Computational Complexity - Turing and RAM Machines

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo . felicioni @ polimi . it

May 10, 2021

[1] Mostly based on Nicholas Mainardi's material.

What is complexity analysis?

Evaluate how efficient is something

- We want to understand how much the size of the input impacts on computation time
- We are interested in an asymptotic behaviour: we are considering $n \to \infty$
- Usually, we analyse the upper bound for the running time (bounds on the used space can be useful too!)
- Usually, we drop constant terms, as they are "negligible" since we take the running time with an input size $n \to \infty$[a]

[a] this approximation should be taken with a proper grain of salt in practice: 10^{15} is constant, but it's not exactly negligible. However large constants are quite rare

Asymptotic Bounds

How to determine if a function is asymptotically negligible?

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \Omega(g(n)) \wedge f(n) \notin \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)) \wedge f(n) \notin \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0 \Rightarrow f(n) \in \Theta(g(n))$

Examples

- $f(n) = e^n, g(n) = n2^n : \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^{\log_2(e^n)}}{2^{\log_2(n2^n)}} = \lim_{n\to\infty} \frac{2^{n\log_2(e)}}{2^{n\log_2(2)+\log_2(n)}} = \lim_{n\to\infty} 2^{n\log_2(e)-n-\log_2(n)} = 2^\infty = \infty$
- $f(n) = \log_e(n), g(n) = \log_2(n) : \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_e(n)}{\log_2(n)} = \lim_{n\to\infty} \frac{\frac{\log_2(n)}{\log_2(e)}}{\log_2(n)} = \frac{1}{\log_2(e)} = \log_e(2)$

A classic: TM recognizing $a^n b^n$

Counting in unary

- We consider a k-tapes TM recognizing $L = a^n b^n, n \geq 0$: how many steps does it take to do so?
- Consider the common algorithm which employs a memory tape as a stack and emulates a DPDA (writes one marker symbol for each a, deletes one for each b)
- It performs an action for each letter of the string, and, when the input has been scanned, the computation either accepts or rejects
- The complexity is $O(2n)$ (considering n as the number of a in the string)
- Consequently, dropping the constant we get $O(n)$ as the asymptotic complexity

Still $a^n b^n$

Single Tape Turing Machine

- In case we have a single tape machine, we need to write the markers on the only tape we have
- Thus, for each `a` on the input tape, we should scroll back to the initial position, write a marker, and go back to the next `a`
- To count the `b`s the procedure is analogous: scroll back, erase a symbol, scroll forward to the next `b`
- Summing up, there are
  - While reading `a`s, we need to scroll back and forth 2 cells more: $\sum_{i=0}^{n-1} 3 + 4i = 3n + 2n(n-1)$
  - n sweeps performed over the tape cells containing the `a`s and the `b`s and the counters being erased ($2n+1$ cells at most): $n(2(2n+1)) = 4n^2 + 2n$
- The total complexity is $4n^2 + 2n^2 + 3n$, that is, $O(n^2)$

A more compact way of counting $a^n b^n$

Binary counting on a k-tapes TM

- Suppose we want to save tape space and decide to count the $\mathtt{a}$s in binary, does the time complexity change?
- When counting in unary, incrementing the counter is a constant time action (write one symbol on the memory tape)
- Counting in binary, we need to actually perform the addition, thus we need to take care of the possible carries
- The worst case for the carry propagation needs to rewrite all the $\log_2(n)$ (binary) digits of the number
- We thus have an total complexity of $O(n \log_2(n))$ (n additions, taking at most $\log_2(n)$ write actions each)

A more compact way of counting $a^n b^n$

Binary counting on a single tape TM

- Same problem as before, this time using a single tape TM
- For each $\mathtt{a}$, we need to rewind the head, and add 1 to the binary counter which we keep before the input string
- As there are n $\mathtt{a}$s, and incrementing a counter takes at most $log_2(\mathrm{counter})$, the cost is $O(n(n+\log_2(n))) = O(n^2+n\log(n))$
- The same thing can be said of decrementing the counter for each $\mathtt{b}$ on the tape, thus the total complexity is $O(n^2+n\log(n))=O(n^2)$
- Space complexity? Still $O(n)$, since the space complexity on a single tape TM is always $\Omega(n)$ (due to the slots of the tape employed for the input string)
- Thus, this strategy does not alter asymptotically both time and space complexities

Are FSA faster?

Is there a problem solved faster by an FSA than by a TM?

- This is equivalent to saying "is there a language recognised by an FSA, asymptotically faster than a TM?"
- No: I can always emulate an FSA with a TM through simply blocking all the tape heads except the one on the input
- The same formulation holds when comparing a PDA with a k-tapes TM as one tape can be used to emulate the stack

Analysing the RAM machine

A not-so-abstract machine

- Memory only machine: all the memory cells can be accessed in $O(1)/O(log(n))$ (linear/logarithmic cost criterion)
- By default the first operand is the one stored in memory location 0 when the instruction is executed
- Minimal RISC assembly with conditional jumps, `halt` instruction stops computation
- Operations employing an immediate value are denoted by an $=$ symbol (e.g. `add=`,`sub=`)

Analysing the RAM machine

Analyzing the computation of $n^n, n > 0$ on a RAM machine

```
READ  1          \
LOAD= 1           |   This code is run only once
STORE 3           |-> Loads the value of n
LOAD  1          /    in Mem[1]: O(1) cost
LOOP: STORE 2    \
LOAD  1           |   This loop is run a number
MUL   3           |-> of times depending on Mem[2],
STORE 3           |   a copy of n, so it runs n
LOAD  2           |   times: O(n) cost
SUB=  1           |
JGZ LOOP         /
WRITE 3           | -> Writes n^n: O(1) cost
HALT
```

Time complexity is mainly due to the loop $\Rightarrow O(n)$

Logarithmic Cost vs Constant Cost

With RAM machine, sometimes we cannot consider a single instruction to be executed in constant time!

- Generally, an arithmetic unit can add in a single step only a limited number of bits
- Or a memory cell can store a limited number of bits, thus multiple stores may physically occur for each `STORE`
- Therefore, even simple operations may take a time dependent on the size of the data (i. e. the number of bits)
- Generally, each machine can perform operations in a single step on integers which can be represented by w bits, where w is the word size (e.g. $w = 32$ for 32 bits CPU)

$$\Downarrow$$

Logarithmic cost means that each operation is about $O(\log_w(n)) = O(log(n))$, since the logarithm is asymptotically independent from the chosen base

Analysing the RAM machine: Logarithmic Cost

Analyzing the computation of $n^n, n > 0$ on a RAM machine

```
READ  1            \
LOAD= 1             |   This code is run only once
STORE 3             |-> Loads the value of n
LOAD  1            /    in Mem[1]: O(log(n)) cost
LOOP: STORE 2      \    This loop is run a number
LOAD  1             |   of times depending on Mem[2],
MUL   3             |   a copy of n, so it runs n
STORE 3             |   times, each one computing
LOAD  2             |-> n^i * n, maximized by n^n * n:
SUB=  1             |   O(n log(n^n) log(n)) = O(n(n log^2(n))) =
JGZ LOOP           /    = O(n^2 log^2(n))
WRITE 3             | -> Writes n^n: O(log(n^n)) =O(n log(n))
HALT
```

Time complexity is mainly due to the loop $\Rightarrow O(n^2 \log^2(n))$

Logarithmic cost criteria: when?

RAM machines for Languages

- Does it make sense to use log cost criteria to recognize a Regular language with a RAM machine?
  - No, since it is equivalent to constant cost. REG languages can be recognized by RAM machine with finite memory independent from the length of the input string. Indeed, it is sufficient one memory cell to read the input character and one to store the current state; the recognizer can be fully specified in the program code (e.g. with a giant `switch/case`)
- Does it make sense to use log cost criteria for CF languages?
  - Yes: we need arbitrarily long counters ...
    - $\hookrightarrow$ But at most $O(\log(n))$ additional cost for memory operations $\Rightarrow$ not so interesting
- Does it make sense to use log cost criteria for recursively enumerable (RE) languages?
  - Yes: we may use an arbitrary amount of memory

Doubling a binary encoded number

Single tape TM

- Input: n, a binary encoded number; output: $2n$ binary encoded
- To double a number in binary, it is sufficient to add an extra 0 as the least significant digit
- If the input head starts on the least significant digit: $O(1)$: it's just "move by one place and write 0"
- If the input head starts on the most significant digit: $O(\log_2(n))$: reach the least significant digit, and then add the extra 0
- Does it change anything if the TM has k-tapes? In the former case, we need to write the output, thus $O(\log_2(n))$ complexity

Doubling a binary encoded number

RAM Machine

The program on a RAM machine is quite straightforward:

```
READ 0
MUL= 2
WRITE 0
```

- Each of these instructions need to write $O(\log_2(n))$ bits in the memory cell 0 (the multiplication is with a constant, thus its cost is $O(\log(n)\log(2)) = O(\log(n))$)
- Therefore, the time complexity is $O(\log_2(n))$, using logarithmic cost

Sorting Quickly

Sort a lowercase alphabetic string with a single tape TM

- Strategy: scan n times the string, pick the lowest character and output it, erasing it from the input
- We need to store the lowest character in the tape
- Thus, for each of the n scans, for each one of the n characters we need to move to the cell storing the lowest character in order to compare them $\Rightarrow O(n^3)$ complexity
- Do we really need the tape to store the lowest character?
- We can use a state of the TM to store this information!
- The possible lowest characters are 26, thus we can store this value employing 26 states
- The complexity becomes $O(n^2)$, since we no longer move to the cell storing the lowest character for each character read

Sorting Quickly

Sort a lowercase alphabetic string with a k-tapes TM

- Smarter method: employ a 26-tapes TM, scan once the input, writing a symbol on one of the tapes, employing them as 26 different counters for all the letters of the alphabet
- Then, output a letter for each symbol on the 26 memory tapes, starting from the tape representing `a`s
- Complexity: one scan of the input string ($O(n)$) plus one re-emission of the sorted input ($O(n)$) $\Rightarrow$ Total: $O(n)$
- What if we have less tapes? Scan the string 26 times: the first scan copies all the a in the output tape, the second scan all the b and so on until the last scan $\Rightarrow$ $O(26n) = O(n)$
- What about RAM machine? incrementing the counters for each letter costs $O(\log(n))$, thus the complexity is $O(n \log(n))$

Writing RAM Programs

Write a RAM program which computes the function $f : \mathbb{N} \setminus 0 \mapsto \mathbb{N}$:

$$f(x) = \begin{cases} \sqrt{x^x} & \text{if x is even} \\ x^3 & \text{otherwise} \end{cases}$$

High level algorithm:

$\mathrm{F}(x)$

```
1  res ← 1
2  if x mod 2 = 0
3      then exp = x/2
4           while exp > 0
5               do res ← res * x
6                  exp − −
7      else  res ← x * x * x
8  return res
```

Writing RAM Programs

RAM Program

```
READ 1                  STORE 2
LOAD 1                  LOAD 3
DIV= 2                  MUL 1
MUL= 2                  STORE 3
SUB 1                   LOAD 2
JLZ ODD                 JUMP LOOP
LOAD 1                  ODD: LOAD 1
STORE 3                 MUL 1
LOAD 1                  MUL 1
DIV= 2                  STORE 3
LOOP: SUB= 1            END: WRITE 3
JZ END                  HALT
```

Writing RAM Programs

Estimating Complexity: Constant Cost

- Time complexity is dominated by the loop
- The loop is run $\frac{x}{2}$ times
- Each loop time complexity is $O(1)$
- Thus, time complexity is $\frac{x}{2} \in O(x)$

Estimating Complexity: Log Cost

- Time complexity is dominated by the loop
- The loop is run $\frac{x}{2}$ times
- Each loop computes $x^{i-1} * x$, with i being at most $\frac{x}{2}$
- Each loop body takes $O(\log(x^{\frac{x}{2}}) \log(x))$, because of the multiplication
- Thus, $T(n) \leq \frac{x}{2} \log(x^{\frac{x}{2}}) \log(x) \in O(x^2 \log^2(x))$

Writing RAM Programs

Write a RAM program which determines if a number is prime:

RAM Program

```
READ 1
LOAD= 1
SUB 1
JZ COMPOSITE
LOAD= 2
LOOP:STORE 2
LOAD 1
SUB 2
JLEZ PRIME
LOAD 1
DIV 2
MUL 2
SUB 1
JZ COMPOSITE
LOAD 2
ADD=1
JUMP LOOP
PRIME: WRITE= 1
HALT
COMPOSITE: WRITE= 0
HALT
```

Writing RAM Programs

Estimating Complexity: Constant Cost

- The time complexity is dominated by the loop
- This is executed at most $n - 2$ times
- The loop body has $O(1)$ cost
- Total complexity is $O(n)$

Estimating Complexity: Logarithmic Cost

- The only difference is that the loop body has cost $O(\log^2(n))$, because of the division/multiplication
- Total complexity is $O(n \log^2(n))$
- In this case, logarithmic cost is *pointless*: it simply adds a $\log^2(n)$ factor, but no more, since we are working with numbers no bigger than n

Writing RAM Programs

Do we need to do all those divisions?

- Is it possible to reduce the complexity of the previous algorithm?
- Notice that, if n is composite, through trying all the possible dividers we will find the smallest of them first
- How big can the smallest divider be at most? No bigger than $\lfloor\sqrt{n}\rfloor$ as otherwise the other factor would be smaller than it!
- Rewriting the previous algorithm exiting the loop at $\lfloor\sqrt{n}\rfloor + 1$!
- Problem in rewriting: we need to compute $\lfloor\sqrt{n}\rfloor + 1$ before
- We can easily do it in $\lfloor\sqrt{n}\rfloor + 1$ time by iteratively trying all possible integers i until $i^2 > n$
- Complexity: $2(\lfloor\sqrt{n}\rfloor + 1) \in O(\sqrt{n})$ constant cost, $O(\sqrt{n}\log(\sqrt{n})\log(\sqrt{n})) = O(\sqrt{n}\log^2(n))$ logarithmic cost

Writing RAM Programs

RAM Program

```
READ 1
LOAD= 1
SUB 1
JZ COMPOSITE
LOAD= 2
LOOP: STORE 3
MUL 3
SUB 1
JGZ COMPUTED
LOAD 3
ADD= 1
JUMP LOOP
COMPUTED: LOAD= 2
LOOP2:STORE 2
SUB 3
JZ PRIME
LOAD 1
DIV 2
MUL 2
SUB 1
JZ COMPOSITE
LOAD 2
ADD=1
JUMP LOOP2
PRIME: WRITE= 1
HALT
COMPOSITE: WRITE= 0
HALT
```

Writing RAM Programs

We can avoid to compute $\lfloor \sqrt{n} \rfloor + 1$ separately ...

Improved Version

```
READ 1                      MUL 2
LOAD= 1                     SUB 1
SUB 1                       JZ COMPOSITE
JZ COMPOSITE                LOAD 2
LOAD= 2                     ADD=1
LOOP: STORE 2               JUMP LOOP
MUL 2                       PRIME: WRITE= 1
SUB 1                       HALT
JGZ PRIME                   COMPOSITE: WRITE= 0
LOAD 1                      HALT
DIV 2
```

Same asymptotic complexity($O(\sqrt{n})$), but a bit better in terms of memory usage and code size

Efficiently Computing $\sqrt{n}$

Let's focus on the computation of $\lfloor \sqrt{n} \rfloor + 1$:

```
READ 1                    LOAD 2
LOAD= 1                   ADD= 1
LOOP: STORE 2             JUMP LOOP
MUL 2                     COMPUTED: WRITE 2
SUB 1                     HALT
JGZ COMPUTED
```

- The loop is executed $O(\sqrt{n})$ times
- Due to the multiplication, the i-.th iteration costs $O(\log^2(i))$, thus the overall complexity is $\sum_{i=1}^{\sqrt{n}} \log^2(i)$
- Let's estimate upper and lower bounds for this summation ...

Efficiently Computing $\sqrt{n}$

- Upper bound: $\sum_{i=1}^{\sqrt{n}} \log^2(i) < \sum_{i=1}^{\sqrt{n}} \log^2(\sqrt{n}) = \sqrt{n}\log^2(\sqrt{n}) = \frac{1}{4}\sqrt{n}\log^2(n) = O(\sqrt{n}\log^2(n))$
- Lower bound:
  $\sum_{i=1}^{\sqrt{n}} \log^2(i) > \sum_{i=1}^{\frac{\sqrt{n}}{2}-1} \log^2(i) + \sum_{i=\frac{\sqrt{n}}{2}}^{\sqrt{n}} \log^2(\frac{\sqrt{n}}{2}) >$
  $\sum_{i=\frac{\sqrt{n}}{2}}^{\sqrt{n}} \log^2(\frac{\sqrt{n}}{2}) = \frac{\sqrt{n}}{2}\log^2(\frac{\sqrt{n}}{2}) = \Omega(\sqrt{n}\log^2(n))$

In conclusion:

1. $\sum_{i=1}^{\sqrt{n}} \log^2(i) = O(\sqrt{n}\log^2(n))$
2. $\sum_{i=1}^{\sqrt{n}} \log^2(i) = \Omega(\sqrt{n}\log^2(n))$

$$\Downarrow$$

Computing $\lfloor\sqrt{n}\rfloor + 1$ with this strategy costs $\Theta(\sqrt{n}\log^2(n))$

Efficiently Computing $\sqrt{n}$

Can we do better to compute $\lfloor\sqrt{n}\rfloor + 1$?

- The previous algorithm computes $i^2, i = 1, \ldots \lfloor\sqrt{n}\rfloor + 1$ with a multiplication
- Key property of perfect squares: $\forall i (i+2)^2 - (i+1)^2 = (i+1)^2 - i^2 + 2 \implies (i+2)^2 = (i+1)^2 + (i+1)^2 - i^2 + 2$
- High-level algorithm based on this property:

$\textsc{SQRT}(n)$

```
1  inc ← 1
2  acc ← 1
3  res ← 1
4  while acc ≤ n
5        do inc ← inc + 2
6           acc ← acc + inc
7           res + +
8  return res
```

Writing RAM Programs

RAM Program

```
READ 1
LOAD= 1
STORE 2
STORE 3
LOOP: STORE 4
SUB 1
JGZ END
LOAD 2
ADD= 1
STORE 2
LOAD 3
ADD= 2
STORE 3
ADD 4
JUMP LOOP
END: WRITE 2
HALT
```

Computational complexity:

- $O(\sqrt{n})$ iterations which execute only LOAD, ADD and SUB of numbers as big as $i^2 \Rightarrow \sum_{i=1}^{\sqrt{n}} \log(i^2) = O(\sqrt{n}\log(n))$
- we save a $\log(n)$ factor by avoiding multiplications!

Quick questions

Can I recognize ...

- Any deterministic CF language with a k-tapes MT in $\Theta(n)$ (constant cost criterion)
  - Yes: just emulate the corresponding recognizer DPDA employing a tape as stack
- Any deterministic CF language with a single tape MT in $\Theta(n)$ (constant cost criterion)
  - No: it needs to move the head back and forth on the single tape to emulate the stack
- Any deterministic CF language with a RAM machine in $\Theta(n)$ (logarithmic cost criterion)
  - No: Take $a^n b^n$: the RAM will need an unbounded counter to store n, thus recognizing it in $O(n \log(n))$

Quick questions

Can I recognize ...

- Any REG language with a single tape TM in $\Theta(n)$
  - Yes: the state of the computation is memorised in the control automaton, no need to rewind the input tape
- Any REG language with a RAM machine in $\Theta(n)$
  - Yes: same as before, just need to store the state, which is an integer whose size is independent from n, the control automaton will do the whole job
- No CF language in less than $\Theta(\log(n))$ *spatial* complexity with a k-tapes TM
  - No: take as a simple counterexample $L = \{a\}$. I can recognise it in $O(k)$ space.