Lists, Stacks, Queues

in Python

Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2020/index.html

Outline

Digression I: measuring time

Using `time`: `import time` OR `from time import *`

`time.time()`

usually 01.01.1970

- return time in **seconds** since *epoch* as floating point number
- underlying system may not measure time with better precision than 1 second
  ⇒ fractional part of returned time not reliable

`time.perf_counter()`

- as above, but uses system clock with highest available resolution
- appropriate for measuring short durations
- measures *elapsed time* or *wall-clock time*
  ⇒ dependent on system load

`time.process_time()` ✓

- as above, but measures time taken by current process
  ⇒ independent of system load

Digression I: measuring time

Example:

```
from time import *

start_time = time() # try perf_counter(), process_time()
L = [ i**3 for i in range(100000) ]
sleep(2)            # does this affect the measured time?
end_time = time()
print("Time taken: %.4f seconds\n" % (end_time - start_time))
```

Digression I: measuring time

Using `timeit`: `import timeit` OR `from timeit import *`

```
timeit(stmt='pass', setup='pass', timer=time.perf_counter,
number=1000000, globals=None)
```

- returns time in seconds required to run `stmt` $\times$ `number` times
- `setup` : initialisation code (if any) to be executed (not included in measured time)
- `timer` : which system timer to use
- `globals` : specifies *namespace* (global variables, function names, etc.)
- all arguments are optional

```
repeat(stmt='pass', setup='pass', timer=default, repeat=5,
number=1000000, globals=None)
```

- as above, returns a list containing `repeat` timing values

Positional vs. keyword / named arguments VS. default parameter values: see
https://stackoverflow.com/questions/9450656/positional-argument-v-s-keyword-argument

Digression I: measuring time

Examples

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

timeit('[func(42) for func in (f,g,h)]', globals=globals())
timeit(number=100000000)
timeit(number=100)
```

Digression I: measuring time

Using `timeit` from the command line

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement ...]
```

- `n` : how many times to execute `statement`
- `r` : how many times to repeat (default 5)
- `s` : initialisation
- `p` : use `time.process_time()` instead of `time.perf_counter()`
  (i.e., measure process time, not wallclock time)
- `u` : unit to be used (nsec, usec, msec, or sec)

Example:

```
python3 -mtimeit -s'import your_filename' 'your_filename.f()'
```

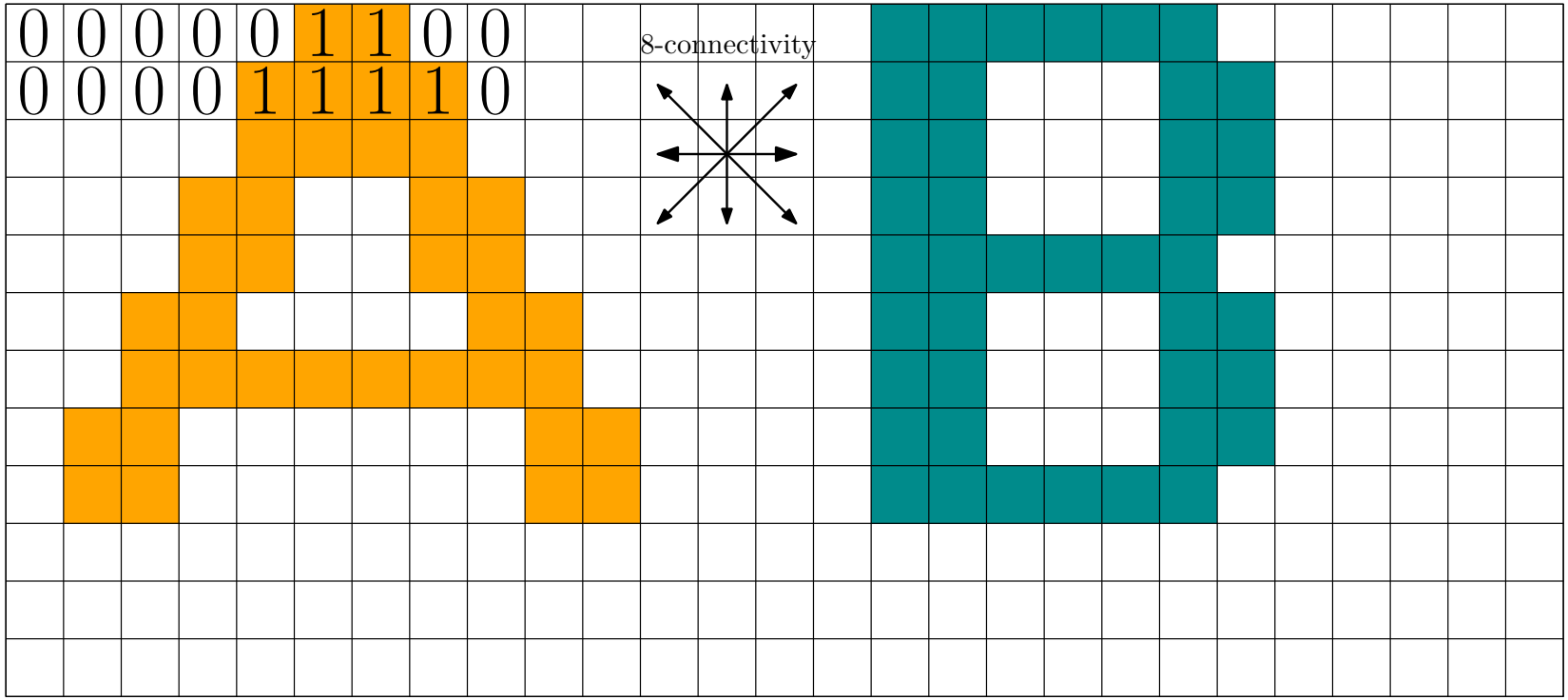
Outline

Designing data structures

- **Question:** How to store / organise data?
- **Counter-questions:**
  1. Can we choose what language to code in?
  2. If yes, how performance-critical is your application? How much time do we have as programmers?
  3. What kind of operations will be needed? $\longrightarrow$ defines *API*
- **Steps:**
  1. Choose appropriate data types supported by the language.
  2. Implement desired operations using the above.
  3. Make a library / package when satisfied.
  4. Repeat if necessary.

Stacks (LIFO)

Problem: Find the *connected components* in an image.



Operations:

- PUSH: insert at top
- POP: remove and return element at top
- LENGTH or HEIGHT

Queues (FIFO)

Problem: Maintain a list of patients waiting to consult a doctor.

Operations:

- INSERT or ENQUEUE: insert at end
- DELETE or DEQUEUE: remove and return element at front
- LENGTH or SIZE
- ISEMPTY, ISFULL

Lists

Problem: Maintain a database of student information for the mentor committee: roll number, name, stream, courses taken, etc.

Operations:

- INSERT
  - at beginning, end, other position (by index)
  - before / after specific element
- DELETE
  - at beginning, end, other position (by index)
  - before / after specific element
- LENGTH: number of elements in the list
- GET: return the value of an item by index
- SEARCH
- ITERATE or FOREACH
- SORT: on a specified attribute

Recap: lists in Python

- Slicing: `L[i : j : k]`
  - `i` = starting index (included), default = 0
  - `j` = ending index (excluded), default = `len(L)`
  - `k` = step / stride, default = 1
  - Slice assignment: `S[i:j:k] = I`
    - similar to deleting and inserting where deleted
    - assignment to *basic slices* (`S[i:j]`) need not match in size
    - assignment to *extended slices* (`S[i:j:k]`) must match in size
- `L.append(x)` `L.insert(i, X)` `L.extend(I)`
- `L.pop(i)` : delete and return last item (or item at index `i`)
- `L.copy()` : make **shallow** copy

What about `L.push()` ?

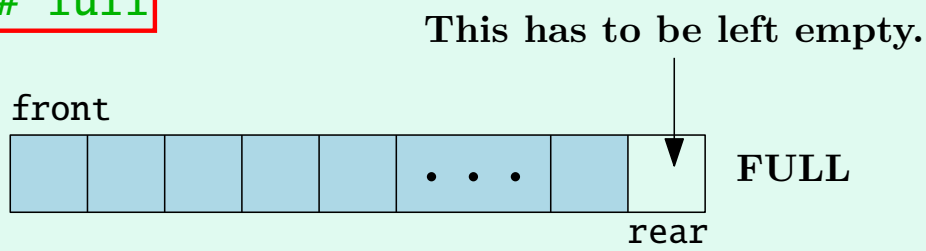
Recap: lists in Python

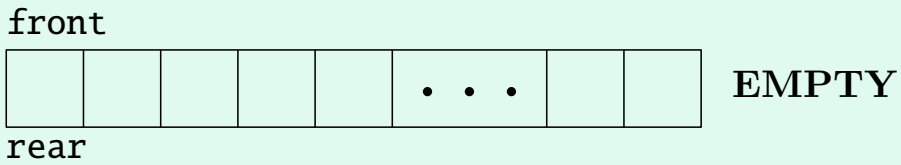
- `L.index(X, i, j)` : returns index of first occurrence of `X` in `L[i:j]`
  exception (error) if X is not found
- `L.sort(key=None, reverse=False)` (also see `sorted(L)`
- `L.reverse(key=None, reverse=False)` (also see `reversed(L)`

Alternative implementation of queues

```
def enqueue(L, x) :
    global rear
    if (rear + 1) % size == front : # full
        print("Queue is full")
    else :
        L[rear] = x
        rear = (rear + 1) % size

def dequeue(L) :
    global front, rear, size
    if front == rear : # empty
        print("Queue is empty")
    else :
        x = L[front]
        front = (front + 1) % size
        return x
```

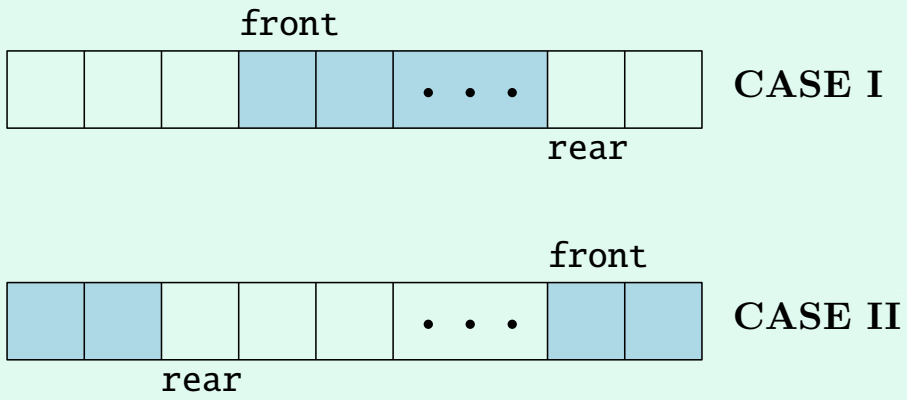




Alternative implementation of queues

```
def length(L) :
    if front <= rear :
        return rear - front
    else :
        return size - (front - rear)

def display(L) :
    if front <= rear :
        print(L[front:rear])
    else :
        print(L[front:] + L[:rear])
```



Digression III: **random**

```
import random
random.randint(1, 100) # The interval is [1, 100)
random.random()        # The interval is [0, 1.0)
```

In-built lists vs. alternative implementation of queues

```
for N in [ 10, 100, 1000, 10000, 100000 ] :
    queue_alt.size = N+1
    Q = [ 0 for i in range(queue_alt.size) ]

    start = time.process_time();
    for i in range(N) :
        queue_alt.enqueue(Q, i) # random.randint(0,N)
    for i in range(N) :
        x = queue_alt.dequeue(Q)
    end = time.process_time();
    print("Time for %d elements (alternative implementation): %.4f" % (N, end -
     start))

    start = time.process_time();
    for i in range(N) :
        Q[i] = i # random.randint(0,N)
    for i in range(N) :
        x = Q.pop(0)
    end = time.process_time();
    print("Time for %d elements (default implementation): %.4f" % (N, end -
     start))
```

In-built lists vs. alternative implementation of queues

Size	Default	Alternative	
10	0.0000	0.0000	
100	0.0000	0.0001	
1000	0.0003	0.0006	*Consistent but unexplained*
10000	**0.0103**	**0.0061**	
100000	**1.0669**	**0.0545**	

Lessons learnt

- The easiest way (for you) is not necessarily the best.
- A more general / sophisticated / powerful data structure with many features is not necessarily the best for a task.

Lessons learnt

- The easiest way (for you) is not necessarily the best.
- A more general / sophisticated / powerful data structure with many features is not necessarily the best for a task.

What is a linear list good for?

- If you need to access by index (aka *random access*)
- If insertions / deletions only happen at the ends
  (not good if insertions / deletions in the middle are frequent)

Lessons learnt

- The easiest way (for you) is not necessarily the best.
- A more general / sophisticated / powerful data structure with many features is not necessarily the best for a task.

What is a linear list good for?

- If you need to access by index (aka *random access*)
- If insertions / deletions only happen at the ends
  (not good if insertions / deletions in the middle are frequent)

More questions

- When might frequent insertions / deletions in the middle be needed?
- What data structures to use in such cases? (coming soon)

Problems I

1. Write a program that takes two positive integers (say M, N) as inputs. It should first enqueue N random integers one by one into a queue, and then dequeue the N elements. This process should be repeated M times.
   Use the following in turn to store the queue:
   (a) a usual Python list, and
   (b) the alternative implementation suggested above,

   and compare the times taken by the two implementations.

Problems II

2. Given a list of numbers as input, write a program to compute the nearest larger value for the number at position i (nearness is measured in terms of the difference in array indices). For example, in the array $[1, 4, 3, 2, 5, 7]$, the nearest larger value for 4 is 5. Implement a naive, $O(n^2)$ time algorithm, as well as an $O(n)$ time algorithm for this problem. Compare the run times of your algorithms, and interpret the results.
3. Given a set of sorted numbers as user input, write a program to remove the duplicate elements.

Problems III

4. There are N petrol pumps $P_1, P_2, \ldots, P_N$ arranged in a clockwise direction along a circular road. Consider a truck with a fuel tank that is initially empty, but which has infinite capacity. The truck will initially fuel up at some P_i and move in a clockwise direction along the circular road. Each time it reaches a petrol pump, it will take up all the petrol available at that pump.

   Write a program to determine the first P_i such that if the truck starts from P_i, it will be able to completely traverse the circle and return to P_i.

   The amount of petrol that every petrol pump has (in litres), and the distance from that petrol pump to the next petrol pump (in km) will be given to you as command line arguments. Assume that the truck needs 1 litre of fuel to travel 1 km.

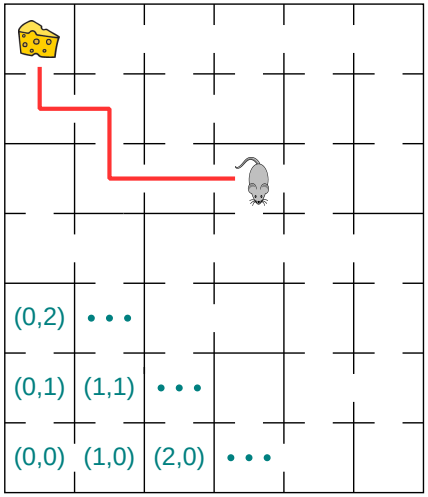
Problems IV

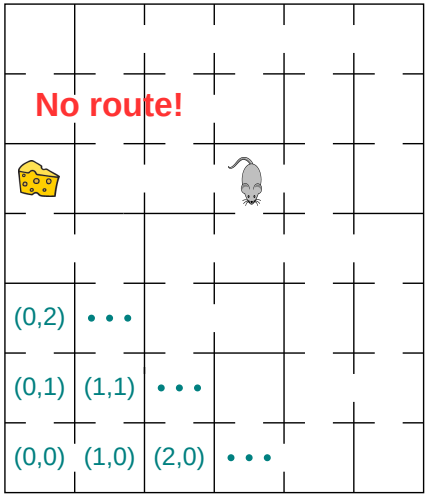
Example: Let there be 4 petrol pumps having 4, 6, 7, and 4 litres of petrol respectively. Let the distance between these pumps be 6, 5, 3, and 5 km respectively. Then your program should output 2, since the first pump from which the truck can complete a circular tour is the 2nd petrol pump.

Also think about when such a tour will *not* be possible.

Problems V

5. Suppose that you have an $m \times n$ maze of rooms. Each adjacent pair of rooms has a connecting door that allows passage between the rooms. However, some of the doors are locked, while the rest are open. A mouse sits in room number (s, t) and there is fabulous food for the mouse at room number (u, v). Your task is to determine whether there exists a route for the mouse from room (s, t) to room (u, v) through the open doors.





Problems VI

The idea is to start searching for a path from room (s, t) by looking at neighbouring rooms $(s_1, t_1), \ldots, (s_k, t_k)$ that can be reached from (s, t), and then those rooms that can be reached from each (s_i, t_i), and so on. A previously visited room should not be revisited during the search. You may maintain an $m \times n$ array of flags to keep track of visited rooms. Write programs to solve this problem using each of the following search strategies in turn.

(a) Use a stack to implement the search. Initially, push (s, t) on the empty stack. Subsequently, as long as the stack is not empty, pop a room from the stack. Let this room be (x, y). If $(x, y) = (u, v)$ (the target room), the search ends. Otherwise, push each adjacent room that can be visited from (x, y) but has not been visited so far. If the stack becomes empty without reaching (u, v), then there is no (s, t)–(u, v) path.

Problems VII

(b) Implement the search using a queue. Initially add (s, t) to an empty queue. Subsequently, as long as the queue is not empty, remove the room (x, y) from the head of the queue. If $(x, y) = (u, v)$, then report success and return. Otherwise, enqueue all unvisited rooms adjacent to (x, y). If the search stops, i.e., the queue becomes empty, report failure.

Input file format:

```
10 # number of test cases
# Test case 1
7 6 # number of rows, columns
3 4 # x, y coordinates of mouse
0 6 # x, y coordinates of cheese
44 # number of doors
0 0 1 0 # x1, y1, x2, y2 coordinates of rooms joined by door 1
0 0 0 1 # x1, y1, x2, y2 coordinates of rooms joined by door 2
...
```

Problems VIII