Introduction to Data Structures

Authors: Darren Yao, Benjamin Qi, Allen Li, Neo Wang
Contributors: Nathan Wang, Abutalib Namazov

What a data structure is, (dynamic) arrays, pairs, and tuples.

Language: Python Edit This Page

TABLE OF CONTENTS

A **data structure** determines how data is organized so that information can be used efficiently. Each data structure supports some operations efficiently, while other operations are either inefficient or not supported at all. Since different operations are supported by each data structure, you should carefully evaluate which data structure will work best for your particular problem.

Lists

The default way to store data in Python is using a **list**, which can automatically resize itself to accommodate more elements. We can add and delete elements at the end in $\mathcal{O}(1)$ time. A list can be initialized as follows:

Copy PYTHON

```
arr = []
```

Python lists are **generic**. This means that they can store any kind of data type, including objects. For example, the following code creates a dynamic array and adds the numbers 1 through 10 to it:

Copy PYTHON

```
for i in range(1, 11):  # Note that range(i, j) includes i, but does not include j
    arr.append(i)
```

In Python, we can give a dynamic array an initial size. The code below creates a dynamic array with 30 zeroes.

Copy PYTHON

```
arr = [0] * 30
```

Iterating

We can use a regular for loop to iterate through all elements of a list.

Copy PYTHON

```
arr = [1, 7, 4, 5, 2]

```

```
for i in range(len(arr)):
    print(arr[i], end=" ")
print()

for element in arr:
    print(element, end=" ")
print()
```

We can also use **iterators**. An iterator allows you to traverse a container by pointing to an object within the container. `iter(arr)` returns an iterator pointing to the first element of the list `arr`.

Copy PYTHON

```
arr = [4, 2, 0, 0, 5]
it = iter(arr)

print(next(it))  # 4
print(next(it))  # 2
print(next(it))  # 0
```

Inserting and Erasing

Copy PYTHON

```
arr = []
arr.append(2)  # [2]
arr.append(3)  # [2, 3]
arr.append(7)  # [2, 3, 7]
arr.append(5)  # [2, 3, 7, 5]
arr[1] = 4
# sets element at index 1 to 4 -> [2, 4, 7, 5]
```

List Comprehensions

List comprehensions are extremely useful for simplifying a python for loop that modifies/creates a list into one expression. The general syntax is: `[ expression for item in list if conditional ]`

An example is provided in the code block below.

Copy PYTHON

```
# If a number is odd, add the number times 2 into the array
old_list = [2, 5, 3, 1, 6]
new_list = []
for i in old_list:
    if i % 2 == 1:
        new_list.append(i * 2)
print(new_list)  # [10, 6, 2]
# Simplified one liner with list comprehension
# Recall the form [ expression for item in list if conditional ]
# expression: i * 2
# list: old_list
# conditional: i % 2 == 1 (only include item i if it satisfies the conditional)
new_list = [i * 2 for i in old_list if i % 2 == 1]
print(new_list)  # [10, 6, 2]
```

A very applicable use of list comprehensions for competitive programming in particular is creating an integer list from space separated input:

Copy PYTHON

```python
# Example input: 5 3 2 6 8 1
# Note that the conditional in the list comprehension is optional, and defaults to True if not provided
arr = [int(x) for x in input().split()]
print(arr)  # [5, 3, 2, 6, 8, 1]
```

For more information on list comprehensions, including nesting them to create multidimensional lists, refer to the below resources.

RESOURCES

PythonForBeginners	**List Comprehensions in Python**	Basic list comprehension tutorial
GFG	**Nested List Comprehensions in Python**	Nesting list comprehensions

Pairs

If we want to store a collection of points on the 2D plane, then we can use a dynamic array of **pairs**.

While Python doesn't have a specific class just for pairs, 2-element **tuples** give nearly the exact same functionality. The only issue is that you can't modify the elements since tuples are immutable.

On the other hand, Python has built-in comparison support for tuples. When comparing, it looks at the first elements of each pair, then the second, and so on and so forth.

Copy PYTHON

```python
"""
Output:
(5, 'asdf')
5
True
"""

p1 = (5, "asdf")
print(p1)
print(p1[0])  # access the first element of the tuple

p2 = (6, "asdf")
print(p1 < p2)
```

Memory Allocation

One thing to keep in mind when using arrays is the memory limit. Usually the USACO memory limit is 256 MB. To estimate how many values can be stored within this limit:

1. Calculate the total memory size in bytes: for 256 MB, that's $256 \cdot 10^6$.
2. Divide by the size, in bytes, of an `int` (4), or a `long long` (8), etc. For example, the number of `int`s that you are able to store is bounded above by $\frac{256 \cdot 10^6}{4} = 64 \cdot 10^6$.
3. Be aware that **program overhead** (which can be very significant, especially with recursive functions) will reduce the amount of memory available.

Quiz

How do you count the number of items in a `list`? Suppose we named the list `l`.

1

```python
len(l)
```

PYTHON

2

```python
count(l)
```

PYTHON

3

```python
size(l)
```

PYTHON

4

```python
l.length()
```

PYTHON

← Previous Question 1 of 3 Skip →

Problems

Nothing to see here! To reiterate, arrays of fixed size should suffice for essentially every Bronze problem, but dynamic arrays, pairs, and tuples can greatly simplify implementation at times. You'll see some examples of these in the following module.

Module Progress: Not Started
