Algorithms

Exercise 1: Complexity and sorting

1 Properties of algorithms

1. You have started an internship in the software engineering group at Aperture Science, where your expert knowledge on the properties of algorithms is required to kickstart a couple of new projects. Below, you will be given the constraints for each project as well as a selection of algorithms. Discuss which algorithm you think is the most suitable one for each project. As part of a compulsory test protocol, you must also justify your answers.

   (a) The Aperture Science Potato Pi is a mobile, starch-based computer whose computational power vastly outweighs its memory. For its intended purpose of accompanying test subjects through optional test protocols, an algorithm is required which can generate new recipes for cake. The following three algorithms have been proposed at the most recent bring-your-daughter-to-work day:

      i. *quick rhubarb* has a worst-case time complexity in $\mathcal{O}(n!)$ and a worst-case space complexity in $\mathcal{O}(1)$.

      ii. *merge rhubarb* has a worst-case time complexity in $\Omega(1)$ and a worst-case space complexity in $\Omega(1)$.

      iii. *shuffle rhubarb* has a worst-case time complexity in $\mathcal{O}(\log n)$ and a worst-case space complexity in $\mathcal{O}(n^3)$.

   (b) The Aperture Science Weighted Vapour Box will bring PC gaming to test chambers, but each Weighted Vapour Box will have different hardware specifications. The operating system of the Weighted Vapour Box will have to monitor a test subject's performance at all times in order to detect when motivating messages such as "There is a 5% chance that the next question will not involve Java" are required. Which of the following algorithms is the most suitable?

      i. *binary rhubarb* has a worst-case time complexity in $\Theta(n^2)$ and a worst-case space complexity in $\Theta(n)$.

      ii. *heap rhubarb* has a worst-case time complexity in $o(n^2)$ and a worst-case space complexity in $\Theta(n \log n)$.

      iii. *priority rhubarb* has a worst-case time complexity in $\mathcal{O}(2^n)$ and a worst-case space complexity in $\Omega(n)$.

   (c) The Aperture Science £500 Heavy Angry Turrets Machine is a wearable computing device which actively listens for voice commands. The CPU consumes more energy the more it is being utilised, while the built-in memory always requires a constant amount of energy.

      i. *circular rhubarb* has a best-case time complexity in $\Theta(1)$ and a best-case space complexity in $\Theta(1)$.

      ii. *selection rhubarb* has an average-case time complexity in $\mathcal{O}(n^2)$ and a worst-case space complexity in $O(n \log n)$.

      iii. *bubble rhubarb* has a worst-case time complexity in $\mathcal{O}(n^3)$ and an average-cast space complexity in $\Theta(n)$.

2. Describe each of the following functions in terms of $\mathcal{O}(g(n))$ for some $g(n)$.
   (a) $f(n) = 4 * 3 + 2$
   (b) $f(n) = 2n + 7$
   (c) $f(n) = 8n * n + n$
   (d) $f(n) = n^2 + \log n$

3. Suppose that you are writing software for a real time system. *I.e.* you are required to design algorithms which need to terminate within a set amount of time. Using asymptotic notation, what is the ideal time complexity for such algorithms?

4. Suppose that we are using a fictional, functional programming language called DNH with ML-like syntax. In this language, all functions are *pure* in the mathematical sense: given the same set of inputs, a function will always compute the same result. We define a higher-order function called *map* which applies a function to every element of a list:

$$\begin{array}{lll} \textbf{fun} \quad map\ f\ [] & = & [] \\ \quad | \quad map\ f\ (x :: xs) & = & f\ x :: map\ f\ xs \end{array}$$

   (a) A student at a *different* university is asked to apply a function g, followed by a function f to elements of a list xs. The student defines the following:

$$map\ f\ (map\ g\ xs)$$

   How many loops are there and how many iterations are there in total?

   (b) Propose a better solution. Explain why your solution is better.

   (c) (Advanced) Prove using induction on lists that your solution produces the same results.

2 Revenge of Hutton's Razor

5. The asymptotic analysis of programs usually requires us to "count instructions". In other words, we need to find functions which describe how many instructions an algorithm will require with respect to the size of its input. To demonstrate why this is tricky, recall[1] the definition of Hutton's Razor from the second OOP coursework. We will now extend the language with a multiplication operator and a single immutable variable named n. In this new form, we will think of every expression as a function of n even if n is not used.

$$e \quad = \quad e + e \mid e * e \mid v \in \mathbb{N} \mid n$$

   The evaluation function has been updated with cases for the new types of expressions. We have also given it a new parameter named x which represents the value of the n variable.

$$\begin{array}{lllll} [\![\cdot]\!] & & : & e \to \mathbb{N} \to \mathbb{N} \\ [\![v]\!] & x & = & v \\ [\![n]\!] & x & = & x \\ [\![e_0 + e_1]\!] & x & = & [\![e_0]\!]\ x + [\![e_1]\!]\ x \\ [\![e_0 * e_1]\!] & x & = & [\![e_0]\!]\ x * [\![e_1]\!]\ x \end{array}$$

[1] Or look at `http://www.cl.cam.ac.uk/~mbg28/oop-ex2.pdf`

As an example, let us evaluate $4 + (8 * n)$ using the evaluation function. Note that the evaluation function now requires two arguments: an expression to evaluate and a value for n. We arbitrarily choose 42 as the value for n:

$$
\begin{aligned}
& [\![4 + (8 * n)]\!]\ 42 \\
= \quad & \{ \quad \text{1. applying } [\![\cdot]\!] \text{ using the case for addition} \quad \} \\
& [\![4]\!]\ 42 + [\![8 * n]\!]\ 42 \\
= \quad & \{ \quad \text{2. applying } [\![\cdot]\!] \text{ using the case for values} \quad \} \\
& 4 + [\![8 * n]\!]\ 42 \\
= \quad & \{ \quad \text{3. applying } [\![\cdot]\!] \text{ using the case for multiplication} \quad \} \\
& 4 + ([\![8]\!]\ 42 * [\![n]\!]\ 42) \\
= \quad & \{ \quad \text{4. applying } [\![\cdot]\!] \text{ using the case for values} \quad \} \\
& 4 + (8 * [\![n]\!]\ 42) \\
= \quad & \{ \quad \text{5. applying } [\![\cdot]\!] \text{ using the case for variables} \quad \} \\
& 4 + (8 * 42) \\
= \quad & \{ \quad \text{6. arithmetic} \quad \} \\
& 4 + 336 \\
= \quad & \{ \quad \text{7. arithmetic} \quad \} \\
& 340
\end{aligned}
$$

For all natural numbers n, describe how many evaluation steps each of the following expressions will require in terms of $\mathcal{O}(g(n))$ for some function $g(n)$. Our previous example, $[\![4 + (8 * n)]\!]\ 42$ required 7 evaluation steps.

(a) $[\![4 + 8]\!]\ n$

(b) $[\![23 * 42]\!]\ n$

(c) $[\![n * 15]\!]\ n$

(d) $[\![n * (n * 16)]\!]\ n$

6. We now wish to compile these new forms of expressions to machine code for our abstract machine. In order to support the n variable, we make the following changes:

   - We introduce an immutable register which stores the value of n and a **VAR** instruction which pushes the value of that register onto the stack.

   To make things interesting, we won't implement multiplication as a single instruction. Instead, we will make the following changes so that multiplication can be performed using repeated addition:

   - We introduce a program counter (a register) which stores the index of the instruction we are currently executing.
   - We add a **POP** instruction which pops a value off the stack and discards it.
   - We add a **SUB** instruction which pops two values off the stack, subtracts the second value from the first and pushes the result back onto the stack.
   - We add a **JUMP** v instruction which adds v to the program counter.
   - We add a **JUMPZ** v instruction which adds v to the program counter if the value on the top of the stack is 0.

$$
\begin{array}{lcl}
C[\![\cdot]\!] & : & e \to p \\
C[\![v]\!] & = & \textbf{PUSH}\ v \\
C[\![n]\!] & = & \textbf{VAR} \\
C[\![e_0 + e_1]\!] & = & C[\![e_1]\!]; \\
 & & C[\![e_0]\!]; \\
 & & \textbf{ADD} \\
C[\![e_0 * e_1]\!] & = & \textbf{PUSH}\ 0; \\
 & & C[\![e_1]\!]; \\
 & & C[\![e_0]\!]; \\
 & & \textbf{JMPZ}\ 7; \\
 & & \textbf{PUSH}\ 1; \\
 & & \textbf{SUB}; \\
 & & \textbf{LOAD}\ -2; \\
 & & \textbf{LOAD}\ -2; \\
 & & \textbf{ADD}; \\
 & & \textbf{STORE}\ -3; \\
 & & \textbf{JMP}\ -8; \\
 & & \textbf{POP}; \\
 & & \textbf{POP};
\end{array}
$$

Figure 1: Compilation rules for our language

- We add a **LOAD** v instruction which looks up the vth value on the stack (from the top down) and pushes that value onto the stack.
- We add a **STORE** v instruction which pops a value off the stack and stores it in the vth position on the stack.

The compilation rules for our new language are shown in Figure 1. Multiplication now works as follows: first we push 0, which represents the intermediate result of the multiplication. Then we push the two operands onto the stack. We will use the second operand as a counter. If the second operand is 0, we jump 8 instructions ahead (**JUMP** 7, because the program counter will be incremented by 1 after we have executed the **JUMP** instruction) to the first of the two **POP** instructions. These instructions remove the two operands from the stack and leave us with the result. If the second operand/counter is not zero, we decrement it by one using **PUSH** 1; **SUB**. We then add the first operand to the intermediate result using **LOAD** -2; **LOAD** -2; **ADD** and overwrite the intermediate result with the resulting value using **STORE** -3.

(a) Apply $C[\![\cdot]\!]$ to each of the four expressions from the previous question and show the resulting code.

(b) Describe the worst-case time complexity of the code for each expression.

(c) Are your answers different than for the worst-case time complexities of the expressions? If so, why? Discuss the implications of your answers.

3 Sorting

7. Below is the ML code for a Quicksort-like algorithm:

```
fun  quick []        =  []
   | quick [x]       =  [x]
   | quick (a :: bs) =  let fun  partition (left, right, []) : real list  =  (quick left) @ (a :: quick right)
                              | partition (left, right, x :: xs)          =  if x ≤ a then partition (x :: left, right, xs)
                                                                             else partition (left, x :: right, xs)
                        in partition ([], [], bs) end;
```

   What is the *space* complexity of this implementation? Explain your answer using extracts from the code.

8. Now implement Quicksort in Java, but using only $\mathcal{O}(\log n)$ additional space. Explain why your solution meets this requirement.

9. The real performance of a sorting algorithm often depends not only on the input size, but also on the order of the elements. Suppose that you have just designed a new sorting algorithm which you believe is much better than any existing one. You now wish to test how it performs on various types of data. Describe the different data sets you would use.

10. Suppose that you have a function called *isSorted* which tests if a list is sorted. Do you think it would make sense to call this function first before attempting to sort anything? Explain.