Lecture 29: Limits of Computation
10:00 AM, Nov 11, 2019

Contents

1 Paths through sorting code ............................................. 1

2 Limits on computation .................................................. 1

2.1 Decision Problems ................................................ 1

2.2 Classes of Problems ................................................ 2

2.3 Halting .............................................................. 2

3 Summary .............................................................. 4

Objectives

By the end of this lecture, you will know:

• Understanding that there are limits to computation

• Exploring classes of problems P and NP

• Proving the Halting Problem

1 Paths through sorting code

Every path from root to leaf represents a path of execution through the tree. There has to be as many leaves as the number of ways to sort permutations of the tree. Ever distinct shuffle has a different path through the tree. If we have an input length of n, there are at least n! number of leaves. Look at the slides for a more mathematical proof. The runtime for comparison based sorting is n log n.

2 Limits on computation

An ideal computer would have infinite memory and time. Unfortunately, we don’t have ideal computers and thus have to be mindful of the runtime of our programs and how much space they require.

We earlier proved that comparison based sorting had to take time at least proportional to n log n. Subset-sum on the other hand seems to need at least $2^n$ time.
Why does subsets require exponential time whereas sorting is only $n \log n$? What is the difference between these problems? Are some computational problems inherently harder than others?

### 2.1 Decision Problems

Decision problems consume some input and produce a boolean output. For example the question subset-sum asks is:

Is there a subset of these weights that adds up to that total? Here, the inputs are the weights $(1 \ 2 \ 8 \ 3 \ 5)$ and the total is 13. This seems to require $2^n$ time to solve.

However, if we want to verify whether a particular subset is in the solution to subset-sum, that is a different problem. The question subset-sum-check asks is:

Are these items a subset of weights that add up to the required total? Here, the inputs also include the purported subset $(0 \ 13)$.

How hard is this problem? There are two steps, first we need to check if the items really add up to the required target. Secondly, we need to check if the input is a valid subset, that is, checking if all the items are in the original set.

If the original list has length $n$, the first part is $O(n \rightarrow n)$ since we only need to go through the list once to add the items. In our example, $0 + 13 = 13$, which is the target, so we are good. Checking whether all the items are in the original set requires going through the set once for each item in our purported subset. Worst case, our subset is of length $n$, and so the second part is $O(n \rightarrow n^2)$. Here, it turns out neither 0 nor 13 are found after going through the list once for each. In total, our program is $O(n \rightarrow n^2)$, which we say is solvable in polynomial time.

### 2.2 Classes of Problems

Some decision problems are known to be solvable in polynomial time. The class of all such problems is called $P$. subset-sum does not appear to be in class $P$.

Some decision problems can be verified in polynomial time, like subset-sum-check above. These problems are said to be in $NP$.

A big unsolved question in Computer Science is: are there any problems in NP that are not in P? In other words, are there any problems that are easy to verify but hard to solve?

Many people think yes; that the the fact that you can verify a problem easily does not necessarily mean you can find a solution equally easily, and the gap between P and NP is seemingly large. However, no one has a proof either way. If you come up with one, it’s worth a million dollars!

### 2.3 Halting

David Hilbert, an early computer scientist believed that it can be proven that there exist no computational problems that we cannot solve, and that it is possible to build a machine that would give us a true or false answer for any decision problem.

What other questions can we ask about programs? We could ask whether a certain program uses a lot of memory, or whether a program correctly computes a desired value.
Another interesting question to ask is whether a program will terminate (or halt) on an idealized computer, or will it run forever? Take the following example of a broken procedure to compute length:

```
(define (b-len alod)
  (cond
   [(empty? alod) 0]
   [(cons? alod) (+ 1 (b-len alod))]))
```

This program does not modify the recursive input to be smaller in any way, and thus will run forever.

Easier questions to ask regarding this would be: Does this program halt before it has performed 10,000 operations? Or does this program halt within 3 minutes? It is possible to answer these questions just by empirical observation.

Interesting things happen, however when we try to define a procedure that determines when given a procedure as an input, whether it will run forever.

```
;; input: proc, a procedure, represented as a string
;; data, a datum that the procedure can consume, represented as a string
;; output: true if (proc data) terminates, false otherwise
(halt? proc data)
```

Assume that we have written halt. We have this idea of procedures consuming other procedures—that is what your Rackette program did. If we wrote Rackette in Racket instead of Reason, we could feed our Rackette source code to our Rackette procedure as input!

```
(define rackette-as-string "(define (evaluate exp) ... )")
```

We could then apply halt? to this procedure.

```
(halt? rackette-as-string rackette-as-string)
```

We can write a generalisation of this called halts-on-self to check for any program, whether it will halt when applied to itself.

```
(define (halts-on-self s)
  (halt? s s))
```

Now let us define some more procedures.
The `loop-if-halts-on-self` procedure takes as input a program `s` (expressed, say, as a long string), and it uses `halts-on-self` to predict whether that procedure would terminate if given as input the program itself. If input program `s` would terminate, then `loop-if-halts-on-self` calls `loop`, and so our program never terminates. If `s` would not terminate, then our program just returns true.

Now imagine that we apply the `loop-if-halts-on-self` procedure to itself! To do so we can define `qs` to be the same program as a string.

```
(define qs "(define (loop-if-halts-on-self s) ... )")
```

Then we evaluate,

```
(loop-if-halts-on-self qs)
```

Does it halt? Let’s consider both possibilities:

1. Assume that it halts. That means that the if-condition evaluated to false. The if-condition called the `halts-on-self` procedure on `qs`, so the `halt?` procedure must have predicted that the `qs` program does not halt when given itself as input.

2. Assume that it does not halt. That means that the if-condition evaluated to true, so the `halt?` procedure must have predicted that the `qs` procedure halts when given itself as input.

If we assume that it halts, we can infer that it does not halt. That is a contradiction. If we assume that it does not halt, we can infer that it does halt, which is also a contradiction. What was our initial assumption? That it is possible to write the `halt?` program. We are forced to conclude that the `halt?` procedure cannot exist.

## 3 Summary

### Ideas
- Some decision problems seemingly cannot be solved in polynomial time
- Even if verifying the solution of the problem can
- No one knows if there really is a distinction between the difficulty of these
- The Halting procedure is an example of a computationally impossible problem.
• Computation has limits

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback.