CSC-1S003-EP Introduction to Algorithms

TD2: Complexity Analysis, Recurrences

February 13th, 2025

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 $2^{f(n)} = 2^{\Theta(g(n))}$

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. $T(n) = 3\,T\left(\frac{n}{3}\right) + \sqrt{n}$
7. $T(n) = 2\,T\left(\frac{n}{4}\right) + n^{0.51}$
8. $T(n) = 4\,T\left(\frac{n}{2}\right) + 7n$
9. $T(n) = 3\,T\left(\frac{n}{4}\right) + n \log n$
10. $T(n) = 3\,T\left(\frac{n}{3}\right) + \frac{n}{2}$
11. $T(n) = 7\,T\left(\frac{n}{3}\right) + n^2$
12. $T(n) = 6\,T\left(\frac{n}{3}\right) + n^2 \log n$
13. $T(n) = 5\,T\left(\frac{n}{2}\right) + 10n + 3n^2$
14. $T(n) = 4\,T\left(\sqrt{n}\right) + \log^2 n$
15. $T(n) = 2\,T\left(\frac{n}{8}\right) + \frac{1}{n+1} + \sqrt[3]{n}$
16. $T(n) = 65\,T(n-3) + 4^n$

Exercise 3: complexity analysis of (nested) loops

Give the asymptotic complexity of the following Python definitions of `f(n)`, taking `n` itself as the input size (all comparisons, arithmetic operations and `math.sqrt` are assumed to cost $O(1)$):

1.
```
def f(n):
    for i in range(n):
        j = i
        while j > 0:
            # some instructions of cost O(1)
```

```
            j -= 1
    for j in range(n):
        # some instructions of cost O(1)
```

2.
```
import math
def f(n):
    for i in range(int(math.sqrt(n))//2):
        # some instructions of cost O(1)
    j = 0
    while j < int(math.sqrt(n))//4:
        # some instructions of cost O(1)
        j += 1
    for k in range(j+7):
        # some instructions of cost O(1)
```

3.
```
import math
def f(n):
    for i in range(int(math.sqrt(n))//2):
        # some instructions of cost O(1)
        for j in range(int(math.sqrt(n))//4):
            # some instructions of cost O(1)
            for k in range(j, j+7):
                # some instructions of cost O(1)
```

4.
```
import math
def f(n):
    for i in range(int(math.sqrt(n))//2):
        # some instructions of cost O(1)
        for j in range(i, i+7):
            # some instructions of cost O(1)
            for k in range(j, j+7):
                # some instructions of cost O(1)
```

5.
```
import math
def f(n):
    h = int(math.sqrt(n)) // 2
    for i in range(h):
        # some instructions of cost O(1)
        for j in range(i * i):
            # some instructions of cost O(1)
            for k in range(1, j):
                if j % h == 0:
                    # some instructions of cost O(1)
```

6.
```
import math
def f(n):
    h = int(math.sqrt(n)) // 2
    for i in range(h):
        # some instructions of cost O(1)
        for j in range(i * i):
            # some instructions of cost O(1)
            for k in range(1, j):
```

```
                    if j % h != 0:
                        # some instructions of cost O(1)
```

7.
```
import math
def f(n):
    h = int(math.sqrt(n)) // 2
    for i in range(h):
        # some instructions of cost O(1)
        for j in range(i * i):
            # some instructions of cost O(1)
            if j % h == 0:
                for k in range(1, j):
                    # some instructions of cost O(1)
```

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
```

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

```
    if c > 0:
        l.append(1)
    rev(ml)
    rev(nl)
    rev(l)
    return l
```

What is the asymptotic complexity of `mystery` as a function of the maximum 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).

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 from 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 a 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?

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. That is, he believed that no general multiplication algorithm existed with asymptotic complexity better than $\Theta(n^2)$. 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 $\Theta(1)$;
- getting the number of digits of an arbitrary integer: cost $\Theta(1)$;
- multiplying or dividing an arbitrary integer by 10^n, or taking modulo 10^n: cost $\Theta(n)$;
- adding or subtracting two n-digit numbers: $\Theta(n)$
- multiplying two 1-digit numbers: $\Theta(1)$

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).

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:

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

[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 $O(n)$.

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?

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 $\Theta(1)$), what is the asymptotic complexity of `matmul`?

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 $\Theta(n^2)$ (it is linear in the number of elements of the matrix), the complexity of the divide and conquer algorithm suggested above obeys the recurrence

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

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?

Exercise 11: superpolynomial and subexponential

Can you find a *single* 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 $\varepsilon > 0$?