Homework Complexity 1

c.pierik

December 2023

For the following exercises, you are free to hand them in either on paper (make a photo and upload) or as text file (such as a .pdf or a .py coding file, or anything that doesn't require specific proprietary software to open). You can choose on a per-exercise basis, whatever is most convenient.

0 Reminder and notation

Logarithms are the inverse of exponentiation and $\log_b(x)$ is used to answer the question "b to the power of what equals x? For example, $\log_2(8) = 3$ because $2^3 = 8$. If you need a refresher, use for example `https://mathinsight.org/logarithm_basics`

Logarithms always have this base b, which determines which 'base element' we are exponentiating. However, they frequently appear in the wild without the base explicitly mentioned. Depending on the field you are working in, the default logarithm differs:

- In *computing science*, $\log(x) = \log_2(x)$ (the 'binary logarithm');
- In *mathematics*, $\log(x) = \log_e(x)$ (the 'natural logarithm');
- In engineering and most other scientific fields, $\log(x) = \log_{10}(x)$ (the 'common logarithm').

In my lectures and homework, we will use the following convention:

- $\log(x)$ means $\log_2(x)$;
- $\ln(x)$ means $\log_e(x)$;
- the base b will be explicitly mentioned for any base other than 2 or e.

1 Exercise One

Sort the functions below according to their growth (so following $\mathcal{O}(\cdot)$). You can give the answer as a sequence of numbers (for example: $1-5-9-8-7-4-2-3-6$). Start with the slowest growing function.

9-6-1-5-3-8-7

1. n
2. $n - n^3 + 7n^5$
3. 2^n
4. n^2
5. $n\ln(n)$
6. $\sqrt{n}$
7. $n!$
8. e^n
9. $\ln(\ln(n))$

2 Exercise Two

1. 6×10^7

Suppose we have a computer which can perform 1 million ($= 10^6$) operations per second. The nine formulas below denote the running time of some algorithms (measured in number of operations) depending on the number of elements n we feed to the algorithm. Determine for each algorithm how many elements can be processed in 1 minute.

2. 7745
3. 391
4. 11
5. 3950157
6. 8649296
7. 25
8. 7
9. 1
10. 12

1. n
2. n^2
3. n^3
4. $n!$
5. $n\ln(n)$
6. $n\log(n)$
7. 2^n
8. $n\sqrt{n}$
9. n^{100}
10. 4^n

3 Exercise Three

For each pair of functions below, indicate whether $f(n) = \mathcal{O}(g(n))$ **or** $g(n) = \mathcal{O}(f(n))$ **or** both. See slide 30 of the lecture to find a reminder on the definition of the big-O notation (or, better yet, use your own alternative sources to get a better understanding).

3.1 A

1. $f(n) = \sqrt{n}$, $g(n) = \ln(n^2)$; $f(n) = g(n)$ $g(n) = \ln(n^2)$
2. $f(n) = \log(n)$, $g(n) = \ln(n)$; $f(n) = O(g(n))$ $g(n) = O(f(n))$
3. $f(n) = n$, $g(n) = \log(n)$; $f(n) \neq O(g(n))$ $g(n) = O(f(n))$
4. $f(n) = n\ln(n) + n$, $g(n) = \ln(n)$; $f(n) \neq O(g(n))$ $g(n) = O(f(n))$
5. $f(n) = 10$, $g(n) = \ln(10)$; $f(n) = O(g(n))$ $g(n) = O(f(n))$
6. $f(n) = 2^n$, $g(n) = 10n^2$; $f(n) \neq O(g(n))$ $g(n) = O(f(n))$
7. $f(n) = 2^n$, $g(n) = 3^n$. $f(n) = O(g(n))$ $g(n) \neq O(f(n))$

3.2 B

As in section A, but now also **prove it**. That is, give a c and an N to show that $f(n) \leq c \cdot g(n)$ for every $n > N$ (or the other way around, or both!). See the bottom of slide 30 of the lecture (page 62 of the pdf) for an example.

$$f(n) = \log(n^2), g(n) = \log(n)$$

$f(n) = \log(n^2) = 2\log(n)$

$c = 2$ $N = 1$

$2\log(n) \leq c \cdot \log(n)$

$2\log(n) \leq 2 \cdot \log(n)$ $n > 1$

$f(n) = O(g(n))$

$c = \frac{1}{2}$ $N = 1$

$\log(n) \leq \frac{1}{2} \cdot 2\log(n) = \log(n)$ $n > 1$

$g(n) = O(f(n))$

Exercise 4

```
In [1]: def calculate_harmonic_number(n):
            harmonic_sum = 0.0
            for k in range(1, n+1):
                harmonic_sum += 1/k
            return harmonic_sum

        calculate_harmonic_number(10)
```

```
Out[1]: 2.9289682539682538
```

Exercise 5

```
In [4]: def calculate_factorial(n):
            if n < 0:
                raise ValueError("a non-negative integer.")
            if n == 0:
                return 1
            result = 1
            for i in range(1, n + 1):
                result *= i
            return result

        calculate_factorial(5)
```

```
Out[4]: 120
```

4 Exercise Four

The n-th harmonic number H_n is the sum of the reciprocals of the first n natural numbers:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

Write a Python function that takes a (natural) number n as input and returns H_n. Hand in your program together with its asymptotic complexity and a small justification (explain why you think that that is its complexity).

5 Exercise Five

In mathematics, the factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n such as

$$\begin{cases} n! = n \times (n-1)! \\ 0! = 1 \end{cases}$$

Write a Python function that takes a (natural) number n as input and returns $n!$. Hand in your program together with its asymptotic complexity and a small justification (explain why you think that that is its complexity).

6 Exercise Six

In the Lecture, we discussed the 'Insertion Sort' sorting algorithm. You will now look at another sorting algorithm: Bubble Sort. The idea behind bubble sort is as follows: Given a list of numbers, start at the left and look at the first two elements. If they are in the wrong order, switch them. Then look at elements two and three. If they are in the wrong order, switch them. Repeat this until you have reached the end of the list. You now know one thing for sure: the largest element of the whole list is now at the very end. If you do this a second time, the second largest element will now be at the second latest place in the list. So repeat this until all elements are in the right order. The following gif visualises it nicely: `https://upload.wikimedia.org/wikipedia/commons/c/c8/Bubble-sort-example-300px.gif`. A pseudocode implementation of this algorithm can be found below.

First, try to understand the algorithm (both conceptually, and the pseudocode). Then, analyse its cost in the same way we did for the insertion sort algorithm in the lecture (slide 24, pages 44-52 of the pdf). Make clear how you get your final formula (e.g. by using numbered costs c_i for line i of the algorithm, or some other way). Hand in you analysis and speculate about its asymptotic complexity. You do **not** have to do a best-case or worst-case analysis.

Algorithm 1: Bubble Sort

```
Input: a list A
for i = 1 to A.length − 1 do
    for j = A.length down to i + 1 do
        if A[j] < A[j − 1] then
            exchange A[j] with A[j − 1]
```

$T(n) = \sum_{i=1}^{n-1} (n-i) = \frac{(n-1)n}{2} = O(n^2)$

7 Exercise Seven

Give the asymptotic time complexity (so just the big O notation, not the whole analysis as in Exercise Six) of the following five algorithms, and justify your answers:

Algorithm 2:

```
Input: n
s = 0
i = 0
while i < n do
    s = s + 1
    i = i + 1
```

$O(n)$

Algorithm 3:

```
Input: n
s = 0
i = 0
while i < n do
    j = 0
    while j < n do
        s = s + 1
        j = j + 1
    i = i + 1
```

$O(n^2)$

Algorithm 4:

Input: n
$s = 0$
$i = 0$
while $i < n$ **do**
  $j = 0$
  while $j < n \times n$ **do**
    $s = s + 1$
    $j = j + 1$
  $i = i + 1$

$O(n^3)$

Algorithm 5:

Input: n
$s = 0$
$i = 0$
while $i < n$ **do**
  $j = 0$
  while $j < i$ **do**
    $s = s + 1$
  $i = i + 1$

$\frac{n(n-1)}{2}$

$O(n(n-1)/2)$ $O(n^2/2 - n/2)$

$O(n^2)$

Algorithm 6:

Input: n
$s = 0$
$i = 0$
while $i < n$ **do** n
  $j = 0$
  while $j < i$ **do** $1+2+\cdots+(n-1)$
    $k = 0$
    while $k < j$ **do** $1+2+\cdots+(j-1)$
      $s = s + 1$
      $k = k + 1$
    $j = j + 1$
  $i = i + 1$

$\sum_{i=1}^{n-1} \sum_{j=1}^{i} j = \sum_{i=1}^{n-1} \frac{i(i+1)}{2}$

$\frac{1}{2} \sum_{i=1}^{n-1} (i^2 + i) = \frac{1}{2} \left(\sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i \right)$

$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

$\sum_{i=1}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$

$T(n) = \frac{1}{2} \left(\frac{n(n-1)(2n-1)}{6} + \frac{n(n-1)}{2} \right)$

$O(n^3)$