Some examples of Time Complexity

Exponentiation of small numbers

Slow version

Consider the complexity of calculating the x^y for a nonnegative integer y.

We will assume that adding, multiplying, dividing, and comparing numbers each take (at most) a constant amount of time.

(This assumption makes sense as long as x, y, and x^y are all small enough to fit in a 'machine word')

$z := 1$;
while $y > 0$ **do**
    $z, y := z \times x, y - 1$

The body of the loop is $\Theta(1)$. The number of iterations is y, so the algorithm's time complexity is $\Theta(y)$.

Fast exponentiation

Now let's look at the 'fast' version

$z := 1$;
while $y > 0$ **do**
    if odd(y) **then** $z, y := z \times x, y - 1$
    else $x, y := x^2, y/2$

Again the loop body is constant time ($\Theta(1)$).

The number of iterations is described by

$f(y) = 0$, if $y = 0$
$f(y) = 1 + f(y - 1)$, if $y > 0$ and y is odd
$f(y) = 1 + f(y/2)$, if $y > 0$ and y is even

Consider y in binary notation, for example $y = 21 = 10101_{(2)}$

What sequence of values does y take on?

$$10101_{(2)}$$
$$10100_{(2)}$$
$$1010_{(2)}$$
$$101_{(2)}$$
$$100_{(2)}$$
$$10_{(2)}$$
$$1_{(2)}$$
$$0_{(2)}$$

$f(21) = 7$. The exact value for $f(y)$ is the number of 0s + 2 times the number of 1s $-$ 1

The length of y in bits is decreased by 1 every 1 or 2 iterations. The number of iterations is thus limited to 2 times the number of bits required to represent y

$$f(y) \leq 2 \times \lceil \log_2 y \rceil \leq 2 \times (\log_2 y + 1)$$

Thus the time for the algorithm is $\Theta(2 \log_2 y + 2) = \Theta(\log y)$.

Modular arithmetic on large natural numbers

Large *natural numbers* can be represented using arrays of words where each word can hold a value in $\{0, ..B\}$, where B is a base such that $2 \leq B \leq 2^w$ with w being the width of a word in bits. The number represented by an array X is

$$x = \left(\sum_{i \in \{0,..|X|\}} B^i \times X(i) \right)$$

Suppose we want to compute $x^y \bmod B^N$ with x, y, and z represented by arrays X, Y, and Z of size N.

How long do the operations in the fast exponentiation algorithm take?

All operations are done modulo B^N.

- Provided B is even, we can determine whether y is odd by determining whether $Y(0)$ is odd. $\Theta(1)$
- Adding two numbers takes N word level additions and there are $N - 1$ 'carry' additions. So we have $\Theta(N)$

- Multiplying using the method traditionally taught to children produces N partial products of size N each (we can discard upper digits). For example with $B = 10$ and $N = 5$ we have

$$\begin{array}{r} 87654 \\ \times\ 45676 \\ \hline 25924 \\ +\ 35780 \\ +\ 92400 \\ +\ 70000 \\ +\ 60000 \\ \hline 84104 \end{array}$$

- Each digit of each partial product can be computed in constant time, so computing one partial product is $\Theta(N)$. There are in total N partial products, so it takes $\Theta(N^2)$ to compute all the partial products. We can also do the $N - 1$ additions in $\Theta(N^2)$. (If we are clever we can reduce the time by not adding the inevitable 0 digits. However, over $1/2$ the $\Theta(N^2)$ digits must be processed, so the algorithm is still $\Theta(N^2)$.)
- Subtracting 1 and dividing an even number by two each take $\Theta(N)$ time.

Now let's revisit the algorithm

$z := 1$;
while $y > 0$ **do**
    if odd(y) **then** $z, y := (z \times x) \bmod B^N, y - 1$
    else $x, y := x^2 \bmod B^N, y/2$

We'll assume (for simplicity) $B = 2^w$ for some natural $w > 1$.

Each iteration has one multiplication, so each iteration is $\Theta(N^2)$. In the worst case, y is the largest number possible, $y = B^N - 1 = 2^{Nw} - 1$. The number of iterations is $2Nw - 1$. The total time complexity is $\Theta(wN^3)$.

Exercise: Generalize the above argument to the case of $B \leq 2^w$.

Note: We have changed the way that we measure the size of the input since the previous examples. There the input size was the "value" of the numbers: x, y.

Here we are measuring the "size" of the numbers in words (N) and the size of the words in bits w.

(Both measures are commonly used for arithmetic problems!)

How do these relate? In the worst case $y \cong 2^{Nw}$. So $\log_2 y \cong Nw$. In the case where $N = 1$ we have

$$\Theta(wN^3) = \Theta(w) = \Theta(\log y)$$

so the results of this section generalize the results of the previous.

Question: Is it better to use a big base or a small base? Suppose our machine offers a choice of 32 bit or 16 bit arithmetic. Suppose we need to compute $x^y \bmod 2^{2048}$.

It takes $N = 64$ words to represent a number when $w = 32$. The number of operations is about cwN^3 for some constant c. So about $c2^{23}$ are needed.

Using $w = 16$ and $N = 128$ we get $c2^{25}$, which is 4 times more. So unless the 16 bit arithmetic is much faster than the 32 bit arithmetic, we are better off with the larger base.

Challenge:. There are quicker ways to multiply. Can you find one?

Searching a Maze

Consider searching a maze that consists of 'places' and 'tunnels'.

- There is 1 'Root' place that has no way in and 2 ways out
- 'Branch' places have 1 way in and 2 ways out.
- 'Leaf' places have 1 way in and no way out.

All leaf places are N tunnels away from the root place.

0 or more of the leaf places have a pot of gold.

Problem: You start at the root place and must figure out whether there is any gold in the maze.

while true **do**
    if there is gold, stop
    else if there is an unexplored out-going tunnel,
        traverse it and mark it as explored
    else if there is an in-coming tunnel,
        traverse it
    else stop

Time complexity. We will assume that each operation in the loop is $\Theta(1)$.

The maze forms a directed tree with a branching factor of 2.

There are $2^{N+1} - 1$ places and $t = 2^{N+1} - 2$ tunnels.

In the worst case —no gold— the algorithm traverses each tunnel twice and then stops. So the number of iterations is $1 + 2t$ or $2^{N+2} - 3$.

So the algorithm is $\Theta(2^N)$.

Some more problems on mazes

Input: A maze of N places and with up to 1 tunnel connecting each pair of places. (Tunnels may cross without connecting.)

Problem: Is every place reachable from each other place?

Problem: Is there a path that traverses each tunnel exactly once and returns to the starting place?

Problem: Is there a path that visits each place exactly once and returns to the starting place?

Problem: Each place needs to be painted. But, for reasons of fashion, any two places connected by a tunnel must be painted different colours. How many colours are needed, minimum?

Find the "best" algorithm you can for each problem.

What is the time complexity of each algorithm in terms of N?