Algoritmia

Outline

First Things

Data and Types

- Two general data types:
  - Scalar: `int`, `float`, `complex`, `bool`, `str`
  - Containers: contain an arbitrary number of scalars or of other containers
- In Python everything is an object (and so are, for instance, ints)
  - If `o` is an object, typing `o.` + `tab` in a shell lists its methods
  - Also `dir(o)`
  - Try `dir(1)`
- Python variables are not strongly typed
  - Types are implicitly assigned and checked at runtime

```
a = 1; type(a)
a = 'a'; type(a)
```

  - Explicit type checking with `isinstance(object, type)` or `type(object)is xxx`
- Type casting possible
- Special “value" `None` : absence of value

Variables and Expressions

- Variables: **names** of objects (not synonyms of memory positions, as in C)
  - `a = 3` is **not an assignment** but a **binding** of `a` with the object `3`
- Python has a number of reserved words:

  `and, print, while, class, lambda, min, ...`
  - They correspond to types, operators or built in functions
- Leading and trailing single `_` and double `__` are often used for special meanings
  - Good discussion in The Meaning of Underscores in Python
- Expressions often work as in C
  - `+=, -=, *=` : OK
  - `++, --` do not exist
  - `a // b` : integer division (also `a/b` in Python 2.X if `a, b` integers)
  - `1 / 2 = 0` in Python 2.X, `0.5` in Python 3.X
  - `a**b` : power

Variable Bindings

- Recall that variables in Python are in fact **names**
- At first sight they look more or less as in C, but there are clear cut differences
- Again, there are not assignments but **bindings** between names and objects
- Names in Python are very different from names in C
  - In Python variable names **are not** synonyms of memory addresses where the variable values are stored
- Global variables: defined elsewhere and identified as `global name`
- Same use (and same problems) as in C

Scope Rules

- Python follows the LEGB scope Rule
- **L, Local**: names assigned in any way within a function and not declared global in that function
- **E, Enclosing function locals**: names in the local scope of any and all enclosing functions, from inner to outer
- **G, Global** (module): names assigned at the top-level of a module file, or declared global within the file
- **B, Built-in** (Python): names preassigned in the built-in names module

Variables and Bindings Examples

- Sometimes things may not behave as expected:

```
a = []; b = a; a.append(1); b.append(2)
print (a); print (b)
```

```
a = 10; b = a; a+=1
print (a, b)
```

- Swapping variables is also much different than in C:

```
a, b = b, a
print (a, b)
```

Q: why/how does this work?

Flow Control

- Code blocks are identified by their **indentation**:
  - Recommendation in PEP 0008 – Style Guide for Python Code: 4 white spaces, no tabs
  - Results in highly structured code
  - But watch out for silly errors (as mixing spaces with tabs)
- Selection: `if condition:/elif condition:/else:`
- Iterations through `while` and `for`; no `do while` construction
- While iteration:

```
while condition:
    code block
```

- For iteration:

```
for var in sequence:
    code block
```

- `sequence` has to be an **iterable** object such as strings, and lists, tuples, files, ...

Loop Control Statements

- `break` : the loop terminates and execution goes to the statement immediately following the loop
- `continue` : the remainder of the loop body is skipped and execution goes to checking the loop's condition
- `pass` : used when a statement is required but do not want any command or code to executed
  - For instance, to leave temporarily an empty code block
- Try always to iterate over existing iterables and avoid C-style thinking over Python loops:

```
#on Python 3 do this only if needed
for i in range(1000000):
    print i

#never do this!!
for i in list(range(1000000)):
    print i
```

- `range(N)` defers the creation of the list element until it is needed

Strings

Strings

- Alphanumerical characters between `'` or `"`: `a = 'aaa'`
- First **immutable** object: their individual elements cannot be changed
- Standard operators overload on strings:

  `str1 + str2, int_ * str_, str1 < str2`

- `len(string)` returns its number of characters
- String elements accessible by indices: `a[0], a[-1], a[-2]`
- **Slicing** is used for substring access:

  `a[1: 3], 'abc'[1: 3], a[ : -1]`

  - `sss[F: L]` extracts values of indices `F` to `L-1`
- **Extended slicing**: `sss[F: L: s]` extracts values of indices `F` to `L-1` by step `s`
  - `s[ : : -1]` inverts the `s` string (or any array)

String Methods

- Very useful tools for string handling
- `s.lower(), s.upper()` : return lower or upper case versions of `s`
- `s.isalpha(), s.isdigit()` : tests if all the chars in `s` are of the corresponding type
- `s.find(string)` : searches for `string` and returns the first index where it begins or -1 if not found
- `s.replace(s_old, s_new)` : returns a string with `s_old` replaced by `s_new`
  - `s.replace('', '')` trims all blank space in `s`
- `s.split(delim)` : returns a list of substrings separated by the given delimiter
  - `s.split()` splits `s` over any sequence of white space characters

String Methods II

- The `separator.join(sequence)` construct uses Python's `join` function to put together the `sequence` list of strings separated by the string `separator`

```
s = 'XYZ'.join( ['a', 'b', 'c', 'd'] )
```

- `join` is the "inverse" of `split` :
  - `s.split('XYZ')` splits `s` in its substrings delimited by `XYZ`
- Frequent use when replacing bash files with Python scripts:
  - Form a Unix command string `str_cmd`
  - Execute it with `os.system(str_cmd)`

Pythonic Printing: `format` Method

- Apply the `format` method to a string mixing text and formating code
- The format contains one or more format codes (fields to be replaced) embedded in constant text
- The format codes are surrounded by `{ }`
- Inside `{ }` one has a positional parameter, plus `:` , plus a format string

```
"Second argument: {1:3d}, first one: {0:7.2f}".format(47.42,11)
"Art: {a:5d},  Price: {p:8.2f}".format(a=453, p=59.058)
"various precisions: {0:6.2f} or {0:6.3f}".format(1.4148)
```

- More in Python3 Tutorial: Formatted Output - Python Course

Functions

Functions

- Definition

```
def name(parameters):
    function body
```

- Function call: expression with value the returned value or `None`
- Call by value or by reference? In fact none of them:
  - In C the terms value or reference correspond to variables as synonyms of memory addresses
  - In Python immutable objects are usually called by (a kind of) value and mutable by (a kind of) reference (but watch out!)
- Python uses **call by object** or **call by object reference**: if you pass a mutable object into a function/method
  - It gets a reference to that same object and can be mutated with effects in the outside scope
  - But if the object's name is rebound in the function, the outer scope will know nothing about it and no further outside changes are made

An Example

- Bisection search for square root (from Guttag, p. 28):
- The following Python code yields approximate values to $\sqrt{x}$ for a given `x >= 1.0` with precision `eps`:

```
def bisect_sqroot(x, eps=1.e-3):
    """... docstring ..."""
    assert x >= 1., "error: input {0:f} < 1.".format(x)

    left = 1.; right = x; sqr = (left+right)/2
    while abs (sqr**2 - x) > eps:
        if sqr**2 < x:
            left = sqr
        else:
            right = sqr
        sqr = (left+right)/2
    return sqr
```

- Exercise: change things to get a function `cube_root(x, eps)` that approximates the cubic root of $x \geq 1$

Calling Functions

- When a function is called
  1. The function's **frame** and **namespace** are created
  2. If needed, parameter expressions are evaluated and parameter names are bound to their results
  3. The function body is executed (and more names may be added to the name space) until a return is reached
  4. The return value is bound according to the function call expression and the namespace is (usually) destroyed
- Multiple returned values are possible (well, in fact, no: they are actually tuples)
- Values are bound to parameters either positionally or through the formal parameter names
- This is exploited using **default values**

Argument Default Values

- Argument order may be changed if we use default values

```
def print_name(first_name, last_name, reverse):
    #function's body: exercise

#callable as:
print_name('Jose', 'Dorronsoro', False)
print_name(last_name='Dorronsoro', first_name='Jose', reverse=False)
```

- Default values are defined in the form `arg=value`

```
def print_name(first_name, last_name, reverse=False):
    #...

#callable as:
print_name('Jose', 'Dorronsoro')
print_name('Jose', 'Dorronsoro', True)
```

Passing Arguments

- In more detail: when a function is called,
  - The **positional arguments** are actually packed up into a **tuple** (`args`)
  - The **keyword arguments** are packed up into a **dict** (`kwargs`) with the variable names as keys
- Tuples are ordered and immutable, so we **cannot move** positional arguments around
- Dicts are not ordered and their objects are accessed through their keys; thus we **can move** `kwargs` around
- But cannot use a non keyword argument after a keyword one:

```
print_name('Dorronsoro', first_name='Jose', False) #error
```

- More on tuples and dicts below

Docstrings

- Given by a string contained between two triple quotes (`'''docstring '''` , `"""docstring """`) right after the `def` sentence
- Standard content:
  - A one line description of the function.
  - A full description after an empty line
  - A description of its parameters, returns and their types
- Standard parameter format: reStructured text (reST) / Sphinx, which can be used to generate documentation automatically
  - Other frequent options: Google and Numpy formats
- **Very important**: every function must have them
- More on Stack Abuse: Python Docstrings

Docstring Use

- Example with Google's style

```
def bisect_sqroot(x, eps):
    """Computes a square root of x by the bisection method.

    Returns an approximation to the square root of x up to a
         precision eps

    Args:
        x (float): the number whose square root we want
        eps (float): the precision wanted

    Returns:
        sqr (float): the approximate square root
    """
    ...
```

  - `help(bisect_sqroot)` in shell displays arguments and docstring
  - `bisect_sqroot(` in shell opens window with help
- `pdoc --html -o . my_module.py` writes a file `my_module.html` with (among others) the docstring info
  - Installed through the `pdoc3` module

Euclid Meets Python

- Python tries to build a kind of programming culture: PEP 0008 – Style Guide for Python Code, The Elements of Python Style
- The Zen of Python contains short design guiding principles
- Pythonic code follows this one:
  There should be one — and preferably only one — obvious way to do it
- An example (?): Euclid's algorithm

```
def mcd(x, y):
    while(y):
        x, y = y, x % y
    return x
```

- By the way: in Python (almost) everything is `True` except 0 and "empty" things: `[]`, `""`, `set()`

Structured Types

Structured Types

- Python has five structured types: strings, tuples, lists, dicts and sets
- Recall that **strings** are ordered sequences of chars, each accessible through an index
- They are immutable
- They have a large set of very useful methods
- Strings can be concatenated, indexed and sliced, and we can find their length through `len`
- `str(object)` transforms `object` into a string with results depending on what the object is
- More generally `type(object)` transforms when possible `object` into a another of type `type` with results depending on how the object is defined/programmed

Tuples

- **Tuples**: **immutable** ordered sequences of values possibly of different types accessible through an index
- Examples

```
a = ('a', 1, 'b', 2); b = 'a', 1, 'b', 2
a == b
```

- **Empty** tuple: `tup =( )` ; **one element** tuple: `tup =( 3, )`
- Tuples are **immutable**: their individual elements cannot be changed
- Tuples can be concatenated, indexed and sliced, and we can find their length through `len`
- Apparent multiple returns in functions are actually handled as tuples
- Tuples are the immutable cousins of lists

Lists

- **List**: **mutable** ordered sequences of values possibly of different types, each accessible through an index
- Perhaps the most used structured type in Python
- Lists can be concatenated (`+`), indexed and sliced
- Empty list: `l =[ ]`
- `len(l)` returns the number of objects
- Implemented as dynamic arrays
  - Adding or removing items at the end is fast
  - Not so in other positions
  - Efficient data structure for stacks (but not so for queues)

List Methods

- Some list methods: `l.append(object)`, `l.count(object)`, `l.sort()`, `l.reverse()`, `l.remove(object)`, `l.insert(index, object)`, `l.pop(index)`
- Some of them such as `sort()`, `reverse()` are in place and return `None`
  - `sorted(l)` returns a sorted version of `l`
- `l.index(object)` returns the index where `object` is or raises an exception
  - To just check whether `elem` is in `l`, simply use `if elem in l:`
- The function `tuple` changes (freezes) a list into a tuple
- The function `list` changes (thaws) a tuple into a list
- List **comprehension** is an efficient way to generate particular lists

```
oddN = [2*n+1 for n in range(10)]
```

  - Comprehension also works for dicts and sets

`zip` and `enumerate`

- `zip` joins several lists of the same lenght in a single list of tuples made of the elements on each list

```
l_1 = range(10)
l_2 = [i*i for i in l_1]
#OK
for a, b in zip(l_1, l_2):
        print(a * b)
#less so
for l in zip(l_1, l_2):
        print (l[0] * l[1])
```

- `enumerate` allows to iterate on a list and its indices:

```
l_2 = [i*i for i in l_1]

for i, sq in enumerate(l_2):
    print ("the square of {0:2d} is {1:4d}".format(i, sq))
```

Dictionaries

- **dict**: built in implementation of ADT Dictionary
- Can be seen as unordered lists with elements of the form `key:value`
  - Elements are **accessed by key values** and not indices
- Empty dict: `d = { }`
- Adding elements: `d.update({ 'a':'alpha'}) , d['a']='alpha'`
- The `keys()` method returns a (kind of) list with the (unordered) key values
- The `values()` method returns a (kind of) list with the `dict` values
- The `items()` method returns a (kind of) list of key–value tuples
- We can iterate on the keys of a dict `d` : `for k in d:`
- The statement `k in d` returns `True` if the key `k` is in the dict `d`

`args` and `kwargs` Revisited

- We can define functions with an arbitrary number of **positional** and **keyword** arguments using `*args` and `**kwargs`
- In the following definition

```
def do_something(*args, **kwargs):
    # whatever ...
```

  Python assumes that `do_something` will get a collection with a variable number of positional arguments and then a second collection with a variable number of keyword arguments
- If we call it as

```
do_something(pa1, pa2, pa3, kw1=kwa1, kw2=kwa2)
```

  the tuple `(pa1, pa2, pa3)` and the dict `{'kw1':kwa1, 'kw2':kwa2}` are passed to the function's body (which must know to handle them!!)
- Typical uses:
  - Writing higher order functions that pass arbitrary values to inside functions
  - Understanding others' code

Sets

- **set**: collection of different elements
- Initialization: `s = set()`
- Some methods:
  - `add, pop` : adds an object, removes and returns an object
  - `remove, clear` : removes an object, removes all objects
  - Membership: `in, not in`
  - `union, intersection, difference, symmetric_difference`
  - `issubset, issuperset`
- `len(s)` : number of objects in `s`
- `set(iterable)` : builds a set with the **unique** objects in the iterable

Removing Duplicates

- A usual task is to remove duplicate elements in a list
- Doing it a la C:

```
l_1 = [1, 2, 3, 1, 2, 3]
l_2 = []

#to avoid
for item in l_1:
    if item not in l_2:
        l_2.append(item)

print(l_2)
```

- The Pythonic way:

```
l_1 = [1, 2, 3, 1, 2, 3]

#much better
l_2 = list( set(l_1) )

print(l_2)
```

Arrays

- We can see Python lists as vectors and nested lists as matrices
- However, a much better way of working with arrays in Python is by using the Numpy module
- Is is usually imported as `import numpy as np` and its objects have a very rich set of methods:

```
m = np.zeros(5, 5)   # a  square 5 x 5 matrix of 0s
c = a + b            # element wise addition of arrays
c= a * b             # element wise multiplication of arrays
c = a.dot(b)         # standard multiplication of two arrays
m = -np.eye(5)       # negative 5 x 5 identity matrix
a / 5.               # element wise division
a = np.array([1, 2, 3])     #conversion of list into array
a.T, np.linalg.det(a)       # transpose and determinant
a.mean(axis=0)       # means of the columns of a
a.mean()             # mean of the flattened array
```

- Indexing and slicing of Numpy arrays is possible
- And much, much more!

Files and Modules

Working with Files

- Files are used through a **file handle**:

```
f_name = open('file', 'w')
```

  - A handle can be opened also with `'r'`, `'a'`, `'b'`, `'t'`, ...

- Once the handle `f_name` is defined, we can then use:

```
f_name.read(size) # to read the next size bytes
f_name.read() # to return a string with the entire file
f_name.readline() # to return a string with the next line
# to return string list with each of the file lines:
f_name.readlines()
f_name.write(string)
#to write the strings in the list S as file lines:
f_name.writelines(S)
f_name.close()
```

- In Python a file is a sequence of lines; thus we can loop through a file

```
f_name = open('file', 'r')
for line in f_name:
    print line[ : -1] #-1 avoids the final line break
```

Modules

- Files `*.py` containing statements, function definitions, global variables, etc.
- The `import` statement binds a module within the scope where the import occurs

  ```
  import my_module as mm
  ```

- If the file `my_module.py` has been changed after its import, it has to be reloaded to update the previous binds:

  ```
  reload(mm)
  ```

  - `reload` performs syntactical checking
  - Easier automatic reload in IPython with the `autoreload` extension
- Module functions (or variables) are used through object (dot) notation: `mm.funcname( ... )`

Module Variables

- Python modules can be run by `python module.py [arg_1, ...]`
- Or, better, by `module.py [arg_1, ...]` if the first line in `module.py` is the Python shebang `#!/usr/bin/env python`
- When the Python interpreter reads a source file, it defines some special variables and executes its (executable) code
  - If `xxx.py` is directly run from the Python interpreter, the special `__name__` variable is set to `'__main__'`
  - If `xxx.py` is being imported from another module, `__name__` is set to `'xxx'`
- Usually the following elements appear in a module to be run as a standalone program:

```
def main(...args...):
    #main's body

if __name__ == "__main__":
    main(...args...)
else:
    #lo que sea
```

Important Modules

- There are Python modules for almost everything: see for instance UsefulModules
- Modules that are often imported are
  - `sys, os` for OS–related tasks (see next)
  - `math` for standard math operations
  - `matplotlib` for plotting
  - `numpy` for linear algebra, `scipy` for scientific computing
  - `pandas` for index–field computing with tables
  - `sklearn` for machine learning
  - `statsmodels` for statistics

The `os` and `sys` Modules

- `os` provides interfaces to operating system dependent functionality, such as, for instance
  - `os.chdir(path)` changes the interpreter's active directory
  - `os.system(command)` executes the command in the string `command` in a subshell
- `sys` provides access to some interpreter variables and to functions that interact strongly with the interpreter.
- `sys.path` is a list of strings that specifies the search path for modules
  - Add new dirs using `sys.path.append()`
- `sys.argv` is a list containing command-line arguments
  - Thus `len(sys.argv)` gives the number of command-line arguments
  - `sys.argv[0]` is the script name

Algorithm Basics

Algorithms

- Many definitions, none too precise
- Adapting from Wikipedia

  a set of rules that precisely define a sequence of operations to perform some task and which eventually stop
- Usually written in **pseudocode**:
  - Intermediate between natural language and computer code
  - Not necessarily directly understandable by a computer but close by
  - Simple Python often fine in “simple" tasks
- Three building blocks:
  - **Sequential blocks**
  - **Selections**
  - **Repetitions** or **Loops**

Sequences, Selections and Loops

- **Sequential blocks**: blocks of (ordinary) sentences that execute sequentially in their entirety
  - Sentences may have just straight computations or several calls to functions
  - Order of execution according to gravity's law
  - In Python: blocks made of sentences with same indentation
- **Selections**: sentences where execution branches to different blocks according to some condition
  - In Python: `if condition:`, `elif condition:`, `else:`
- **Repetitions** or **loops**: a sentence block is repeated while some condition holds
  - In Python: `while condition:` and also `for` iterations

First Example: Euclid's Algorithm

- Computes the g.c.d. of two positive numbers a, b by repeatedly computing $r = a\%b$ and replacing a by b and b by r while $r > 0$
- In Python:

```
def euclid_gcd(a, b):
    while b > 0:
        #print(a, b)
        r = a % b
        a = b
        b = r

    return a
```

- Or more concise (pythonic?):

```
def euclid_gcd(a, b):
    while b > 0:
        a, b = b, a % b

    return a
```

Algorithm Design

- Algorithm design and writing (and programming in general) is usually done on an ad–hoc basis
- It is a **creative act**: must follow programming rules but also requires **imagination, creativity and experience**
  - The same happens with ordinary writing: we cannot fill an empty page just with grammar rules
  - Programming also requires hard work, lots of practice and, also, **quite a bit of algorithm reading**
- Sometimes we can take advantage of general **design techniques**
  - Derived from long experience in problem solving and algorithm analysis
  - Cannot be applied as automatic rules of thumb
  - But may have a wide range of applicability
- We will consider three here: **greedy** algorithms, **divide and conquer** (a.k.a. **recursive**) algorithms and **dynamic programming**

Algorithm Efficiency

Things To Keep In Mind

- First of all **algorithms must be correct**
  - A fast but wrong algorithm is useless
- It is also desirable that they do not require (much) extra memory
- It is also highly desirable that they are as **fast** as possible
  - But ... an algorithm must “read" its inputs
  - If there are many and they are large, the algorithm will likely be slow
  - But desirable execution times should not be far above from the same “order of magnitude" than its inputs’ size
- How do measure execution times?

Estimating Execution Times I

- Full details in the Análisis de Algoritmos course
- First of all, **forget about just measuring actual times**
  - They depend on the language, the machine, the programmer and, of course, the inputs
  - They are thus too context–dependent to allow meaningful **generalizations**
- We focus instead on **abstract times** measured by counting the **key operations** the algorithm performs on a given input
- For iterative algorithms we usually look for the key operation on the innermost loop
  - Counting how many times these key operations are performed will give us a good estimate of the time their algorithms will take

Matrix Multiplication

- Well known (and quite costly!!) algorithm: $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$
- Simple (and quite bad) Python code:

```
def matrix_multiplication(m_1, m_2):
    """ ..."""
    n_rows, n_interm, n_columns = \
        m_1.shape[0], m_2.shape[0], m_2.shape[1]

    m_product = np.zeros( (n_rows, n_columns) )

    for p in range(n_rows):
        for q in range(n_columns):
            for r in range(n_interm):
                m_product[p, q] += m_1[p, r] * m_2[r, q]

    return m_product
```

- Key operation: clearly `m_1[p, r] * m_2[r, q]`
- How many? Assuming square matrices with N rows/columns, obviously $N \times N \times N = N^3$
  - Substantially larger than problem size $N^2 + N^2 = 2N^2$

The o, O and Θ Notations

- We cannot always give precise abstract time estimates
- We introduce notation to handle these cases
- We assume that the cost functions are positive and increasing (this should be the case with the abstract execution times of algorithms)
- Given such a pair f, g, we say that $f = o(g)$ if $\frac{f(N)}{g(N)} \to 0$ when $N \to \infty$
- Also, $f = O(g)$ if we can find C and N_C s. t. $f(N) \leq Cg(N)$ if $N \geq N_C$
- Finally, $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$
- Examples: $n_{MM}(A, B) = \Theta(N^3)$, $n^e_{LS}(N) = O(N)$

`timeit` Magic Command

- But we shouldn't forget about actual times
- IPython's magic command `timeit` is very handy to measure execution times of code snippets
- Very easy to run and interpret

```
timings = %timeit -n 1000 -r 10 -o [k**2 for k in range(10)]

print('\n', dir(timings)[-9 : ], '\n')
print("all_runs", np.array(timings.all_runs).round(3))
print("repeat", timings.repeat)
print("loops", timings.loops)
print("compile_time", timings.compile_time)
print("average_time", timings.average)
print("best_time", timings.best)
print("worst_time", timings.worst)
```

From Abstract Times to Real Times

- We can apply `%timeit` to our matrix multiplication as follows

```
a = np.ones((100, 100)); b = np.eye(100)
%timeit -n 10  -r 10 matrix_multiplication(a, b)
```

- If it gives us a time estimation of, say, 1 second, what can we expect for matrices with dimensions 500?
  - Since $500 = 5 \times 100$, $500^3 = 125 \times 100^3$, i.e., 125 times bigger
  - Thus, we should expect `%timeit` to report about 125 seconds
  - Even if we do not perform precise measurements, we can say that the actual time needed to multiply two $N \times N$ matrices is $\Theta(N^3)$
- This is not %100 precise, but gives a ball park estimate
- Hence, `matrix_multiplication` is very quite costly
- And watch out: straight Python code can be quite slow because of language overheads
  - Heavy duty libraries such as `numpy, pandas, scipy` or `sklearn` do their work in C or C++ compiled code
  - Check this with `%timeit -n 10 -r 10 a.dot(b)`
  - And by the way, in Python *code as little as possible, using good libraries instead*

A Survey of Basic Data Types and Structures

Data Structures

- Algorithms work on data
- Single variables are OK for simple algorithms
- **Data structures**: ways to organize more complex data for more advanced algorithms
- Simplest data structures: **strings, lists, arrays**
- More advanced: **dictionaries, sets**
- All of them built in in Python (arrays better through Numpy)
- Advanced structures: **linked lists, trees, graphs**
  - Available in Python through imported modules

ADTs VS DSs

- Abstract data type (ADT): a collection of items plus a set of primitives acting on them
- Primitives define and specify the most relevant operations on an ADT
- Example: **stacks**, which are a sorted collection of objects accessible only from its **top**
- Stack primitives: `ini`, `pop` and `push`
- ADTs are abstract (!!) so decisions and details are needed to implement them
- Fisrt step: decide which DS to use

Examples of ADTs and DSs

- ADTs from Prog II: stacks, queues, priority queues (?)
- DS from Prog II: arrays, linked lists, binary trees, binary search trees, heaps, general trees, graphs (?)
- DS from EDyL: graphs
- ADTs from An Alg: ADT dict
- DS from An Alg: max heaps, AVLs, hash tables
- DS from BdDs: B and B+ trees
- ADTs in AEDAs: **priority queues, disjoint sets**

PQs over Min Heaps

Min Heaps

- A **heap** is a binary quasi complete tree
- A **min heap** is a heap T such that for each node T' of T we have

$$\text{info}(T') < \min\{\text{info}(\text{left}(T')), \text{info}(\text{right}(T'))\}$$

  - Thus the smallest element is at the root
- There are two main dynamic primitives on min heaps:
  - `extract` the smallest element of the min heap, i.e., that at the root
  - `insert` a new element in the min heap
- Both allow efficient implementations if the min heap is stored on a Numpy array (better) or a list

Min Heaps on Arrays/Lists

- We assume a heap with n elements stored in an array with indices `0, ..., n-1`
- The left and right children of the node at index `i` are `2*i+1, 2*i+2`
- The parent of the node at index `i` is `(i-1)// 2`
- The key tool to process min heaps is the `heapify` function that at a given node makes sure that **the value at that node ends at its proper place in a min heap**

Heapifying from a Node

- A possible Python code of `heapify` at a node `i` of a heap `h` is

```
def heapify(h: List, i: int):
    while 2*i+1 < len(h):
        next_i = i
        if h[next_i] > h[2*i+1]:
            next_i = 2*i+1
        if 2*i+2 < len(h) and h[next_i] > h[2*i+2]:
            next_i = 2*i+2
        if next_i > i:
            h[i], h[next_i] = h[next_i], h[i]
            i = next_i
        else:
            return
```

- The cost of `heapify` is clearly $O(\text{prof}(h)) = O(\lg n)$

Extracting the Root of a Min Heap

- To extract the root we simply
  - Remove the root node
  - Move the last node to the root
  - Apply `heapify` from the root on the reduced heap
- The extraction cost is clearly $O(\lg n)$

Inserting a New Node in a Min Heap

- To insert a new node/value into an existing min heap we
  - Put the new node at the end of the array containing the heap
  - While the new node is smaller than its parent, we swap their places in the min heap
- A possible Python code is

```
def insert_min_heap(h, k):
    h += [k]
    j = len(h) - 1

    while j >= 1 and h[(j-1) // 2] > h[j]:
        h[(j-1) // 2], h[j] = h[j], h[(j-1) // 2]
        j = (j-1) // 2
```

- The insertion cost is clearly again $O(\lg n)$

Priority Queues

- They are queues with the standard primitives `insert, remove` but where
  - Each item has a **priority** which determines its place in the PQ after the insertion
  - We assume smaller values have a higher priority
- `insert(x)` puts `x` “after" the elements in the PQ with smaller priority and “before" those with a larger one
  - Before and after do not have to be literally true
- `remove` extracts the first element maintaining the other elements’ priority

PQs over Min Heaps

- Conceptually we build a min heap where each node contains its priority and, if needed, a pointer to its data
- The min heap is built according to the priorities
- The PQ primitives reduce to the min heap ones
  - `insert` is just the insertion on the min heap
  - `remove` is just removing the root of the min heap readjusting it afterwards
- The cost of both is thus $O(\lg N)$

The Disjoint Set Abstract Data Type

Disjoint Set

- A **Disjoint Set** (DS) over a universal set U is a dynamic family S of disjoint subsets of U (i.e., a **partition** of U), each of which is **represented** by a certain element x and that has the following primitives:
  - `init_DS(U, S)`: receives the universal set U and returns the initial S as the famility of atomic subsets $\{\{u\} : u \in U\}$
  - `find(x, S)`: receives an element $x \in U$ and returns the representative of the subset S_x of S that contains x
  - `union(x, y, S)`: receives two representatives x, y, computes their union $S_x \cup S_y$ and returns a representative of the subset $S_x \cup S_y$

Observations on the Disjoint Set

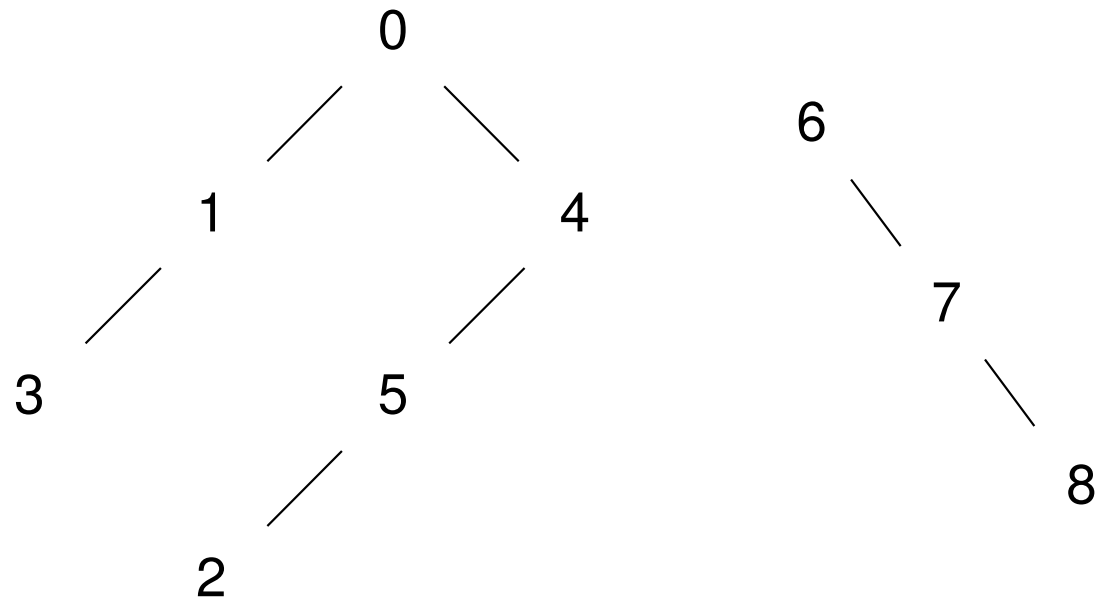
- The subsets of a Disjoint Set are never split
  - They can only change to bigger subsets
  - The Disjoint Set is never empty
- After `init_DS` we start with a partition with $|U|$ subsets;
  - Thus, the maximum number of unions is $|U| - 1$
- There are several choices for the DS's data structure
  - An example: a set of lists for the subsets and a pointer array to them for the representative
  - We will opt for a simpler one

Trees as DSs for Disjoint Sets

- Our data structure stores the subsets in a DS as a particular kind of (forests of) trees
- The representative x of a subset S is at the **root** of the subset tree T_S
- The cost of `union(x, y, S)` is then just $O(1)$, as we simply make, say, T_{S_y} a child subtree of the x root
- To implement `find(u, S)` first we need a fast way to locate the tree of u and, then, to go from the u node to the root
- This can be easily done if we place the subsets on a table `p[ ]` where:
  - `p[u]` is the index of the father of `u`
  - `p[x]=-1` for a root `x`, i.e., a representative

An Example

- For a subset partition over the universal set `[0, 1, 2, 3, 4, 5, 6, 7, 8]`



the associated table (starting at index `0`) would be
`[-1, 0, 5, 1, 0, 4, -1, 6, 7]`

Union and Find over Trees

- To initialize the DS we simply set `p[i]=-1` for all `i`
- The simplest pseudocode for `find` is

```
def find(u, p):
    while p[u] != -1:
        u = p[u]
    return u
```

- The pseudocode for a naive `union` is

```
def union(x, y, p):
    p[y] = x    #join second tree to first
    return x
```

Improving Union

- Since the cost of `find` is $O($ height $(T_x))$ it is clear that we should join the shorter tree into the taller one
  - This will make future finds cheaper
- For this we need to keep a tree's height h
  - We simply can change $p[x]$ at the root x from -1 to $-h$
- We then change the pseudocode for `union` as

```
def union_height(x, y, p):
    if p[y] < p[x]:          #T_y is taller
        p[x] = y; return y
    elif p[y] > p[x]:        #T_x is taller
        p[y] = x; return x
    else:                    #T_x, T_y have the same height
        p[y] = x; p[x] -= 1; return x
```

- We also change the while condition on `find` to

```
while p[u] >= 0:
```

The Cost of Find

- **Proposition.** *If prof(T) is the depth of a DS tree T and we apply union by heights, we have prof(T) $\leq \lg |T|$, with $|T|$ the number of nodes in T*
- **Proof Sketch:**
  - Use induction on $|T|$, with an obvious base case $|T| = 1$
  - Assume prof(T') $\leq \lg |T'|$ for $|T'| < |T| = k$ and that we join T_y into T_x with $|T_x \cup T_y| = k$
  - If prof(T_y) $<$ prof(T_x),

$$\text{prof}(T_x \cup T_y) = \text{prof}(T_x) \leq \lg |T_x| \leq \lg |T_x \cup T_y|$$

    and the same argument works when prof(T_x) $<$ prof(T_y),
  - If prof(T_y) $=$ prof(T_x) and, say, $|T_y| \leq |T_x|$,

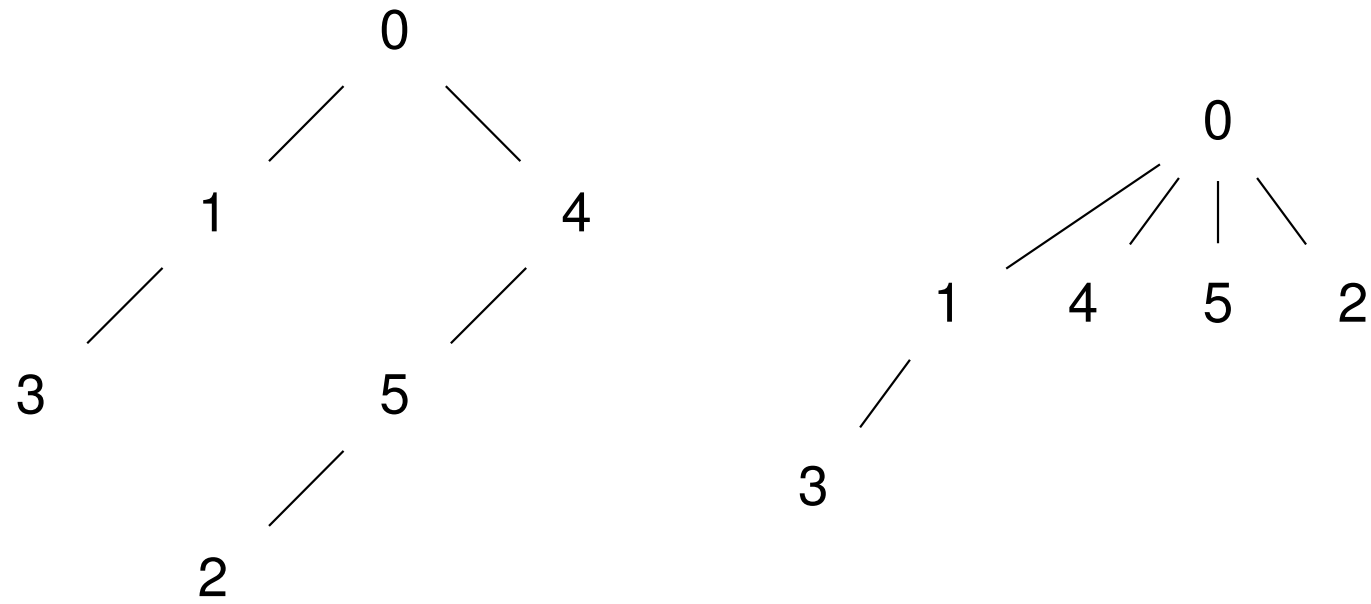
$$\begin{aligned} \text{prof}(T_x \cup T_y) &= 1 + \text{prof}(T_y) \leq 1 + \lg |T_y| = \lg\ 2|T_y| \\ &\leq \lg |T_x \cup T_y| \end{aligned}$$

Improving Find

- Thus, the cost of `find(x, p)` is also $O(\log |S_x|) = O(\log N)$
- Moreover, we can further improve on this
- Observe that when finding the representative of u we also find the **representative of all the v between u and the root of its tree**
- We can thus change `find` to update `p[v]` for all v between u and the root
- In other words, we can **compress the path** from u to the root

The Effect of Path Compression

- Left: tree state after `find(2)`; right: state after `find_cc(2)`



Find with Path Compression

- Recall that after finding the representative of u, we also know it for all the other nodes between u and the root of the tree
- We thus improve `find` as follows:

```
def find_cc(u, p):
    # find the representative
    z = u
    while p[z] >= 0:
        z = p[z]

    # compress the path from u to the root
    while p[u] >= 0:
        y = p[u]
        p[u] = z
        u = y
    return z
```

Path Compression and Union by Rank

- The problem is now that, after `find`, we no longer have in $-p[x]$ the tree's height
- We do nothing about this other than calling $-p[x]$ the tree's **rank**
- We change nothing on `union` although it is no longer a union by height but a **union by rank**
- However the joint cost of unions and finds considerably improves
- **Proposition:** *If on a DS with N elements we do L unions by rank and* $M = \Omega(N)$ *path compression finds, the overall cost is*

$$O(L + M \lg^* N)$$

The $\lg^*$ Function

- We define $\lg^* H = K$ if K is the smallest integer such that after K binary logs we have

$$\lg(\ldots \lg(\lg H)\ldots) \leq 1$$

- For instance $\lg^* 65536 = \lg^* 2^{16} = 4$, but then

$$\lg^* 2^{65536} = 1 + \lg^* 2^{16} = 5$$

- Now 2^{65536} is a huge number:
  - Find out how many digits its decimal expression has (easy)
  - Then try to write it using millions, billions, googols and so on! ;-)
- For practical purposes $\lg^* H = O(1)$
- Disjoint sets can be applied in several graph algorithms

Some Graph Definitions

- Recall some definitions from EDyL
- A **path** in a graph $G = (V, E)$ is another graph $\pi = (V_\pi, E_\pi)$ with nodes $V_\pi = \{(u_0, \ldots, u_K)\} \subset V$ and edges $E_\pi = \{(u_i, u_{i+1}), 0 \leq i < K\} \subset E$
- An undirected graph $G = (V, E)$ is **connected** if for every pair $u, v \in V$ there is a path π in G from u to v
- The **connected** components of an undirected graph are the maximal connected subgraphs of G
- A **cycle** π in a graph $G = (V, E)$ is a path that starts and ends at the same point
- A **tree** is an undirected connected graph that is also **acyclic**, i.e., there are no cycles in E
- G is **weighted** if each edge (u, v) has an associated **cost** $c(u, v) \in \mathbf{R}$

DSs and Connected Components

- A first application of DS is to find the connected components of an undirected graph $G = (V, E)$
  - Simply initialize a DS $\mathcal{S}$, and
  - For each $(u, v) \in E$, if $p \neq q$ are the representatives of u, v in $\mathcal{S}$ at that moment, apply `union` on p, q
- It is easy to see that, at the end, the different subsets in $\mathcal{S}$ contain the connected components of G
  - Just reason by induction that, at every step, the subsets of $\mathcal{S}$ are connected
- The cost will be $O(|E| \log^* |V|)$, slightly worse than the linear $O(|E|)$ cost of the method based on depth first search

Implementing Kruskal's Algorithm

Minimum Spanning Trees

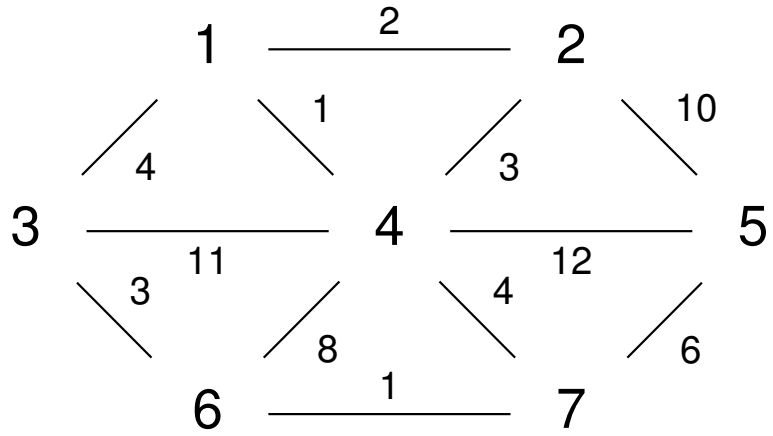
- T is a **spanning tree** for $G = (V, E)$ if $T = (V, E_T)$ with $E_T \subset E$
- If G is weighted, the **cost** of an ST T is

$$c(T) = \sum_{(u,v) \in E_T} c(u, v)$$

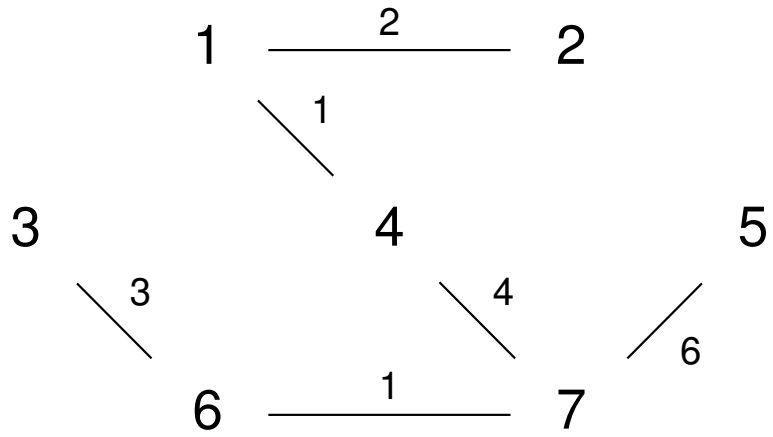
- $T = (V, E_T)$ is a **minimum spanning tree** (MST) for the undirected graph $G = (V, E)$ if for any other ST $T' = (V, E'_T)$ we have $c(T) \leq c(T')$
- Notice that there may be several minimum spanning trees in a graph but the **minimum cost is unique**

MST Examples

- On the graph

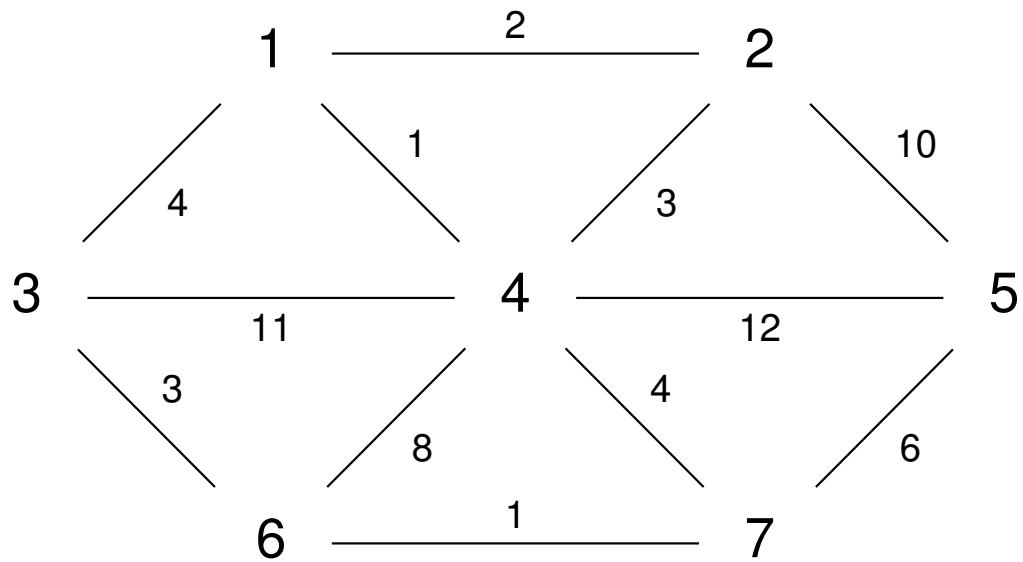


a first MST with cost 17 is



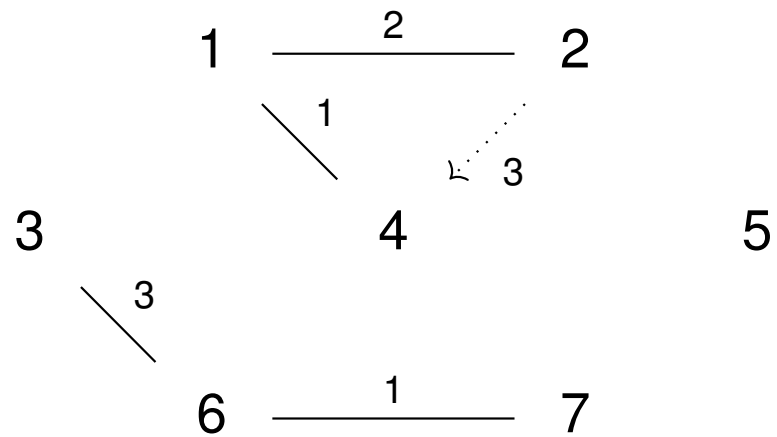
A First Look at Kruskal's Algorithm

- Main idea: sort the edges of E in a PQ by increasing costs and build a graph (forest?) of partial STs
  - Starting from single node trees $T_u = (\{u\}, \emptyset)$ and
  - Adding edges from the PQ that do not produce cycles
- Consider the previous example



How to Apply Kruskal?

- Solving ties lexicographically, the sorted edges are
  `(1,4), (6,7), (1,2), (2,4), (3,6), (1,3), (4,7), (5,7), (4,6), ...`
- We add edges to the partial ST as



- But trying to add `(2,4)` we get a **cycle**, so we drop it and add next `(3,6)`
- We keep going until we (hopefully) get an MST

Elements of Kruskal's Algorithm

- To implement Kruskal we need a PQ, a way of storing the selected edges and a way to maintain the forest of partial subtrees and to detect cycles
- No problem with the PQ and we can simply gradually build the final MST graph over the Kruskal forest of the partial subtrees
- At first sight maintaining trees and detecting cycles in them looks complicated and costly
- However, observe that (u, v) **gives a cycle iff** u **and** v **are in the same subset** $V_{T'}$ of the vertices of a tree T' in the Kruskal forest
  - `2` and `4` are in the set `{1, 2, 4}`
  - Thus we do not need to work with trees but with **subsets**
- We do this working with a Disjoint Set and its primitives allow us to write a Kruskal pseudocode

Kruskal over Disjoint Sets

- We work with union by rank and path compression:

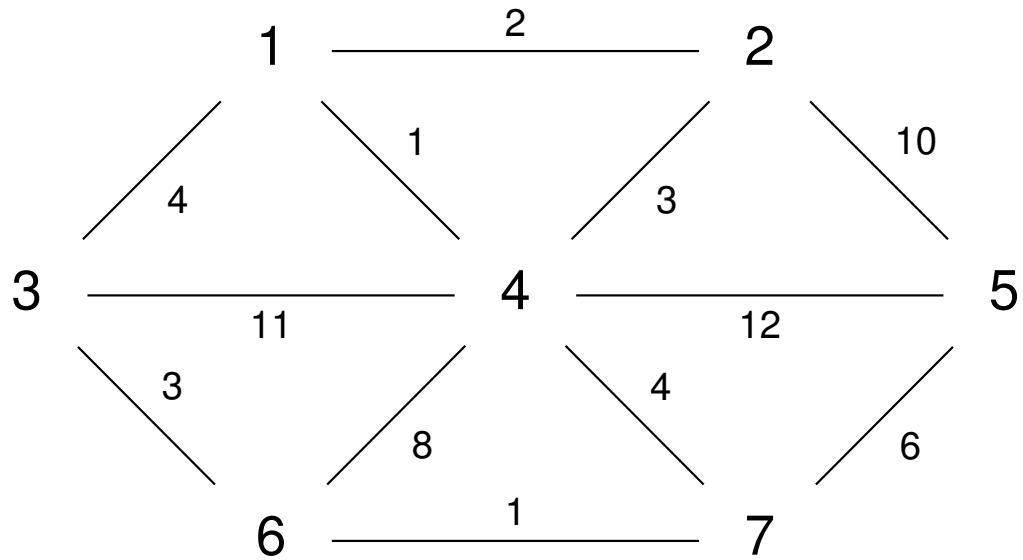
```
def kruskal(G):
    mst = []                                #empty list for the MST
    ds = init_DS(V)                         # 1
    Q = pq()

    for all (u, v) in E:
        Q.put( (c(u, v), (u, v)))           # 2

    while not Q.empty:                      # 3
        _, (u, v) = Q.get()                 # 4

        x = find_cc(u, ds)                  # 5
        y = find_cc(v, ds)

        if x != y:
            mst.append((u, v))              # 6
            union(x, y, ds)                 # 7

    return (V, mst)
```

The Cost of Kruskal's Algorithm

- Clearly the cost of (1) is $O(|V|)$ and that of (2) is $O(|E| \log |V|)$
- The cost of (4) accumulated over (3) is again $O(|E| \log |V|)$
- Since the single cost of (6) and (7) is $O(1)$ and only happens when `x!=y`, their accumulated costs are $O(|V|)$
- Finally, since we must do at least one `find_cc` for each node, the total number is $\Omega(N)$ and, therefore, the cost of (5) accumulated over (3) is $O(|E| \lg^* |V|)$, that is, essentially $O(|E|)$
- Summing things up, the cost of Kruskal is $O(|E| \lg |V|)$, dominated by the PQ operations
- In particular the DS operations do not penalize the algorithm

Applying Kruskal's Algorithm

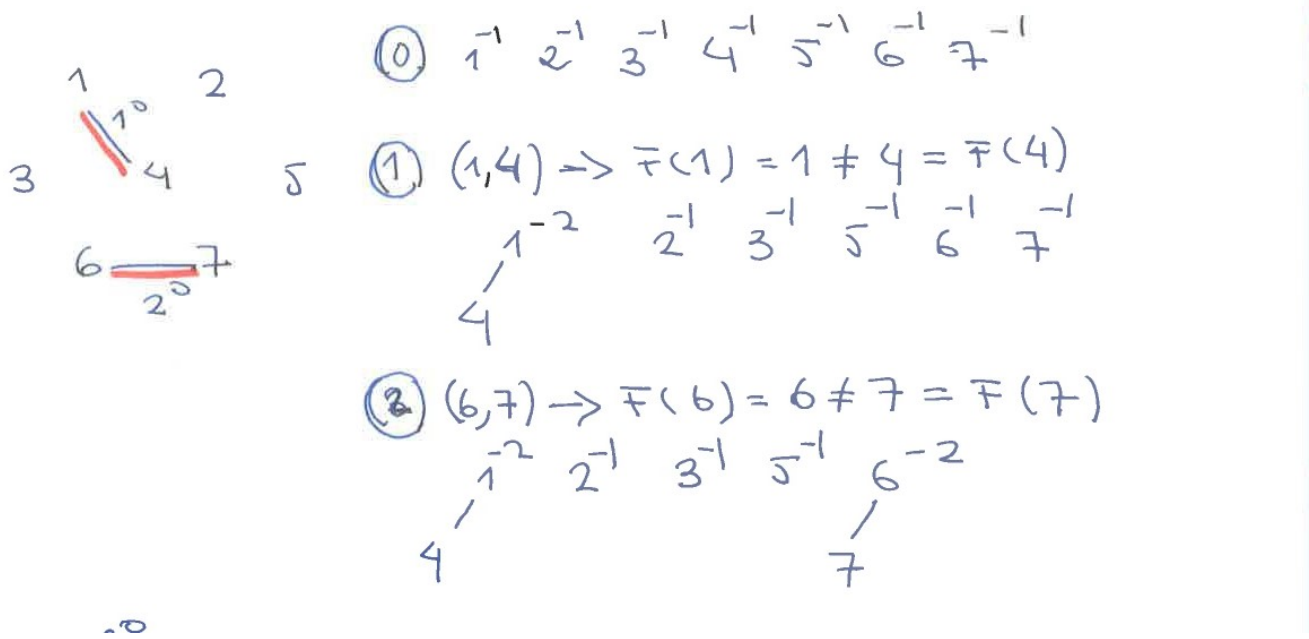
- We apply it on the previous graph



- The PQ is `(1,4), (6,7), (1,2), (2,4), (3,6), (1,3), (4,7), (5,7), (4,6), (2,5), (3,4), (4,5)`

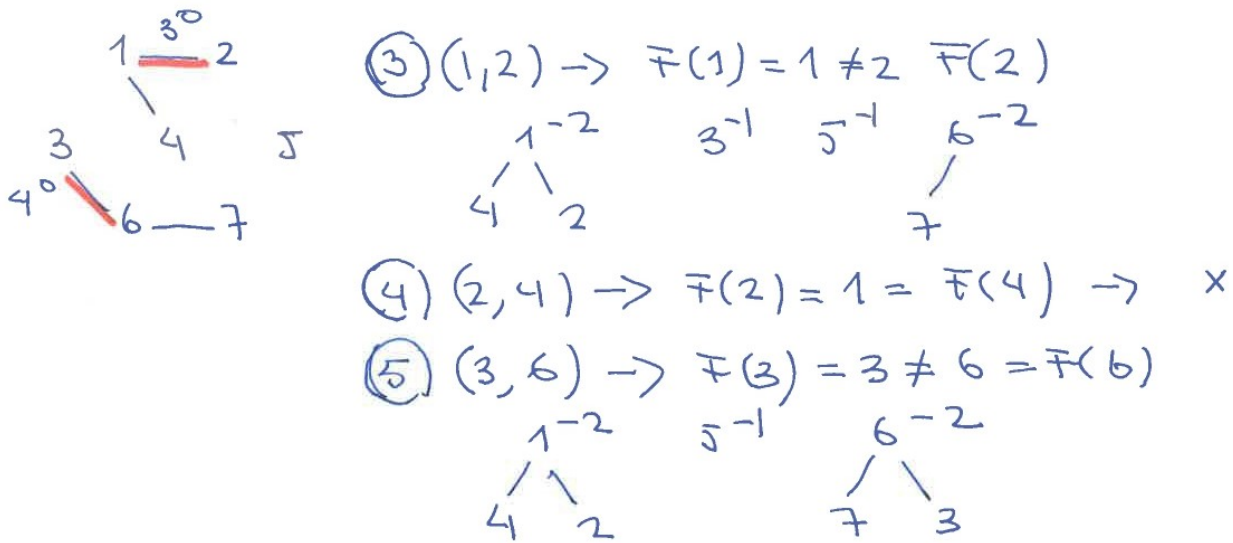
Applying Kruskal's Algorithm (II)

- We maintain separately the Kruskal forest and the DS forest



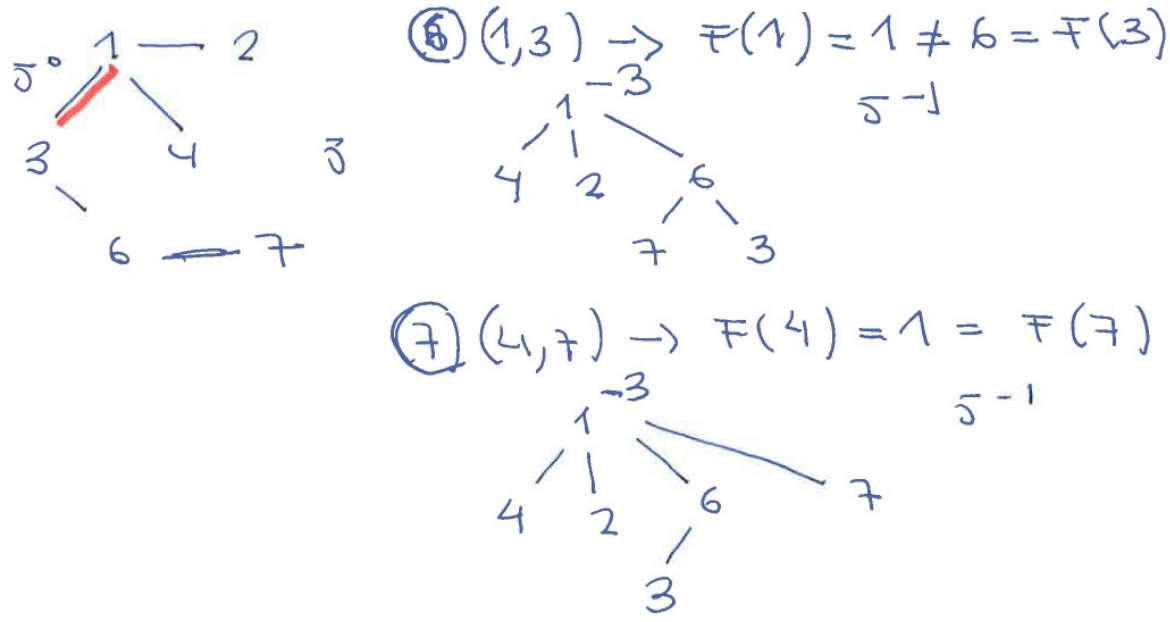
Applying Kruskal's Algorithm (III)

- We process the remaining edges from the PQ



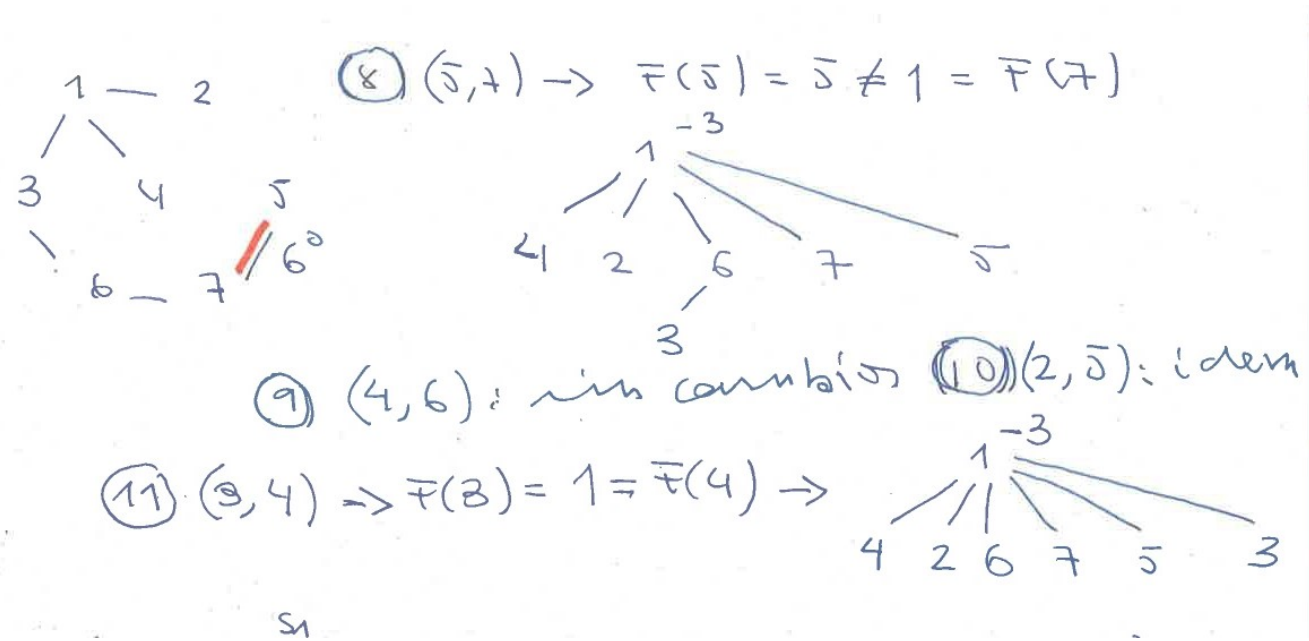
Applying Kruskal's Algorithm (IV)

- We process the remaining edges from the PQ



Applying Kruskal's Algorithm (V)

- We process the remaining edges from the PQW until it is empty
- The MST may not change but the DS forest may



Greedy Change

The Greedy Approach to Optimization

- Greedy algorithms try to solve a **global** optimization problem making at each step a **locally** optimal choice
- This is a natural approach in optimization and usually yields efficient algorithms
  - Sometimes it works: the Dijkstra, Prim and Kruskal (seen in EDyL) algorithms for graphs are greedy
  - Often it doesn't: we'll see this next or the greedy change and knapsack algorithms
  - Another example of an incorrect greedy algorithm is the Nearest Neighbor for the in Traveling Salesman Problem
- In general, greedy algorithms are easy to design but hard to prove correct (and sometimes impossible!)

The Change Problem

- Assume we have work as supermarket cashiers and our clients want change in as few coins as possible
- How can we proceed?
- Simplest idea: give at each step the largest coin smaller than the amount that remains to change
- Example: how to give change of 3,44 euros?
  - Easy: one 2 euro coin, one 1 euro coin, two 20 cent coins, two 2 cent coins
- Have to write down the algorithm but the general idea is **greedy**:
  - We try to minimize **globally** the total number of coins
  - Using **locally** at each step the largest coin possible to minimize the amount still to change

The Greedy Change Algorithm

- We will work with an ordered list of coin values, say

  `l_coin_values = [1, 2, 5, 10, 20, 50, 100, 200]`

  and save the number of coins of each type used in a dict

```
def change(c: int, l_coin_values: List) -> Dict:
    """
    docstring: write it always!!!
    """
    d_change = {}

    for coin in sorted(l_coin_values)[ : : -1]:
        d_change[ coin ] = c // coin
        c = c % coin

    return d_change
```

- But ... it doesn't work for general coin systems: Try to give change of `7` with coins `1, 3, 4, 5`
- But everybody uses it; may it be correct for some coin systems

The Fractionary Knapsack Problem

0–1 Knapsack

- Assume we have N elements with integer weights w_i and values v_i and a knapsack that stands a maximum weight W
- We want a choice of elements $i_1, \ldots, i_K$ such that $\sum_j w_{i_j} \leq W$ and $\sum_j v_{i_j}$ is maximum
- Mathematically, we want to solve a **Constrained Optimization Problem**:

$$\max \sum_{1}^{N} v_i x_i \text{ subject to } \sum_{1}^{N} w_i x_i \leq W \text{ and } x_i \in \{0, 1\}$$

- The “0–1" name derives from the constraint $x_i \in \{0, 1\}$

A Greedy Solution?

- There is a natural greedy solution: order the i by decreasing relative values $\pi_i = \frac{v_i}{w_i}$ and select the first K ones such that

$$\sum_{1}^{K} w_{i_j} \leq W < \sum_{1}^{K+1} w_{i_j}$$

- The cost is $O(N)$ (plus that of sorting) but **the solution may not be optimal**
- Consider 3 elements with values $\{15, 11, 10\}$, weights $\{5, 4, 4\}$ and $W = 8$:
  - The relative values are $\{3, 2.75, 2.5\}$
  - The greedy knapsack takes element 1 with value 15 but the $\{2, 3\}$ knapsack has a value of 21
- However we'll see next that the greedy idea works for **Fractionary** Knapsacks, where we can take any fraction of the weight w_i of an element
  - That is, if we go from an integer–valued problem to a real–valued one

The Fractionary Knapsack Problem

- Again we have N elements with integer weights w_i and values v_i and a knapsack that stands a maximum weight W.
- We want a choice of weights w'_i, $0 \leq w'_i \leq w_i$, of the elements $i_1, \ldots, i_K$ such that $\sum_j w'_{i_j} \leq W$ and $\sum_j v_{i_j}$ is maximum.
- In other words, we may take **any weight fraction** of each element.
- If $\pi_i = v_i / w_i$, the constrained optimization problem we have to solve now is

$$\max \sum_{1}^{N} \pi_i w'_i \text{ subject to } \sum_{1}^{N} w'_i \leq W \text{ and } 0 \leq w'_i \leq w_i.$$

- Now we have **real** constraints (and not integer ones).

The Greedy Knapsack

- We recall the greedy selection: order the i by decreasing relative values $\pi_i = \frac{v_i}{w_i}$ and
  - Select the first K ones such that

$$\sum_{1}^{K} w_{i_j} \leq W < \sum_{1}^{K+1} w_{i_j}.$$

  - Select a fraction $W - \sum_1^K w_i$ of the element $K+1$
- Thus, the greedy knapsack is then
  - $w_i^g = w_i$ when $i \leq K$,
  - $w_{K+1}^g = W - \sum_1^K w_i$ and
  - $w_i^g = 0$ for $K+2 \leq i \leq N$.
- Its cost is $O(N \lg N)$ for sorting and $O(N)$ to select the knapsack.
- The greedy knapsack didn't work in the 0–1 case but now it does.

Optimality of the Greedy Knapsack

- **Proposition:** *For any other selection w'_i we have*

$$V' = \sum_1^N w'_i \pi_i \leq \sum_1^N w^g_i \pi_i = V^g.$$

- To prove it notice that $w^g_i \geq w'_i$ for $i = 1, \ldots K$ and thus,

$$\begin{aligned} V^g - V' &= \sum_1^K \pi_i (w^g_i - w'_i) + \pi_{K+1}(w^g_{K+1} - w'_{K+1}) - \sum_{K+2}^N \pi_i w'_i \\ &\geq \pi_{K+1} \left(\sum_1^{K+1} w^g_i - \sum_1^N w'_i \right) \\ &\geq \pi_{K+1} \left(W - \sum_1^N w'_i \right) \geq 0. \end{aligned}$$

- Therefore, $V^g \geq V'$ and the greedy knapsack is optimum.

Huffman Coding

Coding Alphabets

- Assume an alphabet $\mathcal{A} = \{\alpha_1, \ldots, \alpha_M\}$ and a symbol set $\Sigma = \{\sigma_1, \ldots, \sigma_S\}$.
  - For instance, we could take as $\mathcal{A}$ the ascii characters and $\Sigma = \{0, 1\}$.
- A **codification** of $\mathcal{A}$ in terms of Σ is a variable lenght representation of each α_j as $\beta_j = (\sigma_{j_1} \ldots \sigma_{j_{\ell_j}})$ with $\sigma_i \in \Sigma$.
- We call $\mathcal{C} = \{\beta_1, \ldots, \beta_M\}$ the **code**
- In a **prefix** (or instantaneous) code, no $\beta_i \in \mathcal{C}$ is a prefix of any other β_j.
- We can interpret the character codes of a prefix code as the **leaves of a tree** where each node has at most $S = |\Sigma|$ sons.
- If $\Sigma = \{0, 1\}$, the symbols of a **binary prefix code** (BPC) $\mathcal{C}$ are in the leafs of a binary tree $T_{\mathcal{C}}$.

File Compression

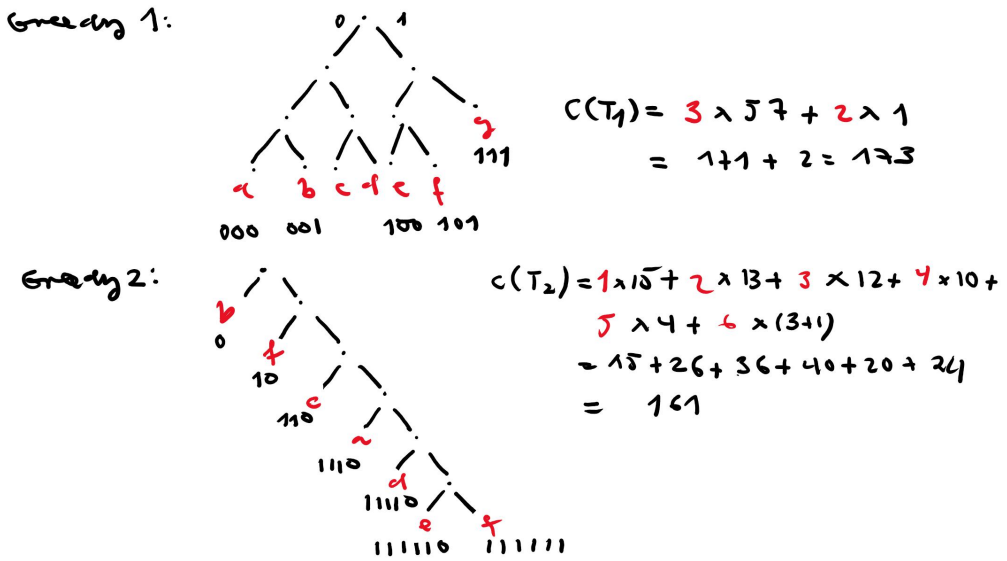
- Assume a file F is made of characters from $\mathcal{A} = \{\alpha_1, \ldots, \alpha_M\}$ and each character α_j appears in F with (absolute) frequency f_j.
- If $\mathcal{C} = \{\beta_1, \ldots, \beta_M\}$ is a BPC for $\mathcal{A}$ and $\ell_j =$ lenght (β_j), the **size in bits** of F compressed by $\mathcal{C}$ will be

$$\tau(F) = \sum_{1}^{M} f_j \ell_j.$$

- Natural question: **how to choose $\mathcal{C}$ so that $\tau(F)$ is minimum?**
- Natural answer: try something greedy.
  - Greedy idea 1: place the symbols as the leaves of a binary tree with depth $\lceil \lg M \rceil$.
  - Greedy idea 2: give one bit to the character most frequent, two to the second and so on.
  - But neither one is optimal.

Greedy Coding

- **Example:** Assume F is made of characters a to g with respective frequencies 10, 15, 12, 4, 3, 13, 1.

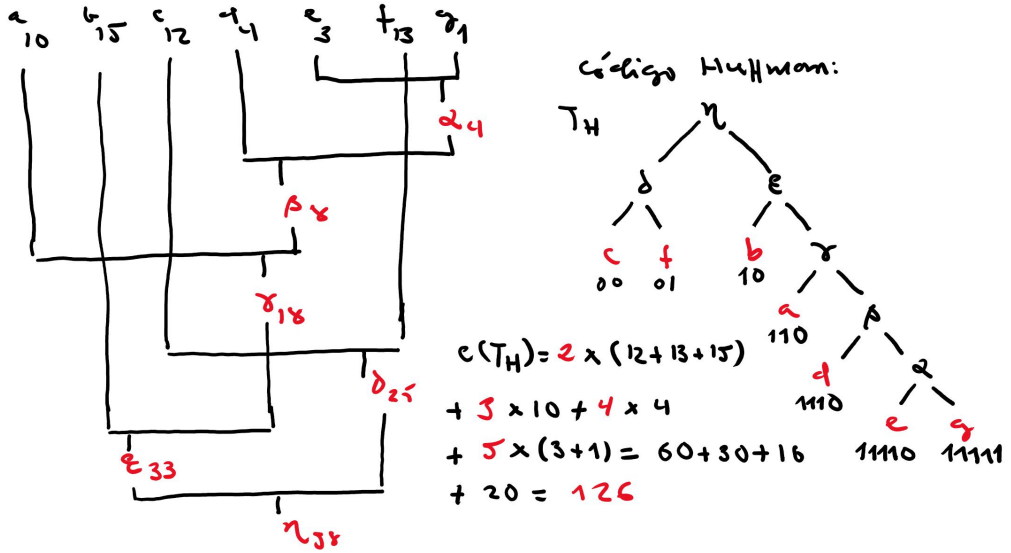


The Huffman Approach

- Huffman's is an iterative greedy algorithm.
- At each step **the two less frequent (meta)–characters of $\mathcal{A}$ are merged as a new meta–character with frequency the sum of the frequencies of the merged ones.**
- Each meta–character can be seen as a partial BPC
  - We thus work with a forest of partial BPCs.
- After $M - 1$ such merges we arrive at a single meta–character ρ that we put at the root of a binary tree.
- We obtain the BPC tree $T_{\mathcal{C}}$ **de–merging the various meta–characters in the inverse order in which they were created**.

A Huffman Code

- **Example:** Assume F is made of characters a to g with respective frequencies 10, 15, 12, 4, 3, 13, 1.
- The Huffman process and final coding are the following



Some Facts on Huffman Coding

- There may be several different Huffman BPCs for the same file F.
- Huffman coding requires three steps:
  - A pass through F to compute character frequencies, with cost $O(N)$ if F has N characters.
  - The actual computation of the BPC, with a cost $O(M \lg M)$ if we keep the meta-characters in a min–heap.
  - A second pass through F to compress it, with cost $O(N)$.
- The total cost is $O(N)$ but with two passes through F.
- There are one–pass alternatives such as the Lempel–Ziv family of compression algorithms.

Optimality of Huffman Codes

- **Theorem.** *Huffman codes $\mathcal{C}_H$ are optimal BPCs in the sense that if $\mathcal{C}'$ is another BPC for F, $\tau_{\mathcal{C}_H}(F) \leq \tau_{\mathcal{C}'}(F)$.*
  The proof relies on two lemmas.
- **Lemma 1.** *There is an optimal BPC for which the two less frequent characters are at the deepest leaves, have the same lenght and differ only in the last bit.*
- **Lemma 2.** *Let Σ be an alphabet with frequencies f_j and Σ' another in which we merge the two less frequent characters c_i, c_j in another c' with frequency $f_i + f_j$ and leave the other characters and frequencies unchanged. Assume that T' is a BPC for Σ' and T is obtained by demerging T'. Then, if T'* ***is an optimal BPC for Σ', T is an optimal BPC for Σ.***

Proving the Lemmas

- Just a few pointers to how to prove them
- To prove the first lemma, one assumes that c_1, c_2 are the less frequent characters and that c'_1, c'_2 are the two “deepest" characters in an optimum T, with $f'_1 \leq f'_2$.
- Since $f_1 \leq f'_1$ and $f_2 \leq f'_2$ but $\ell_1 \leq \ell'_1$ and $\ell_2 \leq \ell'_2$, is easy to see that interchanging them diminishes $\tau(F)$.
- The second lemma captures the optimal substructure of any optimal code.
- To prove it one writes down $\tau_T(F)$ in terms of $\tau_{T'}(F)$ and then checks that if T is not optimal, neither is T'.

End of the Argument

- The optimality of a Huffman code follows from these lemmas.
  - Lemma 1 is applied in each of the merging steps.
  - Lemma 1 also ensures the starting condition.
  - Lemma 2 ensures that at each merging/demerging step of the algorithm we go from an optimal code to another.
- The iterative demerging phase starts from a obviously optimal single leaf BPC and uses Lemma 2 at each step.
- We must thus arrive at an optimal Huffman BPC at the end of the demerging iterations.
- Lemma 2 is a kind of **loop invariant**: a condition which is true after each iteration and that at the end guarantees the algorithm to be correct.

Loop Invariants

- Recall: a loop invariant is a condition that remains true after each loop and that “leads" the algorithm towards a correct solution
- The standard way to prove the correctness of an iterative algorithm is to find an adequate loop invariant for its iterations
- Example: loop invariants for InsertSort or BubbleSort to sort a table T between indices p and u
  - InsertSort: after iteration i, $i = p + 1, \ldots, u$, the subtable $T[p], \ldots, T[i]$ is sorted
  - BubbleSort: after iteration i, $i = u, \ldots, p + 1$, the subtable $T[i], \ldots, T[u]$ is sorted
- In both cases we easily check that the loop invariant assertion is correct after each iteration
- Thus, after iterations u or $p + 1$, respectively, the table is sorted

Information and Entropy

Information and Disorder

- Natural question: Given a file F, how much can it be compressed?
- Intutively, the amount of “information” in F should establish a bound on how much it can be compresssed.
- New question: **How do we measure information?**
- Intuitively, the more “disordered" the content of a file is, the larger the information needed to describe it.
- In other words, we can assimilate information to disorder which leads to a new question:
  How do we measure disorder?

Entropy

- Disorder in the physical world is well studied and understood through Thermodynamics using a key physical magnitude, **entropy**.
- Claude Shannon introduced entropy in information theory.
- The **entropy** of a discrete probability distribution $P = (p_1, \ldots, p_M)$ is given by

$$H = H(P) = \sum_1^M p_i \lg \frac{1}{p_i} = -\sum_1^M p_i \lg p_i.$$

- Intuitively, we may think of characters α_i appearing with probabilities p_i and where the information of the i–th character needs $\lg \frac{1}{p_i}$ bits to be stored.
  - Longer characters have “more information” (because they are less frequent!!).
- Thus, we can see the entropy H as the average information in bits of the characters $\alpha_1, \ldots, \alpha_M$.

Entropy Properties

- Obviously $H(P) \geq 0$ for any discrete distribution P.
- Obviously $H(0, \ldots, 1, \ldots, 0) = 0$.
  - In other words, since only one of the M characters will actually happen, there is no information in the character set.
- The entropy of the uniform distribution is $H\left(\frac{1}{M}, \ldots, \frac{1}{M}\right) = \lg M$.
- **Proposition.** *The uniform distribution has the largest entropy; i.e., we have $H(P) \leq \lg M$ for any P.*
  - To prove it, consider for any $P = (p_1, \ldots, p_M)$ the Lagrangian

$$L(P) = -\sum_1^M p_i \lg p_i + \lambda(1 - \sum p_i),$$

solve $\frac{\partial}{\partial p_i} L(P) = 0$ and plug it back into the constrain $\sum p_i = 1$.

Entropy and Optimal Compression

- Assume F has N characters with absolute frequencies f_i.
- The relative frequencies $p_i = f_i/N$ form a discrete probability distribution P.
- If we compress F with a BPC code $\mathcal{C}$ of M symbols with ℓ_i bits, its average character lenght in bits $ACL_{\mathcal{C}}$ is

$$ACL_{\mathcal{C}}(F) = \sum_{1}^{M} \ell_i \frac{f_i}{N} = \sum_{1}^{M} \ell_i p_i.$$

- **Theorem.** *For any BPC code $\mathcal{C}$, we have*

$$ACL_{\mathcal{C}}(F) \geq H(p_1, \ldots, p_M). \tag{1}$$

- In other words, the entropy H of the relative character frequencies in a file gives a bound for the **minimum average number of bits per character**.

Kraft's Inequality

- **Proposition.** *If the symbols in a BPC $\mathcal{C}$ have lenghts ℓ_i, then*

$$\sum_{1}^{M} 2^{-\ell_i} \leq 1. \tag{2}$$

- To prove it, recall that ℓ_i is the depth of the i–th symbol in $T_{\mathcal{C}}$ and choose $K \geq \ell_i$ for all i.
- Let T the binary tree of depth K built by growing $T_{\mathcal{C}}$ adding to each of its leaves descendants up to depth K.
- The maximum number of leaves in T is 2^K but each leaf of $T_{\mathcal{C}}$ adds $2^{K-\ell_i}$ leaves to T.
- We must thus have $\sum_1^N 2^{K-\ell_i} \leq 2^K$, and, therefore, (2) follows.

Proving the Entropy Bound

- Denoting the entropy by H and $ACL(\ell_1, \ldots, \ell_M)$ by L, we have:

$$\begin{aligned}
H - L &= \sum_1^M p_i \left(\lg \frac{1}{p_i} - \ell_i \right) = \sum_1^M p_i \lg \frac{2^{-\ell_i}}{p_i} \\
&= \frac{1}{\log 2} \sum_1^M p_i \log \frac{2^{-\ell_i}}{p_i} \leq \frac{1}{\log 2} \sum_1^M p_i \left(\frac{2^{-\ell_i}}{p_i} - 1 \right) \\
&= \frac{1}{\log 2} \sum_1^M p_i \left(\frac{2^{-\ell_i} - p_i}{p_i} \right) = \frac{1}{\log 2} \sum_1^M \left(2^{-\ell_i} - p_i \right) \\
&= \frac{1}{\log 2} \left(\sum_1^M 2^{-\ell_i} - 1 \right) \leq 0.
\end{aligned}$$

where we have used that $\log x \leq x - 1$ and Kraft's inequality.

Shannon Coding

- In general $\lg 1/p_i$ will not be an integer but $\lceil \lg 1/p_i \rceil$ is.
- Shannon (or Shannon–Fano) codes build a BPC with at most $\lceil \lg 1/p_i \rceil$ bits per character.
- If $\mathcal{S}$ is a Shannon code, we have

$$ACL_{\mathcal{S}} = \sum_{1}^{M} p_i \left\lceil \lg \frac{1}{p_i} \right\rceil \leq 1 + \sum_{1}^{M} p_i \lg \frac{1}{p_i}.$$

- Thus the average number of bits per character in a Shannon code is at most 1 above the entropy optimum.
- To build a Shannon code,
  - We sort the alphabet symbols by decreasing frequencies,
  - And iteratively split them in left and right parts so that their probability sum is about the same.
- We assign then a bit 0 to the left part and a bit 1 to the right one.

Shannon Coding II

- It can be shown that this ensures lenghts such that $\ell_i \leq \lceil \lg 1/p_i \rceil$.
- **Example:** Assume again F is made of characters a to g with respective frequencies 10, 15, 12, 3, 4, 13, 1.
- The Shannon coding process is:

nb	c	f										codigos				
2	*b*	15	**15**									*0*	*0*			
2	*f*	13	**28**									*0*	*1*			
2	*c*	12	40	12	**12**							*1*	*0*			
3	*a*	10	50	10	22	10	**10**					*1*	*1*	*0*		
4	*d*	4	54	4	26	4	14	4	**4**			*1*	*1*	*1*	*0*	
5	*e*	3	57	3	29	3	17	3	7	3	**3**	*1*	*1*	*1*	*1*	*0*
5	*g*	1	58	1	30	1	18	1	8	1	4	*1*	*1*	*1*	*1*	*1*
	tot	*58*		*30*		*18*		*8*		*4*						
	mit	*29*		*15*		*9*		*4*		*2*						

- In this case it coincides with the Huffman optimum code.

A Huffman and Shannon Example

- We can check directly that for characters a to e with frequencies 15, 7, 6, 6, 5, the Huffman code gives a better compression.

The Traveling Salesman Problem

The Traveling Salesman Problem

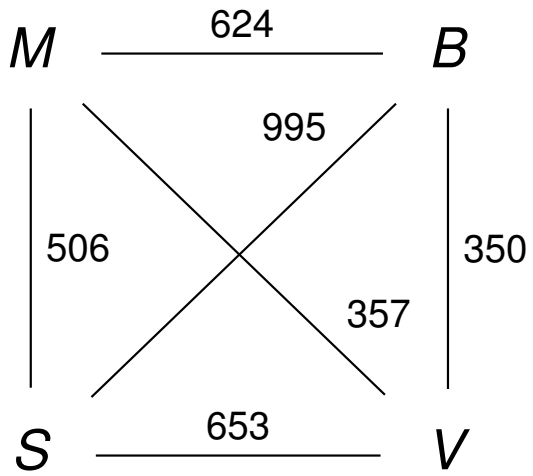
- We give for TSP a natural greedy algorithm that **doesn't work**
- We have a set of N cities $c_0, \ldots, c_{N-1}$ and we know their pairwise distances $d_{i,j} = \text{dist}(c_i, c_j)$
- At a given time we can go from any city to another
- We want to visit them in a **circuit** starting at c_0 and **minimizing the total distance** of the circuit
- Natural greedy strategy: from city c_i we go to the **nearest city not visited yet**

A TSP Example

- Simple example:

```
cities = ["madrid", "barcelona", "sevilla", "valencia"]
```

- The (complete) graph is



- And distances can be given by a 4×4 Numpy matrix

A Greedy Algorithm for TSP

- A first Python greedy code can be the following:

```
dist_m = np.random.randint(0, 200, (num_cities, num_cities) )
dist_m = (dist_m + dist_m.T) // 2
dist_m = dist_m - np.diag( np.diag(dist_m) )

def greedy_tsp_circuit(dist_m: np.ndarray) -> np.ndarray:
    num_cities = dist_m.shape[0]
    circuit = [0]
    while len(circuit) < num_cities:
        current_city = circuit[-1]

        # sort cities in ascending distance from current
        options = np.argsort(dist_m[ current_city ])

        # add first city in sorted list not visited yet
        for city in options:
            if city not in circuit:
                circuit.append(city)
                break

    return np.array(circuit + [0])
```

Is It Correct?

- No, and it is not difficult to find simple examples that show it
- In fact, the general forms of the change, knapsack and TSP problems are **among the hardest problems** in algorithmics
- They are examples of **NP complete problems**
  - Essentially the only way to find always the optimal solution is to enumerate and analyze all possible solutions
- But companies must solve thousands of TSP problems every morning
- What to do in practice?
  - Design **approximate algorithms** that ensure solutions not too far away from the optimal ones
- How to design approximate algorithms?
  - Often by clever greedy analyses!!!

Approximation Algorithms

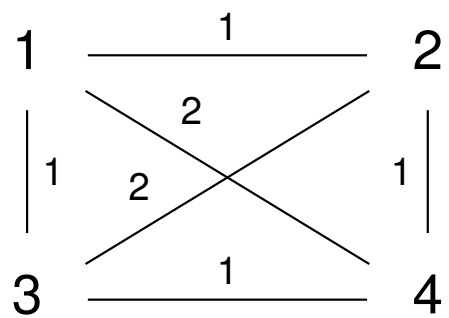
- Alternative: **approximate** algorithms
- **Definition:** Given an optimization problem $\mathcal{P}$, an **approximate algorithm** for $\mathcal{P}$ **with bound** $\lambda \geq 1$ is an algorithm A that for every input I returns a solution $s_A(I)$ such that

$$c^*(I) \leq c(s_A(I)) \leq \lambda c^*(I)$$

  with $c^*(I)$ the optimal cost for $\mathcal{P}$ on I
- NN is not exactly an approximate algorithm for TSP, since its bound is $O(\log |V|)$ and depends on $|V|$
- We will get an approximate algorithm for TSP through MSTs and Eulerian Circuits

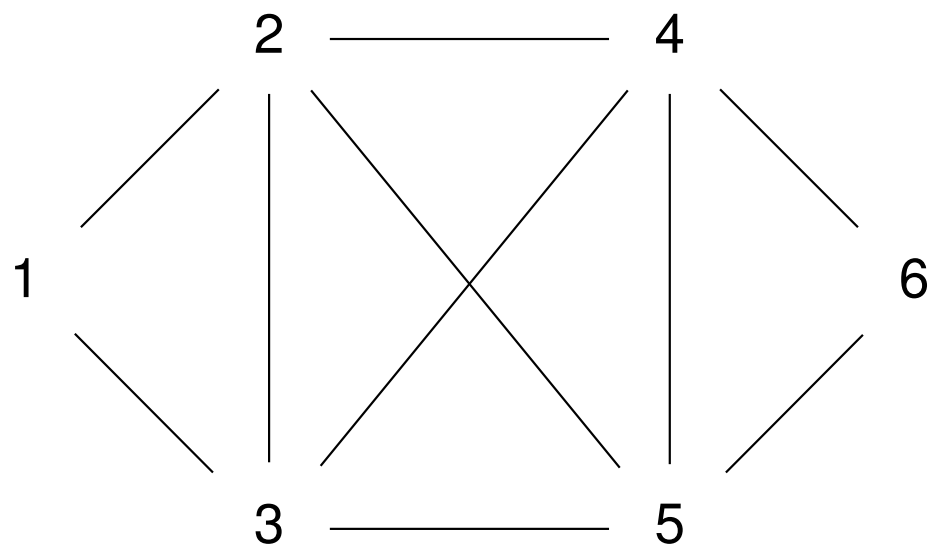
Minimum Spanning Trees

- We recall some graph definitions
- A **tree** is an undirected, connected, acyclic graph
- Given a (connected) undirected graph $G = (V, E)$, a **spanning tree** is a tree $T = (V, E_T)$ where $E_T \subset E$
- If G is a weighted graph with cost function c, a spanning tree T is **minimum** if $c(T) = \sum_{(u,v)\in E_T} c(u, v) \leq c(T')$ for any other spanning tree T'
- For example, the edges $(1, 2), (1, 3), (3, 4)$ define a minimum spanning tree T with cost $c(T) = 3$ for



Eulerian Circuits

- Given a (connected) undirected graph $G = (V, E)$, a **Eulerian circuit** is a circuit on G which traveses all the edges in E **only once**
- The **degree** $d(u)$ of a node in G is the number of edges that start (or end) at u
- **Euler's Theorem** G contains an Eulerian circuit if and only if $d(u)$ is even for every node $u \in V$
- For example, the graph below has Eulerian circuits



Approximation Algorithms for TSP

- **Proposition:** *If the cost function is Euclidean, i.e., it verifies*

$$c(u, v) \leq c(u, w) + c(w, v) \text{ for all } u, v, w \in V,$$

then there is an approximate algorithm for TSP with $\lambda = 2$

- The concrete algorithm is:

```
def euclideanTSP(g, c):
    """Apply all steps by inspection
    """
    find a MST t on g
    duplicate its edges to obtain a graph g_1

    #now each node in g_1 has degree 2 and there is an EC
    find a EC p_1 in g_1

    shortcut seen edges in p_1 to get circuit p
    return p
```

Approximation Algorithms for TSP II

- **Proof sketch:**
  - Let T_1, π_1 and π be the MST, the EC and the circuit returned by the algorithm
  - Let π^* be an optimal circuit and remove an edge on π^* to get a spanning tree T^*
  - Since T_1 is an MST, we have $c(T_1) \leq c(T^*) \leq c(\pi^*)$
  - By the Euclidean distance property, if we shortcut the segment $u \to w \ldots \to z \to v$ to $u \to v$, we have

$$\underbrace{c(u,v)}_{\pi} \leq \underbrace{c(u,w) + c(w,x) + \ldots + c(z,v)}_{\pi_1}$$

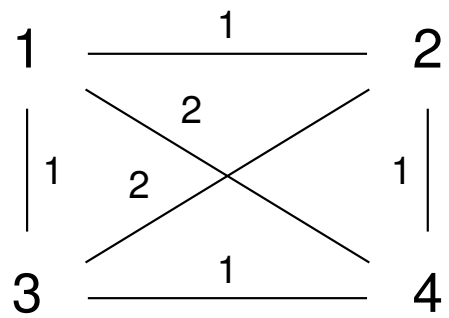
  - We then we conclude that

$$c(\pi) \leq c(\pi_1) = 2c(T_1) \leq 2c(\pi^*)$$

- The **Christofides** algorithm improves this to $\lambda = 1.5$ (see this article in Wired for more about the algorithm)

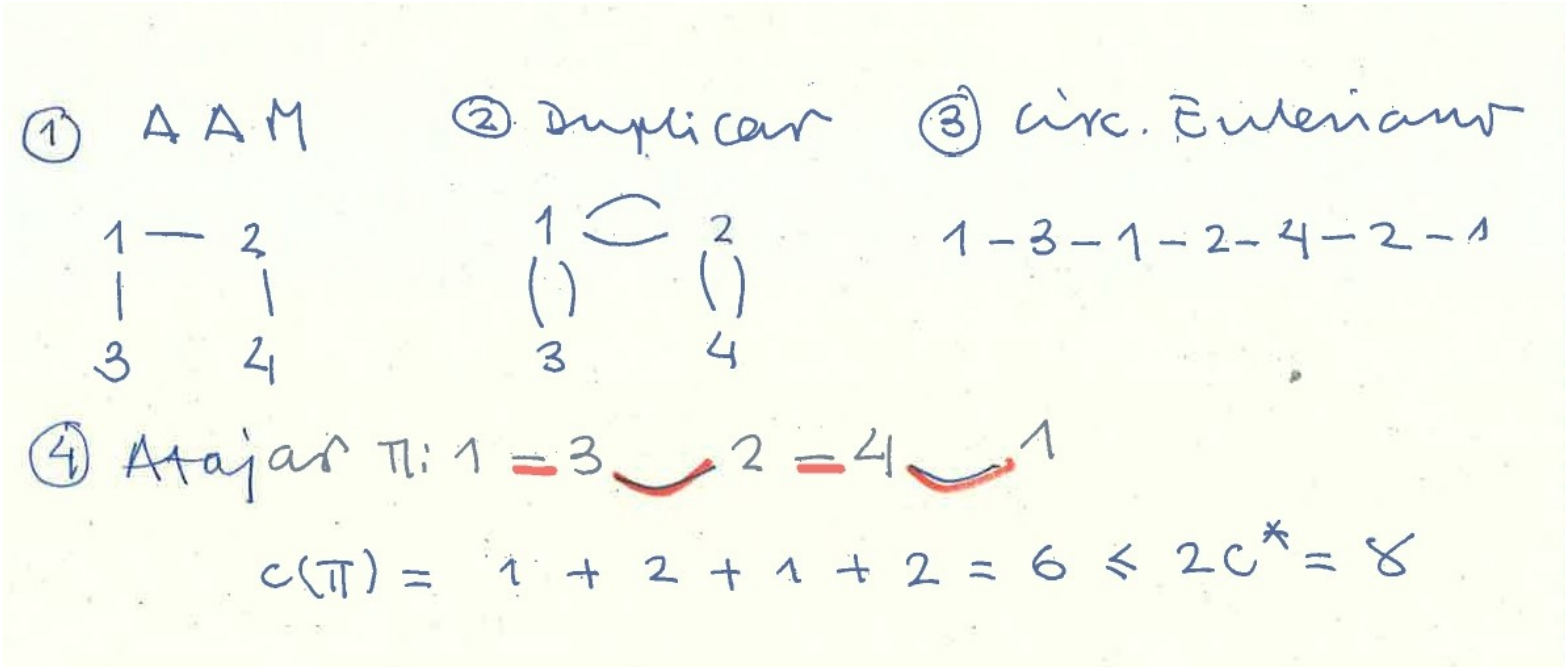
Approximation Algorithms for TSP III

- To learn more: Johnson, McGeoch, The Traveling Salesman Problem: A Case Study in Local Optimization
  - Or the movie The Travelling Salesman
- **Example**



Applying The Algorithm

- The steps to find an approximate TSP solution



Basic Examples

Recall: Divide And Conquer

- Recursive algorithms often derive from a **Divide and Conquer** strategy:
  - Divide a problem P in M subproblems P_m
  - Solve these separately getting solutions S_m
  - Combine these solutions in a solution S of P
- Classical examples are recursive sorting algorithms such as **MergeSort** and **QuickSort**, searching algorithms such as **Binary Search** or many tree (and graph) algorithms
- In general Recursion is a powerful design tool
  - There is no other way to devise algorithms for problems such as the Hanoi Towers puzzle
- But recursive algorithms can be very inefficient, for instance, when the subproblems aren't substantially smaller
- And even in this case, recursion always imposes extra hidden costs

Binary Search

- Binary search over sorted lists is good example of a recursive algorithm

```
def bin_search(t, key, first, last):
    if last >= first:
        mid = (first + last) // 2
        if t[mid] == key:
            return mid
        else:
            if t[mid] < key:
                return rec_bb(t, mid+1, last, key)
            else:
                return rec_bb(t, first, mid-1, key)
```

- It is simple and elegant but also recursion is easy to remove

Efficiency and Recursion

- Recursive algorithms are effective in the sense that they actually solve the base problems
- But in general they are not **efficient**, as they are always costly:
  - Recursion has to be removed somehow before execution
  - Sometimes that is easy, as is the case when there is **tail** recursion:
- Tail recursion: there is no executable code after the return of a recursive call
  - This happens in binary search
  - Also in the second recursive call in Hanoi
- But in general automatic recursion removal requires a considerable stack overhead
  - Extensive hand-crafting is needed to end up with truly efficient algorithms

Non Recursive Binary Search

- Tail recursion is very easy to remove

```
def bin_search(t, key, first, last):
    while first <= last:
        mid = (first + last) // 2

        if key == t[mid]:
            return mid
        elif key < t[mid]:
            last = mid-1
        else:
            first = mid+1
```

- And execution times are better (specially in compiled languages)

Recursive Matrix and Number Multiplication

Recursive Matrix Multiplication I

- If A, B are $N \times N$ matrices, the standard algorithm to multiply $C = A \times B$ has a cost N^3
- If we assume N even, breaking A, B and C in $N/2 \times N/2$ blocks A_{ij}, B_{ij}, C_{ij}, and computing

$$C_{ij} = A_{i1} B_{1j} + A_{i2} B_{2j}$$

  suggests that one $N \times N$ multiplication can be reduced to eight $N/2 \times N/2$ multiplications
- In turn, this suggests a simple recursive matrix multiplication algorithm, *RMM*
- Assuming N even, its cost $T_{RMM}(N)$ verifies

$$T_{RMM}(N) = 8\, T_{RMM}\left(\frac{N}{2}\right)$$

Recursive Matrix Multiplication II

- Unwinding $T_{RMM}(N)$ for $N = 2^K$, we get

$$T_{RMM}(N) = 8\,T_{RMM}\left(\frac{N}{2}\right) = 8^2\,T_{RMM}\left(\frac{N}{2^2}\right) = \ldots = 8^K\,T_{RMM}\left(\frac{N}{2^K}\right)$$

  and since $T_{RMM}(1) = 1$, we get

$$T_{RMM}(N) = 8^K = (2^K)^3 = N^3$$

- Thus, there is no gain over the standard algorithm

Strassen's Algorithm

- But if we can decompose one $N \times N$ multiplication into M multiplications of $N/2 \times N/2$ matrices, the recurrence becomes

$$T_{RMM}(N) = M \times T_{RMM}\left(\frac{N}{2}\right)$$

that unwinds as

$$T_{RMM}(N) = M^K = (2^{\lg M})^K = (2^K)^{\lg M} = N^{\lg M}$$

- Thus, we improve on the classical algorithm if $M < 8$
- Strassen's algorithm achieves this with $M = 7$
- It uses seven matrices $M_1, \ldots, M_7$ and formulae such as $C_{11} = M_1 + M_4 - M_5 + M_7$, $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$ and so on
- See en.wikipedia.org/wiki/Strassen_algorithm

Recursive Number Multiplication

- The same ideas can be used for number multiplication: if A has an even number N of bits we can write it as

$$\begin{aligned} A &= a_{N-1}2^{N-1} + \ldots + a_{N/2}2^{N/2} + a_{N/2-1}2^{N/2-1} + \ldots + a_0 \\ &= \left(a_{N-1}2^{N/2-1} + \ldots + a_{N/2}\right)2^{N/2} + a_{N/2-1}2^{N/2-1} + \ldots + a_0 \\ &= A_1 2^{N/2} + A_0 \end{aligned}$$

- If AB also has N bits and we decompose A and B as above, we have

$$AB = A_1B_1 2^N + (A_1B_0 + A_0B_1)2^{N/2} + A_0B_0,$$

which suggests a recursive number multiplication algorithm *RNM*

Analyzing Recursive Multiplication

- Its cost function in terms of binary multiplications verifies

$$T_{RNM}(N) = 4 \times T_{RNM}\left(\frac{N}{2}\right) \text{ and } T_{RNM}(1) = 1$$

- If $N = 2^K$, unwinding $T_{RNM}(N)$ gives

$$T_{RNM}(N) = 4^K \times T_{RNM}\left(\frac{N}{2^K}\right) = 4^K = (2^K)^2 = N^2$$

- We thus get the same cost of the standard number multiplication algorithm
- But we can easily improve on this

Karatsuba's Algorithm

- Notice that we can also write

$$AB = A_1 B_1 2^N + ((A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0)\, 2^{N/2} + A_0 B_0,$$

  requiring only 3 multiplications of $N/2$–bit numbers (but 4 sums)
- If we build a recursive algorithm RNM_K on this, its cost function verifies $T_{RNM_K}(N) = 3 \times T_{RNM_K}\left(\frac{N}{2}\right)$ and $T_{RNM_K}(1) = 1$
- For $N = 2^K$, it unwinds as

$$T_{RNM_K}(N) = 3^K = (2^K)^{\lg 3} = N^{\lg 3}$$

- This plus a clever implementation constitute Karatsuba's method for (large) number multiplication, used for instance in Public Key Cryptography

The Selection Problem

The Selection Problem

- Given a table T of size n and an index k, return the **value of the element that occupies the k–th position in a sorting of T**
- Simplest solution: sort T and return $T[k]$
- Its cost is $O(n \log n)$, too large if we have to find the maximum $k = n - 1$ or the minimum $k = 0$, for which standard algorithms have cost $n - 1$
- An **adversary analysis** can be used to show that this gives a lower bound: any algorithm must have a cost $\Omega(n)$
- An important case is the computation of the **median** with $k = \lfloor n/2 \rfloor$
- Q: can we solve the general selection problem in linear time?

Tweaking QuickSort

- One idea is to somehow mix QuickSort and Binary Search: assuming $p = 0$, $u = n - 1$,
  - Compute the approximate middle index `m` returned by the `split` routine of QuickSort and return `T[m]` it if `k == m`
  - If `k < m` look for the k–th element in the left subtable
  - If `k > m` look for the $k - m - 1$–th element in the right subtable
- The known analysis of QuickSort hints at some results:
  - The algorithm will have cost $\Omega(n)$, since the cost of `split` is $n - 1$
  - If we use the first element as the pivot in `split` and want to find the last element, the cost will be $\Theta(n^2)$
  - But there is only one recursive call so we should get a linear–cost average case

Quick Select

- The Python code to find the element that ocuppies position `t[k]` with $p \leq k \leq u$ in the sorted list could be

```
def split(t: List[int], p: int, u: int, pivot=0) -> Tuple[int, List[
    int]]:
    mid_val = t[p + pivot]
    a = np.array(t)

    idx1 = a < mid_val
    idx2 = a > mid_val

    return idx1.sum(), list(a[idx1]) + [mid_val] + list(a[idx2])

def qselect(t: List[int], p: int, u: int, k: int, pivot=0) -> int:
    if k > u or k < p:
        return

    m, t1 = split(t, p, u) #middle index

    if k == m:
        return t[k]

    elif k < m:
        return qselect(t1, p, m-1, k)

    else:
        return qselect(t1, m+1, u, k)
```

- Now `k` is not an ordinal position but an index

Quick Select II

- Now the Python code to find the `k` -th element of the sorted list could be:

```
def split2(t: List) -> Tuple[List, int, List]:
    mid = t[0]
    t_l = [u for u in t if u < mid]
    t_r = [u for u in t if u > mid]
    return t_l, mid, t_r


def qselect2(t: List, k: int) -> int:
    if len(t) == 1 and k == 0:
        return t[0]

    t_l, mid, t_r = split2(t)
    m = len(t_l)
    if k == m:
        return mid
    elif k < m:
        return qselect2(t_l, k)
    else:
        return qselect2(t_r, k-m-1)
```

- Here `k` refrers to standard order position

Average Case of Quick Select I

- **Proposition.** *If $A_{QSel}(n,k)$ denotes the average number of key comparisons (KCs) that QSelect does, we have $A_{QSel}(n,k) \leq 4n$ for all k*
- **Proof sketch:** We write and solve a recurrence equation
  - Assuming $p = 0$, $u = n-1$, we have

$$n_{QSel}(T, 0, n-1, k) = n - 1 + \nu,$$

    with $n - 1 = n_{split}(T, 0, n-1)$ and ν the number of recursive KCs
  - If `split` returns m and $k = m$, then $\nu = 0$
  - When $k < m$ we have

$$\nu = n_{QSel}(T, 0, m-1, k) \simeq A_{QSel}(m, k)$$

  - And when $k > m$,

$$\nu = n_{QSel}(T, m+1, n-1, k-m-1) \simeq A_{QSel}(n-m-1, k-m-1)$$

Average Case of Quick Select II

- Assuming all cases equiprobable, we have

$$\begin{aligned} A_{QSel}(n,k) \quad \simeq \quad & n-1+ \\ & \frac{1}{n}\sum_{m=k+1}^{n-1} A_{QSel}(m,k)+ \\ & \frac{1}{n}\sum_{m=0}^{k-1} A_{QSel}(n-m-1,k-m-1) \end{aligned}$$

- And applying induction on this we arrive at $A_{QSel}(n,k) \leq 4n$

Improving the Worst Case

- The worst case of QuickSelect (and of QuickSort) comes from the bad behavior over ordered tables of the pivot used in `split`
- We need to improve pivot selection, ensuring that both subtables are large enough
- Assuming $N = 10L + 5$, we can use QuickSelect itself proceeding as follows:
  - Split T into $2L + 1$ consecutive subtables T_i of 5 elements
  - For each subtable T_i obtain its median m_i: it can be done by MergeSort with at most 8 key comparisons
  - Obtain the median m^* of $T' = [m_1, \ldots, m_{2L+1}]$ by recursive QuickSelect with this kind of pivoting and return it as the pivot
- It can be shown that with this pivot, **the sizes of the left and right subtables are** $\leq \frac{7}{10}N$
- We call QuickSelect with this pivoting **QSelect5**

The Cost of QSelect5

- Set $W_{QS5}(N) = \max_{\sigma \in \Sigma_N, k} n_{QS5}(\sigma, k)$; then
  - Computing the $N/5$ medians has a cost of $8\frac{N}{5}$ kcs
  - It then recursively applies QuickSelect5 over a table with $N/5$ elements, with a cost at most $W_{QS5}(\frac{N}{5})$
  - We also have $|\sigma_{left}|, |\sigma_{right}| \leq 7N/10$
- Thus, the overall cost of QuickSelect5 can be bounded as

$$\begin{aligned} n_{QS5}(\sigma, 1, N, k) &= n_{Pivot5}(\sigma, 1, N) + n_{split}(\sigma, 1, N) + \\ &\quad \max_m \{ n_{QS5}(\sigma_{left}, k), n_{QS5}(\sigma_{right}, k - m) \} \\ &\leq \frac{8N}{5} + W_{QS5}\left(\frac{N}{5}\right) + N - 1 + \\ &\quad W_{QS5}\left(\frac{7N}{10}\right) \end{aligned} \tag{3}$$

Worst Case of QuickSelect5

- **Proposition:** *We have* $W_{QS5}(N) \leq 26N$
- ***Proof sketch:*** We will apply induction using (3)
- Assuming $W_{QS5}(N') \leq 26N'$ for $N' < N$, we have using (3):

$$\begin{aligned} W_{QS5}(N) &\leq \frac{8N}{5} + N - 1 + W_{QS5}\left(\frac{N}{5}\right) + W_{QS5}\left(\frac{7N}{10}\right) \\ &\leq \frac{8}{5}N + N - 1 + \frac{26}{5}N + 26\frac{7}{10}N \end{aligned}$$

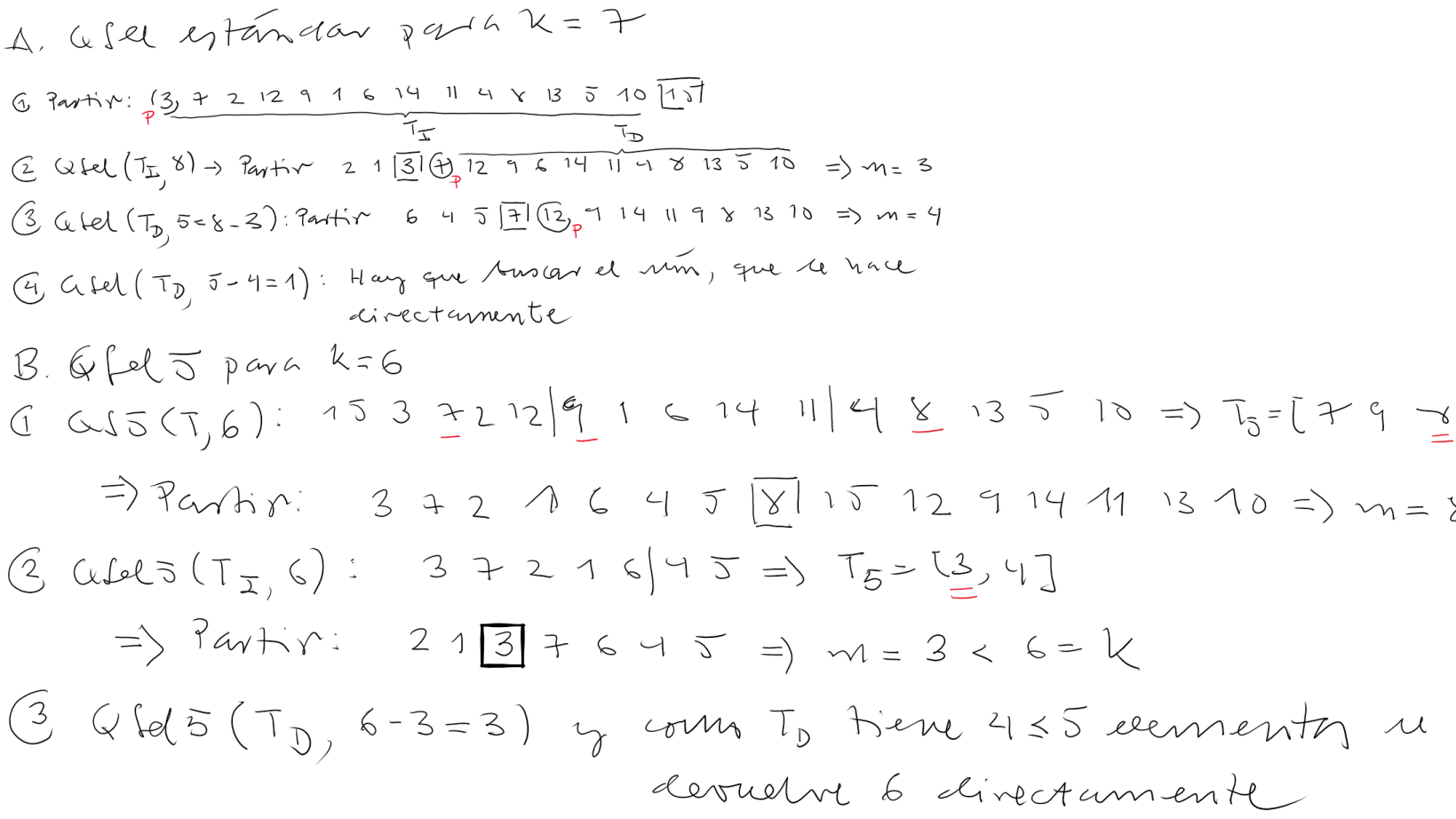
- And putting everything together, we have

$$W_{QS5}(N) \leq \frac{16 + 10 + 52 + 182}{10}N - 1 \leq 26N$$

Examples

- Evolution of `QuickSelect` and `QuickSelect5` over

  `15 3 7 2 12 9 1 6 14 11 4 8 13 5 10.`



What About QuickSort?

- Taking as the pivot the first element of a table, the average case cost of QuickSort was $O(N \lg N)$ but the worst case cost was $O(N^2)$
- The reason was that a first element pivot will be in the average near the middle of the table, but it may also be at the beginning or the end
- This was also the case with the first version of QuickSelect but was fixed in QuickSelect5
- If we use a median of 5 pivot, an $O(N \lg N)$ worst case for QuickSort can also be achieved

Basic Concepts on Graphs

1. Python Basics
2. Algorithms
3. Abstract Data Types and Data Structures
4. Greedy Algorithms
5. Recursive Algorithms
   - Basic Examples
   - Recursive Matrix and Number Multiplication
   - The Selection Problem
   - Basic Concepts on Graphs
   - Depth First Search and Edge Classification
   - Graph Connectivity
   - DAGs and Topological Sort
6. Dynamic Programming

Definitions

- Graph: Pair $G = (V, E)$ of a set V of vertices (nodes) and a set E of edges (u, v) with $u, v \in V$
- In general, graphs are **directed**, as edges imply direction: in (u, v) we “go" from u to v
- **Undirected** graphs: $(u, v) \in E$ iff $(v, u) \in E$
- **Unweighted** graphs: we only consider edge structure
- **Weighted** graphs: edges (u, v) have weights w_{uv}
  - These could be, for instance, distances between cities
- **Multigraphs**: there might several edges between two vertices and also between a vertex and itself

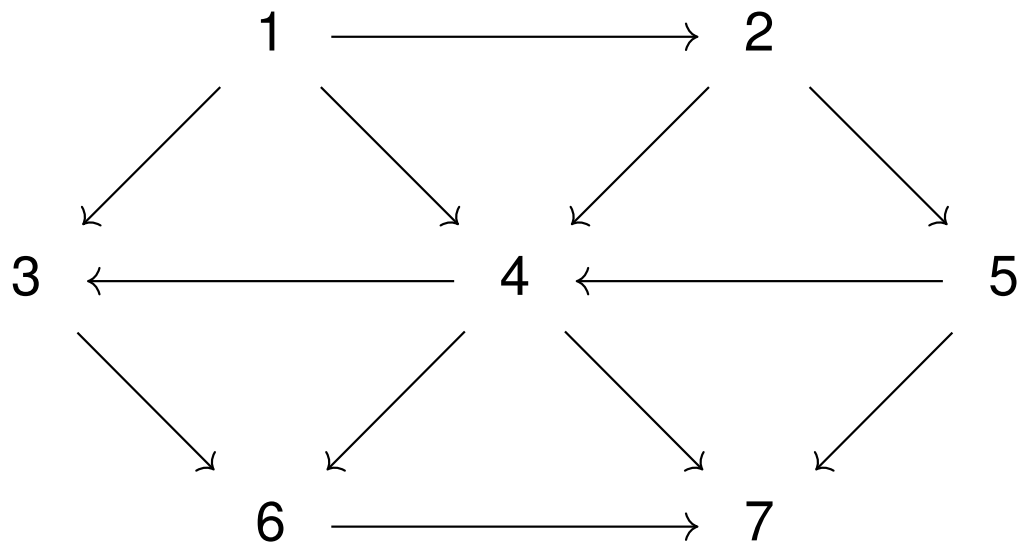
Storing an Unweighted Graph

- **Adjacency matrix:** Assume $V = \{1, \ldots, N\}$; then if $(i, j) \in E$, we define $m_{ij} = 1$, and $m_{ij} = 0$ otherwise
  - Cannot be used for multigraphs
  - By convention $m_{ii} = 1$ (although sometimes we may consider $m_{ii} = 0$)
  - Memory cost: $\Theta(|V|^2) = \Theta(N^2)$
- **Adjacency list:** We can consider a pointer table $T[\ ]$ where $T[i]$ points to a linked list
  - If $(i, j) \in E$, then j is in one of nodes in the list pointed to by $T[i]$
  - Memory cost: $\Theta(|V|) + \Theta(|E|)$
  - No problem for multigraphs
- For standard graphs the cost is always $O(|V|^2)$ for both methods, since we then have

$$|E| \leq |V|(|V| - 1) = O(|V|^2)$$

An Example

- Consider the following directed graph:



The Adjacency Matrix

- The first rows of the adjacency matrix are

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ & & & \cdots & & & \end{pmatrix}$$

The Adjacency List

- Partial adjacency list: we use a lexicographic order

$$
\begin{array}{llllllll}
1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 \\
2 & \rightarrow & 4 & \rightarrow & 5 & & \\
3 & \rightarrow & 6 & & & & \\
4 & \rightarrow & 3 & \rightarrow & 6 & \rightarrow & 7 \\
 & & & \ldots & & & &
\end{array}
$$

The Size of a Graph

- While $|V|$ and $|E|$ are in general independent, we may expect $|V| = O(|E|)$ for interesting graphs
  - $|E|$ will usually give G's size
- G is **dense** if $|E| = \Theta(|V|^2)$
- G is **sparse** if $|E| \ll |V|^2$
- If G s dense, the adjacency matrix storage is more efficient; if G is sparse, adjacency lists are better
- We will usually work with adjacency lists, using adjacency matrices for special algorithms

Depth First Search and Edge Classification

Depth First Search (DFS)

- Recursive DFS is one of the two main algorithms for graph traversal (the other is Breadth First Search)
- In DFS, starting from a node *u*, we recursively apply

```
def dfs(u, G):
    s[u] = True
    do_something_before_dfs(u)
    for all w adjacent to u:
        if s[w] == False:
            p[w] = u
            dfs(w, G)
    do_something_after_dfs(u)
```

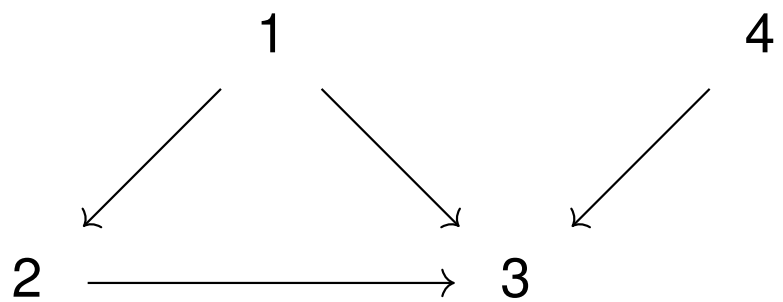
- `s[ ], p[ ]` are global tables to store seen nodes and previous node values
- The table `p[ ]` defines the **DFS tree** of the call from `u`

Depth First Search II

- We may have to restart DFS if not all nodes have been processed, for which we need a driver for DFS

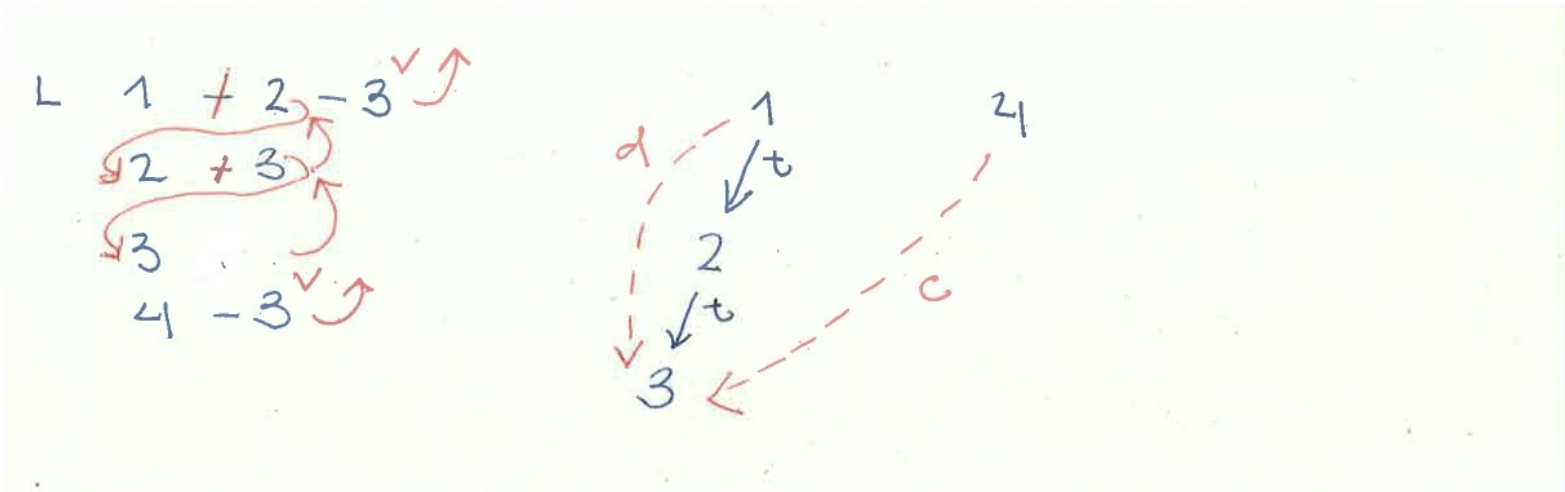
```
def driver_dfs(G):
    s[ ] = False; p[ ] = None
    for all u in V:
        if s[u] == False:
            dfs(u, G)
    return p[ ]
```

- The driver also initializes the global tables `s[ ], p[ ]`
- At the end the table `p[ ]` contains the **DFS forest**
- If `do_something` has cost $O(1)$, the joint cost of `driver_dfs` and `dfs` is clearly $O(|E|)$
- An example:



Applying DFS

- The DFS evolution and the DFS forest are



Edge Classification by DFS

- DFS induces a classification on the edges of a directed graph G
  - **Tree edges**: (u, v) where $u = p[v]$
  - **Back (ascending) edges**: (u, v) where $v = p[\ldots p[u]\ldots]$ (one or more p)
  - **Forward (descending) edges**: (u, v) where $u = p[\ldots p[v]\ldots]$ (with at least 2 steps on $\texttt{p}$)
  - **Cross edges**: any other $(u, v) \in E$
- If G is undirected and (u, v) is a forward edge, then (v, u) is a back edge
  - Thus, we will not distinguish then between forward and back edges
  - Also if (u, v) is a tree edge we don't count (v, u) as an ascending edge
- We prove later that if G is undirected there are no cross edges

Parenthesis Theorem

- Assume we have a counter c in DFS and consider 2 time–stamps:
  - **Discovery**: `d[u] = c; c += 1` , updated when DFS **starts** on u
  - **Finish**: `f[u] = c; c += 1` , updated when DFS **ends** on u
- Obviously $d_u < f_u$
- **Parenthesis Theorem.** *For a graph G and $u, v \in V$, consider the intevals $I_u = (d_u, f_u)$, $I_v = (d_v, f_v)$. Assuming $d_u < d_v$ we either have $I_v \subset I_u$, or $I_u \cap I_v = \emptyset$*
- **Proof sketch:** Assume $d_u < d_v$;
  - If $f_u < d_v$, obviously $I_u \cap I_v = \emptyset$
  - And if $f_u > d_v$, DFS started recursively on v before finishing with the already started u; thus, the recursion on v must finish before that of u and $f_v < f_u$
  - Thus, $I_v \subset I_u$

DFS for Discovery and Finish

- We add a global counter `c` and two extra global tables `d[ ], f[ ]`
- The DFS pseudocode to compute also `d[ ], f[ ]` is

```
def dfs_d_f(u, G):
    s[u] = True
    d[u] = c; c += 1
    for all w adjacent to u:
        if s[w] == False:
            p[w] = u
            dfs_d_f(w, G)
    f[u] = c; c += 1
```

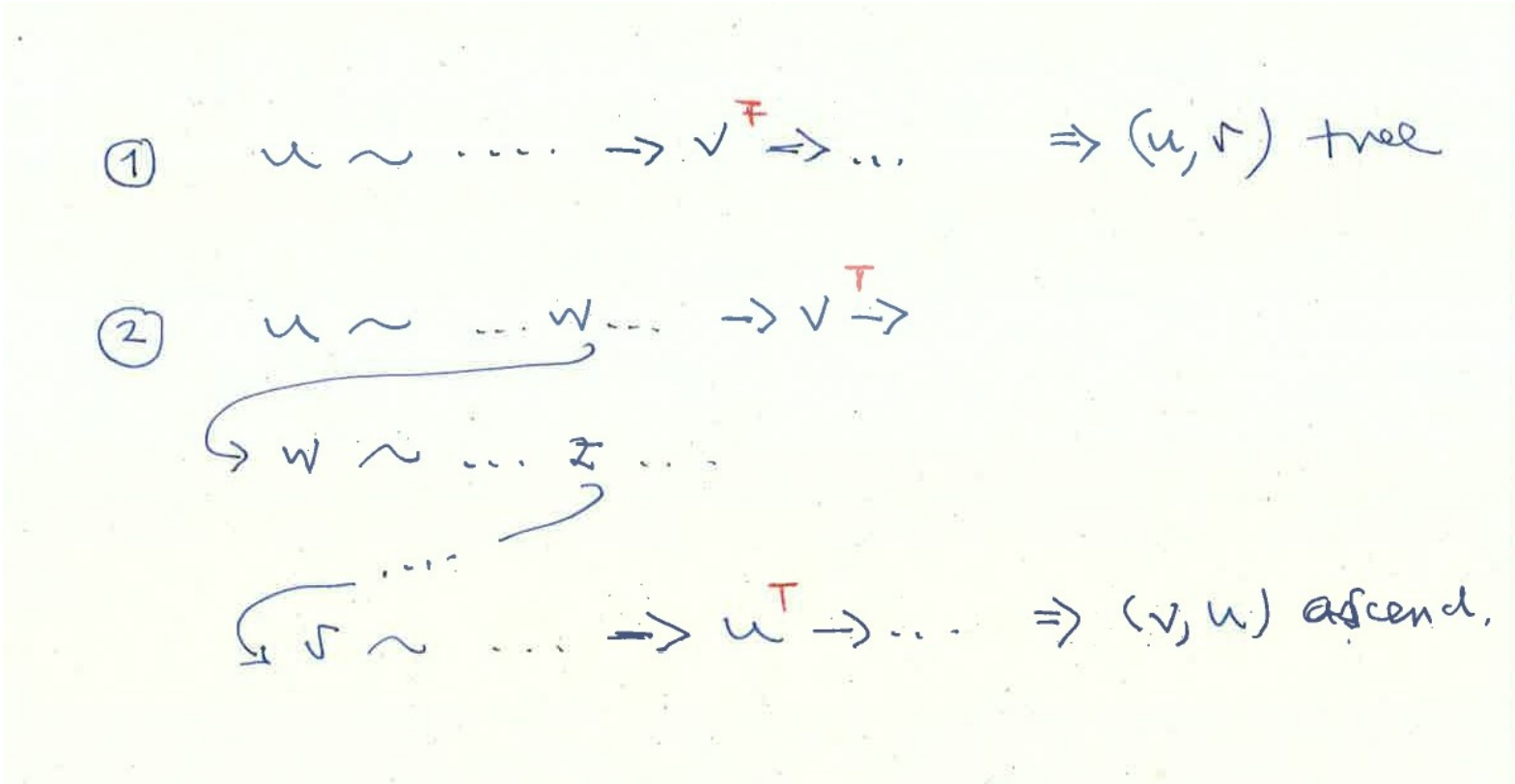
- `d[ ], f[ ]` are two new global tables to store discovery and finish times
- A driver `driver_dfs_d_f` also initializes `c=0, d[ ], f[ ]`, calls `dfs_d_f` as needed and returns `p[ ], d[ ], f[ ]`

No Cross Edges in Undirected Graphs

- **Proposition.** *If G is undirected there are no cross edges*
- **Proof sketch:** Take $(u, v) \in E$:
  - Assume $d_u < d_v$; then we have $f_v < f_u$ for v is adjacent to u
  - If $s[v] = F$ when we arrive at v, then (u, v) is a tree edge
  - And if $s[v] = T$ when we arrive at v, we have processed $L[v] \Rightarrow$ we have processed $(v, u) \Rightarrow (v, u)$ must be either a tree or a back edge
  - Thus, (u, v) is either a tree or a forward edge
- Thus, in no case is (u, v) a cross edge

No Cross Edges in Undirected Graphs II

- We sketch the previous arguments.



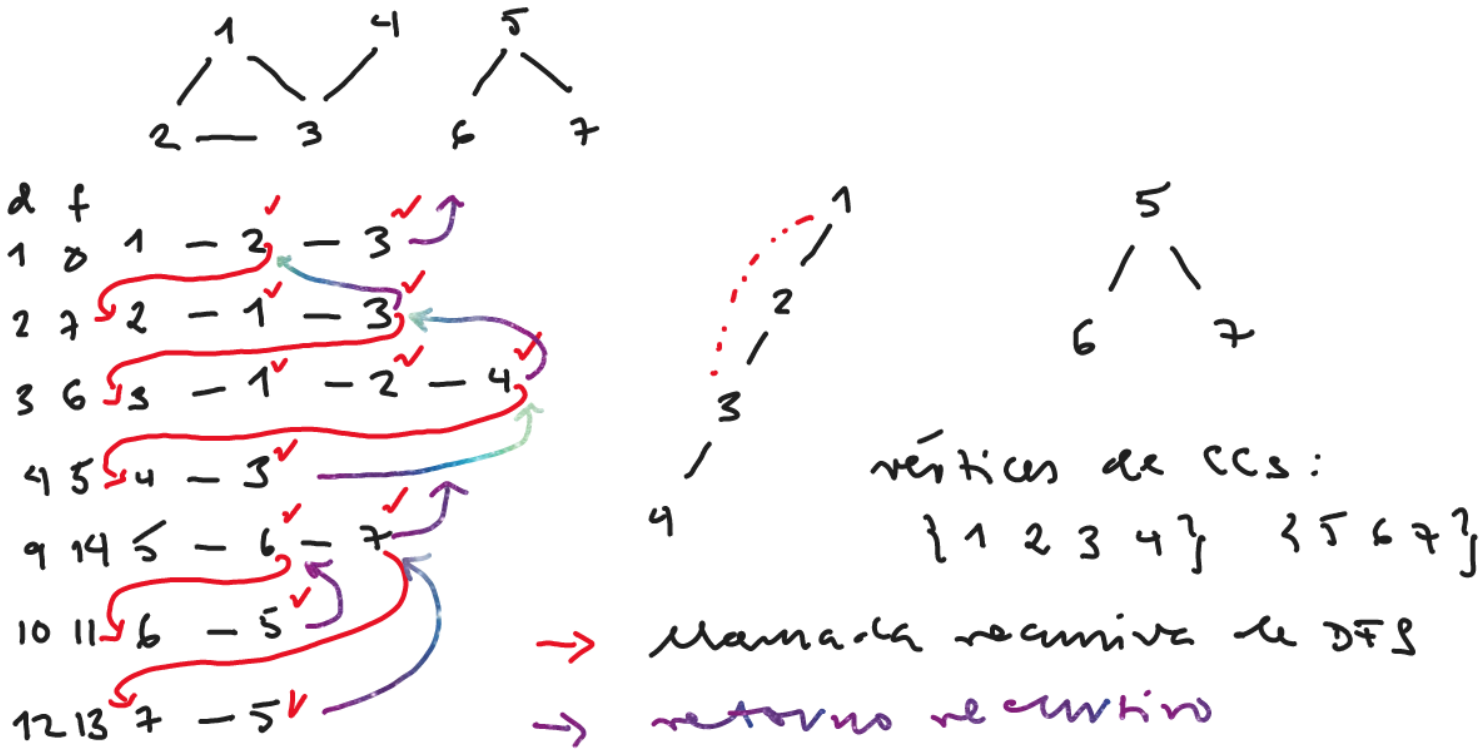
Graph Connectivity

Undirected Graph Connectivity

- Recall that an undirected graph $G = (V, E)$ is connected if **for every pair** $u, v \in V$ **there is a path** π **in** E **from** u **to** v
- **Connected component**: a maximal connected subgraph of G
- If $G_i = (V_i, E_i)$ are the connected components of G, the V_i are a **partition** of V and the E_i of E
- If we order the vertices of G as $V = V_1 \cup \ldots \cup V_K$, then the adjacency matrix M is **block diagonal** with the blocks M_k being the adjacency matrices of the G_k
- DFS can be used to give the connected components of G through the table $p[\,]$ and restarting DFS as needed
  - BFS and its driver can also be used to give the connected components of G through the table $p[\,]$
- The cost of DFS connectivity is $O(|E|)$, better than the cost of the alternative algorithm that uses disjoint sets

An example

- We compute the connected components of the undirected graph



Directed Graph Connectivity

- A directed graph $G = (V, E)$ is **weakly connected** if its extension to an undirected graph by “doubling" single edges (u, v) is connected
- A directed graph $G = (V, E)$ is **strongly connected** if for every pair $u, v \in V$ there is a path π in E from u to v
- DFS is also used in **Tarjan’s Algorithm** to obtain the strong components of a graph
- Tarjan’s algorithm basically obtains the strong components by
  - Computing DFS’s ending times on G and
  - Applying again DFS to the transpose graph G^T in the order inverse to the ending times

Tarjan's Algorithm

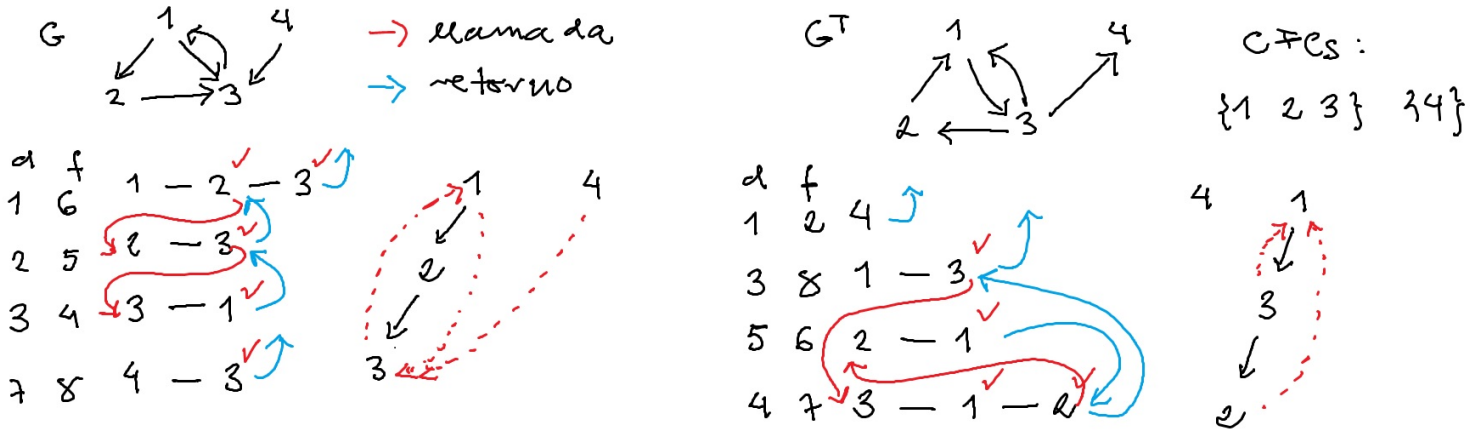
- The transpose of a graph G with adjacency matrix m is the graph G^T with matrix m^t
- The Tarjan pseudocode is

```
def tarjan_cfc(G):
    p, d, f = driver_dfs_d_f(G)

    s[ ] = False
    p[ ] = None
    for u in reverse(f[ ]):
        if s[u] == False
            dfs(u, G^T)
    return p[ ]
```

- Then each one of the trees in the DFS forest in `p[]` has the nodes of a strongly connected component

Applying Tarjan's Algorithm

- An example:



DAGs and Topological Sort

Directed Acyclic Graphs

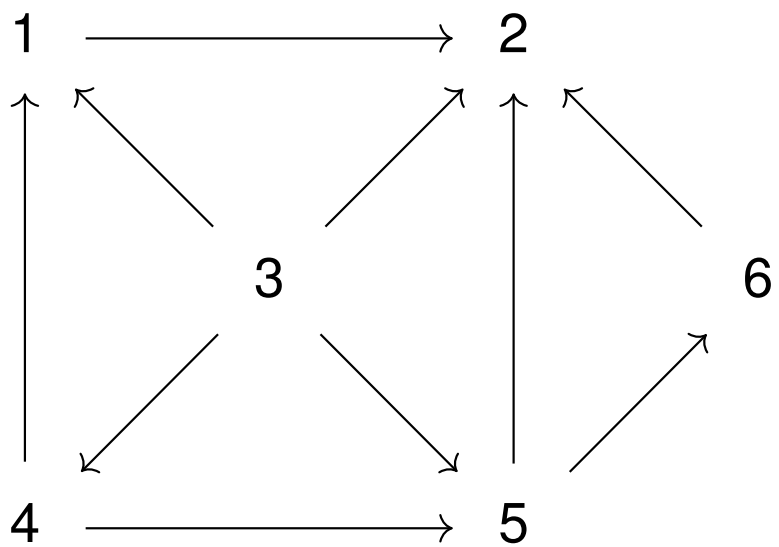
- A **directed acyclic graph** (DAG) is a directed graph without cycles
- **Proposition:** G is a DAG iff there are no ascending edges in G
  - If (v, u) is ascending, there is a path from u to v in the DFS forest, and adding (v, u) results in a cycle
  - Conversely, assume π is a cycle, $u \in V_\pi$ is the first node processed in DFS and $\pi = (u, \ldots, v, u)$
  - Then it can be shown that v descends from u in the DFS forest and, thus, (v, u) is ascending
- DFS can be used to detect cycles in a graph by slightly modifying our previous AP algorithm
- DAGs can be used to model many other problems of interest, such as topological node ordering

Topological Sort

- Recall: $\leq$ is a **total order** if either $u \leq v$ or $v \leq u$ or both
- A **topological sort** (TS) in a DAG $G = (V, E)$ is any total ordering of its vertices s.t. if $(u, v) \in E$, then $u \leq v$
  - Notice that there may be several different topological orderings
- If G is a DAG, a topological sort can be obtained by
  - Applying DFS
  - Adding a vertex u at the beginning of a linked list **after DFS ends its process on it**
- Then **the list order is a topological sort** of G: if $(u, v) \in E$
  - DFS ended at u **after** all the vertices adjacent to u, with v among them, have been processed
  - Therefore, v has been added to the TS list **before** u, and hence $u \leq v$
- The cost of TS on DAGs is thus $O(|E|)$

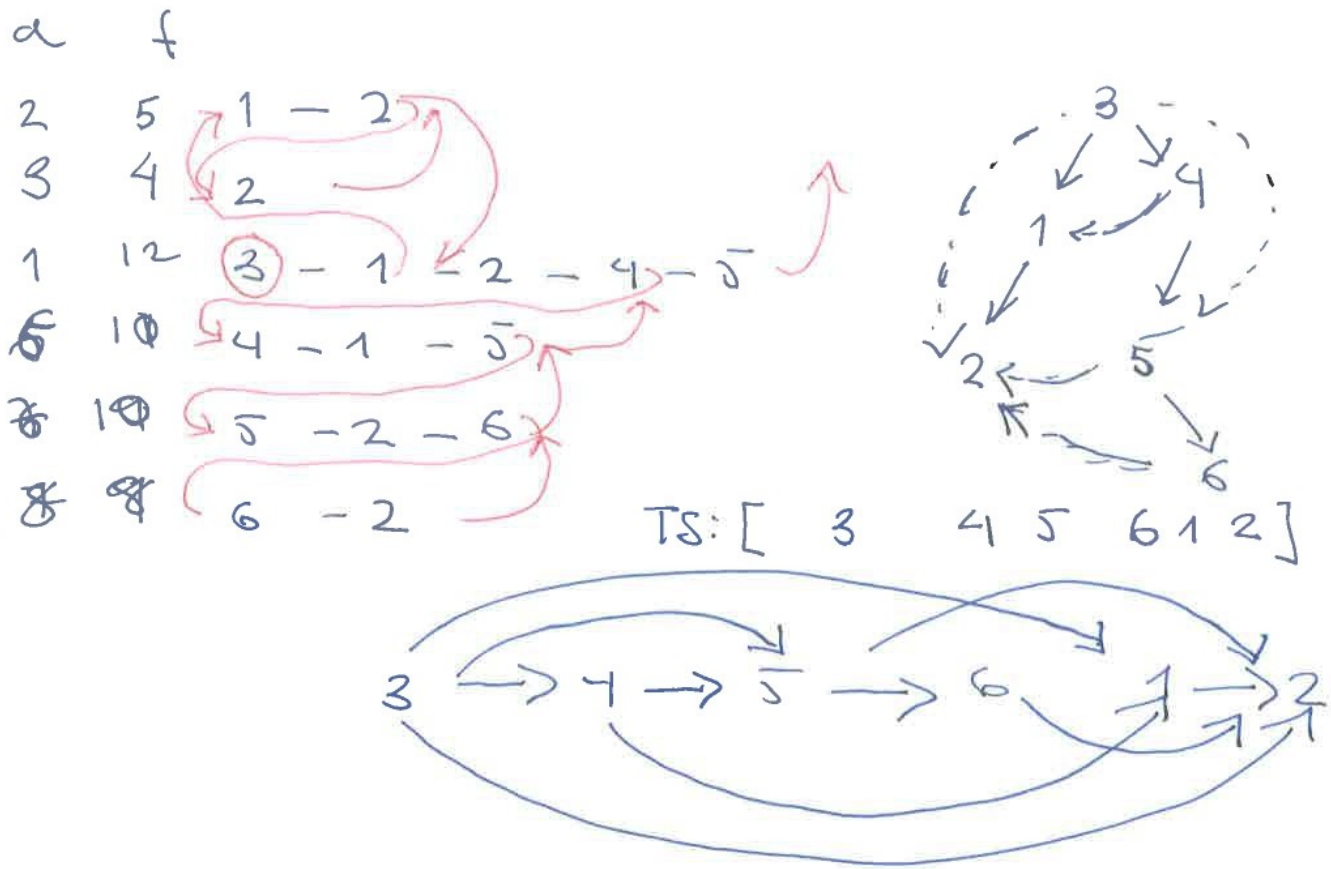
Applying Topological Sort

- An example:



Applying Topological Sort II

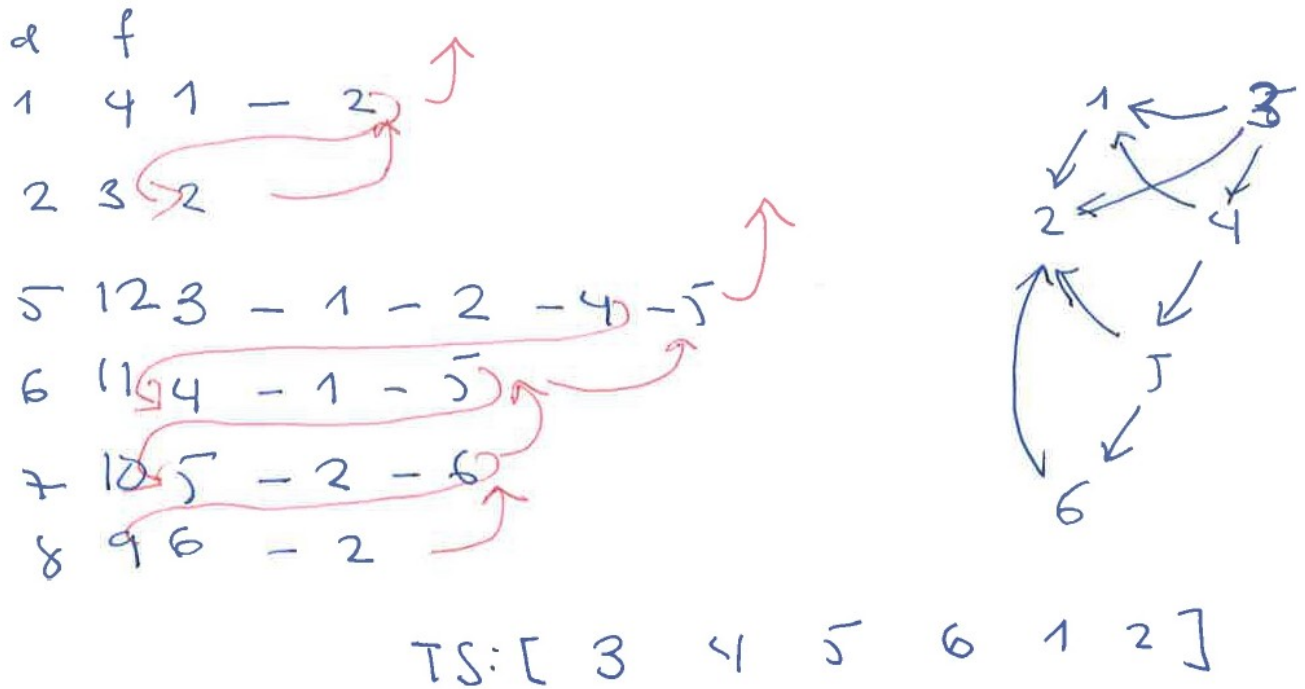
- We apply DFS computing the discovery and finish times



- TS can also be obtained by reversed finish times

Applying Topological Sort II

- TS can also be obtained by reversed finish times



The Change Problem

Recall: The Greedy Change Algorithm

- Assuming we work with an ordered list of coin values, say
  `1, 2, 5, 10, 20, 50, 100, 200, ...,`
  we can save the number of coins of each type used in a dict

```
l_coin_values = [1, 2, 5, 10, 20, 50, 100, 200, ...]

def change(c: int, l_coin_values: List) -> Dict:
    """
    """
    d_change = {}

    for coin in sorted(l_coin_values)[ : : -1]:
        d_change[ coin ] = c // coin
        c = c % coin

    return d_change
```

Does It Work?

- Recall that it does for Euro coin system
- But not when trying to give change of 7 maravedis with coin values

  `1, 3, 4, 5`
- What is the answer of the algorithm?
  - Correct answer: just 2 coins, one of 4 maravedis and one of 3
- Recall that this often happens with greedy algorithms
  - They are very natural but may give wrong results!!
- Let's try a **dynamic programming approach**

A General Change Algorithm

- Starting point: **get a good name/notation**
- Assume we want to change an amount C with $v_1, \ldots, v_N$ coin denominations
- Let $n(i, c)$ be the minimum number of coins to change an amount c using only the first i coins
  - The number we want is $N(N, C)$
- Now, if coin i doesn't enter the change, we obviously have $n(i, c) = n(i - 1, c)$
- But if coin i enters the change, we can use one of the i coins with value v_i and compute afterwards $n(i, c - v_i)$
  - We thus have $n(i, c) = 1 + n(i, c - v_i)$ in this case
- And there aren't other options

The DP Change Algorithm

- Therefore , we arrive for $i = 1, \ldots, N$ at $n(i, 0) = 0$, $n(1, c) = c$ and

$$\begin{aligned} n(i, c) &= \min\{n(i-1, c), 1 + n(i, c - v_i)\}, \quad v_i \leq c \leq C \\ &= n(i-1, c), \quad c < v_i \end{aligned}$$

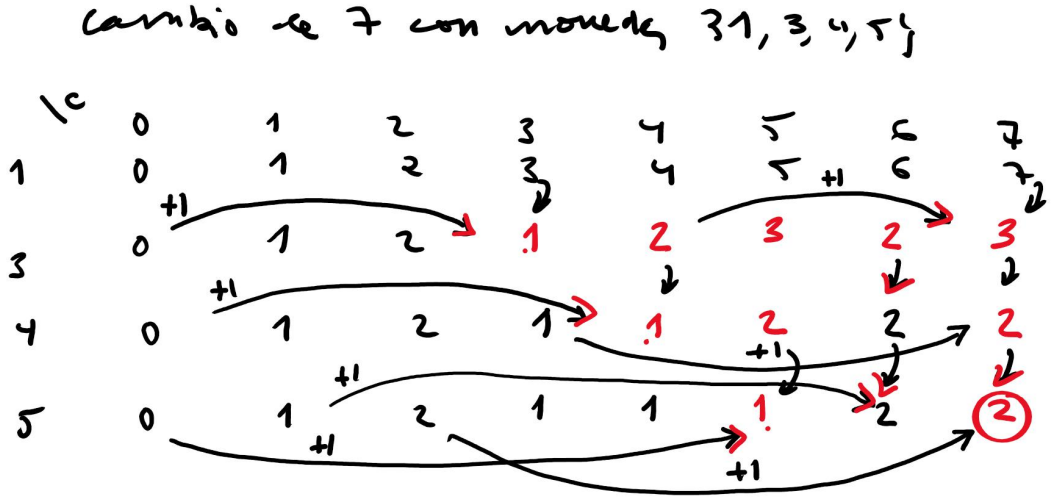
- We thus have to fill an $N \times C$ matrix with a cost

$$O(N \times C) = O(N \times 2^{\lg C})$$

  and the optimal number of coins is $n(N, C)$
- Notice also that the algorithm is more or less obviously correct
- But the cost is linear on N but **exponential** on C
  - Much worse than the linear complexity of the greedy algorithm

Giving DP Change

- Giving change via DP of 7 with coin values `1, 3, 4, 5` , we fill the matrix $n(i, c)$ from top to bottom rows and from left to right columns



The Knapsack Problem

0–1 Knapsack

- Recall that we have N elements with integer weights w_i and values v_i and a knapsack that stands a maximum weight W
- We want a choice of elements $i_1, \ldots, i_K$ such that $\sum_j w_{i_j} \leq W$ and $\sum_j v_{i_j}$ is maximum
- Mathematically, we want to solve a **Constrained Optimization Problem**:

$$\max \sum_{1}^{N} v_i x_i \text{ subject to } \sum_{1}^{N} w_i x_i \leq W \text{ and } x_i \in \{0, 1\}$$

- The “0–1" name derives from the constraint $x_i \in \{0, 1\}$
- We saw that the problem has a natural greedy solution that, however, is not correct

Dynamic Programming Solution

- Starting point (again): **get a good name/notation**
- For $2 \leq i \leq N$, $w_i \leq w \leq W$, let $v(i, w)$ be the optimum value of a knapsack with the first i elements and bound w
- If i is **in** the optimal selection for $v(i, w)$ we must have

$$v(i, w) = v_i + v(i-1, w - w_i)$$

- And if i is **not in** the optimal selection for $v(i, w)$, we must have

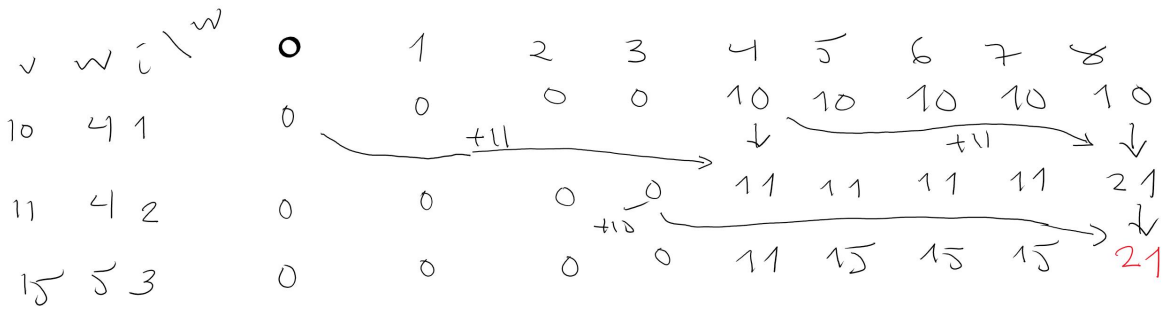
$$v(i, w) = v(i-1, w)$$

- Thus for these i, w combinations we have

$$\begin{aligned} v(i, w) &= \max\{v_i + v(i-1, w-w_i), v(i-1, w)\} \text{ if } w_i \leq w \leq W \\ &= v(i-1, w) \text{ if } w < w_i \end{aligned}$$

The Bottom Up Algorithm

- Obviously $v(i, 0) = 0$ and adding an “extra" element with 0 weight and value, we have $v(0, w) = 0$ for all w
- It is also clear that $v(i, w) = v(i - 1, w)$ if $w < w_i$
- Notice that to compute the i–th row $v(i, w)$ we just need to have already computed the previous row $v(i - 1, w)$
- Again we get the optimal value $v(N, W)$ going from top to bottom and from left to right
- **Example:** 3 elements with values $v = \{15, 11, 10\}$, weights $w = \{5, 4, 4\}$ and $W = 8$:



But . . .

- The iterative DP algorithm has to fill a $N \times W$ matrix
- Since computing $v(i, w)$ has a $O(1)$ cost, the overall cost is

$$O(N \times W) = O(N \times 2^{\lg W})$$

- Since the problem size is $O(N + \lg W)$, we see that the cost of the DP solution is also exponential in problem size
- In fact, the **decision versions** of the change, 0–1 Knapsack and Traveling Salesman problems are examples of **NP–complete** problems
  - Decision version of TSP: given a cost bound C, can we find a tour with cost $\leq C$?
- NP–complete problems are an important subclass of the **NP** problems: decision problems for which we can check the correctness of solutions with polynomial cost
- The class **P** is made of problems which we can solve with polynomial cost
- Obviously $\mathbf{P} \subset \mathbf{NP}$ but whether $\mathbf{P} = \mathbf{NP}$ is a famous unsolved problem in computer science

DP String Algorithms

Editing Strings

- Assume we have two strings `S` and `T` and want to edit, i.e., transform one into the other according to the following allowable operations
  - Change one character in either string
  - Insert a character in either string
  - Remove a character from either string
- Notice that removing a remove a character from string `S` is equivalent to adding a character to string `T`
- Also we can keep one of the strings fixed and do all the change/add/remove operations on the other

The Edit Distance

- The **Edit Distance** `d(S, T)` between `S` and `T` is the minimum number of edit operations that we have to make to turn, say, `T` into `S`
- For instance, the edit distance between `unnecessarily` and `unescessaraly` is 3:

```
unne-cessari-ly
  d i      di
un-escessar-aly
```

  - On the top string we delete an `'n'` , insert an `'s'` and change `'i'` into `'a'`
  - Conceptually the change operation is equivalento to an insert + delete combination
  - We mark **indels** (i.e., either an insertion or a deletion) with `-`: insert (`-` on top) or delete (`-` on bottom)
- The edit distance can also be used for **approximate string searches**: given a string `S` find another `T` in a string list so that their edit distance is minimal

A Dynamic Programming Solution

- Given the full strings S and T with M and N characters respectively, consider the substrings

$$S_i = [s_1, \ldots, s_i], \quad T_j = [t_1, \ldots, t_j]$$

- If $d_{i,j}$ is the edit distance between S_i and T_j, we want to find $d_{M,N} = \text{dist}(S, T)$
- Observe that if $s_i = t_j$, then $d_{i,j} = d_{i-1,j-1}$
- And if $s_i \neq t_j$ we have three options
  - Change t_j into s_i; then $d_{i,j} = 1 + d_{i-1,j-1}$
  - Remove t_j from T_j; then $d_{i,j} = 1 + d_{i,j-1}$
  - Remove s_i from S_i; then $d_{i,j} = 1 + d_{i-1,j}$

Filling The DP Matrix

- We thus arrive to the following equations for the Edit Distance problem

$$\begin{aligned} d_{i,j} &= d_{i-1,j-1} \text{ if } s_i = t_j; \\ &= 1 + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}\} \text{ if } s_i \neq t_j \end{aligned}$$

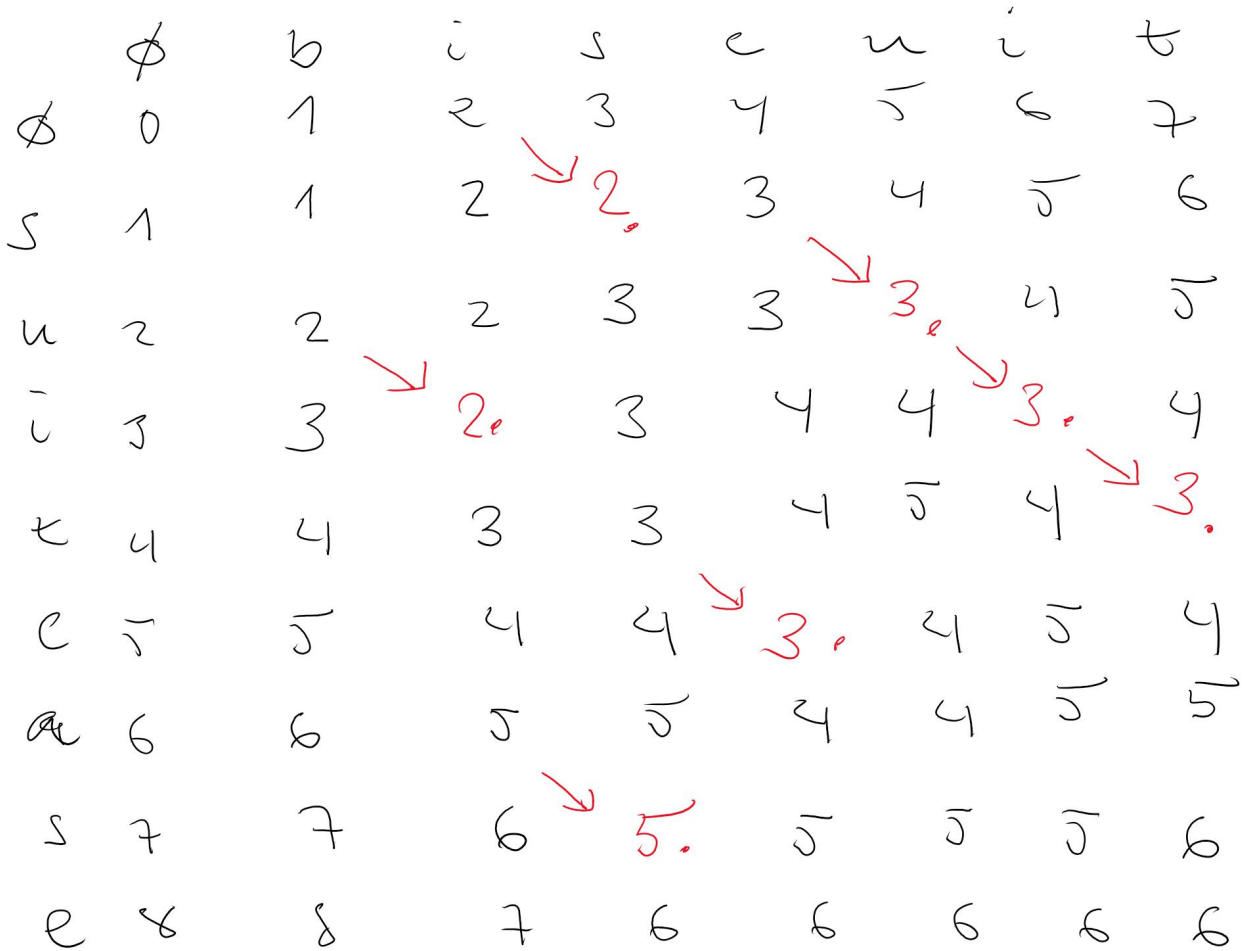
  with the boundary conditions $d_{i,0} = i,\ d_{0,j} = j$, where

$$d_{i,0} = \text{dist}(S_i, \emptyset), \quad d_{0,j} = \text{dist}(\emptyset, T_j)$$

- The cost is clearly $O(M \times N)$, no longer NP–complete but, still, costly
- And an initial memory cost is also $O(M \times N)$, just awful, as problem size is $O(M + N)$
  - But, in fact, quite easy to alleviate if we only care about $d_{M,N}$

An Example

- Example: find the edit distance between `biscuit` and `suitcase`



The Longest Common Subsequence

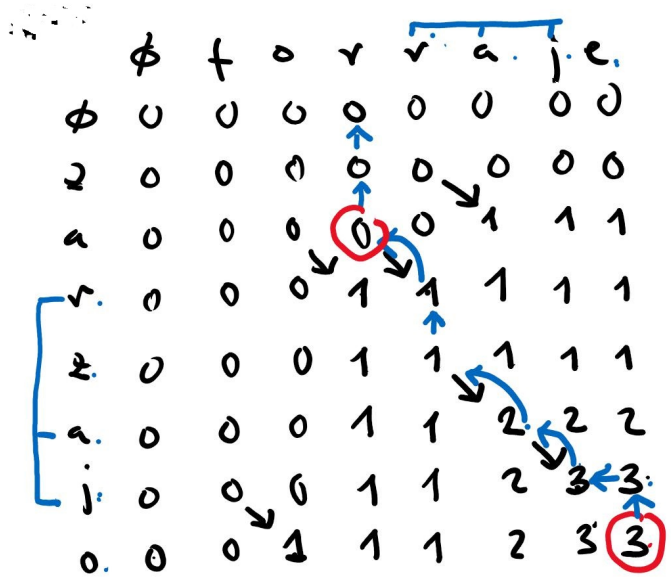
- Given again strings S, T, we want to find the longest (non necessarily consecutive) common subsequence (LCS) to both
- As before, let first $\ell_{i,j}$ be the length of the LCS between S_i and T_j. We have now:

$$\begin{aligned} \ell_{i,j} &= 1 + \ell_{i-1,j-1} \text{ if } s_i = t_j; \\ &= \max\{\ell_{i,j-1}, \ell_{i-1,j}\} \text{ if } s_i \neq t_j \end{aligned}$$

  with the boundary conditions $\ell_{i,0} = 0,\ \ell_{0,j} = 0$
  - Again this results in an easy to apply algorithm with cost $O(M \times N)$
  - Example: find the LCS between `biscuit` and `suitcase`
- Good (and easy) exercise: write down a Python function to find the length of the LCS
- Very good exercise: modify the previous Python function so that it gives one of the possible LCS

An LCS Example

- Example: find the longest non-consecutive common sequence between `forraje` y `zarzajo`
- We first compute the LCS matrix
- We then trace the algorithm's steps from the optimal LCS length value to get a common subsequence



S_c = [raj]
no consecutivos

Efficient Matrix Multiplication

Optimal Ordering in Matrix Multiplication

- Given N matrices $A_1, \ldots, A_N$ with sizes $c_{i-1} \times c_i$, $0 \leq i \leq N$, the order in which we compute $A_1 \times \ldots \times A_N$ doesn't matter mathematically for the final solution
- But it affects greatly the computation cost
- Recall that computing $A_i A_{i+1}$ requires $c_{i-1} \times c_i \times c_{i+1}$ number multiplications
- Consider 4 matrices A_1, A_2, A_3, A_4 with respective sizes 50×10, 10×40, 40×30 and 30×5
  - The cost of the order $A_1(A_2(A_3 A_4))$ is 10,500 products
  - But the cost of the order $((A_1 A_2)A_3)A_4$ is 87,500 products
- We want to find how to order matrix multiplication so that the overall cost is minimum

EOS for Matrix Multiplication Ordering

- We will derive a PD algorithm by determining the **explicit optimal structure** (EOS) of the problem
- Let $m_{L,R}$ be minimum number of number products needed to compute $A_L \ldots A_R$
  - We have $m_{i,i+1} = c_{i-1} \times c_i \times c_{i+1}$
  - We set $m_{i,i} = 0$ and we want to compute $m_{1,N}$
- To arrive at the EOS property, assume that the optimal ordering is $(A_L \ldots A_j)(A_{j+1} \times A_R)$; then

$$m_{L,R} = c_{L-1} \times c_j \times c_R + m_{L,j} + m_{j+1,R}$$

- We don't know j but we can simply compute all the possible $m_{L,R}$ above and take the minimum, i.e., for $L < R$,

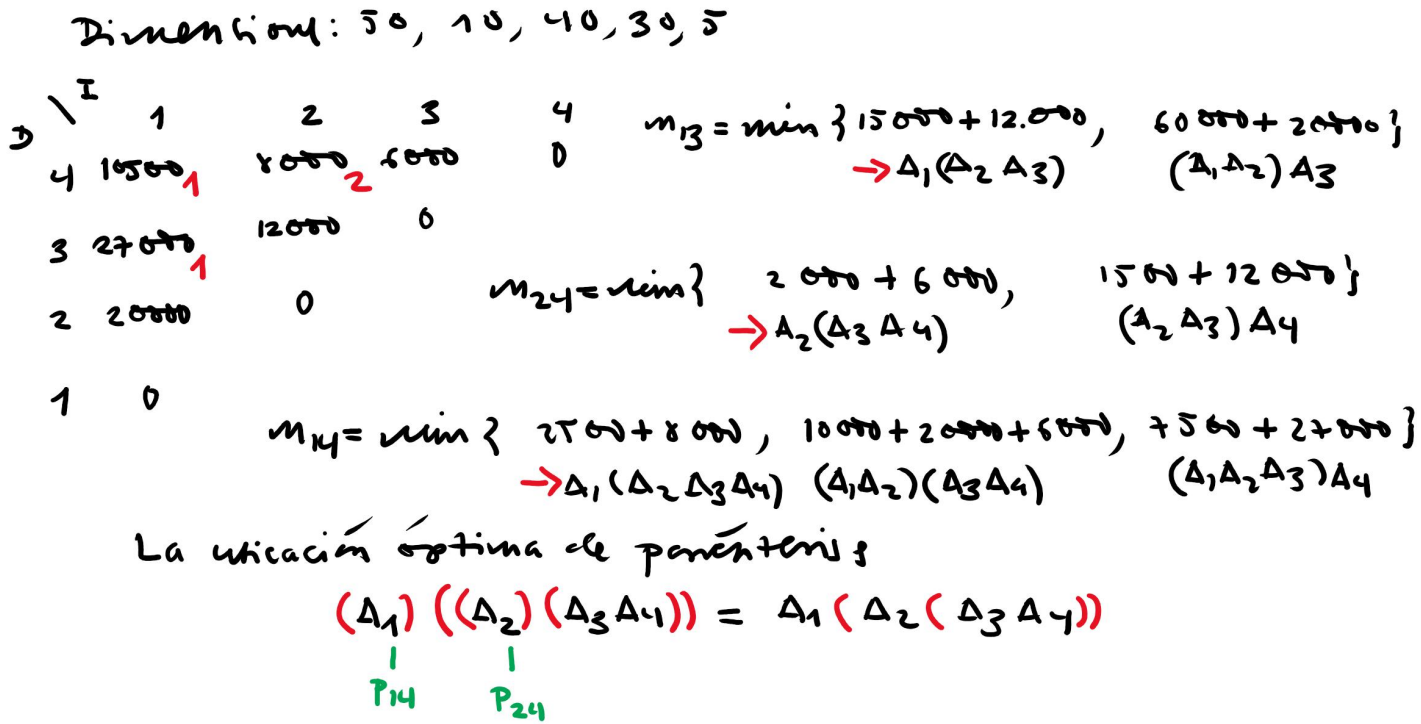
$$\begin{aligned} m_{LL} &= 0, 1 \leq L \leq N \\ m_{LR} &= \min_{L \leq j \leq R-1} \{c_{L-1} \times c_j \times c_R + m_{Lj} + m_{j+1R}\}, 1 \leq L < R \leq N \end{aligned}$$

DP Algorithm

- The EOS property suggests an iterative bottom up algorithm
  - Compute first the $m_{i,i+1}$ for $1 \leq i \leq N-1$ values
  - Compute then the $m_{i,i+2}$ for $1 \leq i \leq N-2$ values and keep on going
  - Until we arrive at $m_{1,N}$
- Since we have to compute $O(N^2)$ values m_{ij} for $2 \leq j \leq N$ and $1 \leq i < j$ at a cost of $O(j-i) = O(N)$ for each term, the overall cost will be $O(N^3)$
  - Exercise: derive the precise value
- For easier visualization, we can store the $m_{L,R}$ values in a triangular matrix $a_{i,j} = m_{j,N-i}$, i.e., the L values define the columns and the R values define the rows in inverse order

An Example

- Consider again the matrices A_1, A_2, A_3, A_4 with respective sizes 50×10, 10×40, 40×30 and 30×5



Parallel Computation Models

Parallel Computers

- They are a collection of similar processors connected so that they can coordinate computations and data exchange
- Simple examples: multi core/multi thread CPUs
- Goal: if p processors are available, distribute w basic operations among them so that they are performed in $\frac{w}{p}$ time
- Of course, not always possible: consider putting on two socks and shoes
  - One pair of hands (i.e., processors) has to perform 4 operations
  - Two pairs of hands can perform them in 2 time units
  - But the same is true for 3 or more processors

Random Access Machine (RAM)

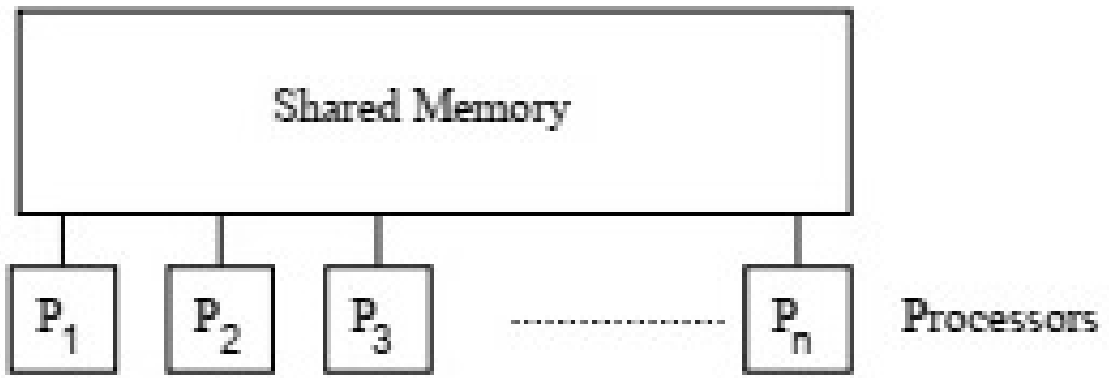
- It is the basic model for single processor computations
- It comprises a single processor (CPU) connected to a memory system
- The CPU performs basic operations such as memory read/write, arithmetic or logical operations, etc.
- Each is assumed to take one time step
- Multiprocessor models generalize the basic RAM model

Parallel Memory Models

- Parallel machines differ first on their underlying memory models
- In **Local Memory** models memory is distributed among the processors
  - Each processor has direct access to its own memory
  - But needs to send requests to access memory in other processors
- In **Modular Memory** models there are p processors and m memory modules, all connected through a network
  - Each memory access requires a network request
- In **Parallel RAM** models each processor has a small local memory and all can access a single, large global memory
  - A single step is needed for global memory access
  - Memory access is done in parallel
  - It is the model more widely used

Parallel RAM

- A schematic representation of a PRAM machine would be



Operation and Instruction Models

- Operation models can be synchronous and asynchronous
- In **asynchronous** models each processor has its own clock
- In **synchronous** models a single global clock controls all processors
- In the **Multiple Instruction Multiple Data** (MIMD) model each processor can execute a different set of steps
- In the **Single Instruction Multiple Data** (SIMD) model all processors execute the same steps

PRAM Read and Write

- PRAM is a synchronous, shared memory model
- We assume **concurrent reads** (CR): all processors can read concurrently from the same memory location
- Mechanisms have to be implemented to solve concurrent writes
- In **Exclusive Write** (EW) no concurrent writes in the same memory location are allowed
- In **Concurrent Write** (CR) concurrent writes are allowed under some conditions
- The most common assumption is a **common write**: all concurrent processors must try to write the same value
- Standard options are the **CREW** and **common CRCW** models
- Pure PRAM models are not realistic but offer a clear first description level
- However, we will not use them and will work with a higher abstraction

Work-Depth Models

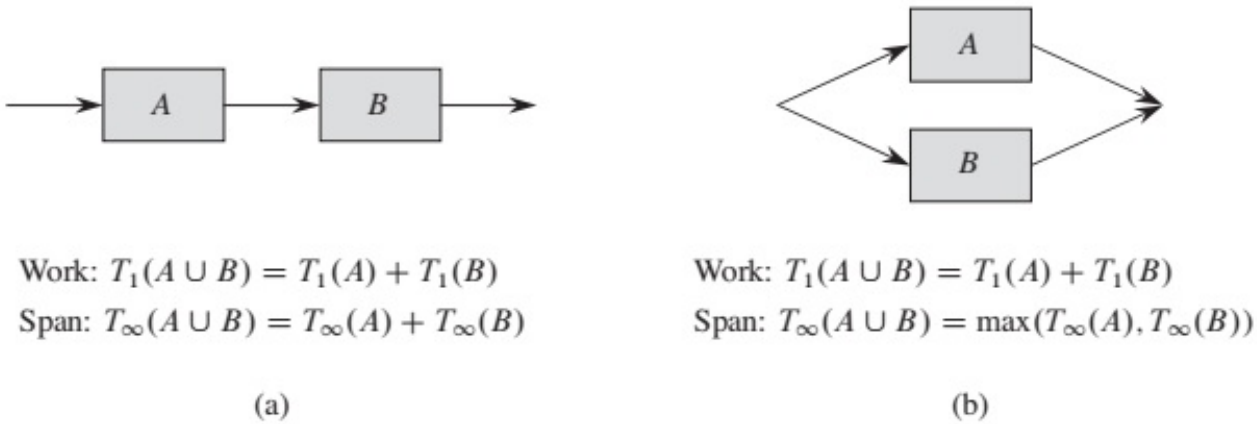
- Focusing on the processors makes difficult to write/understand parallel algorithms
- The **work-depth** (or **work-span**) model abstracts the problem details and concentrates on the algorithms
- The goal is to get familiar pseudocode adding some parallel instructions
- `spawn` starts a new, child parallel computation (a **thread**) on a different processor
- `sync` stops the father process until all spawned children have finished
- **strands**: sequences of consecutive statements without any parallel instruction
- The work-depth model is not realistic either, but simplifies pseudocode writing and time analysis

Some Notation

- We will take as running time either the number of steps an algorithm needs or the number of basic operations it performs
- Assuming p processors, $T_p(n)$ denotes the running time/number of steps of an algorithm on a problem of size n
- $T_1(n)$, the running time with 1 processor, is called the **work**
  - It is just the running time of the initially parallel code once we remove the parallel instructions
- $T_\infty(n)$ denotes the **span** (or **depth**), i.e., the running time without any limit in the number of processors available
  - It is an ideal, not a practical estimate but will be used to get reasonable time estimates without complicating the parallel code
- For the sock-shoe problem of size $n = 4$, we have $T_1(4) = 4$ but $T_\infty(4) = 2$

Computing Work and Span

- We can visualize the meaning of work and span



Work and span computations. (Taken from Cormen, Leiserson et al.

- On the left the subcomputations are performed sequentially and total work and span are the sum of partial work and span
- On the right the work doesn't change, but the joint span is the maximum of the substasks' spans

Sequential Fibonacci

- The standard psc is

```
def fib(n):
    if n <= 1:
        return 1

    return fib(n-1) + fib(n-2)
```

- Let $T(n)$ the number of sums `fib` performs; then $T(0) = T(1) = 0$ and

$$T(n) = 1 + T(n-1) + T(n-2)$$

- If $H_n = 1 + T(n)$, is easy to see that $H_n = F_n$, the n-th Fibonacci number
- Therefore

$$T(n) = F_n - 1 = \Theta(\phi^n), \text{ where } \phi = \frac{1+\sqrt{5}}{2}$$

i.e., a quite large cost

Parallel Fibonacci

- The new psc with three strands is

```
def par_fib(n):
    if n <= 1:                  # strand 1
        return 1

    else:
        x = spawn par_fib(n-1)
        y = par_fib(n-2)        # strand 2
        sync
        return x + y            # strand 3
```

  - Removing `spawn, sync` we get the previous algorithm
- Now we have $T_1(n) = F_n - 1 = \Theta(\phi^n)$ and for T_∞ we have The cost is now $T_\infty(0) = T_\infty(1) = 0$ and

$$T_\infty(n) = 1 + \max\{T_\infty(n-1), T_\infty(n-2)\} = 1 + T_\infty(n-1)$$

- This solves to $T_\infty(n) = n - 1$, much better than before
  - But this implicitly assumes an infinite number of processors
  - We have to deal with this

The Work and Span Laws

- Since at one time step p processors do p units of work, in T_p time they will perform $p \times T_p$ total units of work
- Since obviously the total work performed must be $\geq T_1$, we arrive at the **Work Law**:

$$T_1 \leq p \times T_p \text{ or } \frac{T_1}{T_p} \leq p$$

  - The **speed up** $\frac{T_1}{T_p}$ when working with p processors is always $\leq p$
- On the other hand, since a machine with ∞ processors can always emulate one with p processors, the **Span Law** follows

$$T_\infty \leq T_p$$

  - The **parallelism** of an algorithm is the ratio $\frac{T_1}{T_\infty}$
- By the Span Law, $\frac{T_1}{T_\infty} \geq \frac{T_1}{T_p}$, i.e. speed up $\leq$ parallelism
- In practice we must consider the number of processors available
  - This requires **scheduling**: to assign threads to processors

Greedy Scheduling

- In **greedy scheduling** we will assume that *at each time step, as many strands are assigned to processors as it is possible*
- **Theorem.** *With p processors and greedy scheduling, a computation with work T_1 and span T_∞ can be run in time*

$$T_p \leq \frac{T_1}{p} + T_\infty$$

- *Proof.* We consider first **complete steps**: each processor receives a ready strand and prove that they are at most $\left\lfloor \frac{T_1}{p} \right\rfloor$
- If not, the total work at these steps would be

$$p\left(\left\lfloor \frac{T_1}{p} \right\rfloor + 1\right) = p\left\lfloor \frac{T_1}{p} \right\rfloor + p \geq p\left(\frac{T_1}{p} - 1\right) + p = T_1$$

- And this is a contradiction, as more work would be performed than it is possible

Greedy Scheduling II

- *Proof (cont.).* We consider next **incomplete steps**, i.e., steps where there are less strands ready than there are processors
  - Hence, each strand gets a processor
- Then the number of incomplete strands must be $\leq T_\infty$
  - Just notice that on T_∞ we assume there are always more processors than strands
  - Hence, all steps in T_∞ are incomplete
- Putting both together we have for the number of steps with p processors

$$T_p \leq \left\lfloor \frac{T_1}{p} \right\rfloor + T_\infty \leq \frac{T_1}{p} + T_\infty$$

Greedy Scheduling III

- **Corollary.** *If T_p^* is the running time of an optimal scheduler in a machine with p processors, the running time of any greedy scheduler verifies $T_p \leq 2T_p^*$*
- *Proof.* Recall that by the Work Law, $T_p^* \geq \frac{T_1}{p}$ and by the Span Law, $T_p^* \geq T_\infty$. Therefore,

$$T_p \leq \frac{T_1}{p} + T_\infty \leq 2 \max\left(\frac{T_1}{p}, T_\infty\right) \leq 2T_p^*$$

- **Corollary.** *If $T_\infty \ll \frac{T_1}{p}$, then $T_p = \Theta\left(\frac{T_1}{p}\right)$, i.e., we get an approximately speed up of p*
  - Thus, we get a p speed up when the parallelism $\frac{T_1}{T_\infty}$ is large
- *Proof.* Since then $T_\infty \ll \frac{T_p}{p} \leq \frac{T_1}{p}$, it follows that

$$T_p \leq \frac{T_1}{p} + T_\infty = O\left(\frac{T_1}{p}\right)$$

and, since by the Work Law, $T_p \geq \frac{T_1}{p}$, we have $T_p = \Theta\left(\frac{T_1}{p}\right)$

Parallel Matrix-Vector Products

Parallel Loops

- We can use `spawn` and `sync` to parallelize loops, as in

```
for i = 0 to n-1:
    x[i] = spawn f(i)
x[n] = f(n)
sync
...
```

  where `f(i)` are some computations
- To simplify pseudocodes we will use the construct `par_for`

```
par_for i = 0 to n:
    x[i] = f(i)
...
```

- The implementations of a parallel `for` usually add an overhead of lg n but we will able to ignore it in most cases

Parallel Matrix-Vector Multiplication

- We apply parallel loops first in a parallel matrix-vector product

```
def par_matr_vect(a, x):
    n = a.shape[0]
    y = vector(n)
    par_for i = 0 to n-1:
        y[i] = 0
    par_for i = 0 to n-1:
        for j = 0 to n-1:
            y[i] += a[i, j] * x[j]
    return y
```

- Here $T_1(n) = n^2$ and the overhead $\lg n$ of the parallel `for` can be ignored as we have

$$T_\infty(n) = \Theta(n + \lg n) = \Theta(n)$$

- The parallelism is thus here $\frac{T_1(n)}{T_\infty(n)} = \Theta(n)$, quite substantial
  - Achievable with n processors
  - This is basically the speed up attained by GPUs

A Wrong Version

- At first sight we could think of a possibly better version of matrix-product products:

```
def par_matr_vect(a, x):
    n = a.shape[0]
    y = vector(n)
    par_for i = 0 to n-1:
        y[i] = 0
    par_for i = 0 to n-1:
        par_for j = 0 to n-1:
            y[i] += a[i, j] * x[j]
    return y
```

  - This would result in $T_\infty(n) = \Theta(\lg n + 1) = \Theta(\lg n)$
  - But it is dangerous, as several parallel instructions in the `par_for j = 0 to n-1:` loop would try to write in the same memory location
- This situation is called a **determinacy race** and is an example of **race conditions**
- The simplest way to handle race conditions is just to avoid them by ensuring that **strands that operate in parallel are independent**
  - This is not the case in the code above

Parallel Matrix Multiplication

- The previous approach also works for matrix multiplications

```
def par_matr_mult(a, b):
    n = a.shape[0]
    c = matrix(n, n)
    par_for i = 0 to n-1:
        par_for j = 0 to n-1:
            c[i, j] = 0
            for k = 0 to n-1:
                c[i, j] += a[i, k] * b[k, j]
    return c
```

- Here $T_1(n) = n^3$ but $T_\infty(n) = \Theta(n + \lg n) = \Theta(n)$
- Thus the parallelism is here $\frac{T_1(n)}{T_\infty(n)} = \Theta(n^2)$, again quite substantial
  - Achievable with n^2 processors
- But we can further improve it following a parallel recursive approach

Recursive Parallel Matrix Multiplication

- We will parallelize the recursive matrix multiplication algorithm

```
def par_rec_matr_mult(a, b, c):
    n = a.shape[0]
    if n == 1:
        c_11 = a_11 b_11
    else:
        t = matrix(n, n)
        partition a, b, c, t into n/2 submatrices
        a_11, a_12, ..., t_22
        spawn par_rec_matr_mult(a_11, b_11, c_11)
        spawn par_rec_matr_mult(a_12, b_21, t_11)
        ...
        spawn par_rec_matr_mult(a_21, b_12, c_22)
        par_rec_matr_mult(a_22, b_22, t_22)
        sync
        par_for i = 0 to n-1:
            par_for j = 0 to n-1:
                c_ij = c_ij + t_ij

    return c
```

Recursive Parallel Matrix Mult. II

- Here we are using the block matrix multiplication formulae

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

- We start by storing in the matrices C_{ij} the index $k = 1$ multiplications

$$C_{ij} = A_{i1}B_{1j}$$

and in the matrices T_{ij} the index $k = 2$ multiplications

$$T_{ij} = A_{i2}B_{2j}$$

- After ths `sync` we arrive at the final sum matrix C_{ij} as

$$C_{ij} = C_{ij} + T_{ij}$$

Recursive Parallel Matrix Mult. III

- Here we have again $T_1(n) = n^3$
- The (simplified) recurrence for $T_\infty(n)$ is $T_\infty(1) = 1$ and

$$T_\infty(n) = \lg n + T_\infty\left(\frac{n}{2}\right)$$

  whose solution is $T_\infty(n) \simeq (\lg n)^2$
- Thus the parallelism is here $\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n^3}{(\lg n)^2}\right)$, very high
- Using Strassen's approach the parallelism would be $\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n^{\lg 7}}{(\lg n)^2}\right)$, also very high

Parallel Sums

- We enlarge our parallel instructions with an **apply-to-each** instruction

```
[ f(a) for a in l if cond(a) == True]
```

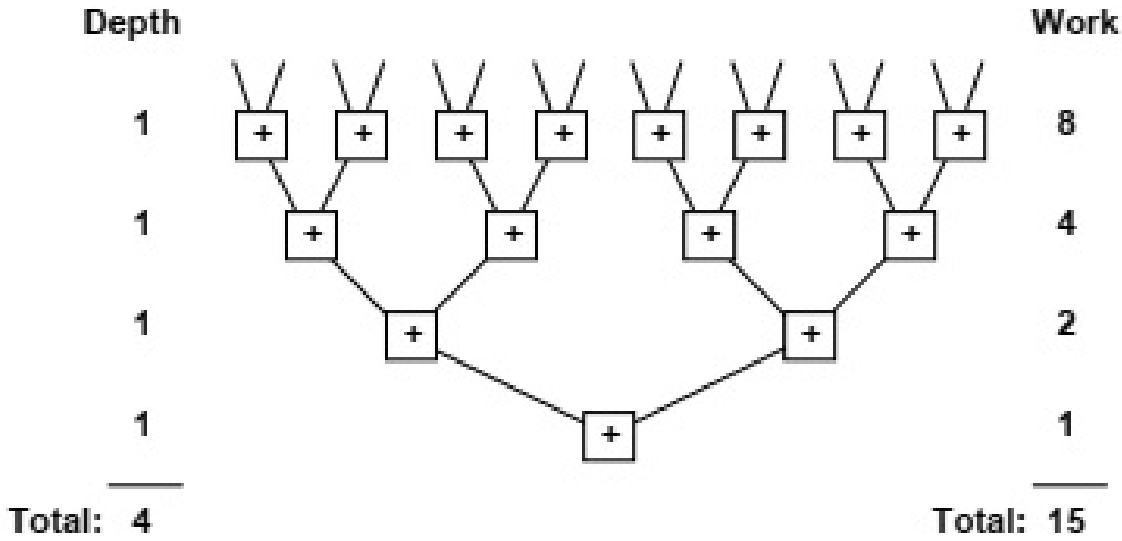
  which applies the function `f` in parallel to each element `a` in the list `l`
  - We mimic the list comprehension syntax of Python
- We use it to define a parallel sum algorithm

```
def par_sum(t):
    """Only when len(t) = 2**k for some k"""
    if len(t) == 1:
        return t[0]
    else:
        tt = [ t[2*i] + t[2*i+1] for i in range(lent(t)/2)] ]
        return par_sum(tt)
```

  - Here we just assume `len(t)` to be a power of 2

Parallel Sums II

- Notice that it is quite easy to transform it in an iterative **tournament** algorithm
- For instance, we can sum16 elements as follows



Parallel Sums III

- We assume that apply-to-each can be implemented with $O(1)$ cost
- The work and span recurrences are

$$T_1(n) = \frac{n}{2} + T_1\left(\frac{n}{2}\right), \quad T_\infty(n) = 1 + T_\infty\left(\frac{n}{2}\right)$$

  with solutions $T_1(n) = n - 1$, $T_\infty(n) = \lg n$
- The parallelism is thus $\frac{n}{\lg n}$
- We can obviously replace the sum with any binary operation
  - Hence, we can get the max and min of a table in $\lg n$ time

More on Parallel Matrix Multiplication

- Another parallel algorithm for matrix multiplication is

```
def par_matr_mult(a, b):
    n = a.shape[0]
    c = tensor(n, n, n)
    par_for i = 0 to n-1:
        par_for j = 0 to n-1:
            par_for k = 0 to n-1:
                c[i, j, k] = a[i, k] * b[k, j]
    par_for i = 0 to n-1:
        par_for j = 0 to n-1:
            c[i, j, 0] = par_sum(c[i, j, :])

    return c[ : , : , 0]
```

- Here $T_1(n) = n^3$ but $T_\infty(n) = \lg n$ because of the parallel sums
- The parallelism is thus $\frac{n^3}{\lg n}$
  - But requires n^3 processors

Parallel Sorting

Parallel QuickSort

- QuickSort is quite easy to parallelize

```
def par_quicksort(t):
    if len(t) == 1:
        return t
    p = t[rand(n)]                          #random pivot

    left = [ a if a < p for a in t]         #parallel fors
    right = [ a if a > p for a in t]

    left = spawn par_quicksort(left)
    right = spawn par_quicksort(right)
    mid = [ a if a == p for a in t]
    sync

    return left + mid + right
```

- To analyze the algorithm we will approximate the work and span recurrences as

$$T_1(n) = n + 2T_1\left(\frac{n}{2}\right), \quad T_\infty(n) = 1 + T_\infty\left(\frac{n}{2}\right)$$

with solutions $T_1(n) = n \lg n$, $T_\infty(n) = \lg n$ and parallelism n

A First MergeSort

- We can parallelize MergeSort as

```
def par_mergesort(t: List, f: int, l: int):
    """f, l: first and last indices"""
    if f < l:
        m = (f + l + 1) // 2
        if m-1 > f:
            spawn par_mergesort(t, f, m-1)
        if l > m:
            spawn par_mergesort(t, m, l)
        sync
        merge(t, f, m, l)    #standard merge
```

- When $p = 1$ we have standard MS and, hence, $T_1(n) = \Theta(n \lg n)$
- For T_∞ we have $T_\infty(n) = \Theta(n) + T_\infty\left(\frac{n}{2}\right)$ with solution $T_\infty(n) = \Theta(n)$ and a small $\Theta(\lg n)$ parallelism
  - The bottleneck is in the standard, sequential `merge` function
  - We see later how to improve on this to have a **parallel merge** with $T_\infty(n) = \Theta(\lg n)$

Parallel MergeSort

- Assuming a $T_\infty(n) = \Theta(\lg n)$ merge, we can change our previous parallel version of MergeSort

```
def par_mergesort(t: List, f: int, l: int):
    if f < l:
        m = (f + l + 1) // 2
        spawn par_mergesort(t, f, m)
        par_mergesort(t, m+1, l)
        sync
        par_merge(t, f, m, l)
```

- The T_∞ recurrence is now

$$T_\infty(n) = \Theta(\lg n) + T_\infty\left(\frac{n}{2}\right)$$

with the solution $T_\infty(n) = \Theta((\lg n)^2)$

- But the T_1 recurrence is now

$$T_1(n) = n \lg n + 2T_1\left(\frac{n}{2}\right)$$

with the solution $T_1(n) = \Theta(n(\lg n)^2)$, which gives a $\Theta(n)$ parallelism

A Parallel Merge

- Assume that we have to merge the subtables `t[f : m], t[m : l+1]`, which for clarity we denote as `x, y`
- Take `x[i], i in [f, m)` and let `j in [m, l]` be so that `y[j]` is the first element in `y` bigger than `x[i]`
  - If all elements in `y` are smaller than `x[i]` , we take `j = l - m + 1`
- Then, we have
  - `x[i]` has `i-f` smaller elements in `x`
  - `x[i]` has `j-m` smaller elements in `y`
- Thus `x[i]` goes to the position `f + (i-f)+ (j-m)= i + j - m` in the merged table
- We can argue in just the same way with the elements in `y` respect those in `x`

A Parallel Merge II

- Let `bs(x, t, f, l)` be a function that returns the index of the first element of `t[f : l+1]` bigger than `x`
  -
  - We will see later how we can modify binary search to define such a function `bs` with cost $O(\lg n)$
- We define a parallel merge as follows

```
def par_merge(t: List, f: int, m: int, l: int):
    z = vector(l - f + 1)              #first index 0, not f
    par_for i in range(f, m):
        j = bs(t[i], t, m, l)
        z[i + j - m - f] = t[i]        #substract f to move on z
    par_for j in range(m, l + 1):
        i = bs(t[j], t, f, m-1)
        z[j + i - m - f] = t[j]

    t[f : l + 1] = z                   #copy z back into f
```

- Thus, the span T_∞ of `par_merge` will be $\Theta(\lg n)$
  - But its work is $T_1(n) = n \lg n$, worse than that of standard merge
  - A better version with $T_1(n) = n$ can be given

Adapting Binary Search

- We just adapt binary search to search for rankings

```
def bs(x: int, t: List, f: int, l: int):
    if x >= t[l]:   #all elements are smaller than x
        return l + 1

    while f < l:
        mid = (f + l) // 2
        if x <= t[mid]:
            l = mid
        else:
            f = mid + 1
    return l

t = list(range(5))
print(bs(0, t, 0, 4))
>>> 0
print(bs(10, t, 0, 4))
>>> 5
```

Iterative MergeSort

- An alternative to recursive MergeSort on a table of size 2^K is to iteratively merge from left to right consecutive subtables of sizes 1, 2, 4, ... up to 2^{K-1}
- For instance given `[6 4 0 7 5 2 3 1]`, the iterations' evolution would be

```
size 1   pre-merge:  [6 | 4 | 0 | 7 | 5 | 2 | 3 | 1]
size 2   post-merge:     [4 6 | 0 7 | 2 5 | 1 3]
size 4   post-merge:       [0 4 6 7 | 1 2 3 5]
         post-merge:         [0 1 2 3 4 5 6 7]
```

- This idea is easy to parallelize as we can perform in parallel the mergings of the subtables of sizes $2^k, k = 0, \ldots, K-1$
  - It is quite easy to extend this to tables of size a general N

Parallel Iterative MergeSort

- A schematic code could be

```
for i in range(1, k+1):
    par_for j in range(2**(k-i)):
        f = j * 2**i
        l = f + 2**i - 1
        m = (f+l+1)//2

        merge(t, f, m, l)
```

- Simplifying the work of `merge` on a table of size `m` to be just `m` (instead of `m-1`), the work for $N = 2^K$ would be

$$T_1(N) = \sum_{j=1}^{K} \left(2^j \times 2^{K-j}\right) = \sum_{1}^{K} 2^K = K2^K = N \lg N$$

  - At each iteration we have to merge 2^{K-j} subtables of size 2^j
- But the span is just $T_\infty(N) = K = \lg N$
- That is we have the same N parallelism as before, but with an iterative algorithm