PROGRAMMING AND ALGORITHMS II

GABRIEL VALIENTE

ALGORITHMS, BIOINFORMATICS, COMPLEXITY AND FORMAL METHODS
RESEARCH GROUP, TECHNICAL UNIVERSITY OF CATALONIA

2023–2024

PARAMETER PASSING

REFERENCES VERSUS POINTERS

- We assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed.
- This assumption is valid in most systems because a pointer to the array is passed, not the array itself.
- CLRS Problem 4-2 examines the implications of three parameter-passing strategies:
  - An array is passed by pointer.
  - An array is passed by copying.
  - An array is passed by copying only the subrange that might be accessed by the called procedure.

- In Python, if **a** refers to an object and you assign **b = a**, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

- The association of a variable with an object is called a reference. In this example, there are two references to the same object.
- An object with more than one reference has more than one name, so we say that the object is aliased.

- If the aliased object is mutable, changes made with one alias affect the other:

  ```
  >>> b[0] = 42
  >>> a
  [42, 2, 3]
  ```

  Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.
- For immutable objects like strings, aliasing is not as much of a problem. In this example:

  ```
  a = 'spam'
  b = 'spam'
  ```

  It almost never makes a difference whether **a** and **b** refer to the same string or not.

- When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, **delete_head** removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

- The parameter **t** and the variable **a** are aliases for the same object:

```
>>> a = [1, 2, 3]
>>> delete_head(a)
>>> a
[2, 3]
```

- The default mechanism for parameter passing in C++ is pass by value, which causes a separate copy of the data value for each parameter to be made.
- Since a completely separate copy is used, any changes to the formal parameters in the function are not reflected in the actual parameters.
- This allows you to effectively treat the formal parameters as additional local variables since changes made to them do not directly affect other parts of the program.
- As with Python, it does not matter whether the names of the formal and actual parameters match.

```
#include <iostream>
using namespace std;

void f(int a,
       int b) // a and b are the formal
              // parameters
{
  cout << a << " " << b << endl;
  a += 3;
  b += 5;
  cout << a << " " << b << endl;
}

int main() {
  int x = 1, y = 2;
  f(x, y); // x and y are the actual
           // parameters
  cout << x << " " << y << endl;
}
```

```
1 2
4 7
1 2
```

- C++ also supports a second parameter passing mechanism known as pass by reference in which, instead of making a copy of the data value, a reference (the address in memory) to the data is passed.
- With pass by reference in C++, any change, including assigning a new value, to the formal parameter is reflected in the actual parameter.
- Pass by reference is indicated by putting an ampersand (&) in front of the formal parameter (but not the actual parameter).
- The corresponding actual parameter for any formal parameter that uses pass by reference must be a variable, not an expression.

```
#include <iostream>
using namespace std;

void f(int a,
       int &b) // a and b are the formal
               // parameters
{
  cout << a << " " << b << endl;
  a += 3;
  b += 5;
  cout << a << " " << b << endl;
}

int main() {
  int x = 1, y = 2;
  f(x, y); // x and y are the actual
           // parameters
  cout << x << " " << y << endl;
}
```

```
1 2
4 7
1 7
```

- C++ supports marking parameters **const**, which means that the function cannot change the parameter.

```
#include <iostream>
using namespace std;

void f(const int a, int b) {
  a = 2; // this will generate a
         // compiler error
  b = 2; // this is fine
}

int main() {
  int x = 1, y = 2;
  f(x, y); // x and y are the actual
           // parameters
  cout << x << " " << y << endl;
}
```

- C++ vectors are passed by (constant) reference for efficiency reasons.

Python

- Immutable objects (**str**, **tuple**) are passed by object reference but cannot be changed.
- Mutable objects (**list**) are passed by object reference.

C++

- Parameters passed by value cannot be changed.
- Parameters passed by constant reference cannot be changed.
- Parameters passed by reference can be changed.

PYTHON VERSUS C++

LISTS, DICTIONARIES, AND SETS IN PYTHON

- Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items.
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets:

```
[0, 1, [2, [3, [4]]], 5, 'a', 'b', 'c']
```

- A list within another list is nested.
- A list that contains no elements is called an empty list; you can create one with empty brackets or with the **list** constructor. You can assign list values to variables.

```
s = []
t = list()
```

- The syntax for accessing the elements of a list is the same as for accessing the characters of a string. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> t = list(
... map(chr, range(ord('a'), ord('z')+1)))
>>> t[0]
'a'
```

- Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned:

```
>>> t[1] = t[1].upper()
>>> t
['a', 'B', 'c', 'd', 'e', 'f', ..., 'z']
```

- List indices work the same way as string indices:
  - Any integer expression can be used as an index.
  - If you try to read or write an element that does not exist, you get an **IndexError**.
  - If an index has a negative value, it counts backward from the end of the list:

    ```
    >>> t[-1]
    'z'
    ```

- The **in** operator also works on lists:

  ```
  >>> 'z' in t
  True
  >>> 0 in t
  False
  ```

- The most common way to traverse the elements of a list is with a **for** loop. The syntax is the same as for strings:

```
for x in t:
    print x
```

- The indices are needed in order to write or update the elements. A common way to do that is to combine the built-in functions **range** and **len**:

```
for i in range(len(t)):
    t[i] = t[i].upper()
```

- A for loop over an empty list never runs the body.
- Although a list can contain another list, the nested list still counts as a single element.

- The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
```

- The * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> ['a', 'b', 'c'] * 2
['a', 'b', 'c', 'a', 'b', 'c']
```

- The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

- If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list:

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

- Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.
- A slice operator on the left side of an assignment can update multiple elements.

```
>>> s = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t = s.copy()
>>> t[1:3] = ['x', 'y']
>>> s
['a', 'b', 'c', 'd', 'e', 'f']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

- **append** adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

- **extend** takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
>>> t2
['d', 'e']
```

- **sort** arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

- Most list methods are void; they modify the list and return **None**.

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t = t.sort()
>>> t
>>>
```

- There are several ways to delete elements from a list. If you know the index of the element you want, you can use **pop**:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

- **pop** modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

- If you don't need the removed value, you can use the **del** operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

- If you know the element you want to remove (but not the index), you can use **remove**:

```
>>> t = ['a', 'b', 'c'] * 2
>>> t.remove('b')
>>> t
['a', 'c', 'a', 'b', 'c']
```

  The return value from **remove** is **None**.

- A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use **list**:

```
>>> list('spam')
['s', 'p', 'a', 'm']
```

- The **list** function breaks a string into individual letters. If you want to break a string into words, you can use the **split** method. An optional argument called a delimiter specifies which characters to use as word boundaries.

```
>>> s = 'spam-spam-spam'
>>> s.split()
['spam-spam-spam']
>>> s.split('-')
['spam', 'spam', 'spam']
```

- **join** is the inverse of **split**. It takes a list of strings and concatenates the elements. Since it is a string method, you have to invoke it on the delimiter and pass the list as a parameter:

  ```
  >>> t = ['spam'] * 3
  >>> ''.join(t)
  'spamspamspam'
  >>> ' '.join(t)
  'spam spam spam'
  ```

- A dictionary is like a list, but more general. In a list, the indices have to be integers. In a dictionary, they can be (almost) any type.
- A dictionary contains a collection of indices, which are called keys, and a collection of values. Each key is associated with a single value. The association of a key and a value is called an item.
- A dictionary represents a mapping from keys to values. Each key maps to a value.

- For example, we can build a dictionary that maps from three-letter to one-letter amino acid codes. The keys and the values are all strings:

```
>>> d = {'ALA': 'A', 'ARG': 'R',
...      'ASN': 'N', 'ASP': 'D',
...      'CYS': 'C', 'GLN': 'Q',
...      'GLU': 'E', 'GLY': 'G',
...      'HIS': 'H', 'ILE': 'I',
...      'LEU': 'L', 'LYS': 'K',
...      'MET': 'M', 'PHE': 'F',
...      'PRO': 'P', 'SER': 'S',
...      'THR': 'T', 'TRP': 'W',
...      'TYR': 'Y', 'VAL': 'V'}
```

- In general, the order of items in a dictionary is unpredictable:

```
>>> d
{'CYS': 'C', 'ASP': 'D', 'SER': 'S',
 'GLN': 'Q', 'LYS': 'K', 'ILE': 'I',
 'PRO': 'P', 'THR': 'T', 'PHE': 'F',
 'ALA': 'A', 'GLY': 'G', 'HIS': 'H',
 'GLU': 'E', 'LEU': 'L', 'ARG': 'R',
 'TRP': 'W', 'VAL': 'V', 'ASN': 'N',
 'TYR': 'Y', 'MET': 'M'}
```

  However, dictionaries preserve insertion order in Python 3.6.

- The function **dict** creates a new dictionary with no items.

```
>>> d = dict()
>>> d
{}
```

- To add items to the dictionary, you can use square brackets:

```
>>> d['ALA'] = 'A'
```

- This line creates an item that maps from the key 'ALA' to the value 'A':

```
>>> d
{'ALA': 'A'}
```

- This output format is also an input format.

- We use the keys to look up the corresponding values:

```
>>> d['ALA']
'A'
```

- If the key is not in the dictionary, we get an exception:

```
>>> d['ASX']
KeyError: 'ASX'
```

- The **len** function returns the number of items in the dictionary:

```
>>> len(d)
20
```

- The **in** operator tells as whether something appears as a key in a dictionary:

```
>>> 'ALA' in d
True
>>> 'GLX' in d
False
```

- To see whether something appears as a value in a dictionary, we can use the method **values**, which returns a collection of values, and then use the **in** operator:

```
>>> 'A' in d.values()
True
>>> 'Z' in d.values()
False
```

- Suppose we are given a list and we want to count how many times each element appears.

```
>>> histogram('abracadabra')
{'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
```

```
>>> histogram('the big bug bit the \
...            little beetle but the \
...            little beetle bit the \
...            big bug back'.split())
{'the': 4, 'big': 2, 'bug': 2, 'bit': 2,
 'little': 2, 'beetle': 2, 'but': 1,
 'back': 1}
```

- We create a dictionary with elements as keys and counters as the corresponding values. The first time we see an element, we add an item to the dictionary. After that we increment the value of an existing item.

```
def histogram(t):
    d = dict()
    for elem in t:
        if elem not in d:
            d[elem] = 1
        else:
            d[elem] += 1
    return d
```

- Dictionaries have a method called **get** that takes a key and a default value. If the key appears in the dictionary, **get** returns the corresponding value; otherwise it returns the default value.

```
def histogram(t):
    d = dict()
    for elem in t:
        d[elem] = d.get(elem, 0) + 1
    return d
```

- If we use a dictionary in a **for** statement, it traverses the keys of the dictionary.

```
def print_dict(d):
    for k in d:
        print(k, d[k])
```

```
def print_dict(d):
    for k in sorted(d):
        print(k, d[k])
```

```
>>> h = histogram('abracadabra')
>>> print_dict(h)
a 5
b 2
c 1
d 1
r 2
```

- Lists can appear as values in a dictionary. Given a dictionary that maps from elements to frequencies, we might want to create an inverted dictionary that maps from frequencies to elements. Since there might be several elements with the same frequency, each value in the inverted dictionary should be a list of elements.

```
def invert_dict(d):
    inverse = dict()
    for k in d:
        v = d[k]
        if v not in inverse:
            inverse[v] = [k]
        else:
            inverse[v].append(k)
    return inverse
```

```
>>> h = histogram('abracadabra')
>>> h
{'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
>>> invert_dict(h)
{5: ['a'], 2: ['b', 'r'], 1: ['c', 'd']}
```

- Lists can be values in a dictionary, but they cannot be keys. Keys have to be immutable, and lists are not. Tuples are mutable, though.

```
>>> d = dict()
>>> d[[1, 2, 3]] = 'whatever'
TypeError: unhashable type: 'list'
>>> d[(1, 2, 3)] = 'whatever'
```

- Since dictionaries are mutable, they cannot be used as keys, but they can be used as values.

- Python provides another built-in type, called a set, that behaves like a collection of dictionary keys with no values.
- Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.
- Example: Write a function called **has_duplicates** that takes a list and returns **True** if there is any element that appears more than once. It should not modify the original list.

```
def has_duplicates(t):
    d = dict()
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

PYTHON VERSUS C++

VECTORS, MAPS, AND SETS IN C++

- A **vector** is a sequence of elements of the same type that you can access by an index.

```
vector<int> v = {5, 7, 9, 4, 6, 8};
```

- You can also define a **vector** of a given size without specifying the element values, and the elements are given a default value according to the element type.

```
vector<int> v(6);
```

- Indices for a **vector** always start with 0 and increase by 1.
- A **vector** knows its size.

```
for (int i = 0; i < v.size(); ++i)
  cout << v[i] << endl;
```

- The range for a **vector v** is the half-open sequence

  $$[\ \texttt{0, 1, ..., v.size()}\ )$$

  that can be traversed using a range **for** loop.

```
for (int x : v) cout << x << endl;
```

- You can start a **vector** empty and grow it to the desired size as you read or compute the data you want in it.

```
int x;
vector<int> v;
while (cin >> x) v.push_back(x);
```

- Basically, `cin >> x` is true if a value was read correctly and false otherwise, so that the **while** statement will read all the **int** values you give it and stop when you give it anything else.

- A **map** is an ordered container of (key, value) pairs

```python
def histogram(t):
    d = dict()
    for x in t:
        d[x] = (
            d.get(x, 0)
            + 1
        )
    return d
```

```cpp
#include <map>
```

```cpp
map<char, int>
histogram(string t) {
  map<char, int> d;
  for (int i = 0;
       i < t.length();
       ++i)
    ++d[t[i]];
  return d;
}
```

- A **set** is an ordered container of keys

```
def has_duplicates(t):
    return len(
        set(t)
    ) < len(t)
```

```
#include <set>
```

```
bool has_duplicates(
    string t) {
  set<char> s;
  for (int i = 0;
       i < t.length();
       ++i)
    s.insert(t[i]);
  return s.size() <
         t.length();
}
```

PYTHON VERSUS C++

DATA TYPES IN PYTHON

Example (Some Python 3.10.6 data types)

- A **list** of n **int** of 8 bytes each uses about $56 + 8n$ bytes.

```
[sys.getsizeof(list(range(i))) for i in range(10)]
[56, 72, 72, 88, 88, 104, 104, 120, 120, 136]
```

- A **tuple** of n **int** of 8 bytes each uses $40 + 8n$ bytes.

```
[sys.getsizeof(tuple(range(i))) for i in range(10)]
[40, 48, 56, 64, 72, 80, 88, 96, 104, 112]
```

- A **str** of length n uses $49 + n$ bytes.

```
[sys.getsizeof('A' * i) for i in range(10)]
[49, 50, 51, 52, 53, 54, 55, 56, 57, 58]
```

- A **set** of n **int** of 8 bytes each uses about $216 + 48n$ bytes.

```
[sys.getsizeof(set(range(i))) for i in range(10)]
[216, 216, 216, 216, 216, 728, 728, 728, 728, 728]
```

PYTHON VERSUS C++

DATA TYPES IN C++

Example (Some C++ 11.3.0 data types)

Data type	Size (bytes)	
Character	1	`sizeof(char)`
Boolean	1	`sizeof(bool)`
Integer	2	`sizeof(short int)`
Integer	4	`sizeof(int)`
Integer	8	`sizeof(long int)`
Real	4	`sizeof(float)`
Real	8	`sizeof(double)`
Complex	8	`size(complex<int>)`
(List)	24	`sizeof(vector<int>)`
(Tuple)	1	`sizeof(struct ...)`
String	32	`sizeof(string)`
Set	48	`sizeof(set<int>)`
Dictionary	48	`sizeof(map<int, int>)`

- A **vector** x of n **int** of 4 bytes each uses about $24 + 4n$ bytes.

`x.size()`	0	1	2	3	4	5	6	7	8	9	$\cdots$
`x.capacity()`	0	1	2	4	4	8	8	8	8	16	$\cdots$