601:/home/markom/gi-2024$ cd notebooks_work/
markom@RTI91601:/home/markom/gi-2024/notebooks_work$ pwd
/home/markom/gi-2024/notebooks_work
```

Navigating files and directories

- **Absolute** and **relative** path hints

```
cd .                    # current dir
cd /                    # root dir
cd /home/markom
cd ..                   # one level up
cd ../..                # two levels up
cd ~                    # home dir
cd home
cd ~/data/..
Cd                      # home dir
```

Working with files

- **Create** a directory

```
mkdir projects
mkdir -p ../project/data ../project/results
```

- **Removing** a file or directory - be careful, **no undo**

```
rm my_file.txt
rm -r my_dir # note -r option, 'recursive'
```

- **Copying** file or a directory

```
cp data.txt project/data.txt
cp -r project project_backup
```

- **Moving** file or a directory

```
mv project/draft.txt project/final.txt
```

Working with files

- Using **wildcards** to specify multiple files at once
  - `*` represents zero or more other characters
  - `?` represents exactly one character
- Shell **expands** the wildcard to create a list of matching filenames before running the preceding command

```
ls *.txt     # lists all text files in the director
cp *_dat*    # copies all files files that contain _dat in its name
rm -r 2024-02-2? # remove all directories containing data
                 # from the end of February, 2024
```

Redirecting commands I/O

- Redirecting command **input** (`stdin`) from a file
  - Read operator <
- Redirecting command **output** from `stdout` to a file
  - Write operator >
  - Append operator >>
- Counting number of lines in files with `wc`

```
wc -l *.pdb > lengths.txt
```

```
20  cubane.pdb
12  ethane.pdb
 9  methane.pdb
30  octane.pdb
21  pentane.pdb
15  propane.pdb
107 total
```

Unix pipes

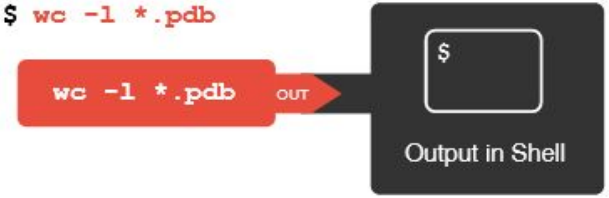
- **Passes** `stdout` of one command to `stdin` of the other with `|`
- Very useful for fast file manipulation
- **Saves time** for writing/reading to hard drive

```
wc -l *.pdb | sort -n

 9  methane.pdb
12  ethane.pdb
15  propane.pdb
20  cubane.pdb
21  pentane.pdb
30  octane.pdb
107 total
```

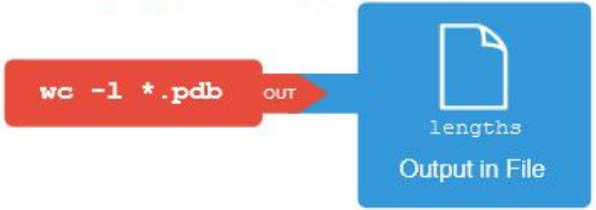
```
wc -l *.pdb | sort -n | head -n 3

9  methane.pdb
12  ethane.pdb
15  propane.pdb
```

Unix pipes

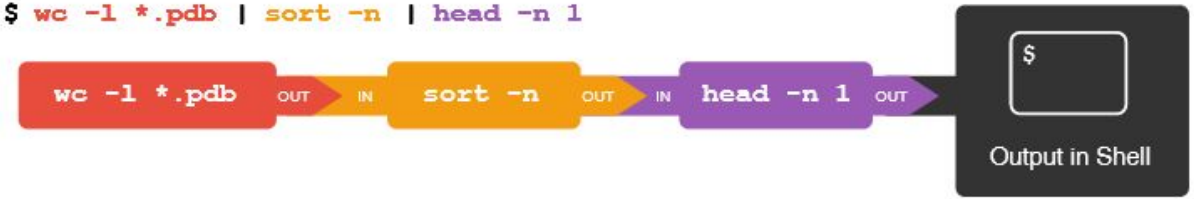
```
$ wc -l *.pdb
```




```
$ wc -l *.pdb > lengths
```




```
$ wc -l *.pdb | sort -n | head -n 1
```



Command chaining operators

- Executing **set** of commands within **one** command
- Every Unix command returns a **value**
  - **Zero** by default, if everything went well
- Operators `&&` `||` and `;`
- Useful for executing more commands within same Docker container

```
untar seq.fasta.tar && bwa mem fq1 fq2 seq.fasta > out.sam
```

Bash variables

- Declaring a variable through **variable assignment**

  `<variable_name>=<value>`

- **Variable expansion** with `$` and `${}`
  - `${}` is used to clearly delimit the variable name

```
my_variable="ggplot2"
echo "My favorite R package is ${my_variable}"
```

- Variable expansion is done by the **shell interpreter** before commands or statements are executed
  - Be careful with using spaces, quotes, or wildcard characters such as `*` or `?` in filenames, as it complicates variable expansion

Loops

- **Repeat command** or set of commands multiple times
- For each item in a list separated by spaces

```
for iterator in list
do
    operation/command using $iterator
done

for filename in basilisk.dat minotaur.dat unicorn.dat  # or *.dat
do
    echo $filename
    head -n 2 $filename | tail -n 1
done

for datafile in *.dat; do cat $datafile >> all.dat; done

for datafile in *.jpg *.png; do  echo "Image $datafile"; done
```

```
Image Logo_HGP.jpg
Image bioinformatics.jpg
Image multiomics.jpg
Image omics.jpg
Image command_syntax.png
Image filesystem.png
Image pipes.png
Image terminal.png
```

Conditional statements

- Slightly **complicated**
  - Single-bracket syntax `[ ]`
  - Double-bracket syntax `[[ ]]`
  - Double-parenthesis syntax `(( ))`
- **General** syntax

  `if <condition>; then <commands>; fi`
- **Single-bracket** syntax conditions based on `test` command
  - File-based conditions
  - String-based conditions
  - Arithmetic (number-based) conditions

`[ -d directory ]`	Directory exists and is directory
`[ -f regularfile ]`	File exists and is file
`-r, -w, -x`	Readable, writable, executable
`==, !=, <, <=, >, >=,`	String comparisons
`[ -n STRING ]`	Non-empty string
`[ -z EMPTYSTRING ]`	Empty string
`[ NUM1 -eq NUM2 ]`	Equal
`[ NUM1 -ne NUM2 ]`	Not equal
`[ NUM1 -gt NUM2 ]`	Greater than
`[ NUM1 -ge NUM2 ]`	Greater equal
`[ NUM1 -lt NUM2 ]`	Less than
`[ NUM1 -le NUM2 ]`	Less equal

Conditional statements

- The **words** *`if`*, *`then`*, *`else`*, *`elif`* and *`fi`* are shell **keywords**
  - They cannot share the same line, unless separated by **semicolon** *;*
- Checking different file properties

```
somefile="images.txt"

if [ -r $somefile ]; then
    cat $somefile;
elif [ -f $somefile ]; then
    echo "The file ${somefile} exists, not readable to the script.";
else
    echo "The file ${somefile} does not exist.";
fi

The file images.txt does not exist.
```

Bash scripts

- Bash **scripts** are simple text files that contain a series of commands we want to automate running rather than running them manually
  - Need **executable** permissions or execute with `bash script.sh`
- Parameterization using **positional arguments**

```
# Makes a backup for of files
# with given extension and prefix
# usage: bash img_backup.sh extension prefix
for file in *."$1"
do
    cp $file $2-$file
done
```

$0	Name of the script
$1 to $9	Arguments to the script
$@	All the arguments
$#	Number of arguments
$?	Return code of the previous command

Text manipulation tools

- **Text files** are widespread in Unix / Linux environments
  - Databases: users, groups, host, services
  - Configuration and log files
  - Commands written as scripts in `bash`, `perl`, `python`
- Extensive support for text **extraction**, **reporting** and **manipulation**
  - Extracting: `head`, `tail`, `grep`, `awk`, `cut`, `uniq`
  - Reporting: `wc`, `find`, `cat`, `less`
  - Manipulation: `sort`, `tr`, `sed`
  - Comparing text files: `diff`

tr - transform

- Manipulates **individual** characters in an input stream
  - Translate, delete, complement characters
- Characters from **one set** are replaced by characters from **another set**
- Always reads from `stdin` and writes to `stdout`

```
echo "agtccaatgct" | tr '[a-z]' '[A-Z]'

AGTCCAATGCT

echo "agtccaatgct" | tr -d 'a'

gtcctgct
```

cut - through the file

- The `cut` command in UNIX is a command for **cutting out** (extracting) the sections from each line of files and writing the result to standard output
  - Useful for fixed text file formats
- It can be used to cut parts of a line by **byte position**, **character** and **field**
- Useful for slicing columns from TSV/CSV files

```
$ cut -c 3,6,8 example.txt     # Extracts character 3,6 and 8 from each line
                               (one based)
$ cut -f 2-4 example.tsv       # Extracts columns 2,3 and 4 from file
                               (TAB is default delimiter)
$ cut -f -3 -d ',’example.csv # Extracts columns 1,2 and 3 from file
```

awk - dig into the file

- Search files for lines that contain certain **patterns**
- `awk` refers to a **program**, and to the **language** used by program
- When a line **matches** the patterns, `awk` performs defined **actions** on that line
- `awk` keeps processing input lines until the **end** of the input file is reached
- Options:
  - `-F fs` To specify a file separator.
  - `-f file` To specify a file that contains awk script.
  - `-v var=value` To declare a variable.

```
$ awk -F: '{print $1}' /etc/passwd  # same as cut -f 1 -d ':' /etc/passwd

$ echo "Hello Tom" > hello
$ awk '{$2="Adam"; print $0}' hello  # Outputs Hello Adam. $0 prins the entire line
```

awk - dig into the file (2)

```
$ awk 'BEGIN {print "The File Contents:"} {print $0} END {print "File footer"}' myfile
# Print contents of the file and add a sentence to the start and end of it

$ awk 'BEGIN{FS=":"; OFS="-"} {print $1,$6,$7}' /etc/passwd
# OFS  Specifies the Output separator, DEFAULT IS " "

$ awk '{if ($1 > 30) print $1}' test.tsv
# Output first column if its value is > 30

$ awk '{if ($1 > 30){x = $1 * 3; print x} else{x = $1 / 2; print x }}' testfile
# awk supports mathematical functions: sin(x) | cos(x) | sqrt(x) | exp(x) | log(x) | rand()

$ awk 'BEGIN{x = "likegeeks"; printf "The output is: %s\n", toupper(x)}'
# C-style elements
```

Source

sed command - non-interactive stream text editor

- Modifying the input as specified by a list of **commands**
- A **single** command may be specified as the first argument to sed

```
$ echo "ATATATAGAATGATGA" | sed 's/TA/CG/'
# s command replaces the first text with the second text pattern

$ sed 's/test/another test/2' myfile
# specifying the occurrence number that should be replaced like this

$ sed -n 's/test/another test/p' myfile # The p flag prints each line with
a matching pattern, -n option to prints the modified lines only.

$ sed '2,3s/test/another test/' myfile # Only lines 2 and 3 are modified
```

Source

sed command - non-interactive stream text editor

```
$ sed '2,$s/test/another test/' myfile
# Modify starting from line 2 to the end

$ sed '2d' myfile
# Deletes 2nd line from the stream, not the original file

$ sed '3,$d' myfile
# Keeps only first two lines from myfile

$ $ sed 'y/123/567/' myfile
# Replaces character 1->5, 2->6, 3->7
```

Source

Grep this!

- Searches for the **pattern inside** the file: `grep pattern file_name`

```
$ grep '^\.Pp' myfile # Find all occurrences of ’.Pp' at the beginning of a line

$ grep -v -e 'foo' -e 'bar' myfile  # To find all lines in a file which do not contain the words `foo' or `bar'

$ grep -B 1 -A 1 'aagtagggttca' hg38.fasta  # Search for a nucleotide sequence and print 1 line before and after any match. It won’t find the pattern if it spans more than 1 line.

$ grep -i "is" demo_file  # Key upper/lower case insensitive

$ grep -iw "is" demo_file  # “is” must be a word, surrounded by spaces
```

Grep this!

```
$ grep -r "GATTACA" *  # Searching in all files recursively using grep

$ grep -v "go" demo_text # Invert search (include if pattern is not found)

$ grep -c "go" demo_text  # count how many lines matches the given pattern

$ grep -n "go" demo_text  # Show line number while displaying the output

$ grep -m 1 pattern file # Stops search after first match

$ grep -E 'pattern1|pattern2' filename  # Look for appearance of any of two

$ grep -E 'Dev.*Tech' employee.txt # Look for Dev and Tech. No AND in grep.
```

How to grep for “Dev” or “Tech” but not “PM”?

Remote work

- Logging to a remote machine
  - `ssh markom@ws2-ec3-amazon.com`
- Remote copy
  - `scp markom@ws2-ec3-amazon.com:/home/markom/results.sam .`
- Download from internet
  - `wget https://dl.dropbox.com/u/154654/results.sam`

Compression of files

- Widely used compression tools
  - `gzip`, `bzip2`
  - Both work on one file - create a **tarball** first!
- Create archive or extract files with `tar`
  - `tar -cf archive.tar file1 file2 # just a tarbal`
  - `tar -czf archive.tar.gz *.sam # compress with gzip`
  - `tar -cjf archive.tar.bzip2 *.sam # compress with bzip2`
  - `tar -xvf /home/markom/examples.tar # extract tarball, verbose mode`
  - `tar -xvzf /home/markom/examples.tar.gz`
  - `tar -xvjf /home/markom/examples.tar.bz2`

How to…?

- Concatenate two tables with the same columns?

```
Name   ID      Available
John   332323 Yes
Mike   343434 No
Steven 323421    YES
```

```
Name ID       Available
Bin  336323   Yes
Vera 373434   Yes
Sara 324441   YES
```

```
$ cp table1.tsv table.tsv && sed 1d table2.tsv >> table.tsv
```

Vim editor

- Interactive ultra fast, keyboard-only text manipulation
- Insert, command and visual mode
- More to know than just: How can I exit?
- Good interactive tutorial
    - https://www.openvim.com/

Resources and additional reads

Presentation available at: github.com/vladimirkovacevic/gi-2024-etf

- Vince Buffalo: Bioinformatics Data Skills
- Dan Gusfield: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge
- Pavel Pevzner, Neils Jones: An Introduction to Bioinformatics Algorithms (Computational Molecular Biology), MIT
- R. Durbin, S. Eddy, A. Krogh, G. Mitchinson: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids , Cambridge University Press
- Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, Alexandru I. Tomescu: Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing, Cambridge University press
- Introduction to Unix
- GeeksForGeeks, Introduction to Linux Shell and Shell Scripting
- Software Carpentry, The Unix Shell
- Ted Laderas, Bash for Bioinformatics
- MIT, The Missing Semester of Your CS Education