**1 Exercise One**

Sort the functions below according to their growth (so following O(·)). 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.

1. n

2. n-n^3+7n^5

3. 2^n

4. n^2

5. n ln(n)

6. √n

7. n!

8. e^n

9. ln(ln(n))

9 – 6 - 1 - 5 - 4 - 2- 3 - 8 – 7

**2 Exercise Two**

Suppose we have a computer which can perform 1 million (= 106) 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.1. n **106 x 60**

2. n2 **√106 x 60**

3. n3 **³√106 x 60**

4. n!

5. n ln(n) **106 x 60**

6 n log(n) **106 x 60**

7. 2n

8. n √n **³√106 x 60**

9. n100  **100√106 x 60**

10. 4n  **n ≈ log(106×60)​ /log(4) ≈ log(6)+7​ /log(4)**

**3 Exercise Three**

For each pair of functions below, indicate whether f(n) = O(g(n)) or g(n) = 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)

there is a c > 0, N > 0 such that f (n) ≤ c · g(n) for every n > N

3.1 A

1. f(n) = √n, g(n) = ln(n2); g(n) = O(f(n))

2. f(n) = log(n), g(n) = ln(n); Both

3. f(n) = n, g(n) = log(n); g(n) = O(f(n))

4. f(n) = n ln(n) + n, g(n) = ln(n); g(n) = O(f(n))

5. f(n) = 10, g(n) = ln(10); Both

6. f(n) = 2n, g(n) = 10n2 ; g(n) = O(f(n))

7. f(n) = 2n, g(n) = 3n f(n) = O(g(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) ≤ c · 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.

Eg:

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

there is a c > 0, N > 0 such that f (n) ≤ c · g(n) for every n > N

So:

n = O(n2)

n2 != O(n)

5n + 2 = O(4n + 3) = O(n)

⇒ Take c = 2, N = 1 to see that 5n + 2 ≤ c · (4n + 3) for all n > N, because 2 · (4n + 3) = 8n + 6

f(n) = log(n2) = 2log(n), g(n) = log(n)

c=3, N=1 for all n > N, we have:

f(n)<c⋅g(n), so f(n) = O (g(n))

c=1, N=1 for all n > N, we have:

g(n)<c⋅f(n), so g(n) = O (f(n))

**4 Exercise Four**

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

黑色的钟表

描述已自动生成

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

def harmonic\_number(n): c1 1

result = 0 c2 1

for i in range(1, n + 1): c3 n

result += 1 / I c4 n

return result c5 1

T(n) = 2n + 3 =O(n)

The loop runs n times, and in each iteration, it performs a constant amount of work (O(1))

Therefore, the overall time complexity is O(n) because the work done in each iterationscales linearly with the size of the input n. The dominant factor influencing the time complexity is the loop that iterates n times.

**5 Exercise Six**

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

黑色的钟表

描述已自动生成

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).def factorial(n): c1 1

result = 1 c2 1

for i in range(1, n + 1): c3 n

result \*= I c4 n

return result c5 1

T(n) = 2n + 3 =O(n)

The asymptotic complexity of this function is O(n) because the loop iterates n times.

and each iteration performs a constant amount of work.

The loop runs n times, and in each iteration, it performs a constant number of operations (multiplication and assignment), which is O(1).The overall time complexity is O(n ) because the work done in each iteration scales linearly with the size of the input n.

The dominant factor influencing the time complexity is the loop that iterates n times.

**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 ci 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 cost times

Input: a list A c1 1

for i=1 to A.length – 1 do c2 n

for j = A.length down to i + 1 do c3 ∑n1(n+1-i)

if A[j] ＜ A[j-1] then c4 ∑n1 (n+1-i)

exchange A[j] with A[j-1] c5 ∑n1 (n+1-i)·J

J = 1

T(n) = 1 + n +（1 + 1 + k）∑n1(n+1-i) = 1 + n + 3 (n+1)· n/2 = O(n^2)

J = 0

T(n) = 1 + n +（1 + 1）∑n1(n+1-i) = 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 answer：

1.

Input: n c1 1

s=0 c2 1

i=0 c3 1

while i＜n do c4 n

s=s+1 c5 n-1

i=i+1 c6 n-1

T(n) = 1+1+1+n+n-1+n-1=3n+1

Time Complexity: O(n)

2.

Input: n

s=0

i=0

while i＜n do

j=0

while j＜n do

s=s+1

i=i+1

i=i+1

T(n) = 1+1+1+n+n-1+(n-1)n+(n-1)n+(n-1)^2+n-1

Time Complexity: O(n2)

3.

Input: n

s=0

i=0

while i＜n do

j=0

while j＜n×n do

s=s+1

i=i+1

i=i+1

T(n) = 1+1+1+n+n-1+n-1+n^3+n+n^3(n-1)+ n^3(n-1) + n-1=O(n^3)

Time Complexity: O(n3)

4.

Input: n

s=0

i=0

while i＜n do

j=0

while j＜i do

s=s+1

i=i+1

T(n)=1+1+1+n+n-1+n-1+n-1+n-2+n-1=O(n)

Time Complexity: O(n)

5.

Input: n

s=0

i=0

while i＜n do

j=0

while j＜i do

k=0

while k＜j do

s=s+1

k=k+1

j=j+1

i=i+1

T(n)=1+1+1+n+1+n+∑ni=1i+∑n-1 i=0i +∑n-1i=1i·∑i=0n-1i +∑n-2 i=0i·∑n-1 i=0i +∑n-1i=0i +n

Time Complexity: O(n^4)

表格

描述已自动生成