CS-UY 1134: Homework 2
Summer 2019

Due July 18, 11:55 PM

Programming Part (70 Points)

Submission Instructions

Write your solution to each programming problem in a separate ".py" file. Name your files "P1.py", "P2.py", etc. There should be 5 files total. In each file, include your implementation of the given functions. Make sure the name of your function is *exactly* the same as the the name we provide in the problem. If a problem has sub-parts, include your solutions to all parts in the same file.

Submit all of your ".py" files (there should be 5 of them, named "P1.py", "P2.py", etc.) to the Gradescope assignment "Assignment 2 – Programming."

Problem 1

For this problem, we will explore a problem known as the *maximum contiguous subsequence sum* problem, and three algorithms for solving it. You will measure the actual running times of the three different algorithms. Read about the problem here: `https://en.wikipedia.org/wiki/Maximum_subarray_problem`.

In this problem, we are given a list, and we need to find a contiguous subsequence within the list which has the greatest sum. As an example, if the list is `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the contiguous subsequence with the greatest sum is `[4, -1, 2, 1]` (between indices 3 and 6 in the list), which has a sum of 6.

Provided with this homework is the file `MaxSubsequenceSum.py`. This file includes the code for three different algorithms for this maximum subsequence sum problem. It also has a Timer class which you will use for timing the code. Download this file. Try to read and understand how the three different algorithms work. The linear time $O(n)$ algorithm is a bit more clever and is known as *Kadane's Algorithm*.

Look at the `SimpleTimer` class to get an understanding of how to use it in your code. You will run and time each of the three algorithms using the following values for n: $2^7 = 128, 2^8 = 256, 2^9 = 512, 2^{10} = 1024, 2^{11} = 2048$, and $2^{12} = 4096$. For each of these values of n, do the following:

1. Create a list of n random integers between -10000 and 10000. You may wish to utilize the `random` module in python.

2. Time how long it takes for the `max_subsequence_sum1` function to find the maximum contiguous subsequence sum for this list. An example of what your code might look like is below.

```
my_timer = Timer()
# Your code to fill a list lst with n values
my_timer.reset()
max_sum, start, end = max_subsequence_sum1(lst)
runtime = my_timer.elapsed()
```

3. Time how long it takes for the `max_subsequence_sum2` function to find the maximum contiguous subsequence sum for the *same list* of n values. An example of what your code might look like is below.

4. Time how long it takes for the `max_subsequence_sum3` function to find the maximum contiguous subsequence sum for the *same list* of n values. An example of what your code might look like is below.

Your code should not take more than 20 minutes to run — it took much less on my computer.

(Note: You may write your code directly in the provided `MaxSubsequenceSum.py` file, but make sure when you submit that you rename it as "P1.py" as per the submission instructions.)

Problem 2

A *nested list* of numbers is a list whose elements are all numbers or other nested lists of numbers. This creates a sort of *hierarchy*. As an example, the list `nested_lst = [1, [2, 3], 4, [5, [6, [7, [8]]], [9], 10]]` is a nested list of numbers.

Give a **recursive** implementation of the function `def flatten_nested_list(lst, low, high)`. The function takes a nested list of numbers and two indices `low` and `high`, indicating the range of indices to be considered. The function should returns a *flattened* version of the sub-list at athe positions `low`, `low+1`, ..., `high`. That is, the function should create a new single-level (non-hierarchical) list that contains all the numbers that appear anywhere in the portion of the input list between index `low` and `high`. For example, using the list above, calling the function `flatten_nested_list(nested_lst, 0, 3)` should return the list `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`.

Problem 3

A **permutation** of a list is a rearrangement of the elements in some order. For example, `[1, 3, 2]` and `[3, 2, 1]` are both permutations of `[1, 2, 3]`.

Implement a **recursive** function `def permutations(lst, low, high)`. The function is gibven a list `lst` and two indices `low` and `high` which indicate the range of indices to be considered. The function should return a list containing all the different permutations of the elements in `lst`. Each permutation should itself be represented as a list.

For example, if `lst = [1, 2, 3]`, the call `permutations(lst, 0, 2)` could return `[[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]`.

Hint: Consider, for example, the list `lst = [1, 2, 3, 4]`. To compute the permutation list of `lst`, first try to find all the permutations of `[2, 3, 4]` and then think how you can modify this to get the permutations of `[1, 2, 3, 4]`.

Problem 4

Write a class, `class MaxStack`, that provides an implementation of the *MaxStack* ADT. A MaxStack supports the following operations:

- `MaxStack()`: The constructor initializes an empty MaxStack.
- `len(max_s)`: Returns the number of elements in `max_s`. Of course, you will need to do this by implementing the `__len__` method.
- `max_s.is_empty()`: Returns True if `max_s` does not contain any elements, and False otherwise.
- `max_s.push(elem)`: adds `elem` to the top of `max_s`.
- `max_s.top()`: Returns the element at the top of `max_s` without removing it. Raises an Exception if `max_s` is empty.

- `max_s.pop()`: Removes and returns the element at the top of `max_s`. Raises an Exception if `max_s` is empty.
- `max_s.max()`: Returns the element in `max_s` with the largest value, without removing it. Raises an Exception `max_s` is empty.

Note: You can assume that the user only inserts integers, so that elements can be compared to each other and the maximum is well-defined.

As an example, your MaxStack should exhibit the behavior below:

```
>>> max_s = MaxStack()
>>> max_s.push(3)
>>> max_s.push(1)
>>> max_s.push(6)
>>> max_s.push(4)
>>> max_s.max()
6
>>> max_s.pop()
4
>>> max_s.pop()
6
>>> max_s.max()
3
```

Implementation Requirements:

1. For the representation of `MaxStack`, your data members should be:
   - A Stack, of type `ArrayStack` (you may use the implementation from lecture, available on NYU Classes)
   - Additional $\Theta(1)$ space for additional data members, if needed.
2. Your implementation should support the `max` operation in $\Theta(1)$ worst-case time. For all other operations, the running time should be the same as in our original Stack implementation.

Hint: You may store the elements of the Stack as a tuple so you can attach with each "real" data element some additional information.

Problem 5

Give an alternate implementation of a queue by utilizing two stacks. Write your implementation in a class, `class Queue`.

Implementation Requirements:

1. For the representation of the Queue, your data members should be:
   - **Two** Stacks, of type `ArrayStack`
   - Additional $\Theta(1)$ space for additional data members, if needed
2. Any sequence of n `enqueue` and `dequeue` operations starting with an empty queue should run in worst-case time $\Theta(n)$ altogether.

Problem 6

Implement a function `def eval_postfix_boolean_exp(boolean_exp_str)` that takes a string containing a boolean expression in postfix notation and evaluates the boolean result of the expression. The input string will contain a single `'&'` to represent AND and a single `'|'` to represent OR. You can assume that all operands and operators are separated by spaces. For example, the expression `"5 2 <"` would return False while the expression `"2 5 <"` would return True. Similarly, the expression `"1 2 <6 3 <&"` would return False while `"1 2 <6 3 <|"` would return True. Your function should support the following operators:

- <: For example, `"2 5 <"` is True, while `"5 2 <"` is False.
- >: For example, `"1 3 >"` is False, while `"3 1 >"` is True.
- `&`: The boolean AND, as described above.
- `|`: The boolean OR, as described above.

You need not support any other operators in your function except these four.

Written Part (30 Points)

Submission Instructions

Answer the following questions and submit your solutions to Gradescope, under "Assignment 2 – Written." Typed answers are preferred, but you may handwrite your answers and submit a PDF image of your answers *as long as you write neatly* and your answers are readable. **You may be marked wrong if we can't read your answer.**

The Problems

1. Create a table showing the actual running times of the three algorithms from Programming problem 1. Include the running times for all of the given input sizes on all three algorithms. An example of what your table might look like:

Actual times:

Input Size	`max_subsequence_sum1` $O(n^3)$	`max_subsequence_sum2` $O(n^2)$	`max_subsequence_sum3` $O(n)$
128	0.020698	0.00056314	1.69e-05
256	0.150249	0.0025115	3.43e-05
⋮			

2. For this problem, we will estimate the running times of the three algorithms for the maximum subsequence sum problem.

   To estimate sunning times, we can utilize the asymptotic analysis. The asymptotic running time (in Big-Oh or Big-Theta) of a function tells us how quickly the running time increases as we increase the input size n.

   Suppose $T(n)$ is the running time of some algorithm on an imput of length n. If we know $T(n)$ for some n, we can use the asymptotic analysis to estimate $T(2n)$, the running time for inputs of size $2n$.

   Suppose $T(n) = \Theta(n)$. So $T(n)$ is linear. Since $T(n)$ is asymptotically linear, we can approximate it with a linear function. To do so, we will write $T(n)$ as $T(n) \approx cn$ for some constant c (though we do not know necessarily what fhe constant is). This will give a pretty rough estimate, but it can still be useful for getting an idea of how the running time increases as n increases. With this, we can approximate $T(2n)$ as follows: $T(2n) \approx c(2n) = 2(cn) \approx 2T(n)$. So we can approximate the running time of the algorithm on an input of size $2n$ as twice the running time of the algorithm on an input of size n. In symbols: $T(2n) \approx 2T(n)$. More generally, for any k, $T(kn) \approx c(kn) = k(cn) \approx kT(n)$.

Now suppose instead that $T(n)$ is not linear but quadratic. So $T(n) = \Theta(n^2)$. We can similarly approximate $T(n)$ with a simple quadratic function: $T(n) \approx cn^2$ for some c (again, we may not know c, but as we will see, we can still use this approximation). Then, we can estimate the running time of the algorithm on an input of size $2n$ as: $T(2n) \approx c(2n)^2 = 4cn^2 \approx 4T(n)$. So we can approximate the running time of the quadratic algorithm on an input of size $2n$ as *four times* the running time on an input of size n.

We can apply the same approach to approximate running times for other polynomials. For example, if $T(n) = \Theta(n^3)$, we can approximate $T(n)$ as $T(n) \approx cn^3$ for some (unknown) constant c. Then, $T(2n) \approx c(2n)^3 = 8cn^3 \approx 8T(n)$. We can use this if $T(n) = \Theta(n^4)$ as well, or if $T(n) = \Theta(n^5)$, and so on.

As a concrete example, suppose we have an algorithm with a quadratic running time, so $T(n) = \Theta(n^2)$. Suppose we have run the algorithm on an input of size 32, and timed it to find it took 125 milliseconds. We might write this as $T(32) = 125$. If we want to estimate the running time of this algorithm on an input of length 64, we can do: $T(64) \approx 4T(32) = 4(125) = 500$.

As a further example, suppose we instead have an algorithm with a cubic running time, and we have timed it on an input of size 32 to find it took 81 milliseconds. So, we might write $T(32) = 81$. Then, we can estimate the running time on an input of rize 64 as: $T(64) = T(2 \cdot 32) \approx 8T(32) = 8(81) = 648$.

Now, you will use this technique to estimate the running time of the functions `max_subsequence_sum1`, `max_subsequence_sum2`, and `max_subsequence_sum3` on larger input sizes. You should read the code and be able to confirm that the function `max_subsequence_sum1` runs in cubic time, with running time $\Theta(n^3)$; `max_subsequence_sum2` runs in quadratic time, $\Theta(n^2)$; and `max_subsequence_sum3` runs in linear time, $O(n)$.

Use the technique above to estimate the running times of each of the three algorithms, using the actual running time your computer used when running the algorithms for $n = 2^7$. Create a new chart with your estimates of the running times for $n = 2^8, 2^9, 2^{10}, 2^{11}$ and 2^{12}. For example:

Predicted times:

Input Size	`max_subsequence_sum1` $O(n^3)$	`max_subsequence_sum2` $O(n^2)$	`max_subsequence_sum3` $O(n)$
256	0.165584	0.00225256	3.38e-05
512	1.324672	0.00901024	6.76e-05
1024	...	...	...
2048	...	...	...
4096	...	...	...