Powering and addition chains

We will see here several methods to efficiently compute x^n, and compare their costs (the cost being defined as the number of multiplications). Download the files `chains.py`, `PowerTree.py` and `test.py`. The files `chains.py` and `PowerTree.py` contain the skeleton code that you need to complete. The file `test.py` contains the code to test your solution.

1 Binary powering and its cost

Question 1. Rewrite the recursive function `bin_pow(x,n)` seen in class, with the only modification that the number of products has to be 0 for $n = 1$. In other words it corresponds to

$$x^0 = 1,\ x^1 = x,\ \text{and for } n \geq 2 :\ x^n = (x^{\lfloor n/2 \rfloor})^2 \text{ for } n \text{ even},\ x^n = x * (x^{\lfloor n/2 \rfloor})^2 \text{ for } n \text{ odd}.$$

Test your method by executing the file `test.py`.

Question 2. Write a recursive function `cost_bin_pow(n)` that returns the number $c(n)$ of multiplications of a call to `bin_pow(x,n)`. Recall that $c(n)$ satisfies the recurrence

$$c(0) = c(1) = 0,\ \text{and for } n \geq 2 :\ c(n) = 1 + c(\lfloor n/2 \rfloor) \text{ for } n \text{ even},\ c(n) = 2 + c(\lfloor n/2 \rfloor) \text{ for } n \text{ odd}.$$

Test your method by executing the file `test.py`.

2 Factorization powering and its cost

An integer n is called composite if it can be written as a product of two integers both greater than 1, and is called prime otherwise. For a composite integer n, the smallest factor of n is the smallest $p \geq 2$ that divides n. Another recursive strategy to compute x^n is based on the following recursive scheme :

1. $x^0 = 1$ and $x^1 = x$,
2. for $n \geq 2$ a prime number, we have $x^n = x * x^{n-1}$,
3. for $n \geq 2$ a composite number, with p the smallest factor and $q = n/p$, we have $x^n = (x^p)^q$.

Question 3. Write a function `smallest_factor(n)` that returns -1 if n is prime and returns the smallest factor of n if n is composite (to gain on the time complexity, note that the smallest factor can not be greater than $\sqrt{n}$). Test your function (e.g. 31 is prime, the smallest factor of 9 is 3, and the smallest factor of 1001 is 7) by executing the file `test.py`.

Question 4. Write a recursive function `factor_pow(x,n)` that returns x^n based on the recursive strategy shown above. Test your method by executing the file `test.py`.

Question 5. Write a recursive function `cost_factor_pow(n)` that returns the number of multiplications of a call to `factor_pow(x,n)` (we do not count the cost of the calls to `smallest_factor`). Note that the cost $c(n)$ satisfies the recurrence : $c(0) = c(1) = 0$ and if $n \geq 2$ then $c(n) = 1 + c(n-1)$ if n is prime, while $c(n) = c(p) + c(n/p)$ if n is composite, with p its smallest factor.

To test your code, you can write a loop to display the comparated costs of `bin_pow(x,n)` and `factor_pow(x,n)` for n from 0 to 40. You should see that both costs are the same for $n \leq 14$, that `factor_pow` is better (cost smaller by one unit) for $n \in \{15, 27, 30, 31, 39\}$ and that `bin_pow` is better (cost smaller by one unit) for $n = 33$.

3 Addition chains

An addition-chain of length ℓ for n is a strictly increasing sequence $[a_0, \ldots, a_\ell]$ of numbers such that $a_0 = 1$, $a_\ell = n$, and for every $k \in [1..\ell]$, there are two indices $i \leq j < k$ such that $a_k = a_i + a_j$. In other words, each non-initial entry z of the sequence can be written as the sum of two entries (possibly twice the same entry) that are on the left of z. For instance, $[1, 2, 4, 6, 10, 14, 16, 24]$ is an addition-chain for $n = 24$. An addition-chain lets us compute x^n using ℓ multiplications :

$$\text{for } k \text{ from 1 to } \ell \text{ compute and store } x^{a_k} = x^{a_i} * x^{a_j}$$

Any 'strategy' to compute x^n using ℓ multiplications can be reformulated as a strategy for finding an addition-chain of length ℓ for n. For instance for binary powering, the addition-chain $L(n)$ associated to n is recursively specified as follows :

— $L(1) = [1]$

— if n is even, then $L(n)$ is obtained by appending n to $L(n/2)$, e.g. $L(6) = [1, 2, 3, 6]$ and $L(12) = [1, 2, 3, 6, 12]$.

— if $n \geq 3$ is odd then $L(n)$ is obtained by appending n to $L(n-1)$, e.g. $L(6) = [1, 2, 3, 6]$ and $L(7) = [1, 2, 3, 6, 7]$.

For factorization powering, the addition-chain $L(n)$ associated to n is recursively specified as follows :

— $L(1) = [1]$

— if $n \geq 2$ is prime, then $L(n)$ is obtained by appending n to $L(n-1)$, e.g. $L(4) = [1, 2, 4]$ and $L(5) = [1, 2, 4, 5]$.

— if $n \geq 2$ is composite, $n = pq$ with p the smallest factor, let $L^{(p)}(q)$ be $L(q)$ where each entry is mutliplied by p, and the first entry (which is p) is removed. Then $L(n)$ is obtained by concacenating $L(p)$ with $L^{(p)}(q)$. For instance $L(3) = [1, 2, 3]$ and $L(5) = [1, 2, 4, 5]$, giving $L^{(3)}(5) = [6, 12, 15]$, and $L(15) = [1, 2, 3, 6, 12, 15]$.

A crucial question (still widely open) to optimize the computation of x^n is thus to compute an addition-chain of n of minimal length ; we let $L^\star(n)$ be this length. We have seen in class that $\lfloor \log_2(n) \rfloor$ is a lower bound, and we have seen that the binary powering method can be done with at most $\lfloor 2 \log_2(n) \rfloor$ multiplications, so we have

$$\lfloor \log_2(n) \rfloor \leq L^\star(n) \leq 2 \lfloor \log_2(n) \rfloor.$$

An addition-chain is called *simple* if every non-initial entry z can be written as the sum of the preceding entry and another entry on the left of z. For instance $[1, 2, 4, 6, 10, 14, 16, 26]$ is a simple addition-chain but $[1, 2, 4, 6, 10, 14, 16, 24]$ is not (the last entry decomposes as $24 = 14 + 10$ and can not be written as $24 = 16 + y$ with y an entry on the left of 24). For binary powering and factorization powering it can easily be checked (by induction) that the addition-chain $L(n)$ associated to n is always simple.

Question 6. Write a function `power_from_chain(x,L)` that takes two arguments : an integer x and a list L whose entries give a simple addition-chain of an integer $n \geq 1$. Your function has to perform a loop on L and returns x^n (it can be useful to use a dictionary structure to store the powers of entries in L). Test your function, e.g. with $x = 2$ and $L = [1, 2, 3, 6, 12, 15]$ (the function should return $2^{15} = 32768$).

4 The power tree construction of addition chains

We will now see a method (based on a certain tree construction) to compute a simple addition-chain of any integer n, which is at least as good as binary powering, and (experimentally) very rarely underperforms factorization powering.

The power tree, due to Knuth (we follow here the presentation in the book 'A guide to algorithm design' by Benoit, Robert and Vivien), is constructed as follows. The root of the tree is 1 (layer 0 of

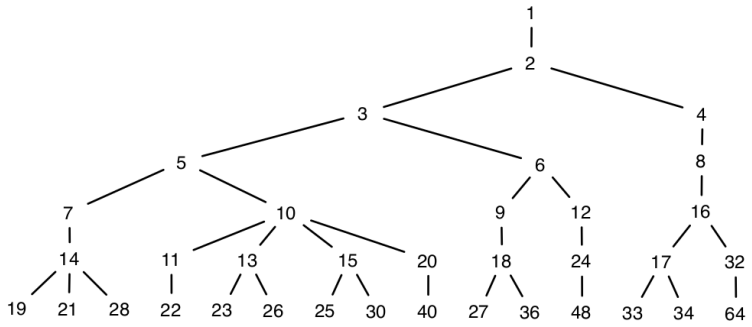

FIGURE 1 – The power tree (layers 0 to 6)

the tree). The tree is then built by induction. For $k \geq 0$, the $(k+1)$-th layer of the tree is defined from the existing layers (layers $0, \ldots, k$) as follows. Consider each node j of the k-th layer from the left to the right, and create nodes $j+1$, $j+a_1$, $j+a_2$,..., $j+a_k = 2j$ at layer $k+1$, as children of node j, in this order from left to right, where $1, a_1, \ldots, a_k = j$ is the path from the root to j. We *do not add* a node in the tree if there is already a node with the same value. Note that with this construction, the path from the root to any node n in the tree forms a simple addition-chain for n.

Download the file `PowerTree.py` from moodle into your project. The file contains the class `PowerTree` that has already a constructor and a method `draw_tree(self)` to display the tree structure. Any object of the class has two attributes : `parent` and `layers`. The attribute `parent` is a dictionary such that `parent[1]`=1, and if $n \geq 2$ is present in the tree then `parent[n]` is the label of the parent of n. And the attribute `layers` is a list of lists, such that `layers[i]` gives the list of the node-labels at level i, ordered from left to right. Initially the constructor only builds the layer 0 of the tree and assigns the parent of 1 to be 1.

Question 7. Complete the methods

- `add_layer(self)` : this methods builds `layers[k+1]` from `layers[0]`,...,`layers[k]`
- `path_from_root(self,k)` : this method returns a list giving the elements of the path from the root to k in the tree (if k is not in the tree the method returns -1)

You can then test your code, by writting commands in order to create a new tree, add a few layers, and display the tree (check that the tree corresponds to the one in Figure 1).

Question 8. Write a method `power_tree_chain(n)` that returns the addition chain associated to n in the power tree. Using this method, write a method `power_tree_pow(x,n)` that computes x^n from the addition chain of the power tree, and write a method `cost_power_tree_pow(n)` that returns the cost of a call to `power_tree_pow(x,n)`.

Question 9. Compare the costs of `bin_pow(x,n)`, of `factor_pow(x,n)` and of `power_tree_pow(x,n)`. You should see that the cost of the 3rd method is the min of the costs of the first two methods, for $n \leq 22$, and the first value for which it is strictly smaller than both is for $n = 23$ (the respective costs are $7, 7, 6$).

Remark. It can be proved that the cost of power tree powering is always at most the cost of binary powering. However for some (very rare experimentally) values of n it may underperform factorization powering (the first such value is $n = 19879$ where the respective costs are $19, 18$, check it !).

Remark. If the layer addition is done by adding the children in decreasing order from left to right, then it gives another tree structure and another way of associating an addition chain to n. With this convention it can be shown that the addition chain of n is the same as with binary powering.