Analysis of Algorithms - CS 323
Lecture #12 – May 18, 2016

Notes by: Kushal Hada

Announcements:

1. Five more points on test two (bringing us up to +10 so far)
2. From Spring 2016, Announcements #56 - Contacting the Instructor Please email the instructor at LT.CS320@Yahoo.com

Homework:

1. Watch a simple proof for the halting problem https://www.youtube.com/watch?v=92WHN-pAFCs

P = NP

Probably the biggest problem is P ?= NP. We can make an analogy to people who can compose quality music vs people who can appreciate quality music.

There is a huge number of possible permutations of musical notes to create a composition but not all of those permutations results in quality music. Many if not most people can tell whether a given permutation makes a good musical composition. We can compare this to verifying a solution to the travelling salesman problem. The verification of a solution is one level of complexity. However, writing a musical composition from the many possible permutations that results in a good musical composition is a different task.

Countably infinite

A set is countably infinite if its elements can be put in one-to-one correspondence with the set of natural numbers. In other words, one can count off all elements in the set in such a way that, even though the counting will take forever, you will get to any particular element in a finite amount of time.[1]

Uncountably infinite

A uncountable set[2] is is an infinite set that contains too many elements to be countable.

How many programs are out there?

A program can be considered as a sequence of characters. A program can be arbitrarily long (from zero instructions to an arbitrarily long n instructions). This means that there are infinitely many programs out there.

We can do better though. Let us say that there is a finite pool of instructions. Therefore, for any given position in our program, there is a finite set of instructions available per position. Thus, we can say that the number of programs out there is countably infinite.

---

[1] http://mathinsight.org/definition/countably_infinite
[2] https://en.wikipedia.org/wiki/Uncountable_set

However, we can prove that the number of functions possible is not countably infinite. The set of all possible functions is uncountably infinite. See Cantor's diagonal argument[3] which proves that that there are infinite sets which cannot be put into one-to-one correspondence with the infinite set of natural numbers.

The proof shows there is no [total](#) [computable function](#) that decides whether an arbitrary program *i* halts on arbitrary input *x*; that is, the following function *h* is not computable (Penrose 1990, p. 57–63):

$$h(i,x) = \begin{cases} 1 & \text{if } \text{ program } i \text{ halts on input } x, \\ 0 & \text{otherwise.} \end{cases}$$

Here *program i* refers to the *i* th program in an enumeration of all the programs of a fixed Turing-complete model of computation.

f(*i*,*j*)		*i*					
		1	2	3	4	5	6
j	1	1	0	0	1	0	1
	2	0	0	0	1	0	0
	3	0	1	0	1	0	1
	4	1	0	0	1	0	0
	5	0	0	0	1	1	1
	6	1	1	0	0	1	0

f(*i*,*i*)	1	0	0	1	1	0
g(*i*)	U	0	0	U	U	0

Possible values for a total computable function *f* arranged in a 2D array. The orange cells are the diagonal. The values of *f*(*i*,*i*) and *g*(*i*) are shown at the bottom; *U* indicates that the function *g* is undefined for a particular input value.

The proof proceeds by directly establishing that every total computable function with two arguments differs from the required function *h*. To this end, given any total computable binary function *f*, the following partial function *g* is also computable by some program *e*:

$$g(i) = \begin{cases} 0 & \text{if } f(i,i) = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

[3] https://en.wikipedia.org/wiki/Cantor%27s_diagonal_argument

The verification that g is computable relies on the following constructs (or their equivalents):

- computable subprograms (the program that computes f is a subprogram in program e),
- duplication of values (program e computes the inputs i,i for f from the input i for g),
- conditional branching (program e selects between two results depending on the value it computes for $f(i,i)$),
- not producing a defined result (for example, by looping forever),
- returning a value of 0.

The following pseudocode illustrates a straightforward way to compute g:

```
procedure compute_g(i):
    if f(i,i) == 0 then
        return 0
    else
        loop forever
```

Because g is partial computable, there must be a program e that computes g, by the assumption that the model of computation is Turing-complete. This program is one of all the programs on which the halting function h is defined. The next step of the proof shows that $h(e,e)$ will not have the same value as $f(e,e)$.

It follows from the definition of g that exactly one of the following two cases must hold:

- $f(e,e) = 0$ and so $g(e) = 0$. In this case $h(e,e) = 1$, because program e halts on input e.
- $f(e,e) \neq 0$ and so $g(e)$ is undefined. In this case $h(e,e) = 0$, because program e does not halt on input e.

In either case, f cannot be the same function as h. Because f was an *arbitrary* total computable function with two arguments, all such functions must differ from h.

This proof is analogous to Cantor's diagonal argument. One may visualize a two-dimensional array with one column and one row for each natural number, as indicated in the table above. The value of $f(i,j)$ is placed at column i, row j. Because f is assumed to be a total computable function, any element of the array can be calculated using f. The construction of the function g can be visualized using the main diagonal of this array. If the array has a 0 at position (i,i), then $g(i)$ is 0. Otherwise, $g(i)$ is undefined. The contradiction comes from the fact that there is some column e of the array corresponding to g itself. Now assume f was the halting function h, if $g(e)$ is defined ($g(e) = 0$ in this case), $g(e)$ halts so $f(e,e) = 1$. But $g(e) = 0$ only when $f(e,e) = 0$, contradicting $f(e,e) = 1$. Similarly, if $g(e)$ is not defined, then halting function $f(e,e) = 0$, which leads to $g(e) = 0$ under g's construction. This contradicts the assumption that $g(e)$ not being defined. In both cases contradiction arises. Therefore any arbitrary computable function f cannot be the halting function h.

five more points on test two to +10

Probably the biggest problem is
p = np ? watch the video.
The real life difference between P & NP
Proposed solution to travelling salesman
problem We can verify the solution in
a deterministic polynomial time
People who can appreciate music
and people who compose quality music
One level of complexity to read a work
and say that's cool but to be able
come up with it is difference betw^n P & NP

Boring proof of Halting Problem
Approximately in 1930s Alan Turing proved
the halting problem is unsolveable.

COUNTABLE
UNCOUNTABLE $\mathbb{R}$ diagonalization arg
Cantor's diagonalization argument to pro
$\mathbb{R}$ real numbers are irrational (binary)
[0,1) Real numbers including 0 and up to but
not including 1. $0, b_{1,1}, b_{1,2}, b_{1,2}, b_{1,3}, b_{1,4}$

$0.b_{2,1}\ b_{2,2}\ b_{2,3}\ b_{2,4}\ \cancel{b_{2,5}} \ldots$

$0.b_{3,1}\ b_{3,2}\ b_{3,3}\ b_{3,4} \ldots$

$0.b_{4,1}\ b_{4,2}\ b_{4,3}\ b_{4,4}$

$\overline{b_{i,j}}$ = inverse of $b_{i,j}$ $0 \to 1\ 1 \to 0$

$r^* = 0.\overline{b_{1,1}}\ \overline{b_{2,2}}\ \overline{b_{3,3}}\ \overline{b_{4,4}} \ldots$
is not on the list. So the premise is wrong
we can find a number NOT on the list.

How many programs are out there?
There are infinitely many programs but countably infinite in any given model.
How many functions are out there?
$f(x) \to y \quad x \in \cancel{\mathbb{Z}}\ \mathbb{N} \quad y \in \cancel{\mathbb{Z}}\ \mathbb{N}$

x	$f_0(x)$ $f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$	$f^*(x)$
0	$f_1(0)$	$f_2(0)$	$f_3(0)$	$f_4(0)$	$f_0(0)+1$
1	$f_1(1)$	$f_2(1)$	$f_3(1)$	$f_4(1)$	$f_1(1)+1$
2	$f_1(2)$	$f_2(2)$	$f_3(2)$	$f_4(2)$	$f_2($ ⋮
3	$f_1(3)$	$f_2(3)$	$f_3(3)$	$f_4(3)$	$f_i(i)+1$
⋮	⋮				

f^* is a new function where $f^* = f_i(i)+1$
countably infinite programs
UNcountably infinite simple integer functions
UNcountably infinite functions can't be solved
we have a program (x) if $Halt(x,x)$ then loop forever else halt

---

NP completeness making a problem look like another (where have I seen this before?)

Median of five to find median of unsorted
Hamiltonial circuit → Travelling Salesman
IS (independent set problem)
subgraph S such that no edges in V_s are adjacent
Vertex Cover problem VC problem
Subgraph S ~~that is complete~~ such that each $v \in V_s$ is adjacent to another vertex
(These two problems complement)
Clique problem (polynomial time reduction) NP complete problems

One question from stuff from exam1
One q from stuff from exam2
Part 2 → part1 is short answers
E.g. what is the worst case time complexity
If quick sort n^2 average case $n \log n$
Best case of bubble sort is $O(n^2)$ insertion
sort best case is $O(n)$ Time complexity of
binary search $O(\log n)$ Minimum time to sort
$O(n \log n)$ Names of main algorithm $n \log$
$1+2+\ldots+n = n(n+1)/2$ $r+r^2+r^3+\ldots \frac{r^{n+1}-1}{r-1}$
$f(n) = f(n-1) + f(n-2) \rightarrow f(n) - f(n-1) - f(n-2) = 0$
$x^2 - x - 1 = 0$ master theory don't have
to invoke it but $T(n) = T(\frac{n}{2}) + f(n)$
$S(k) = S(k-1)$ $n = 2^k$ domain
transformation. what result helps
explain the time complexity?
$T(n) = 2T(\frac{n}{2}) + f(n)$ Domain
transformation is where we rethink the
input. The range transformation is
where we change the output.
Exam1 question will probably have to
do with sorting (so a linear time sort)

A decision problem will have a yes or no
answer. An optimization problem asks what
is the minimum/maximum we can do?

Dynamic programming → Floyd's
Dijkstra's, Prim's algorithm (keep adding one
more element to sets) will not ask dynamic
programming for triangulation. Big integer
multiplication. Bellman Ford SSSP that
handles negative w $O(|V| \cdot |E|)$ time complexity
knapsack problem is NP complete $O(N \times K)$ where
K is the upper bound capacity on knapsack
we incremented by 1, 2, finite precision vs
$O(n \times k)$

8 Bucket sort, Radix sort ← If I describe it then you should be able to say that's a Radix sort.

0-1 Knapsack ~~protolem~~ is NP complete.

0-1 knapsack problem with integer capacity $\Theta(nk)$ where k is capacity of knapsack. k could be much bigger than n so not really linear.

fractional knapsack once you sort by nonincreasing order (value per weight it takes $O(n)$ with greed approach.

matrix multiplication algorithm Divide and conquer sort of like big integer multiplication in two dimensions n is number of rows number of columns

$$T(n) = 8\,T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$T(n) = \Theta(n^3)$$

Using an algoritham $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$

Straussen's algorithm $T(n) = \Theta(n^{\log_2 7})$

focus on the graphs

Chapter 17, there is another version of knapsack problem called knapsack problem

Also Pg 511