Algorithms and Complexity. Exercise session 1

Algorithm analysis

Order Compare the asymptotic order of growth of the following pairs of functions. In each case tell if $f(n) \in \Theta(g(n))$, $f(n) \in O(g(n))$ or $f(n) \in \Omega(g(n))$.

	$f(n)$	$g(n)$
a)	$100n + \log n$	$n + (\log n)^2$
b)	$\log n$	$\log n^2$
c)	$\frac{n^2}{\log n}$	$n(\log n)^2$
d)	$(\log n)^{\log n}$	$\frac{n}{\log n}$
e)	$\sqrt{n}$	$(\log n)^5$
f)	$n2^n$	3^n
g)	$2^{\sqrt{\log n}}$	$\sqrt{n}$

Solution to Order

a) Compute the limit for the ratio of functions.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{100n + \log n}{n + (\log n)^2} = \lim_{n\to\infty} \frac{100 + (\log n)/n}{1 + (\log n)^2/n} = 100.$$

Since the result is a constant we conclude that $f(n) \in \Theta(g(n))$.

b) It is enough to note that $\log n^2 = 2 \log n$, hence $\log n \in \Theta(\log n^2)$.

c)

$$\frac{g(n)}{f(n)} = \frac{n(\log n)^2}{n^2/\log n} = \frac{(\log n)^3}{n} \to 0 \text{ for } n \to \infty.$$

Therefore, $g(n) \in O(f(n))$ and $f(n) \in \Omega(g(n))$.

d) Substitute $m = \log n$ and compare $f_2(m) = m^m$ with $g_2(m) = 2^m/m$:

$$\frac{g_2(m)}{f_2(m)} = \frac{2^m}{m \cdot m^m} = \frac{1}{m}\left(\frac{2}{m}\right)^m \to 0$$

for $n \to \infty$. Therefore $f(n) \in \Omega(g(n))$ for $g(n) \in O(f(n))$.

e) Polynomial functions grow faster than poly-logarithmic functions. Therefore $f(n) \in \Omega(g(n))$ and $g(n) \in O(f(n))$.

Formally, we can prove the following lemma which says that for all constants $c > 0$, $a > 1$ and for all monotonically increasing functions $h(n)$ it holds $(h(n))^c \in O(a^{h(n)})$. If we set $h(n) = \log n$, $c = 5$ and $a = \sqrt{2}$ we have $(\log n)^5 \in O(\sqrt{2}^{\log n}) = O(\sqrt{n})$ as $\left(2^{1/2}\right)^{\log n} = \left(2^{\log n}\right)^{1/2} = n^{1/2}$.

f) $\frac{f(n)}{g(n)} = \frac{n2^n}{3^n} = n\left(\frac{2}{3}\right)^n$. Exponential functions, as $(2/3)^n$, grow faster than polynomial functions (as n). This can be proved by using Hôpital's rule. First we rewrite the expression as: $n\left(\frac{2}{3}\right)^n = n/\left(\frac{3}{2}\right)^n$. Let the numerator and the denominator be, respectively, $r(n) = n$ and $s(n) = \left(\frac{3}{2}\right)^n$. Then, compute the first derivative

$$r'(n) = 1, \qquad s'(n) = \left(\frac{3}{2}\right)^n \ln\left(\frac{3}{2}\right) = \frac{\ln(3/2)}{\left(\frac{2}{3}\right)^n}$$

We can use Hôpital's rule because $r(n) \to \infty$, $s(n) \to \infty$ and $s'(n) \neq 0$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{r(n)}{s(n)} = \lim_{n\to\infty} \frac{r'(n)}{s'(n)} = \frac{\left(\frac{2}{3}\right)^n}{\ln\left(\frac{3}{2}\right)} \to 0$$

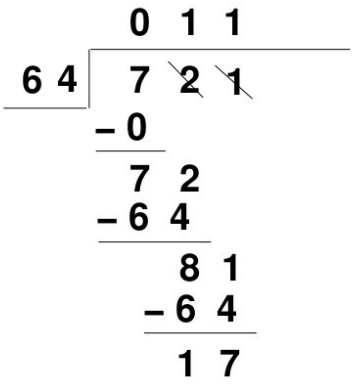
so we conclude that $f(n) \in O(g(n))$.

g) $\sqrt{n} \in \Omega(2^{\sqrt{\log n}})$. It it worth noting that $\sqrt{n} = 2^{\frac{1}{2}\log n}$ and $\lim_{n\to\infty} 2^{\sqrt{\log n} - \frac{1}{2}\log n} = 0$.

□

Division Analyze the schoolbook algorithm for division (stair division). Assume that each bit operation takes unit cost.

Solution to Division
This is the textbook method when 721 is divided by 64:



The algorithm starts by checking how many times 64 goes in 7, which is the most significant digit of the 721. Since $64 > 7$ it goes 0 times. We have $0 \times 64 = 0$, so we subtract 0 from 7. Now we multiply 7 by 10 (the base), move down the next digit of 721, namely 2, and divide 72 by 64 to get the next number in the quotient. We continue in this way, subtract the number of times 64 goes into each two-digit number, move down the next digit, and finish when all of the integer ratio is calculated. The number 17 at the bottom of the division is the reminder.

We can formulate this procedure in the following algorithm which computes $q = b/a$ where a, b are n-bit numbers ($a = a_{n-1}a_{n-2}\cdots a_0$, $b = b_{n-1}b_{n-2}\cdots b_0$) in base B. Let $x \ll y$ be x shifted y steps left.

In order to make it easier to prove the algorithm correctness we make use of pre conditions, post conditions and invariants.

```
Div(a, b, n) =
    PRE= a > 0, b ≥ 0, a and b represented by n bits each.
    POST= qa + r = b, 0 ≤ r < a
    r ← 0
    for i ← n − 1  to 0 do
        INV= (q_{n−1}...q_{i+1}) · a + r = (b_{n−1}...b_{i+1}), 0 ≤ r < a
        r ← (r ≪ 1) + b_i        /* Move next digit down */
        q' ← 0
        a' ← 0
        while a' + a ≤ r do        /* Find max q' s.t. q'a ≤ r */
              INV= r − a < a' = q'a ≤ r
              a' ← a' + a
              q' ← q' + 1
        q_i ← q'
        r ← r − a'
    return ⟨q, r⟩        /* Quotient and remainder */
```

Let us compute now the time complexity of the algorithm. The for-loop runs for n iterations. Each time it performs a constant number of assignments, comparisons, additions and subtractions. Moreover, the while loop goes up to B iterations (since B is the base and $B \times a \geq r$). Since the base can be considered a constant, we compute a constant number of operations, in each iteration of the for-loop. And since all arithmetic operations are done with n-bit numbers, any comparison, addition and subtraction will take time $O(n)$. In total, we have $n \cdot c \cdot O(n) = O(n^2)$.

□

Euclid's algorithm Analyze Euclid's algorithm that finds the greatest common divisor between two integers. Do the analysis in terms of bit cost and unit cost. Below we present the algorithm and assume that $a \geq b$.

```
gcd(a, b)=
    if b|a then
      gcd ← b
    else
      gcd ← gcd(b, a mod b)
```

Solution to Euclid's algorithm

Since the algorithm is recursive, it is not clear if it always terminates, so we start to prove it. In order to prove termination, we usually use a potential variable that has a lower bound (eg zero) and decreases by one at each recursive call. In this algorithm we can use a as a potential variable. Since a is always at least as large as b, the algorithm terminates as soon as $a = b$ or $b = 1$. Thus we have a lower bound on a and we can see that it decreases in each call. The number of calls depends obviously on the input size, so let's calculate how the main parameter, a, decreases. Let a_i be the value of a in the $i - th$ step of the algorithm. We prove the following lemma:

Lemma 1 $a_{i+2} \leq a_i/2$.

Bevis. We know that $a_{i+2} = b_{i+1}$ and $b_{i+1} = a_i \bmod b_i$, so $a_{i+2} = a_i \bmod b_i$. Now assume that $a_{i+2} > a_i/2$. This means that $b_i \geq a_i/2$, which gives a contradiction, as $a_i = a_{i+2} + cb_i > a_i/2 + a_i/2 = a_i$. □

Using the above lemma we can prove that

$$\lceil \log a_{i+2} \rceil \leq \left\lceil \log \frac{a_i}{2} \right\rceil = \lceil \log a_i - \log 2 \rceil = \lceil \log a_i \rceil - 1.$$

This means that in each recursive call, the input size decreases (at least) a bit. Therefore, the total number of calls will be at most $2\lceil \log a \rceil$. In each recursive call we made only modulo and the unit cost operations, which take constant time. Thus, we conclude that the time complexity is $2\lceil \log a \rceil = 2n \in O(n)$. If we analyze the algorithm with respect to bit cost, we must be more accurate. A division between two n-bit integers takes (as we have seen) $O(n^2)$, and so does the modulo operation. In conclusion, if we make $O(n)$ calls and each call takes $O(n^2)$ bit operations, then the total time complexity is $O(n^3)$.

□

Exponentiation with repeated squaring The following algorithm computes a power of 2 with the exponent which is itself a power of 2. Analyze the algorithm in terms of unit cost and bit cost.

Input: $m = 2^n$
Output: 2^m
$power(m) =$
    $pow \leftarrow 2$
    for $i \leftarrow 1$ **to** $\log m$ **do**
        $pow \leftarrow pow \cdot pow$
    return pow

Solution to Exponentiation with repeated squaring
Note that the for-loop takes $\log m = n$ iterations. Since each operation takes a unit time, the time complexity is $O(n)$.

If we use the bit cost, we must examine the cost of each multiplication inside the loop. We assume that the cost for multiplying two l-bit integers is $O(l^2)$ (in practice, there are faster ways).

In each iteration i , let the variable $pow = pow_i$ be 2^{2^i} so each multiplication requires $O((\log pow_i)^2) = O((\log 2^{2^i})^2) = O(2^{2i})$. Adding up all iterations, we have

$$\sum_{i=1}^{\log m} c2^{2i} = c \sum_{i=1}^{\log m} 4^i = 4c \sum_{i=0}^{\log m - 1} 4^i = 4c \frac{4^{\log m} - 1}{4 - 1} \in O(4^{\log m}) = O(4^n).$$

The algorithm has a linear time complexity in terms of unit cost, but exponential in terms of bit cost. □

Incidence matrix product In the lecture we showed how to represent a directed graph $G = \langle V, E \rangle$ by an incidence matrix of size $|V| \times |E|$. The element at position (i, j) is set to 1 iff $v_i \in e_j$, namely, the vertex v_i is covered by the edge e_j. Show what does the elements of matrix BB^T represent (B^T is the transposed matrix of B).

Solution to Incidence matrix product
Note that diagonal elements (i, i) indicate the number of edges having endpoints in node i. The elements at position (i, j) denote the number of edges between nodes i and j. □

Bipartite graphs Describe and analyze an algorithm that determines whether a given graph is bipartite. The time complexity will be linear in the number of nodes and edges of the graph. A graph $G = (N, E)$ is bipartite iff the nodes can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V.

Solution to Bipartite graphs
Perform a DFS visit in the graph and color each node by alternating red and green. If you find an edge between two nodes with the same color, the graph is not bipartite. □
