Queens College of CUNY
Department of Computer Science
Design and Analysis of Algorithms (CSCI 323)
June 2024

Assignment #5
"Estimate, Evaluate and Rank Recurrences"
Due: June 14, 2024

Overview:

In this assignment, we create generic code for the evaluation of recurrence relations, both for its own sake and as a tool to test closed-form formulas that we may derive. To avoid the recomputation associated with recurrences, we store any intermediate results in a dictionary. (This is known in the context of Dynamic Programming as "Memoization".)

In addition, we code the "Master Theorem" used to determine the order of magnitude of recurrences associated with some divide-and-conquer algorithms.

Submissions:

In the Google form. please submit the following.

- Assignment5.py (source code)
- Assignment5.txt (console output including table)
- Assignment5.png (bar graph)

Tasks:

[0] Please see and follow the preliminary instructions in Assignment 1.

[1] Use this function to obtain the content of a function, i.e. the part after the return statement.

```
def func_body(f):
   body = inspect.getsource(f)  # gets the code
   idx = body.index("return")  # get the part after the word return
   return '"' + body[7 + idx:].strip() + '"'
```

[2] Create an empty dictionary to store intermediate results and a helper function ff to efficiently run a function f for input n:

```
dict_funcs = {}
def ff(f, n):
    func_name = f.__name__
    if func_name not in dict_funcs:
        dict_funcs[func_name] = {}
    dict_func = dict_funcs[func_name]
    if n not in dict_func:
        dict_func[n] = f(f, n)
    return dict_func[n]
```

[3] Define a sample function f1, like the one for MergeSort. Try to use the one-line if/else (aka "ternary expression") as it will make it easier to capture the function content in Task 1. Make sure that all quotients are converted to int.

```
def f1(f, n):
    return 0 if n == 1 else 2 * ff(f, int(n/2)) + n
```

[4] Test the function by calling from your main function

```
def call_and_print(func, n, desc):
    print(func.__name__, desc, "for n =", n, "is", ff(func, n))
call_and_print(f1, 256, "f(n) = 2*f(n/2) + n")
```

[5] See all the intermediate values that were also computed by executing:

```
for func in dict_funcs:
    func_name = func.__name__
    print(func_name, dict_funcs[func_name])
```

[6] Once this works, create functions f2 thru f10 corresponding to nine more recurrences from the slides or old exams.

[7] Define a function master_theorem(a, b, c) that determines the order of magnitude of the solution to a recurrence of the form $T(n) = aT(n/b) + O(n^c)$. See See https://www.geeksforgeeks.org/how-to-analyse-complexity-of-recurrence-relation/

[8] Define a function evaluate_functions to evaluate all functions in a list for a set of values of n. Use the function run_searches from an earlier assignment as a starting point, but modify/simplify it as follows:

- replace "searches" with "functions" and the loop with "for func in functions"
- sizes can be called ns and will be a set of different values of n
- instead of search.__name__, use the content of the function per Task 1.
- we only need one trial
- nothing is random here - whatever we compute is what it is
- there is no need to time anything the value stored in the dictionary is the value of the function itself
- we don't need to check any indexes because we aren't searching

[9] Invoke the code from an earlier assignment to print the table.

[10] Invoke the code from an earlier assignment to plot the graph.

Instead of just using labels f1, f2, etc., per Task 1, include what the function is computing

[11] Because the range of inputs and outputs varies so greatly, you may be better off - for the purposes of the table and graph - by taking the log of the input and log of the output. See https://en.wikipedia.org/wiki/Log%E2%80%93log_plot