CSE103 Introduction to Algorithms

TD3: Complexity Analysis, Recurrences

March 25th, 2021

Exercise 1: asymptotic notation

True or false?

1. $n^2 + 100n + 2 = O(n^3)$
2. $75n^3 + 17 = O(n^3)$
3. $49n^4 + n + 15 = O(n^3)$
4. $n^2 + 10n + 6 = \Theta(n^3)$
5. for every real number $\varepsilon > 0$, $\log n = O(n^\varepsilon)$
6. $2n \log^2 n = O(n^2)$
7. $3n^2 \log n = \Theta(n^2)$
8. $2^{\frac{n \log n}{2}} = O(2^n)$
9. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$
10. $f(n) = \Theta(g(n))$ implies $2^{f(n)} = 2^{\Theta(g(n))}$

Solution.

1. True: we know that $n^2 + 100n + 2 = O(n^2)$ and $n^2 \leq n^3$ for all n.
2. True: for all $n > 0$, $17 \leq 17n^3$, so $75n^3 + 17 \leq 92n^3$.
3. False: we have that, for all $n \in \mathbb{N}$, $n^4 \leq 49n^4 + n + 15$, so it is enough to show that $n^4 \neq O(n^3)$. We will show this by contradiction. Let $c > 0$ be a positive real number and suppose that there exists $m \in \mathbb{N}$ such that, for all $n \geq m$, $n^4 \leq cn^3$. Let $d = \min(m + 1, c + 1)$. By definition, $d > c$ and the above inequality applies if we substitute d for n, so we have $d^4 \leq cd^3$. Since $d > 0$, we may divide both sides by d^3 without changing the direction of the inequality, obtaining $d \leq c$, a contradiction.
4. False: it is definitely true that $n^2 + 10n + 6 = O(n^3)$, but the big Theta notation also requires $n^2 + 10n + 6 = \Omega(n^3)$, which is equivalent to $n^3 = O(n^2)$, which is false for similar reasons as point 3.
5. True: the logarithm grows strictly more slowly than any polynomial, even when the degree is close to zero:
$$\lim_{n\to\infty} \frac{\log n}{n^\varepsilon} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\varepsilon n^{\varepsilon-1}} = \lim_{n\to\infty} \frac{1}{\varepsilon n} = 0,$$
where we used l'Hôpital's rule in the first equality. The fact that the limit is 0 implies that, for all $c > 0$, there exists $m \in \mathbb{N}$ such that $\log n \leq cn^\varepsilon$. In particular, if we take $c = 1$, we have that $\log n \leq n^\varepsilon$ for n large enough, which is precisely the definition of $\log n = O(n^\varepsilon)$.
6. True: from the above point, we have in particular that, for n large enough, $\log n \leq \sqrt{n}$, hence $\sqrt{3} \log n \leq \sqrt{3n}$, hence $3 \log^2 n \leq n$, hence $3n \log^2 n \leq 3n^2$, which proves the claim.
7. False, for the same reason as point 4.
8. False: for any chosen real constant $c > 0$, for n large enough we will have $\log c < n \left(\frac{\log n}{2} - 1\right)$ (because the latter function is unbounded), from which we infer $\frac{n \log n}{2} > n + \log c$, from which, since exponentiation is monotonic, we obtain $2^{\frac{n \log n}{2}} > c2^n$.

9. False: consider $f(n) := 2n$ and $g(n) := n$. It is true that $f(n) = O(g(n))$, but $2^{f(n)} = 2^{2n} = 4^n \neq O(2^n) = O(2^{g(n)})$. Indeed, for any fixed $c > 0$, as soon as $c < 2^n$ (which will surely happen for n big enough), we have $c2^n < 2^n 2^n = 4^n$.

   Observe, however, that the statement *is* true if the big O constant is 1. For example, taking point 5, $\log n = O(n^\varepsilon)$ for all $\varepsilon > 0$ holds because $\log n \leq n^\varepsilon$ for n large enough, which implies $n \leq 2^{n^\varepsilon}$ for n large enough, which proves $n = O(2^{n^\varepsilon})$.

10. True. When asymptotic notation (big O, big Omega, big Theta…) is embedded in a formula, we must understand the definition as applied "in place" where the notation is. For example, $h(n) = 2^{\Theta(g(n))}$ means that we have both $h(n) = 2^{O(g(n))}$ and $h(n) = 2^{\Omega(g(n))}$. The former means that we have a real constant $c > 0$ and $m \in \mathbb{N}$ such that $h(n) \leq 2^{cg(n)}$ as soon as $n \geq m$ (notice the position of the constant c). The latter is the dual statement, with $\geq$ in place of $\leq$ and with possibly different constants c', m' in place of c, m.

    Let us prove the implication for the case of big O (the case of big Omega and, therefore, of big Theta is similar). The hypothesis $f(n) = O(g(n))$ means that there is $c > 0$ and $m \in \mathbb{N}$ such that, for all $n \geq m$, $f(n) \leq cg(n)$. Since exponentiation is monotonic, this implies $2^{f(n)} \leq 2^{cg(n)}$, as desired.

Exercise 2: solving recurrences

Give the best asymptotic upper bound you can to the following recurrences (where, in all cases, $T(0) = 1$):

1. $T(n) = 3\,T\left(\frac{n}{2}\right) + n^2$
2. $T(n) = 4\,T\left(\frac{n}{2}\right) + n^2$
3. $T(n) = 16\,T\left(\frac{n}{4}\right) + n$
4. $T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{2n}{5}\right) + n$
5. $T(n) = 2\,T\left(\frac{n}{2}\right) + n \log n$
6. $T(n) = 3\,T\left(\frac{n}{3}\right) + \sqrt{n}$

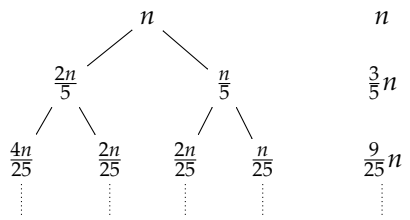
Solution.

1. We may apply the Master Theorem with $a = 3$, $b = 2$ and $c = 2$: we have $a < b^c$, so $T(n) = \Theta(n^2)$.

2. We may apply the Master Theorem with $a = 4$, $b = 2$ and $c = 2$: we have $a = b^c$, so $T(n) = \Theta(n^2 \log n)$.

3. We may apply the Master Theorem with $a = 16$, $b = 4$ and $c = 1$: we have $a > b^c$, so $T(n) = \Theta(n^{\log_4 16}) = \Theta(n^2)$.

4. Our Master Theorem does not apply, so we try with the recursion tree method. The first three levels of the recursion tree look like this:



   The sum of the first three levels gives us the idea that the general formula for level k is $\left(\frac{3}{5}\right)^k n$. Since the tree is unbalanced, we don't know how many levels there will be in total, but luckily

this is not a problem because (if our guess is correct) the costs of the levels follows a geometric pattern and we may sum over infinitely many levels, which certainly gives us an upper bound:

$$T(n) \leq \sum_{k=1}^{\infty} \left(\frac{3}{5}\right)^k n = \frac{n}{1-\frac{3}{5}} = \frac{5}{2}n.$$

We therefore make our guess that, when n is large enough, $T(n) \leq \frac{5}{2}n$. We need to find out what "large enough" means. We start by testing the first few values of $T(n)$, and compare them with the corresponding values of $f(n) := \frac{5}{2}n$, searching for some m such that $T(m) \leq f(m)$:

$$\begin{aligned} T(0) &= 1 & f(0) &= 0, \\ T(1) &= T(0) + T(0) + 1 = 3 & f(1) &= \frac{5}{2}, \\ T(2) &= T(0) + T(0) + 2 = 4 & f(2) &= 5. \end{aligned}$$

Se we are ready to try and prove by induction that, for all $n \geq 2$, $T(n) \leq \frac{5}{2}n$:

- base case ($n = 2$): already established above;
- inductive case ($n > 2$):

$$\begin{aligned} T(n) &= T\left(\frac{n}{5}\right) + T\left(\frac{2n}{5}\right) + n && \text{by definition} \\ &\leq \frac{5}{2}\frac{n}{5} + \frac{5}{2}\frac{2n}{5} + n && \text{by the induction hypothesis} \\ &= \left(\frac{1}{2} + 1 + 1\right) n = \frac{5}{2}n \end{aligned}$$

We may thus conclude that $T(n) = O(n)$. Since the root of the recursion tree is n, this is also a lower bound, hence $T(n) = \Theta(n)$.

5. We try again with the recursion tree method, because this is another recurrence to which our Master Theorem does not apply. In this case, however, the tree is very regular and we may describe it exactly:

   - it is a binary tree, because the coefficient of T in the recurrence is 2;
   - it is balanced, hence of height $\log n$, because the argument of the recursive call to T is $\frac{n}{2}$;
   - each node at a fixed level k has identical weight, given by

   $$n2^{-k}\log(n2^{-k}) = 2^{-k}n(\log n - k).$$

   Since we have a balanced binary tree, the number of nodes at level k is 2^k, so the cost at level k is

   $$2^k 2^{-k} n(\log n - k) = n(\log n - k).$$

   The value of $T(n)$ is obtained by summing level by level:

   $$\begin{aligned} T(n) &= \sum_{k=0}^{\log n - 1} n(\log n - k) = n\log^2 n - n\sum_{k=0}^{\log n - 1} k \\ &= n\log^2 n - n\frac{(\log n - 1)\log n}{2} = \frac{1}{2}(n\log^2 n + n\log n) \leq n\log^2 n. \end{aligned}$$

   We therefore have $T(n) \leq cn\log^2 n$ for any $c \geq 1$, as long as n is large enough. We are left with finding what "n large enough" means. For this, we compute the first few values of T, and compare them with the first few values of $f(n) := cn\log^2 n$:

   $$\begin{aligned} T(0) &= 1 & f(0) &= \text{undefined}, \\ T(1) &= 2T(0) + 1\log 1 = 2 & f(1) &= 0, \\ T(2) &= 2T(1) + 2\log 2 = 6 & f(2) &= 2c. \end{aligned}$$

If we take $c := 3$, we have $T(2) = f(2)$. We take this to be the base case of our inductive proof that, for all $n \geq 2$, $T(n) \leq 3n \log^2 n$. The inductive case is, for $n > 2$:

$$\begin{aligned}
T(n) &= 2\,T\left(\frac{n}{2}\right) + n \log n && \text{by definition} \\
&\leq 2 \cdot 3\frac{n}{2}\left(\log \frac{n}{2}\right)^2 + n \log n && \text{by the induction hypothesis} \\
&= 3n(\log n - 1)^2 + n \log n \\
&= 3n \log^2 n - 6n \log n + 3n + n \log n \\
&= 3n \log^2 n - 5n \log n + 3n \\
&< 3n \log^2 n - 5n \log n + 3n \log n && \text{because } n > 2, \text{ hence } \log n > 1 \\
&= 3n \log^2 n - 2n \log n \\
&< 3n \log^2 n && \text{because } n > 2, \text{ hence } n \log n > 0.
\end{aligned}$$

We have thus established that $T(n) = O(n \log^2 n)$. In this case, the only immediate lower bound we obtain by taking the root of the recursion tree is $\Omega(n \log n)$. In fact, it is possible to prove (using a more general Master Theorem) that we do have $T(n) = \Theta(n \log^2 n)$, but this is out of the scope of CSE 103.

6. The Master Theorem applies here, with $a = 3$, $b = 3$ and $c = \frac{1}{2}$: since $a > b^c$, we have $T(n) = \Theta(n^{\log_3 3}) = \Theta(n)$.

Exercise 3: superpolynomial and subexponential

Can you find a function $f(n)$ which is, at the same time:

- *superpolynomial*: $f(n) \neq O(n^k)$ for all $k \in \mathbb{N}$;
- *subexponential*: $f(n) = O(2^{n^\varepsilon})$ for every real number $\varepsilon > 0$?

Solution. The function $f(n) = n^{\log n}$ is a possible answer:

- it is superpolynomial: obviously $\log n > k$ for any constant $k \in \mathbb{N}$ and for sufficiently big n, so $n^{\log n} > n^k$, which means that $n^{\log n} \neq O(n^k)$;
- it is subexponential: notice that $n^{\log n} = 2^{\log^2 n}$; now, let $\varepsilon > 0$ and recall point 5 of the above exercise: for n sufficiently large, $\log n \leq n^{\frac{\varepsilon}{2}}$ (the big O constant is 1), hence $\log^2 n \leq n^\varepsilon$, hence $2^{\log^2 n} \leq 2^{n^\varepsilon}$.

Exercise 4: complexity analysis of a few simple functions

In the sequel, we will adopt the following cost model:

- comparisons, assignments and swaps have constant cost;
- arithmetic operations on integers have constant cost;
- the instruction `l = [0] * n` (creation of a list of length `n` filled with zeros) has constant cost;
- the `len` function and the `append` list method have constant cost.

Give the asymptotic complexity of the following Python functions:

```
def digadd(c,d,e):
    """
    Parameters
    ----------
    c : int
```

```
        carry bit
    d : int
        digit
    e : int
        digit.

    Returns
    -------
    A pair (c',a) where a is the digit resulting from
    the sum of digits d,e and the carry bit c,
    and c' is the resulting carry bit
    """
    if c < 0 or d < 0 or e < 0 or c > 1 or d > 9 or e > 9:
        return -1,-1
    a = c + d + e
    return a // 10, a % 10

def rev(l):
    """
    Parameters
    ----------
    l : list.

    Returns
    -------
    The list l reversed in place.
    """
    n = len(l)
    for i in range(n // 2):
        l[i],l[n-i-1] = l[n-i-1],l[i]

def shift(l,k):
    """
    Parameters
    ----------
    l : list
    k : int

    Returns
    -------
    The list l with k zeros appended.
    """
    for _ in range(k):
        l.append(0)
    return l
```

Solution.

- The complexity of `digadd` is $O(1)$;
- the complexity of `rev(l)` is $O(n)$, where n is the length of l;
- the complexity of `shift(l,k)` is $O(k)$.

Exercise 5: a mystery function

Instead of using the built-in unbounded integers provided by Python, let us implement unbounded integers as lists of digits, in base 10, with the most significant digit at index 0. That is, the number `1789` will be represented by the list `[1, 7, 8, 9]`. We will restrict to non-negative integers, so there will be no need to represent signs.

Consider the following Python function:

```
def mystery(ml,nl):
    """
    Parameters
    ----------
    ml,nl : list
    two non-negative integers represented as lists of digits

    Returns
    -------
    ???
    """
    h = len(ml) - len(nl)
    if h > 0:
        rev(nl)
        shift(nl,h)
        rev(nl)
    elif h < 0:
        rev(ml)
        shift(ml,-h)
        rev(ml)
    n = len(ml)
    l = [0] * n
    c = 0
    for i in range(n):
        (c,l[n-i-1]) = digadd(c,ml[n-i-1],nl[n-i-1])
    return l
```

What is the asymptotic complexity of `mystery` as a function of the lengths of `ml` and `nl`? Can you tell what it does? (Try and answer just by looking at the code. If you can't figure it out after a while, run it on an example, you'll understand right away).

Solution. The function implements the elementary school algorithm for adding two non-negative integers. If n is the maximum length of `ml`,`nl`, its asymptotic complexity is $O(n)$: the value of `h` is bounded bu n, so the calls to `rev` take $O(n)$, then there is a `for` whose body has complexity $O(1)$ which is executed precisely n times. This loop corresponds to adding the digits of `ml`,`nl` starting from the least significant, that is, "from right to left", carrying a 1 when the sum of the digits exceeds 9.

Exercise 6: one more mystery function

Consider the following Python definitions:

```
def digmul(c,d,e):
    """
    Parameters
    ----------
    c : int
        carry digit
```

```
    d : int
        digit
    e : int
        digit.

    Returns
    -------
    A pair (c',a) where a is the digit resulting d*e+c
    and c' is the carry digit in case the result exceeds 9
    """
    if c < 0 or d < 0 or e < 0 or c > 8 or d > 9 or e > 9:
        return -1,-1
    a = c + d * e
    return a // 10, a % 10

def aux(nl,d):
    """
    Parameters
    ----------
    nl : list
    a non-negative integer represented as list of digits
    d : int
    a digit.

    Returns
    -------
    ???
    """
    if d < 0 or d > 9:
        return -1
    c = 0
    l = []
    n = len(nl)
    for i in range(n):
        (c,a) = digmul(c,d,nl[n-i-1])
        l.append(a)
    l.append(c)
    rev(l)
    return l

def mystery2(ml,nl):
    """
    Parameters
    ----------
    ml,nl : list
    two non-negative integers represented as lists of digits

    Returns
    -------
    ???
    """
    ll = []
    n = len(nl)
```

```
    for i in range(n):
        ll.append(aux(ml,nl[n-i-1]))
        shift(ll[i],i)
    r = [0]
    for i in range(len(ll)):
        r = mystery(r,ll[i])
    return r
```

where `mystery` is the function from the above exercise.

What is the asymptotic complexity of `mystery2` as a function of the lengths of `ml` and `nl`? Can you figure out what it does?

Solution. The function implements the elementary school algorithm for multiplying two numbers. The function `aux`, used as a subroutine, is the special case of multiplying a number by a single digit. If m and n are the lengths of `ml` and `nl`, respectively, the asymptotic complexity of `mystery2` is $O(mn)$.

Exercise 7: Karatsuba multiplication

During a lecture given in Moscow some time in 1960, the great mathematician Andrey Kolmogorov conjectured a $\Omega(n^2)$ lower bound on integer multiplication, where n is the maximum of the number of digits of the two integers being multiplied. About a week later, a 23-year-old student named Anatoly Karatsuba went up to Kolmogorov and showed him the following divide and conquer algorithm:

```
def mul(k, m):
    n = max(number of digits of k,number of digits of m)
    if n == 1:
        return k * m
    h = n // 2
    k1 = k // 10**h
    k2 = k % 10**h
    m1 = m // 10**h
    m2 = m % 10**h
    a = mul(k1, m1)
    c = mul(k2, m2)
    b = mul(k1 + k2, m1 + m2) - a - c
    return a * 10**(h*2) + b * 10**h + c
```

For analyzing the complexity of this algorithm, consider the following cost model:[1]

- assignments, any operation on bounded-size integers (variables in green): cost $O(1)$;
- getting the number of digits of an arbitrary integer: cost $O(1)$;
- multiplying or dividing an arbitrary integer by 10^n, or taking modulo 10^n: cost $O(n)$;
- adding or subtracting two n-digit numbers: $O(n)$
- multiplying two 1-digit numbers: $O(1)$

Assuming that his algorithm actually computes `k*m`, did Karatsuba disprove Kolmogorov's conjecture? Can you *prove* that Karatsuba's algorithm really works, that is, that it multiplies `k` and `m`? (If you don't see at all what's going on, take a look at the next exercise).

[1] Observe that this cost model is realistic: CPUs natively support bounded-size integers (for example, 64-bit integers), on which any operation is constant time. As in the previous exercises, unbounded integers may be implemented as lists of digits in base 10, so digits are bounded-size integers and multiplying two digits has constant cost. The length of such lists is also a bounded-size integer (otherwise the definition would be circular), and extracting it is a constant-time operation. Multiplying, dividing by 10^n or taking modulo 10^n is just shifting a list, so it costs $O(n)$. Adding or subtracting two n-digit numbers represented as lists may be done using the elementary school addition algorithm, which costs $O(n)$.

Solution. Let us call $T(n)$ the complexity of Karatsuba's algorithm, where n is the maximum number of digits of the inputs. Here's a version of the algorithm annotated with costs:

```
def mul(k, m):
    n = max(number of digits of k,number of digits of m)   # O(1)
    if n == 1:   # O(1)
        return k * m   # O(1)
    h = n // 2   # O(1)
    k1 = k // 10**h   # O(n)
    k2 = k % 10**h   # O(n)
    m1 = m // 10**h   # O(n)
    m2 = m % 10**h   # O(n)
    a = mul(k1, m1)   # O(T(n/2))
    c = mul(k2, m2)   # O(T(n/2))
    b = mul(k1 + k2, m1 + m2) - a - c   # O(T(n/2)) + O(n)
    return a * 10**(h*2) + b * 10**h + c   # O(n)
```

Notice that, when costs are linear, we round up $n/2$ to n, because it does not make a difference asymptotically. On the other hand, the $n/2$ inside the cost of the recursive calls is crucial (it's what makes the algorithm be divide and conquer). Overall, we see that the complexity of Karatsuba's algorithm obeys the recurrence

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n).$$

Therefore, a simple application of the Master Theorem (with $a = 3$, $b = 2$ and $c = 1$) gives us $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$, which is *not* $\Omega(n^2)$ and therefore disproves Kolmogorov's conjecture. In particular, Karatsuba's algorithm is asymptotically better than the elementary school algorithm. Being a recursive algorithm, it has a bigger constant though, so in practice it becomes faster only for numbers with more than $\sim$300 digits.

Let us prove the correctness of Karatsuba's algorithm, that is, that it actually computes multiplication. To make notations lighter, let us call $K(k, m)$ the function defined by Karatsuba's algorithm. Let $n > 0$ be the maximum number of digits of k and m. We will prove that $K(k, m) = k \cdot m$ by induction on n:

- $n = 1$: immediate (we enter the `if` and return $k \cdot m$);
- $n > 1$: by definition, the algorithm computes four numbers k_1, k_2, m_1, m_2 such that

  $$k = k_1 10^{\lfloor \frac{n}{2} \rfloor} + k_2,$$
  $$m = m_1 10^{\lfloor \frac{n}{2} \rfloor} + m_2.$$

  Notice that k_1 and m_1 have at most $\lceil \frac{n}{2} \rceil$ digits, whereas k_2 and m_2 have at most $\lfloor \frac{n}{2} \rfloor$ digits. The algorithm then returns the number

  $$K(k_1, m_1)10^{2\lfloor \frac{n}{2} \rfloor} + (K(k_1 + k_2, m_1 + m_2) - K(k_1, m_1) - K(k_2, m_2))10^{\lfloor \frac{n}{2} \rfloor} + K(k_2, m_2).$$

  Since K is applied to numbers with strictly less than n digits, we may apply the induction hypothesis and obtain that the above number is equal to

  $$\begin{aligned} & k_1 m_1 10^{2\lfloor \frac{n}{2} \rfloor} + ((k_1 + k_2)(m_1 + m_2) - k_1 m_1 - k_2 m_2)10^{\lfloor \frac{n}{2} \rfloor} + k_2 m_2 \\ = \; & k_1 m_1 10^{2\lfloor \frac{n}{2} \rfloor} + (k_1 m_2 + k_2 m_1)10^{\lfloor \frac{n}{2} \rfloor} + k_2 m_2 \\ = \; & (k_1 10^{\lfloor \frac{n}{2} \rfloor} + k_2)(m_1 10^{\lfloor \frac{n}{2} \rfloor} + m_2) = km, \end{aligned}$$

  as desired.

Exercise 8: not as clever as Anatoly

Karatsuba's algorithm is based on the observation that, when written in base 10, any $2n$-digit non-negative integer k may be written as

$$k = k_1 10^n + k_2,$$

with k_1 and k_2 two n-digit numbers. This corresponds to "splitting down the middle" the decimal expansion of k: for example, $1789 = 17 \cdot 10^2 + 89$. When two $2n$-digit numbers k and m are written in this way, their product becomes

$$k\,m = (k_1 10^n + k_2)(m_1 10^n + m_2) = k_1 m_1 10^{2n} + (k_1 m_2 + k_2 m_1)10^n + k_2 m_2.$$

This suggests a divide and conquer algorithm: recursively compute the products $k_1 m_1$, $k_1 m_2$, $k_2 m_1$ and $k_2 m_2$ and then compute the above sum to get the result. However, if you inspect Karatsuba's algorithm (as given in the previous exercise), you will see that he computes the 10^n factor in a strange way:

$$k_1 m_2 + k_2 m_1 = (k_1 + k_2)(m_1 + m_2) - k_1 m_1 - k_2 m_2.$$

Can you explain why he did that? To help you see it, suppose we implemented the divide and conquer algorithm as suggested above, *without* Karatsuba's twist. What asymptotic complexity would we get?

Solution. The divide and conquer algorithm described in the exercise works just as Karatsuba's, except that it recursively computes *four* products, that is, the function implementing it would contain four calls to itself, as opposed to the three of Karatsuba's algorithm. Its complexity $T'(n)$ therefore obeys the recurrence

$$T'(n) = 4T'\left(\frac{n}{2}\right) + O(n).$$

An immediate application of the Master Theorem (with $a = 4$, $b = 2$ and $c = 1$) gives us $T'(n) = O(n^{\log_2 4}) = O(n^2)$. So the asymptotic complexity of this algorithm is the same as that of the elementary school algorithm. In fact, since it is recursive, it would result in *worse* running times in practice.

Exercise 9: naive matrix multiplication

We may represent an $n \times n$ matrix (of any numeric type: integers, real numbers, it will not be relevant here) as a list containing n lists of length n. The rows of the matrix will correspond to the inner lists; for example

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{is represented by} \quad \texttt{[[1,2,3],[4,5,6],[7,8,9]]}.$$

The following function computes the product of two $n \times n$ matrices `M` and `N`:

```
def matmul(M,N):
    # we assume that M and N are square matrices of identical size
    n = len(M)
    R = [0] * n  # create the rows of the result
    for i in range(n):
        R[i] = [0] * n  # initialize the entries of the result to 0
    for i in range(n):
        for j in range(n):
            # R_ij = (i-th row of M) scalar product (j-th col of N)
            for k in range(n):
                R[i][j] += M[i][k] * N[k][j]
    return R
```

Under the usual cost model (assignments, creation of lists and arithmetic operations all have cost $O(1)$), what is the asymptotic complexity of `matmul`?

Solution. The asymptotic complexity of naive matrix multiplication is $O(n^3)$, as is clear from the three nested `for` loops bounded by n.

Exercise 10: divide and conquer for matrix multiplication

What if we applied Karatsuba's idea to matrix multiplication? Any $2n \times 2n$ matrix A may be written as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where A_{ij} are $n \times n$ matrices. It is easy to see that matrix multiplication works "blockwise", that is

$$A \cdot B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}.$$

Therefore, we may compute the product of two $2n \times 2n$ matrices by recursively computing *eight* products of $n \times n$ matrices plus four additions (still of $n \times n$ matrices). Since addition of $n \times n$ matrices costs $O(n^2)$ (it is linear in the number of elements of the matrix), the complexity of the divide and conquer algorithm suggested above obeys the recurrence

$$T(n) = 8\,T\left(\frac{n}{2}\right) + O(n^2).$$

How does it compare asymptotically to the naive algorithm of the previous exercise?

In 1968, Volker Strassen found a "Karatsuba-style" optimization (the details are non-trivial, we will not give them here) allowing to use only *seven* multiplications. What is the complexity of Strassen's algorithm?

Solution. An immediate application of the Master Theorem to the divide and conquer algorithm (with $a = 8$, $b = 2$ and $c = 2$) gives $T(n) = O(n^3)$, so there is no improvement over the naive algorithm (in fact, as in the case of integer multiplication, the divide and conquer algorithm will have worse performance because of the recursive calls).

By contrast, since there are only seven recursive calls, the complexity $S(n)$ of Strassen's algorithm obeys the recurrence

$$S(n) = 7\,S\left(\frac{n}{2}\right) + O(n^2).$$

The Master Theorem (with $a = 7$, $b = 2$ and $c = 2$) now gives us $S(n) = O(n^{\log_2 7}) = O(n^{2.8074})$, which, on large matrices, offers a significant improvement with respect to $O(n^3)$.

Over the ensuing decades, people have regularly improved Strassen's upper bound to matrix multiplication, finding more and more sophisticated algorithms with better and better asymptotic complexity. The best algorithm known so far, found by Josh Alman and Virginia Vassilevska-Williams in 2020, has complexity $O(n^{2.3729})$, but it is only of theoretical interest: the constant hidden in the big O notation is so large that it is unusable in practice.