STMC HKOI Training

Lesson 6: Looping structure and arrays (II)

Chan Yan Mong

January 23, 2022

Goal today

Today we will look at more list features and see how we can combine it with loops to do more advance applications

- List slicing, concatenation and membership
- Multiple dimensions list
- Loop breaks
- Many applications:
  - Bomberman Solver
  - Fisher-Yates shuffle algorithm
  - Linear Search
  - Maximum Subarray Sum

List: Membership

- Sometimes we want to check if a member belongs to a list
- Python provide us with an easy way to do it using `in` keyword
- Here are some examples:

```
>> 3 in [1,2,3]
True
>> "Tim" in ['John','Jerry','May']
False
```

List: Concatenation

- We can also *"add"* two list by joining them together
- This is called **concatenation**
- This is done using the + operator
- For example:

```
>> [4,5,6] + [1,2,3]
[4,5,6,1,2,3]
>> ["Tim","Katy"] + ['John','Jerry','May']
['Tim', 'Katy', 'John', 'Jerry', 'May']
```

List: Slicing

- Sometimes we want to make a copy of *part* of the list
- This operation is called **slicing**
- The basic syntax of slicing is:

```
mylist[start:end]
```

- The slice **includes start but exclude end**

List: Slicing

- Here are some examples:

```python
mylist = [0,1,2,3,4,5,6]

mylist[0:1] # Return [0]
mylist[0:3] # Return [0,1,2]
mylist[3:5] # Return [3,4]
mylist[1:5] # Return [1,2,3,4]
```

List: Slicing

- One can also either start or end as blank
- If start is left blank, it's equivalent as putting zero as start
- If end is left blank, it's equivalent as putting `len(<the list you slice>)` as end
- More explicitly

```
mylist[:4] # Same as mylist[0:4]
mylist[3:] # Same as mylist[3:len(mylist)]
```

List: Slicing

- Here are some explicit examples:

```python

mylist = [0,1,2,3,4,5,6]

mylist[:4] # Same as mylist[0:4] so [0, 1, 2, 3]
mylist[2:] # Same as mylist[2:len(mylist)] so [2, 3, 4, 5, 6]

```

List: Slicing

- We can also specify the interval of slicing via the following syntax:

```
mylist[0:6:2] # This will give [0,2,4]
mylist[1:6:2] # This will give [1,3,5]
mylist[2:6:2] # This will give [2,4]
mylist[2:6:3] # This will give [2,5]
```

- Again, you can see the end is always excluded

List: Multidimensional List

- We can also put list inside list to create multidimensional lists (Or multidimensional array as other language will call)
- For example, the following is a 2-dimensional array with 3 rows and 4 columns. We shall denote such array `List(3,4)`

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

List: Multidimensional List

- One can implement the list above using the following python code:

```python
twoDim = [
  [1,2,3,4],
  [5,6,7,8],
  [9,10,11,12]
]

# Be careful about the order of the index
twoDim[0]    # [1,2,3,4]
twoDim[1]    # [5,6,7,8]
twoDim[0][0] # 1
twoDim[1][0] # 5
twoDim[0][2] # 3
twoDim[2][3] # 12
```

List: Multidimensional List

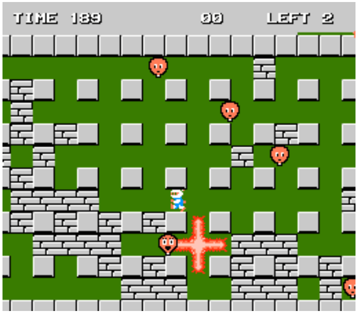
- We can also go higher dimension by keep wrapping list inside list. For example to create a list of form `List(2,2,2)`

```
threeDim = [
  [
    [0,1],
    [2,3]
  ],
  [
    [4,5],
    [6,7]
  ]
]
```

Example: Bomberman (I)

Reference: Ch.3 Sec.2 *"Aha! Visualizing Algorithms: The Basics"* (ISBN: 9863474363)

- Bomberman is an arcade-style maze-based video game developed by Hudson Soft on 1983.
- Since then multiple versions of the game have been released and it remained on the classics
- In the game, players have to place bombs in certain locations to remove walls and eliminate all enemies from the map



Example: Bomberman (I)

- We will now consider a simplified version of the game to explore what we can do with list
- We will make the following simplifications:
  1. The player only has one bomb
  2. The bomb is incredibly powerful. It blows up everything along the row and column it's placed and only stop when it hits a wall
  3. All walls are unbreakable
  4. Enemies will not move
- Now we ask a question: ***Given a map of walls and enemies, where can I put the bomb such that the largest number of enemies are eliminated?***

Example: Bomberman (I)

We will now formulate the problem:

- `map[n][m]` is a $n \times m$ grid index from 0
- Each grid holds a number from $0 - 2$:
  - 0 are empty showspaces
  - 1 are walls
  - 2 are enemies
- A bomb can only be placed on grid that holds 0
- For simplicity we will always make the outmost part of the list walls

For example, a 6×8 play grid can look like:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 & 2 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Example: Bomberman (I)

What happens if we place a bomb? For example, let's say we placed a bomb in `map[3][1]`

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 2 & 0 & 1 \\ 1 & \colorbox{lightgreen}{0} & 2 & 0 & 2 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Example: Bomberman (I)

Then it will destroy everything in the red block

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 2 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 & 2 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Example: Bomberman (I)

Now we can restate our problem

Bomberman (I)

Let `grid[m][n]` be a $m \times n$ grid with walls and enemies defined above. If we can only put one bomb on an empty grid, what is the best position such that we can eliminate most enemies?

To solve that, we will use *brute-force searching*

Example: Bomberman (I)

- **Brute-force searching** is a algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem’s statement.
- In short, it means finding the solution by checking all possible combinations
- Although brute-force searching is the easiest to implement, it’s also often the slowest to run
- However, it is a good starting point for small problems

Algorithm 1 Bomberman Solver

```
best_cell $\leftarrow$ None
max_kill $\leftarrow$ 0
for all cell in grid do
    if cell is empty then
        Place bomb at cell
        kill_count $\leftarrow$ Get kill count for cell
        if kill_count > max_kill then
            max_kill $\leftarrow$ kill_count
            best_cell $\leftarrow$ cell
        end if
        Remove bomb at cell
    end if
end for
```

Example: Card shuffling

- Suppose you want to write a poker game in computer
- One essential element of a poker game will be to shuffle the cards
- *How can you produce a random permutation of cards efficiently?*

Fisher-Yates shuffle

- An efficient way to do it is to use the **Fisher-Yates algorithm**
- The idea is related to how we actually shuffle cards
- Suppose we have 3 cards and we want to put them in random order
- Each time we will draw one out, and place it one after another
- We don't need to check if there are repetition, because the card drawn out is **removed**
- This insight lead to the following algorithm

Fisher-Yates shuffle

Algorithm 2 Fisher-Yates shuffle

Input: *Items* - List of length n
Output: Shuffled list of length n
Program:

```
for all $i = 0$ to $n - 2$ do
    $j \leftarrow$ Random number from $i$ to $n - 1$ (inclusive)
    Swap $Items[i]$ and $Items[j]$
end for
```

Fisher-Yates shuffle

- Let’s actually shuffle one array using the algorithm to see how it works
- Consider the following list of numbers:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

- Now because the length of the array is 4, $n = 4$.
- Also, when we first begin the loop $i = 0$ so we draw a number between $0 - 3$

Fisher-Yates shuffle

- Let's say we draw $j = 2$, then according to the algorithm we have to swap item 0 and 2, that is:

$$\begin{bmatrix} \colorbox{pink}{1} & 2 & \colorbox{lightgreen}{3} & 4 \end{bmatrix}$$
$$\downarrow$$
$$\begin{bmatrix} \colorbox{lightgreen}{3} & 2 & \colorbox{pink}{1} & 4 \end{bmatrix}$$

- This completed the first loop and $i = 1$ now

Fisher-Yates shuffle

- Now since $i = 1$, according to the algorithm we should draw *j* in between $1 - 3$ (Instead of $0 - 3$ in last run)
- So you can see that element 0 has been fixed. No later steps in the algorithm can change it's position.

$$\begin{bmatrix} \colorbox{lightgray}{3} & 2 & 1 & 4 \end{bmatrix}$$

Fisher-Yates shuffle

- Suppose now we draw $j = 3$, then according to the algorithm we have to swap $i = 1$ and $j = 3$, that is:

$$\begin{bmatrix} 3 & 2 & 1 & 4 \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} 3 & 4 & 1 & 2 \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} 3 & 4 & 1 & 2 \end{bmatrix}$$

Fisher-Yates shuffle

- Now *i* $= 2$ and *j* is a random number from $2 - 3$. Suppose this time *j* is 2. Then we need to wap element 2 with itself (i.e. not swapping)

$$\begin{bmatrix} \colorbox{lightgray}{3} & \colorbox{lightgray}{4} & \colorbox{pink}{1} & 2 \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} \colorbox{lightgray}{3} & \colorbox{lightgray}{4} & \colorbox{pink}{1} & 2 \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} \colorbox{lightgray}{3} & \colorbox{lightgray}{4} & \colorbox{lightgray}{1} & 2 \end{bmatrix}$$

Fisher-Yates shuffle

- But now $i = 4 - 2 = 2$ so the algorithm said we are done!
- This make sense because we only have one element left: If we have permuted element 1 to $n - 1$, then autoamtically the nth element will also falls in place because there is only one slot left in the queue
- So finally our shuffled list is:

$$\begin{bmatrix} \colorbox{lightgray}{3} & \colorbox{lightgray}{4} & \colorbox{lightgray}{1} & \colorbox{lightgray}{2} \end{bmatrix}$$

Python implementation

```python
# Fisher-Yates shuffle in python
# Items - List of length n
import random

for i in range(len(Items)-1):
  j = random.randint(i,len(Items)-1) # Get random number

  # Swapping
  temp = Items[i]
  Items[i] = Items[j]
  Items[j] = temp

```

Correctness and Complexity

- As one can easily see, the method does produce a randomly permuted list
- Furthermore to shuffle a list of n we need to loop $n-1$ times
- So when n is large, the number of steps we need is roughly n (i.e. $\mathcal{O}(n)$)

Permutations

- Now the previous problem also raised an interesting question: How many **permutation** are there in general?
- A permutation refers to the rearrangement of orders of a set of objects
- Order matters (Unlike combinations)
- For example, the following all the different permutations of three numbers $\{A, B, C\}$

$$\{A, B, C\}, \{A, C, B\}$$
$$\{B, A, C\}, \{B, C, A\}$$
$$\{C, A, B\}, \{C, B, A\}$$

Counting Permutations

- In general, if we have *n* distinct objects, what is the number of permutations?
- We can count this way:

1. Imagine putting the *n* objects in queue
2. Consider the 1st element in the queue. Since we *n* objects, there are *n* choice.
3. Consider the 2nd element. We are left with $n - 1$ objects, so there are $n - 1$ choice.
4. Consider the 3rd element. We are left with $n - 2$ objects, so there are $n - 2$ choices
5. $\cdots$

- One can quickly see the total number of ways is $n \times (n-1) \times (n-2) \times \cdots 3 \times 2 \times 1$

Counting Permutations

- We can also generalize this counting strategy to the case when we have *n* distinct objects, but only *r* slots
- For example, if we want to form a queue of 3 out of 5 students, we can count the following way:
  1. For the 1st position, there are 5 possibilities
  2. For the 2nd position, there are 4 possibilities
  3. For the 3rd position, there are 3 possibilities
  4. So totally we have $5 \times 4 \times 3 = 36$ ways of doing it
- One can see readily see the general formula is: $n \times (n-1) \cdots \times (n-r+2) \times (n-r+1)$

Permutations

To summarize

Theorem (Permutations of n distinct objects)

The number of ways to form a queue of length n using n distinct objects is:

$$n \times (n-1) \times (n-2) \cdots \times 2 \times 1$$

*We can also write it as $n!$ to save space. This symbol pronouced as **n factorial**.*

Permutations

Theorem (Permutations of n distinct objects in r slots)

The number of ways to form a queue of length r using n distinct objects is:

$$\begin{aligned} P_r^n &= \frac{n!}{(n-r)!} \\ &= n(n-1)\cdots(n-r+2)(n-r+1) \end{aligned}$$

*The symbol P_r^n is called the **r-permutation of n** and is often referred as **nPr***

Some exercise

Examples

1. What are the number of ways for 5 student to form queue? (Ans: $5!$)
2. 30 students from a class is invited to form a queue of 5. What is the numbers of ways to do that? (Ans: P_5^{30})
3. What are the total numbers of ways to shuffle a deck of 52 cards? (Ans: $52!$)
4. What is the number of ways to put 10 *indistinguishable* particles into two bags? (Ans: 2^{10}) (Hint: You cannot use the results previously, just count from scratch)

Linear Searching

- Now we will talk about searching
- Let’s say you are a teacher and you want to have a name list in your hand
- Now you want to write a program that helps you look up student name from their SID
- How can you do it?

Algorithm 3 Linear Search

Input:
TargetSID - Target SID
StudentInfo - List of student info in format: (*SID*,*Name*)
Output: Name of student with *TargetSID*
Program:

```
for all SID, Name in StudentInfo do
    if SID equals to TargetSID then
        Output Name and Exit
    end if
end for
Output ”No entries found”
```

```python
# Python example of Linear Search
targetSID = input('Search Query (SID): ')
studentInfo = [
  ('s192830','Chan Tai Man'),
  ('s192840','Lam Ching Yi'),
  ...
]
targetName = ""

# Linear Search
for i in range(len(studentInfo)):
  sid, name = studentInfo[i]
  if sid == targetSID:
    targetName = name
    break

# Print results
if len(targetName) == 0:
  print('No such entries is found')
else:
  print('The student you are looking for is: ', targetName)
```

Complexity

- Since we at most need to find the required entry, the worse case complexity is n (i.e. $\mathcal{O}(n)$)
- Or we are very lucky and are able to find the answer in the first trial, so the best case is 1 iteration (i.e. $\mathcal{O}(1)$)
- On average, since the element we want to search have $1/n$ probability to be in any of the positions, we will have:

$$\frac{1+2+3+\cdots+(n-1)+n}{n}=\frac{n+1}{2}$$

So when n is large it's $\approx n/2$

Maximum subarray sum

- Finally let's talk about an interesting problem
- Consider an arbitrary 1D list:

$$\begin{bmatrix} -1 & 2 & 4 & -3 & 5 & 2 & -5 & 2 \end{bmatrix}$$

- **Problem:** Can we find a subarray within the list so that the sum of the list is maximum?

Maximum subarray sum

- For example, if we have the list $[1, -3, 6]$. Then the set of all subarrays are:

$$\{[1], [-3], [6], [1, -3], [-3, 6], [1, -3, 6]\}$$

- And the corresponding subarray sums are:

$$\{1, -3, 6, 4, 3, 4\}$$

- So the maximum subarray sum is 6 and that corresponds to subarray $[6]$

Maximum subarray sum

- Another example: If the list is $[2, -1, 7]$ then the subarrays are:

$$\{[2], [-1], [7], [2, -1], [-1, 7], [2, -1, 7]\}$$

- The corresponding sums are therefore:

$$\{2, -1, 7, 1, 6, 8\}$$

- So this time the maximum subarray sum is 8 and it corresponds to subarray $[2, -1, 7]$

Maximum subarray sum

- Can you design a program that do it?
- (Hint: You can brute force the problem very easily)

Algorithm 4 Maximum subarray sum (Naïve)

Input: *n* - Length of array; *arr* - The array
Output: *best* maximum subarray sum
Program:

```
$best \leftarrow 0$
for all $i$ in $0$ to $n-1$ do
    for all $j$ in $i$ to $n-1$ do
        $sum \leftarrow 0$
        for all $k$ in $i$ to $j$ do
            $sum \leftarrow sum + arr[k]$
        end for
        $best \leftarrow \max(sum, best)$
    end for
end for
Output $best$
```

Naïve method is slow

- Note that the naive method is **very slow**
- It has 3 for loops so in general it will take $\mathcal{O}(n^3)$ to run
- We need something faster
- In fact, someone have solved it for us in $\mathcal{O}(n)$ time. The algorithm is named **Kadane's algorithm**

Interesting backstory

- The problem was proposed by Ulf Grenander in 1977 in his research in computer graphics. He derived an algorithm of $\mathcal{O}(n^2)$ himself.
- Later, when the problem is presented to Michael Shamos, another mathematician. He derived a divide and conquer algorithm of $\mathcal{O}(n \log n)$ overnight.
- Yet later, Shamos presented the problem at a seminar which was attended by Jay Kadane and he designed within a minute an $\mathcal{O}(n)$-time algorithm, which is as fast as possible.
- Now we will look at the algorithm by *Kadane*

Kadane's Algorithm

- Kadane begins by solving a similar but simpler problem: *What is the maximum subarray sum that ends in k*?
- To illustrate that, we will consider the following array:

$$\{1, -2, 5, -10, 1, 10\}$$

- Just like before, we will try to brute force all the subarrays (Side question: How many subarrays are there? What about the general case?)
- This time, however, we will group the subarrays that ends at the same elements together

Kadane’s Algorithm

Subarrays Ends at element *k*					
1	2	3	4	5	6
{1}	{1,-2}	{1,-2,5}	{1,-2,5,-10}	{1,-2,5,-10,1}	{1,-2,5,-10,1,10}
	{-2}	{-2,5}	{-2,5,-10}	{-2,5,-10,1}	{-2,5,-10,1,10}
		{5}	{5,-10}	{5,-10,1}	{5,-10,1,10}
			{-10}	{-10,1}	{-10,1,10}
				{1}	{1,10}
					{10}

The maximum subarray sum is the largest among the red subarrays ($\{1, 10\}$)

Kadane's Algorithm

- The key observation is: It is easy to obtain the maximum subarray sum that ends in $k+1$th element given that of k
- In fact, the maximum subarray can that ends in $k+1$th element must either:
  1. Begins at an element from 0 to k and ends at $k+1$; or
  2. Begins at $k+1$ and ends at $k+1$ (i.e. contain only the $k+1$th element)
- For the first case, the maximum subarray must be the maximum subarray that ends in k + the $k+1$th element
- So to find the maximum subarray sum that ends in $k+1$, we only need to compute $\max(\text{Max sum ends at } k + arr[k+1], arr[k+1])$
- Finally $bestSum = \max_{0 \le k \le n-1}\{\text{Max sum ends at k}\}$

This condense to the following, very neat looking algorithm:

Algorithm 5 Maximum subarray sum (Kadane's Algorithm)

Input: *n* - Length of array; *arr* - The array
Output: *bestSum* - maximum subarray sum
Program:

```
$prevSum, bestSum \leftarrow arr[0], arr[0]$
for all $i$ in $1$ to $n - 1$ do
    $prevSum \leftarrow \max(arr[i], prevSum + arr[i])$
    $bestSum \leftarrow \max(prevSum, bestSum)$
end for
Output $bestSum$
```

Complexity and Discussion

- As mentioned before, since we only need to loop the array once. The time complexity is $\mathcal{O}(n)$ - fastest one can get
- **Question:** Can you modify the algorithm above to compute also the beginning and ending position of the subarray with maximum sum?

List practices

Exercise: Number Search

HKOI Online Judge (01023): https://judge.hkoi.org/task/01023