École Polytechnique Fédérale de Lausanne
Prof. F. Eisenbrand

April 12, 2019

Discrete Optimization (Spring 2019)

Assignment 8

Problem 1

Show that:

$$a^{\log(b)} = b^{\log(a)}$$

for any $a, b > 0$.

Solution:
Take the logarithm on both sides.

Problem 2

For each recurrence relation, give the correct asymptotic growth for the following expressions (assuming $T(0) = 0$, $T(1) = 1$ and $T(\cdot) = T(\lfloor \cdot \rfloor)$):

- $T(n) = T(n-1) + 1$
- $T(n) = 16T(n/4) + n^2$
- $T(n) = 7T(n/2) + 5n^2$
- $T(n) = T(n^{1/2}) + \log(n)$
- $T(n) = T(n - \sqrt{n}) + T(5)$

Solution:
For the first 3 recurrence relations, we get: $O(n), O(n^2 \log(n)), O(n^{2+\log_2(7/4)})$. The fourth is $O(\log(n))$, this can be seen as follows: Expanding j times we get

$$T(n) = T(n^{(1/2)^{j+1}}) + \log(n) + 1/2 \log(n) + 1/4 \log(n) + \cdots + (1/2)^j \log(n)$$

The series $\sum_{i=0}^{\infty}(1/2)^i$ converges and is equal to 2. j will have to be of order $\log(\log(n))$ for $n^{(1/2)^{j+1}}$ to be a constant (but since the series converges, the value of j is actually irrelevant for the asymptotic running time. Thus:

$$\log(n) \leq T(n) \leq \sum_{i=0}^{\infty}(1/2)^i \log(n) \leq 2 \log(n)$$

The last is a bit trickier: this can be seen by expanding the expression once:

$$T(n) = T(n - \sqrt{n}) + T(5) = T(n - \sqrt{n} - \sqrt{n - \sqrt{n}}) + 2T(5)$$

We see that in each expansion, the current n in $T(n)$ decreases a bit less than before, in particular, we cannot just argue that we are done in $\sqrt{n}$ steps... To make the argument a bit more formal, we define $n_0 = n$ and $n_i = n_{i-1} - \sqrt{n_{i-1}}$. If we expand $T(n)$ i times, then we have $T(n) = T(n_i) + i * T(5)$. The idea is to regroup the numbers in such a way that for each group, the decrease, i.e. $\sqrt{n_i}$, is more or less the same for all $n's$ in the same group. We do this as follows: for $k = \lceil \log_2(n) \rceil$ we have

$$2^k \geq n = n_0 \geq 2^{k-1}$$

As long as the above holds, we have that $2^{k/2} \geq \sqrt{n} \geq 2^{(k-1)/2}$. This implies that if we expand the recurrence relation at most $2^{(k-1)/2}$ times (but at least $2^{(k-2)/2}$ times if n close enough to 2^k(*)), the new n is such that $2^{k-1} \geq n_{new} \geq 2^{k-2}$ (or, more formally, $n_j \leq 2^k$ for $j \leq 2^{(k-1)/2}$). We have to expand as many times until the resulting n is some constant. This implies the following running time:

$$T(n) \leq O(T(5)) \left(\sum_{k=1}^{\log_2(n)} \sqrt{2}^k \right) = O(T(5)) \frac{1 - \sqrt{2}^{\log_2(n)+1}}{1 - \sqrt{2}} = O(T(5)) \sqrt{2}^{\log_2(n)}$$

$\sqrt{2}^{\log_2(n)} = \sqrt{n}$ by exercise 1. Furthermore, $T(5)$ is simply a constant, so we indeed get $T(n) \leq O(\sqrt{n})$. For the reverse direction, i.e. $T(n) \geq c\sqrt{n}$ for some fixed c and for n large enough, we can proceed analogously, using (*).
A simpler way for this problem (but not quite as general), is to find k such that for our starting n, we have:

$$k^2 + 1 \leq n = n_0 \leq (k+1)^2$$

How many times do we have to expand the expression $T(n)$, such that $n_j \leq k^2$? If $n_j \geq k^2 + 1$, we have that $\sqrt{n_j} \geq k$ and so $n_j \leq n_0 - j * k$. Since $(k+1)^2 - (k^2+1) = 2k$, we see that $n_2 \leq k^2$. This implies that $n_{2k} \leq 0$, meaning that we are done after at most $2k$ steps. Conversely, we also always need one step per iteration, i.e., if for some i and l, $n_i \geq l^2$, $n_{i+1} \geq (l-1)^2$. Since $k = \sqrt{n_0}$, we see that the asymptotic running time is $O(T(5)\sqrt{n}) = O(\sqrt{n})$.
How would you change the argument to solve the asymptotic running time of $T(n) = T(n - \sqrt[3]{n}) + C$?

Problem 3

Complete the algorithm below such that it adds two natural numbers in binary representation $a_0, \ldots, a_{l-1}$, $b_0, \ldots, b_{l-1}$. What is the asymptotic running time (number of basic operations) of your algorithm? Can there be an asymptotically faster algorithm?

Input: Two natural numbers a and b in their binary representation
$a_0, \ldots, a_{l-1}$, $b_0, \ldots, b_{l-1}$.
Output: The binary representation $c_0, \ldots, c_l$ of $a + b$

carry := 0
for $i = 0, \ldots, l-1$
    $c_i = \text{carry} + a_i + b_i \pmod 2$
    carry :=
c_l :=
return $c_0, \ldots, c_l$

Solution:

Input: Two natural numbers a and b in their binary representation $a_0, \ldots, a_{l-1}$, $b_0, \ldots, b_{l-1}$.
Output: The binary representation $c_0, \ldots, c_l$ of $a + b$

carry $:= 0$
for $i = 0, \ldots, l-1$
    $c_i = \text{carry} + a_i + b_i \pmod 2$
    carry $:= (a_i \wedge b_i) \vee (a_i \wedge \text{carry}) \vee (b_i \wedge \text{carry})$
$c_l :=$ carry
return $c_0, \ldots, c_l$

The algorithm performs $O(l)$ basic operations. There cannot be any asymptotically faster algorithm since to correctly compute the sum we need to read all the input, hence we already need $\Omega(l)$ operations.

Problem 4

Show that there are n-bit numbers $a, b \in \mathbb{N}$ such that the Euclidean algorithm on input a and b performs $\Omega(n)$ arithmetic operations.
Hint: Fibonacci numbers

Solution:
Since $F_n = F_{n-1} + F_{n-2}$, clearly the Euclidean division between F_n, F_{n-1} gives $q = 1, r = F_{n-2}$ for any $n \geq 2$, hence $GCD(F_n, F_{n-1})$ will call $GCD(F_{n-1}, F_{n-2})$, which will call $GCD(F_{n-2}, F_{n-3})$, etc., until $GCD(1, 0)$ is called. Hence the Euclidean algorithm performs $\Omega(n)$ recursive calls, hence $\Omega(n)$ arithmetic operations. To complete the proof we need to show that F_n can be represented with n bits: this follows from $F_n \leq 2^n$, which can be easily proved by induction.

Problem 5

Let $A \in \mathbb{R}^{m \times n}$. $A_{j \leftrightarrow i}$ is the matrix A where we switch rows i and j. Similarly, $A_{i \to i+cj}$ is the matrix A where we add c times row j to row i and leave the rest unchanged.

1. Find a matrix S_{ij} such that $S_{ij}A = A_{j \leftrightarrow i}$ and a matrix E_{ij}^c such that $E_{ij}^c A = A_{i \to i+cj}$
2. Given i, j, k, l, c, find i', j', k', l', c' such that:

$$S_{kl}E_{ij}^c = E_{i'j'}^{c'}S_{k'l'}$$

Solution:
$S_{ij} \in \{0,1\}^{m \times m}$ is a permutation matrix, i.e. it has only m entries equal to 1 and no row or column has more than one non zero entry. For indexing reasons, call $S_{ij} = P$. If a row k is not being swapped with another row, then we have $P_{kk} = 1$. To switch rows i and j, we put $P_{ij} = 1$ and $Pji = 1$. All the remaining entries are 0.
$E_{ij}^c \in \mathbb{R}^{m \times m}$ is the identity matrix I_m where we add c to the entry in row i, column j.
For the second part, it is clear that $c' = c$. Also, we can take $k' = k$ and $l' = l$. If $k = l$ (meaning $S_{kl} = I_m$) or if $\{i, j\} \cap \{k, l\} = \emptyset$, we can choose $i = i', j = j'$. If $i \notin \{k, l\}$ but $j = k$ (analogously if $j = l$), then $i' = i$ and $j' = l$. If $j \notin \{k, l\}$ but $i = k$ (analogously if $i = l$), then $j = j'$ and $i' = l$. If $i = k$, $l = j$ (analogously for the other case), then $i' = l$ and $j' = k$.