École Polytechnique — CSE202 Design and Analysis of Algorithms

Powering and addition chains

We study several methods to efficiently compute x^n, and we compare their costs (defined as the number of multiplications). Download the files `chains.py`, `PowerTree.py`, and `test.py` from Moodle. The files `chains.py` and `PowerTree.py` contain the skeleton code that you need to complete. The file `test.py` contains code for testing your solutions.

1 Binary powering and its cost

Question 1. Rewrite the recursive function `bin_pow(x,n)` seen in class, that is,

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

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

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

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

2 Factorization powering and its cost

An integer $n \geq 2$ 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 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 $x^n = x \cdot x^{n-1}$, and
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 increase efficiency, note that the smallest factor cannot 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 :

1. $c(0) = c(1) = 0$,
2. for $n \geq 2$ a prime number, we have $c(n) = 1 + c(n-1)$, and
3. for $n \geq 2$ a composite number, with p the smallest factor, we have $c(n) = c(p) + c(n/p)$.

To test your code, you can write a loop to display the compared 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

For an integer n, an *addition chain of length* ℓ 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 is 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 as follows :

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

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 with 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 $L(12) = [1, 2, 3, 6, 12]$, and
- 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 with n is recursively specified as :

- $L(1) = [1]$,
- 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 $L(5) = [1, 2, 4, 5]$, and
- if $n \geq 2$ is composite, $n = pq$, with p the smallest factor, let $L^{(p)}(q)$ be $L(q)$ where each entry is multiplied by p, and the first entry (which is p) is removed. Then $L(n)$ is obtained by concatenating $L(p)$ 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 cannot 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 with 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 are a simple addition chain of an integer $n \geq 1$. Your function should loop over L and has to return x^n (it can be useful to use a dictionary structure to store the powers of entries in L). Test your method by executing the file `test.py`.

4 The power tree construction of addition chains

We now implement a method (based on a certain tree construction) for computing a simple addition chain of any integer n that is at least as good as binary powering, and (experimentally) very rarely performs worse than 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 ; see also Figure 1), is constructed as follows : The root of the tree is 1 (layer 0 of the tree). The tree is then built inductively. For $k \geq 0$, the $(k+1)$-st layer is constructed by considering each node j of the k-th layer from the left to the right individually. For

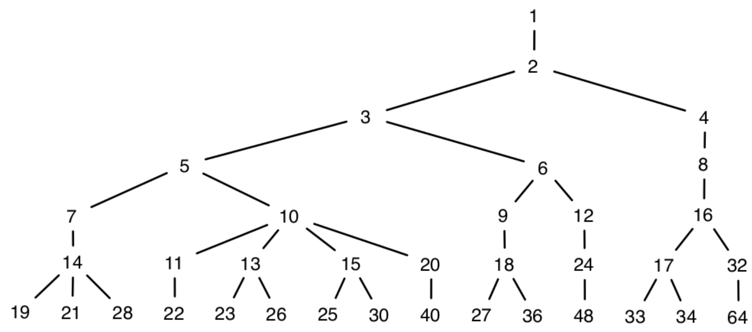

Figure 1 – The power tree with layers 0 to 6.

each such node j, let $(a_i)_{i\in[0..k]}$ with $a_0 = 1$ and $a_k = j$ denote the path from the root to j. We create the new nodes $(j + a_i)_{i\in[0..k]}$ (in layer $k + 1$) as children of node j, in this order from left to right, but we *do not add* a new node if it already exists in in the tree. For example, when constructing layer 3 in the tree in Figure 1, we first create the children of node 3. Since the path from the root to 3 is $(1, 2, 3)$, we would add the nodes $3 + 1$, $3 + 2$, and $3 + 3$. However, since the node 4 $(= 3 + 1)$ is already present in the tree (it is a sibling of 3), we do not add it. We then create the children of 4 analogously. We would add the nodes $4 + 1$, $4 + 2$, $4 + 4$, but we only add node 8, as the other two are already present at this point in time. Since we iterated over all nodes in layer 2, we are done constructing layer 3.

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. 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

— `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), and

— `add_layer(self)` : this methods builds `layers[k+1]` from `layers[0]` to `layers[k]`.

You can test your code by writing commands that 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 with 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. Test your method by executing the file `test.py`.

Question 9. Write a method `cost_power_tree_pow(n)` that returns the number of multiplications of a call to `power_tree_pow(x,n)`, then compare the costs of `bin_pow(x,n)`, of `factor_pow(x,n)`, and of `power_tree_pow(x,n)`. You should see that, for $n \leq 22$, the cost of the 3rd method is the minimum of the costs of the first two methods, 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 proven 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 performs worse than 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.