Algorithms & Data Structures 2019/20
Coursework

Tom Friedetzky & Matthew Johnson

Hand in by 24 January 2020 at 2pm in DUO.

For each question, read carefully what you are required to submit. The files should be submitted together as something.zip. The top level filename doesn't matter as duo will rename it. The names of the other files are important — please follow the instructions. When submitting a python file, please include only the requested functions and remove, for example, any tests you have written while developing your answer. We will use python3 to test your submissions.

1. This question requires you to produce python code to add keys to a hash table. The approach is the same as in Question 3 of Practical 5 (see `adspractical5.pdf` and also `adspractical5solutions.py`); the crucial difference is the way that collisions are handled.

   You must create the code for two functions `hash_quartic.py` and `hash_double.py` that add keys (we ignore the values) to a hash table A of size 23 using the hash function defined by $h(k) = 4k + 7 \bmod 23$. In `hash_quartic.py` collisions should be handled with *quartic* probing. This is a method similar to quadratic probing: with quartic probing, for a key k, you iteratively try to place it in the bucket

   $$A[(h(k) + j^4) \mod 23], \text{ for } j = 0, 1, 2, \ldots$$

   until an empty bucket is found. In `hash_double.py` collisions should be handled using Double Hashing using the secondary hash function $h'(k) = 17 - (k \bmod 17)$. The input to the functions is a list of keys (positive integers) to add to the hash table and the output is the contents of the table where the symbol "$-$" indicates a bucket in the array containing no data. You can assume that the input contains no duplicates. Again, study `adspractical5.pdf` and `adspractical5solutions.py`) to see examples of the format of inputs and outputs.

   Your submission for this question should be a single file `q1.py` containing the two functions `hash_quartic` and `hash_double`. The file should not include any `import` statement. These functions will be tested on a number of inputs. Full marks will be awarded if all tests are passed. Some sample tests are available in `q1test.py`. **[20 marks]**

2. Let i be a positive number. The *child* of i is a number equal to the sum of the factorials of the digits of i. So the child of 123 is 9 and the child of 2357 is 5168 since

   $$1! + 2! + 3! = 1 + 2 + 6 = 9,$$
   $$2! + 3! + 5! + 7! = 2 + 6 + 120 + 5040 = 5168.$$

   Note that we define the factorial of 0 to be 1.

   The *descendants* of i are all the numbers that can be obtained by starting with i and repeatedly finding the child of the last number obtained. The *strength* of i is the number of distinct descendants it has. So the descendants of 4 are 24, 26, 722, 5044, 169, 363601, 1454 since each

number is the child of the one before. Note that the child of 1454 is 169 so this list is complete (if you continue to find children, you will just loop forever through 169, 363601 and 1454). Thus the strength of 4 is 7. You can check that the strength of 7 is 20, and the strength of 1 is 1.

Write a function, using python, called `descendants(n1,n2,k)` which, given three positive integers n_1, n_2 and k such that $n_1 < n_2$, returns the number of integers i, $n_1 \leq i < n_2$, that each have exactly k descendants. Save the function in a file `q2.py`. This file is all that you need to submit for this question. The file should not include any `import` statement and should not contain more than 50 lines of code (excluding blank lines and comments).

Your function will be tested on a range of inputs where each test makes a single call to `descendants(n1,n2,k)`. The test is passed if the correct value is returned. Some of the tests that will be used are available in `q2test.py`. 8 marks will be awarded if all tests are passed. This is easy: aim to make your implementation as efficient as possible so that it can cope with large input values. Up to 6 marks will be awarded for a sensible algorithmic approach. Up to 6 marks will be awarded for an efficient implementation. **[20 marks]**

3. Consider an algorithm for the problem LONGEST ODD PALINDROME. Given a string, it returns the length of the longest consecutive sequence within it that forms a palindrome of odd length. For example, on inputs *eve*, *abba*, *araara* and *banana*, it returns 3, 1, 3 and 5. See `q3test.py` for an implementation of such an algorithm — there it is called `LOP`.

   (a) Design and implement in python an algorithm `LP` that finds the longest palindrome within a string. The palindrome might have odd or even length. The algorithm should call `LOP` which should not be edited. **[5 marks]**

   (b) Design and implement in python an algorithm `LP2` that finds the longest palindrome within a string where the palindrome can wrap around from the last character back to the first. For example, for the string *abacdc*, the algorithm should return 5 since the palindrome *cabac* can be found starting from the final character. For the strings *baxyzabcdc* and *bbacdea*, the algorithm should return 7 and 4. The algorithm should not return a value greater than the length of the string — the palindrome should not wrap around to the extent that it contains the same character for a second time. The algorithm should call `LOP` or `LP`. **[5 marks]**

   Your submission for this question should be a file `q3.py`. containing the functions `LP` and `LP2`, and should not include any `import` statement. The file might be created by writing over `q3test.py`, but please rename it, and when you are done you can safely delete the original `LOP` before submitting. In developing your answer, `LOP` can be seen as a black box — you give it a string and it replies with a number. Studying `LOP` and appreciating its elegance is a rewarding task, but is not needed to answer this question.

Provide your written answers for questions 4, 5 and 6 in a single file `your_id_q456.pdf`. (Note that python files are also needed for Question 6.)

4. Prove or disprove each of the following statements. We will assume that $x > 0$, and all functions are asymptotically positive. That is, for some constant k, $f(x) > 0$ for all $x \geq k$. You will get 1 mark for correctly identifying whether the statement is True or False, and 1 mark for a correct argument.

   (a) $f(x) + g(x)$ is $o(f(x) \cdot g(x))$ **[2 marks]**

   (b) $2^x \cdot x^2$ is $o(2.1^x)$. **[2 marks]**

   (c) $x^2 \cdot \log x$ is $O(x^2)$. **[2 marks]**

   (d) $x^2 \cdot \log x$ is $O(x^3)$. **[2 marks]**

   (e) $7x^5$ is $O(12x^4 + 5x^3 + 8)$. **[2 marks]**

5. For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise state why the Master Theorem cannot be applied. You should justify your answers.

   (a) $64T(n/8) - n^2 \log n$. **[3 marks]**

   (b) $T(n) = 4T(n/2) + n/\log n$. **[3 marks]**

   (c) $2^n T(n/2) + n^n$. **[3 marks]**

   (d) $T(n) = 3T(n/4) + n \log n$. **[3 marks]**

   (e) $3T(n/3) + \sqrt{n}$. **[3 marks]**

6. Consider the generic QuickSort algorithm as we have seen in lectures, but suppose we want to change it as follows:

   - Instead of picking one pivot we pick k many (say, the k many rightmost elements of the current input). Suppose these are $p_0, \ldots, p_{k-1}$ with $p_0 < p_1 < \cdots < p_{k-1}$. Assume the input elements are pairwise distinct. The `partition()` function should then naturally partition the current input into not two but $k+1$ many parts, namely those smaller than p_0, those between p_i and p_{i+1} for $0 \leq i \leq k-2$, and those larger than p_{k-1}, and this $(k+1)$-way QuickSort should then recurse into the induced $k+1$ parts.
   - Instead of recursively calling QuickSort until the list to be sorted has length 1 (which would make it difficult to choose k many pivots if $k > 1$, we will instead use InsertionSort to sort any list of length $2k$ or less. You will of course want to sort the chosen pivots themselves before partitioning; for this please also use your implementation of InsertionSort.
   - You do not have to make this sort in-place.
   - You must at no time use any function for sorting that is not your own QuickSort or your own InsertionSort.

You should:

(a) Submit a single file called `kWayQS.py` which should accept as command line arguments the value of k and the name of a text file containing numerical input data, one integer per line, and output the sorted list containing those integers. A template file is available in DUO.

For example, if we wanted $k = 3$ pivot elements, the input file were called "`test.dat`" and contained

```
3
2
4
6
5
7
10
1
```

then if you ran

```
python3 kWayQS.py 3 test.dat
```

the output should be

```
[1, 2, 3, 4, 5, 6, 7, 10]
```

This part will, in first instance, be marked automatically using a sequence of tests. If your code passes all of those tests then you will be given full marks, otherwise partial marks subject to inspection of your code. **[20 marks]**

(b) Explain the structure of a worst-case input to this algorithm. You should assume $k = O(1)$ but in your explanation not fix it to anything in particular, that is, I'm not interested in an argument assuming that k is 2 or 3 or any other particular value. **[5 marks]**