Lecture 10 - Complexity Theory

Eric A. Autry

Last time: Computer Memory

This time: Complexity Theory

Next time: Review of Data Structures

Midterm regrades due today in class.

HW 4 to be assigned tonight and due next Thursday the 18th.

The Stack

Where can each function store its local memory (the local variables it uses during computation)?

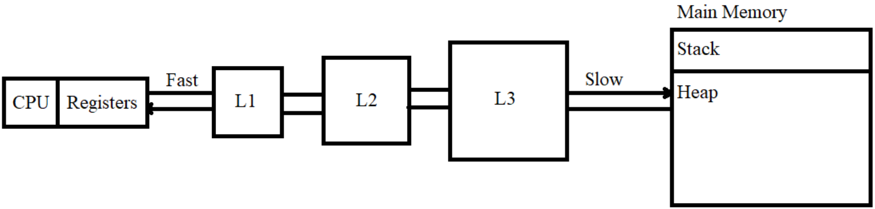
- Each function is allocated space on the stack, a reserved area near the top of the tape.
- The stack is Last In First Out order for allocation/deallocation. This requires storing just one pointer for where we are in the stack. Fast allocation.
- Want to change a variable size dynamically? You cannot (would collide with another function's space on the stack).

The Heap

Store dynamically (changing in size) allocated memory on the heap, a large section of the tape further from the head.

- Each dynamically allocated object could change in size.
- So need complex memory management to ensure data is not overwritten, and garbage collection to free up space on the tape.
- Slower memory access due to the complex bookkeeping.

Caching



We can even allow for multiple levels of Caching, with increased access times as we get closer to the CPU.

Caching

Why is this important?

Let's look at a matrix stored as a 2D array:

$$A = [\,[1,2,3],\ [4,5,6],\ [7,8,9]\,]$$

Consider the following code:

```
for r in rows:
    for c in columns:
        print A[r][c]
```

```
for c in columns:
    for r in rows:
        print A[r][c]
```

Which will run faster? What gets loaded into Cache when?

The code on the right wipes the Cache and loads a new row on each iteration of the inner loop!

Measuring Efficiency

We now know how to implement programs, store data, and perform basic operations.

Now we can focus on the implementation itself: we want efficient solutions.

We measure this through what is called the **computational complexity** (or just ‘complexity’) of the algorithm.

- This is typically reported by giving the runtime T as a function of the input size n, i.e., $T(n)$.
- We like grouping algorithms into complexity classes: groups of algorithms that are all (in some sense) equivalently difficult.

Measuring Efficiency

This is typically reported by giving the runtime T as a function of the input size n, i.e., $T(n)$.

How do we measure T and n? What units do we use?

- We can measure the input size n as:
  - The number of bits required to store the input on the tape,
  - The number of input objects (i.e., number of characters in the input string, length of the input vector, number of rows in an input matrix, etc).

Measuring Efficiency

This is typically reported by giving the runtime T as a function of the input size n, i.e., $T(n)$.

How do we measure T and n? What units do we use?

- We can measure the runtime T as:
  - The number of basic operations required to perform the given task.
  - Why not just report time in seconds?

    Different computers will run at different speeds. The answer would be specific to a single computer, which does not provide enough info for a potential user.

Basic Operations

What do we count as basic operations?

For a Turing Machine algorithm:

- Read from tape.
- Write to tape.
- Move left/right.

Basic Operations

What do we count as basic operations?

For more standard algorithms, we are concerned with operations performed on fixed length data (int, float, char, etc).

A basic operation is:

- FLOP (floating point operation)
  - Read/write a single piece of data from/to memory.
  - Compare two pieces of data.
  - Arithmetic (add, subtract, multiply, divide, exponentiate, mod, etc)

We say that these operations take ‘constant time’ each time they are performed.

Basic Operations

We typically don't worry about memory management (allocation/deallocation, caching), which is machine dependent.

We instead assume that when the algorithm is actually implemented on a specific machine, the programmer will do a good enough job

i.e., ensure that read/write is actually constant time.

Different machines and different implementations will result in different behaviors. Our goal is to report complexity in a general form that is free from these specifics.

Basic Operations

While we don't worry about machine-level memory management, we will consider the complexity of storing data in specific data structures.

We exchange complex allocation for the ability to access data following defined rules.

So read/write operations can affect the overall algorithmic complexity when we use certain data structures (more on those next time).

Ex:

- Hash Table: can access data associated with given keys.
- Binary Search Tree (BST): data stored in specified order
- Min-Heap: can rapidly access the minimum element

Big-O Notation

The most common way to report algorithmic complexity is through the use of Big-O notation:

Consider two functions $f(n)$ and $g(n)$. We say that $f \in O(g)$ if there exists constants $k > 0$ and N such that

$$|f(n)| \;\leq\; k \cdot |g(n)|, \qquad \text{for all } n \geq N.$$

Ex:

$$f(n) = 3n^2 - 2n + 1 \quad \in \quad O(n^2),$$

because

$$3n^2 - 2n + 1 \;\leq\; 3 \cdot n^2, \qquad \text{for } n \geq 1.$$

We ignore the coefficients and all lesser terms!

Our primary concern is behavior of the highest cost term as the input size grows large: the asymptotic complexity.

Complexity Example $\#1$

What is the algorithmic complexity of the palindrome detecting Turing Machine?

1. Read the first letter and cross it off (replace it with an '$\sqcup$').
2. Move to the end of the string that is not yet crossed off.
   - If there was only one letter left, accept because a single letter is always a palindrome (or it was the unimportant middle letter).
3. Read the last letter and cross it off (replace it with an '$\sqcup$').
   - If the letters were not the same, halt and reject.
4. Move to the start of the tape that is not yet crossed off.
   - If there are no letters left, accept because the string had an even length and was a palindrome.
5. Go to step 1.

Cost per iteration: 2 reads, 2 writes, 3 conditionals,
And move across the tape twice...

Complexity Example $\#1$

What is the algorithmic complexity of the palindrome detecting Turing Machine?

Cost per iteration: 2 reads, 2 writes, 3 conditionals (constants)
And move across the tape twice... (variable costs)

Let's count the total sum of the constant costs per iteration.

- There are $n/2$ iterations because we cut off two letters each time.
- So this contributes a total cost of: $\frac{7}{2}n$

How to count the total cost of the movements across the tape?

Complexity Example $\#1$

How to count the total cost of the movements across the tape?

The algorithm works by crossing off a letter, then moving across the tape.

So, when we cross off:

- the 1st letter, we make n moves.
- the 2nd letter, we make $n-1$ moves.
- the 3rd letter, we make $n-2$ moves.
- etc

Total movement:

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{k=1}^{n} k$$

Complexity Example $\#1$

Total movement:

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{k=1}^{n} k$$

Carl Friedrich Gauss:

If n is even:

$$\begin{cases} (n) + 1 = & n+1 \\ (n-1) + 2 = & n+1 \\ (n-2) + 3 = & n+1 \\ \quad \ldots \end{cases}$$

There are $n/2$ of these pairs, so:

If n is odd:

$$\begin{cases} (n) + 0 = & n \\ (n-1) + 1 = & n \\ (n-2) + 2 = & n \\ \quad \ldots \end{cases}$$

There are $(n+1)/2$ of these pairs, so:

$$\sum_{k=1}^{n} k = n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \in O(n^2)$$

Complexity Example $\#1$

What is the algorithmic complexity of the palindrome detecting Turing Machine?

$$T(n) = \frac{7}{2}n + \frac{1}{2}n^2 + \frac{1}{2}n \in O(n^2).$$

Can we do better?

What if we try to make 5 comparisons at once?

What is the best we can do?

How can we get that runtime? Multitape...

Complexity Example $\#2$

What is the algorithmic complexity of the TM that can detect strings with lengths that are powers of 3?

- Scan through the input and cross off two of every three 0s.
- Repeat until only a single 0 left (or reject).

For each iteration we require moving all the way across the original input of size n, so each iteration gives us $O(n)$ cost.

How many iterations?

- Each iteration, the number of unmarked 0s is divided by 3.
- How many times can you repetitively divide a number by 3?

$$\log_3(n)$$

Total Cost:

$$O\left(n \log_3 n\right) = O(n \log n)$$

Important Note about Logs

Base Change Formula:

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

But we can treat $1/\log_c(b)$ as a constant, to write

$$\log_a(x) = \left(\frac{1}{\log_b(a)}\right)\log_b(x) = C\log_b(x) \in O(\log a)$$

So we see that logarithms of any base are all in the same complexity class.

Complexity Classifications

While we often report the Big-O runtime for an algorithm, there are actually five ways to classify the complexity.

Consider two functions $f(n)$ and $g(n)$. We say that:

- $f \in O(g)$ (Big-O, 'f is bounded above by g')

  If there exists constants $k > 0$ and N such that

  $$|f(n)| \;\leq\; k \cdot |g(n)|, \qquad \text{for all } n \geq N.$$

- $f \in o(g)$ (little-o, 'f is strictly bounded above by g')

  If for all constants $k > 0$, there exists a constant N such that

  $$|f(n)| \;<\; k \cdot |g(n)|, \qquad \text{for all } n \geq N.$$

Complexity Classifications

While we often report the Big-O runtime for an algorithm, there are actually five ways to classify the complexity.

Consider two functions $f(n)$ and $g(n)$. We say that:

- $f \in \Omega(g)$ (Big-Omega, 'f is bounded below by g')

  If there exists constants $k > 0$ and N such that

  $$|f(n)| \ \geq \ k \cdot |g(n)|, \qquad \text{for all } n \geq N.$$

- $f \in \omega(g)$ (little-omega, 'f is strictly bounded below by g')

  If for all constants $k > 0$, there exists a constant N such that

  $$|f(n)| \ > \ k \cdot |g(n)|, \qquad \text{for all } n \geq N.$$

Complexity Classifications

While we often report the Big-O runtime for an algorithm, there are actually five ways to classify the complexity.

Consider two functions $f(n)$ and $g(n)$. We say that:

- $f \in \Theta(g)$ (Theta, 'f is bounded both above and below asymptotically by g')

  If there exists constants $k_1 > 0$, $k_2 > 0$, and N such that

  $$k_1 \cdot |g(n)| \;\leq\; |f(n)| \;\leq\; k_2 \cdot |g(n)|, \qquad \text{for all } n \geq N.$$