CSCI3390-Lecture 13: Polynomial time; the class **P**

1 Polynomial versus exponential time; orders of growth

The website contains specifications for two Turing machines. One of these accompanied the first assignment; this is the machine that takes a string of 0's and 1's, and sorts it by moving all the 0's to the left and all the 1's to the right. A high-level description of how it works is:

```
repeat:
    scan right, looking for a 1 followed by 0
    if no such occurrence is found, halt
    otherwise swap the 0 and 1, return to the left end
        of the tape
```

In the solutions to the first assignment, there is a very detailed analysis of how many steps this algorithm requires, but we can make a rough 'back-of-the-envelope' analysis which is actually more pertinent for our new topic: Suppose the length of the input is n. The 'body' of the loop, the portion underneath the word 'repeat' requires no more than $2n$ steps—usually it will be considerably less than this, because the machine will rarely have to scan all the way to the end of the tape to find the first occurrence of 10. The number of passes is equal to the number of 'inversions' in the original input—that is, the number of pairs of positions $i < j$ where the i^{th} symbol is 1 and the j^{th} symbol is 0. The largest possible number of inversions occurs when the input has the form $1^{n/2}0^{n/2}$. Thus the maximum number of passes is $n^2/4$. As a result, the number of steps is no more than

$$n^2/4 \times 2n = n^3/2.$$

But the more pertinent calculation is: 'no worse than a constant times n^2 multiplied by a constant times n, thus bounded by a constant times n^3.' We will say that the running time for this algorithm is $O(n^3)$ (exact definition below).

The second machine executes the following algorithm:

```
repeat:
    scan right, converting 1s to 0s until a 0 is found
    if no 0 is found, halt
    otherwise, change the 0 to 1 and return to start
       of tape
```

On the input 0^n, the machine runs through all the possible patterns of n bits, once for each scan. For example, with $n = 3$, it will produce after each scan,

000,100,010,110,001,101,011,111

Thus this machine can require as many as 2^n steps (and no more than $2n \cdot 2^n$ steps, because each scan forward and back requires fewer than $2n$ steps).

Usually, in analyzing computer algorithms, we are not too much concerned with constant multiples: an algorithm that requires time $2n$ one one machine may require time n on another, because of a faster processor carrying out the same steps, or the constant multiple could just be an artifact of the units of time that we use. We write

$$f(n) = O(g(n))$$

if there is some constant $c > 0$ such that

$$f(n) \leq c \cdot g(n)$$

for all sufficiently large values of n. We interpret this to mean that *asymptotically, g* grows at least as fast as f. Thus if we let $t(n)$ denote the maximum number of steps our sorting machine executes on inputs of length n, we have

$$t(n) = O(n^3),$$

since in the worst case

$$t(n) = n^3/2$$

for all n. Not only is n^3 an asymptotic upper bound for the running time, which is what $O(n^3)$ means, in this case it is also an asymptotic lower bound, since $n^3 = O(t(n))$ as well. We say that algorithms whose running time is $O(n^k)$ for some k are *polynomial-time* algorithms.

In contrast, the counting Turing machine runs much more slowly: its running time grows *exponentially* in n. In fact, if $a > 1$ and $k > 0$, we always have

$$\lim_{n\to\infty} \frac{n^k}{a^n} = 0.$$

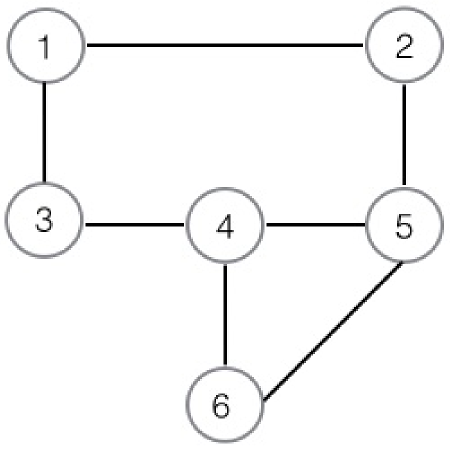

Figure 1: Is there a path from vertex 1 to vertex 6?

We say that $n^k = o(a^n)$. In general, we write $f(n) = o(g(n))$ if the limit of the ratio of $f(n)$ to $g(n)$ is zero.

In practice, what does this mean? Our Turing machine for sorting a string of bits of length 20 will use no more than $20^3 = 80,000$ steps (the true value will actually be quite a bit less than this), while the machine for counting will use at least $2^{20} \approx 1,000,000$ steps. If we try a string of length 40, the first machine will require 640,000 steps—still quite do-able–while the second will need at least one trillion steps, which approaches the limits of what is possible in practice.

In general, we view algorithms requiring polynomial time as *feasible* or *tractable*, and those requiring exponential time as *infeasible* or *intractable.*

2 Example: Graphs

Earlier we discussed an algorithm for finding the shortest path in a graph. Essentially the same algorithm will solve the problem of determining if there *is* a path between two vertices:

Input: A graph G and two vertices u, v.

Output: Yes if there is a path connecting u to v, no otherwise.

We assume we have some nice encoding of the graph, perhaps in the form of *adjacency lists.* Each item below is a list of the vertices adjacent to the vertex at the head of the list:

```
1:2,3
2:1,5
3:1,4
4:3,5,6
5:2,4,6
6:4,5
```

Suppose we want to determine if there is a path from 1 to 6. (There is, of course, which is easy to see from a quick visual inspection of this tiny graph, but we want to describe something that works in general for any graph.) The idea is to maintain a list called a *queue* into which we put first the source vertex 1, then the vertices adjacent to 1, then the vertices adjacent to those, *etc.* We also maintain a checklist so that we do not put a vertex into the queue if it has already been there. Here is the algorithm:

```
Put 1 in the queue
Check 1 off
Repeat:
    if queue is empty, halt and reject
    remove first item u from the queue
    for each neighbor v of u:
        if v is checked off, ignore it
        if v=6, halt and accept
        otherwise, put v in the rear of the queue and check it off
```

For the graph in the example, the queue evolves as follows:

```
1
2 3
3 5
5 4
4
```

and accepts when then neighbor 6 is found. (Incidentally, the same algorithm works for directed graphs.)

Here is a gross overestimate of the running time: If the graph has n vertices, then the statements in the loop marked 'repeatedly' are executed at most n times, once for each vertex removed from the queue. The list of neighbors of the removed vertex must be traversed in the loop labeled 'for'. This list has no more than n elements (actually $n-1$) so the statements under the 'for' are each executed no more than n times. This suggests that our algorithm requires $O(n^2)$ steps, so this problem is solved by a polynomial-time algorithm.

2.1 Polynomial-time algorithms; polynomial-time Turing machines

What if you tried to implement this algorithm with a Turing machine? You might try a 3-tape machine, with one tape used to hold the queue and another to hold the checklist. If the encoding of the graph uses m symbols, you might need m steps to find the head of the list for a given vertex. Further, hunting for the end of the queue and to find if a vertex has been checked off may itself require n steps. Taking everything into account, the number of steps is then something like $O(mn + n^3) = O(n^3)$, since $m < n^2$. If we converted this into an equivalent one-tape machine, we would need $O(n^6)$ steps. Since the length m of the input is always greater than or equal to n, this machine runs in time $O(m^6)$. But the point is, this is *still* polynomial time. So our official definition of a polynomial-time algorithm is a Turing machine that answers 'Yes' or 'No' on every input, and does so in $O(m^k)$ steps, where m is the length of the input and k is some positive integer. Such a TM is called a *polynomial-time Turing machine*. But the real point is this:

> *Polynomial time on a Turing machine is the same as polynomial time for pencil-and-paper algorithms.*

Usually the way we will prove something in in polynomial time is by giving descriptions, analogous to the one above, of a pencil-and-paper algorithm. The transformation of such an algorithm to a single-tape Turing machine will involve the usual tedious back-and-forth manipulations, but the blowup is not worse than polynomial.

3 Example problems with Sudoku

We want to analyze the time complexity of algorithms for two problems related to Sudoku puzzles. When we analyze complexity we are usually interested, as in the examples above, in how the running time grows as a function of the input size. It doesn't make sense to say something is a 'polynomial-time' algorithm if the input size is fixed. So we need to talk not just about the usual 9×9 Sudoku, but a generalized version of the puzzle in which the grid is $N^2 \times N^2$, and the entries are integers in the range $1, \ldots, N^2$. This requires no change in the rules. (The standard version is $N = 3$, and I have seen giant Sudoku puzzles with N as large as 6.)

The first problem is

> *Input:* A completely filled $N^2 \times N^2$ Sudoku grid.
>
> *Output:* Yes if the grid represents a legal solution, No otherwise.

The second takes inputs as they are usually given, with a partially-filled grid.

Input: A partially-filled $N^2 \times N^2$ Sudoku grid.

Output: Yes if it is possible to complete this grid to a solution, No otherwise.

Below is a pencil-and-paper algorithm for the first problem. It maintains a checklist containing each of the integers $1, \ldots, N^2$.

```
for each row:
    clear checklist
    for each entry k in the row
        if k is checked off, halt and reject, otherwise check off k
for each column:
    clear checklist
    for each entry k in the row
        if k is checked off, halt and reject, otherwise check off k
for each subsquare:
    clear checklist
    for each entry k in the row
        if k is checked off, halt and reject, otherwise check off k
halt and accept
```

The algorithm requires N^2 checks for each of the N^2 rows, columns, and subsquares. So in total there are $3N^4$ steps, or $3m$ steps, where m is the input size. This might undercount slightly, since we have not specified how much time it takes to search the checklist to determine if an item has been checked. But at worst, this makes the algorithm $O(N^5) = O(m^{1.25})$. So this is a polynomial-time algorithm, and, as we commented above, it is polynomial-time on a Turing machine as well.

What about the second problem? The naïve algorithm is to try out every assignment of integers $1, \cdot, N^2$ to the cells of the puzzle, and then check each of them using the above algorithm. The number of such assignments is $(N^2)^{N^4}$. If we use $m = N^4$ for the input size, it is

$$(\sqrt{m})^m > 2^m,$$

if $m > 2$, so the total number of steps is at least 2^m, and thus this algorithm requires exponential running time, at least.

4 The class P

A decision problem is said to be in **P** if there is a polynomial-time Turing machine that recognizes it. Since polynomial-time Turing machines halt on every input, this is the same as saying that there is a polynomial-time Turing machine that *decides* it. Our remarks above show that the problem of determining whether two vertices of a graph are connected, and of determining whether a filled Sudoku grid is a legal solution, are both in $\mathbf{P}$.

Have we shown that the problem of determining whether a Sudoku puzzle is solvable is *not* in **P**? No–we have only shown a single algorithm for this problem that is not polynomial time. There are indeed better algorithms, but the question of whether there is a polynomial-time algorithm is open. We do not know whether this problem is in **P**.

By the way, our proof that the path problem in graphs is in **P** depended on one particular way of encoding the graph. Remember that officially decision problems are languages, and the same graph problem is translated into different languages under different encodings. But this problem is in polynomial time for any reasonable method of encoding the graph, for instance, adjacency matrices instead of adjacency lists.

5 Example problems with integers

We will look a problems in which the input is a positive, or a collection of several such integers. For example, we consider the problem, 'is N prime?' The time complexity of such problems depends on how we encode integers. We might encode the integer N by a sequence of N marks, for instance

$$* \quad * \quad * \quad * \quad * \quad * \quad * \quad * \quad * \quad * \quad * \quad * \quad *$$

is the encoding of thirteen. This is *unary* encoding. Alternatively, we could encode thirteen in decimal, as 13, or in binary as 1101. If the usual decimal encoding has n digits, then the unary encoding has as many as $10^n - 1$ symbols, and thus takes exponentially longer to write down. This will make the time complexity of algorithms look very different. However, in problems in which we might deal with large numbers (and cryptograply, for example, routinely uses integers hundreds of digits long) such an encoding is for practical purposes unusable. On the other hand, an input that has n decimal digits, when represented in binary, has about $\log_2 10 \cdot n \approx 3.3n$ digits, so the size of the encoding only changes by a constant multiple. Thus the running time of algorithms is, asymptotically, the same at any base. Unless we say otherwise, we will suppose that our integers are encoded in decimal, or in binary. Which of the two we use makes no difference to the complexity.

Now let's look at this problem:

Input: A positive integer N.

Output: Yes if N is prime, No otherwise.

The naïve algorithm is

```
for each integer k=2,3,...,N-1
    divide N by k.  If remainder is 0, halt and reject
halt and accept
```

I'll leave it as an exercise for you to analyze how much time each division takes, but there are $N - 1 \approx 10^m$ divisions to perform, where m is the length of the input encoded in decimal, so this algorithm requires exponential time.

Is there a better algorithm? You may be aware of the fact that it is only necessary to test potential divisors up to

$$\sqrt{N} \approx 10^{m/2} = \sqrt{10}^m$$

but the running time of the resulting algorithm is still exponential in m.

Are there better algorithms? Yes! In fact, in 2002, a polynomial-time algorithm for this problem was discovered by Aggarwal, Kayal and Saxena. The last two authors were undergraduates. And their paper is called 'PRIMES in **P**.'