Due: 16 August 2024

There are three problems listed below. The theory needed to solve each problem will be discussed at least a week before its due date, and often even earlier.

If you are stuck or confused by any of the problems, please post on Piazza. You may also ask your tutor or lecturer. You are allowed to discuss the problems with your peers and refer to online materials, but you are not allowed to share solutions or copy materials from any source. You may find the academic integrity rules at `https://academicintegrity.cs.auckland.ac.nz/`.

To get full marks you need to **show all working** unless a question explicitly states not to. You should explain your reasoning with a level of detail that is enough to convince another student taking this paper.

For each written question, you should submit via Canvas a PDF file containing your answer. A typed solution is preferred, but a scanned handwritten submission is acceptable if it is neatly written (if it's hard to read, it will not be marked). If typing the assignment, do the best you can with mathematical symbols. For exponents, write something like `2^n` if using plain text. Use LaTeX if you really want it to look good.

Answers to programming questions must be submitted via the automated marker system at `https://www.automarker.cs.auckland.ac.nz/student.php`.

Please try to submit your assignments no later than 5 min before the due time. Late submissions up to 3 days after the deadline will be accepted but will incur a 20% penalty. No further extensions will be granted.

Best of luck, and enjoy the problems!

Problem 1 (12 marks)**.** Consider the following sorting algorithm:

```
a ← list of n elements
Function inssort(a):
    for i ← 2 to n do
        j ← i − 1
        while j ≥ 1 and a[i] < a[j] do
            j ← j − 1
        j ← j + 1
        if j < i then
            t ← erase(a, i)
            insert(a, j, t)
```

a) Design auxiliary functions `insert` and `erase` so that the resulting algorithm is correct. Your functions may add and remove elements to and from the **end** of the list, but not elsewhere. Your functions may copy elements one by one as needed. As a guideline, your functions should behave similarly to the Python `list.pop(index)` and `list.insert(index, element)` functions.

b) Calculate the number of memory writes that `inssort` makes in the best case. We count an operation of the form $a[i] \leftarrow 1$ as a memory write, but not an operation of the form $i \leftarrow 1$. Show an example where the best case is attained.

c) Calculate the number of memory writes that `inssort` makes in the worst case. Show an example where the worst case is attained.

d) Compare the algorithm to the standard swap-based implementation of insertion sort in terms of memory writes. Are there inputs where one algorithm is asymptotically better than the other in terms of memory writes?

Bonus (not marked): Calculate the number of memory writes in the average case.

Problem 2 (4 marks). Solve the recurrence $T(n) = T(n/2) + T(n/4) + \cdots + T(1)$; $T(1) = 1$. You should write the solution, a few sentences explaining how you guessed that solution, and a **formal** proof that the solution is correct.

Problem 3 (4 marks). Comment on the following conversation with a LLM. Which parts of the answer are correct? Which are errors? How can the errors be fixed? Justify your answers. Remember to cite any external sources of information you may use and make sure these are trustworthy.

User Consider the following algorithm:

```
f(n):
  if n == 1 : return 0
  if n%2 == 0 : return 1 + f(n//2)
  else : return 1 + f(3*n+1)
```

Please describe a recurrence corresponding to the number of recursive function calls made on input n and give its closed form.

LLM Certainly! Let's analyze the given algorithm. The function `f(n)` computes the Collatz sequence length for a positive integer `n`. It follows these rules:

1. If `n` is 1, the function returns 0.
2. If `n` is even, it recursively calls `f(n//2)` and adds 1.
3. If `n` is odd, it recursively calls `f(3*n+1)` and adds 1.

To describe the recurrence relation, let's denote the number of recursive calls made by `f(n)` as `T(n)`. We have:

- Base case: When `n = 1`, `T(1) = 0`.
- Recursive cases:
  - If `n` is even: `T(n) = 1 + T(n//2)`
  - If `n` is odd: `T(n) = 1 + T(3*n+1)`

Now, let's find the closed form for this recurrence. We'll consider the worst-case scenario, assuming that `n` is a power of 2 (i.e., `n = 2^k`).

a. For even `n`:

- We have `n = 2^k`, so `n//2 = 2^(k-1)`.
- The recurrence becomes: `T(n) = 1 + T(2^(k-1))`.
- Unrolling the recurrence, we get: `T(n) = 1 + 1 + T(2^(k-2)) = 2 + T(2^(k-2))`.
- Repeating this process, we find that `T(n) = k`.

b. For odd `n`:

- We have `n = 2^k + 1`.
- The recurrence becomes: `T(n) = 1 + T(3*(2^k + 1) + 1)`.
- Simplifying, we get: `T(n) = 1 + T(3*2^k + 4)`.
- Since `3*2^k + 4` is still an odd number, we repeat the process.
- Eventually, we reach `T(n) = k + O(1)`.

Therefore, the closed form for the number of recursive calls made by `f(n)` is:

$$T(n) = \begin{cases} k & \text{if } n \text{ is even } (n = 2^k) \\ k + O(1) & \text{if } n \text{ is odd } (n = 2^k + 1) \end{cases}$$

In both cases, the dominant term is `k`, which represents the number of bits needed to represent `n`. Thus, the overall complexity is logarithmic: $T(n) = O(\log n)$.

Last edited: 2024/08/09 13:30:03 +12'00'