Cracking the GATEs of IITs/IISc

Manas Thakur

PACE Lab, IIT Madras

Content Credits

- *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Hopcroft et al.
- *Algorithms*, TMH edition. Dasgupta et al..
- *Principles of Compiler Design*. Alfred V. Aho and Jeffrey D. Ullman.
- *https://en.wikipedia.org*
- *https://images.google.com*

Outline

- Problems
- Algorithms
- Programming
- Translation
- Research in Systems

Ways to begin a talk: The Overdone Overview

A Simple Problem

- Design a machine to determine whether a given program P1 prints “Hello World!”.

```
int main() {
    printf(“Hello World!”);
    return 0;
}
```

A Simple Problem (Cont.)

```
int main() {
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; ++x) {
            for (y=1; y<=total-x-1; ++y) {
                z = total-x-y;
                if (exp(x,n)+exp(y,n) == exp(z,n)) {
                    printf("Hello World!");
                }
            }
        }
        ++total;
    }
    return 0;
}
```

```
int exp(int i, int n) {
    int ans, j;
    ans = 1;
    for (j=1; j<=n; ++j) {
        ans *= i;
    }
    return ans;
}
```

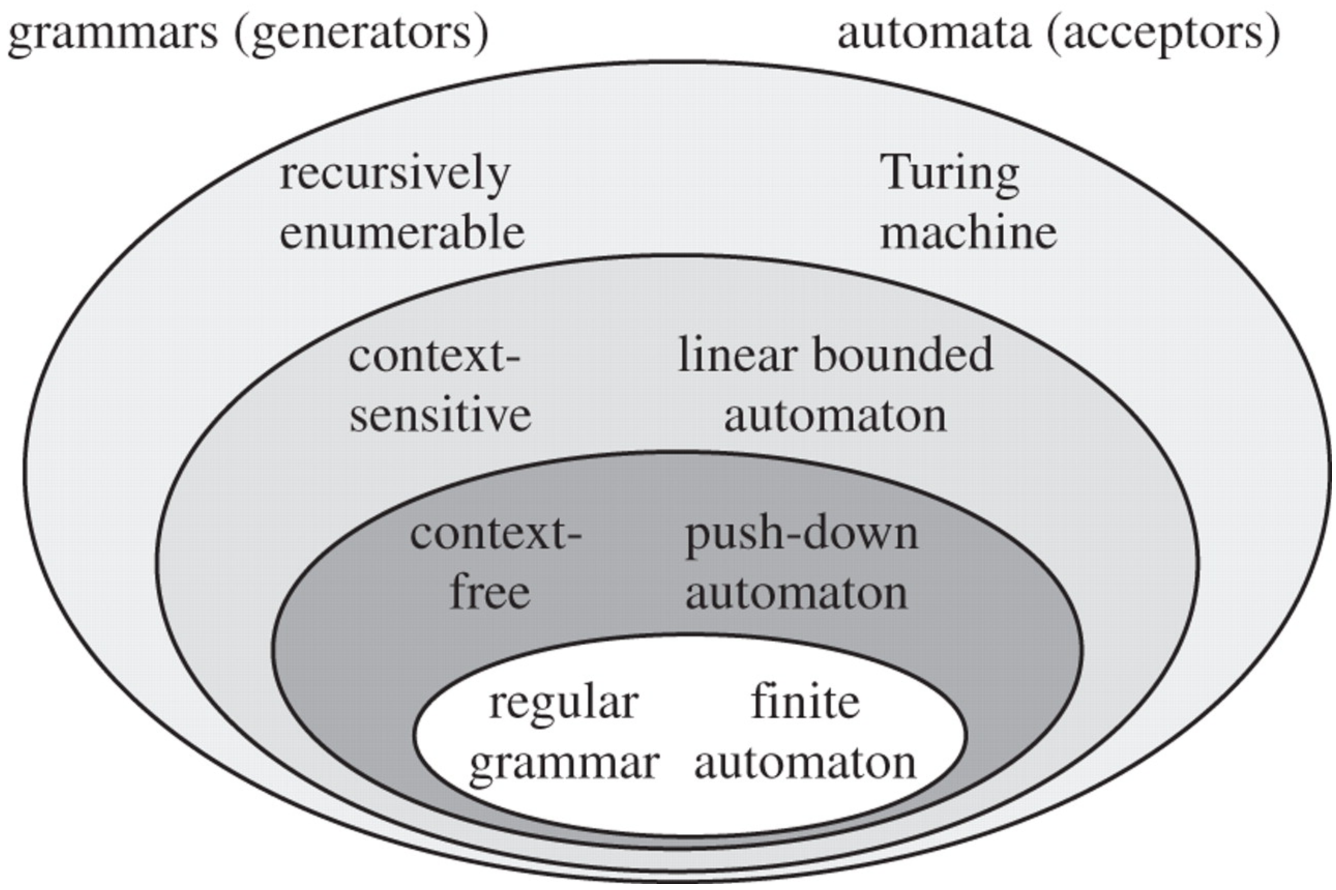
Expressing problems as language-membership tests

- **Step 1:** Represent problem instances as *strings* over a finite alphabet.
  - Our program P1 is essentially a string of characters.
- **Step 2:** Design a machine M1 that accepts *valid* strings.
  - Outputs *yes*, if P1 prints "Hello World!".
  - Outputs *no*, if P1 does not print "Hello World!".
- The language accepted by M1 is:

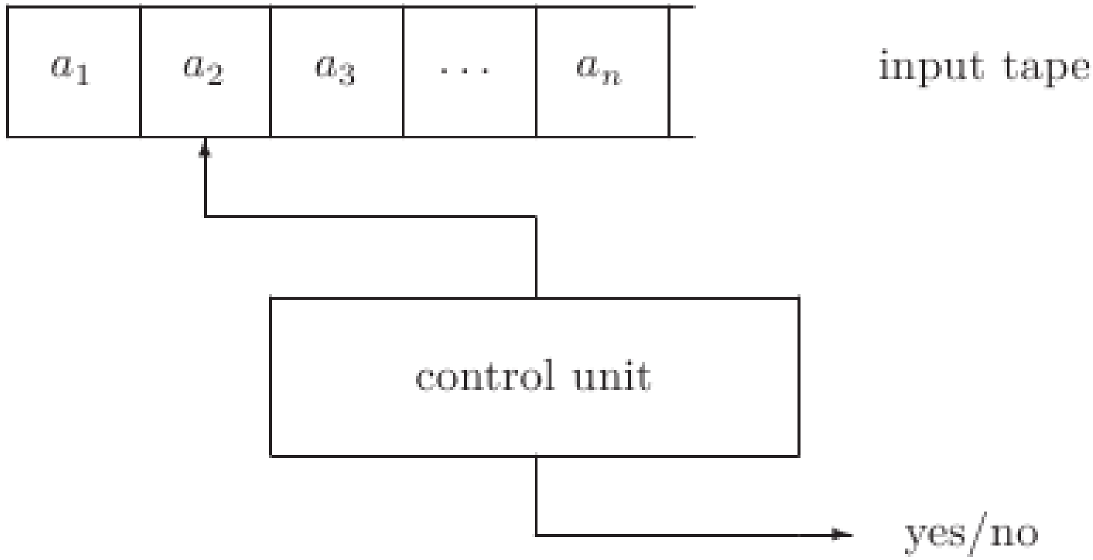
```
L(M1) = { w | w is a program that prints "Hello World!" }
```

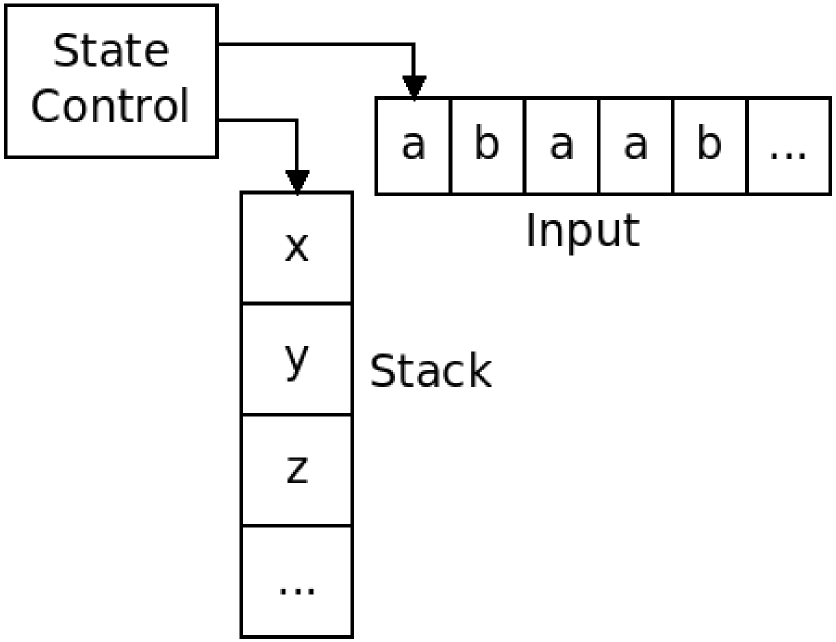
- If M1 always terminates and prints *yes* or *no*, it ***decides*** P1; else it ***recognizes*** P1.

The Chomsky Hierarchy of Languages

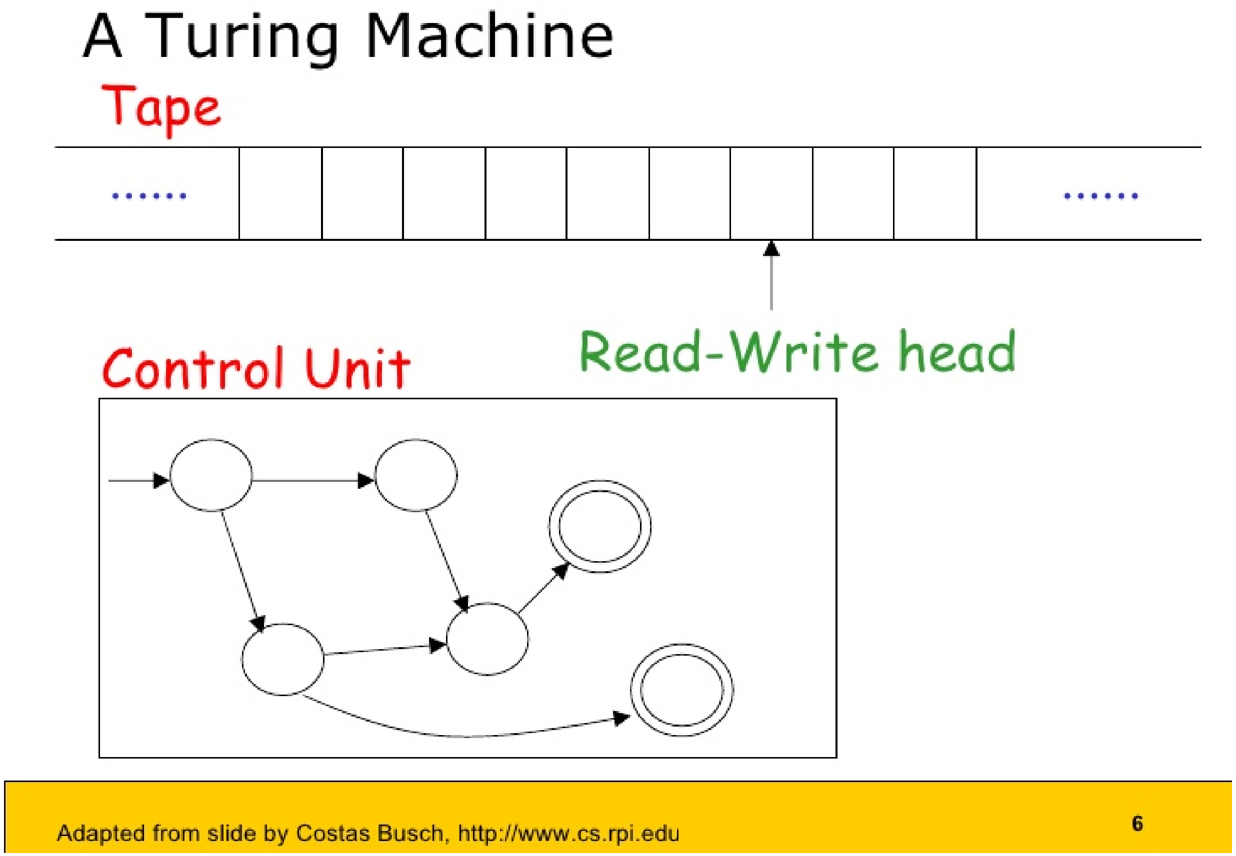


DFA and PDA: A Quick Recap





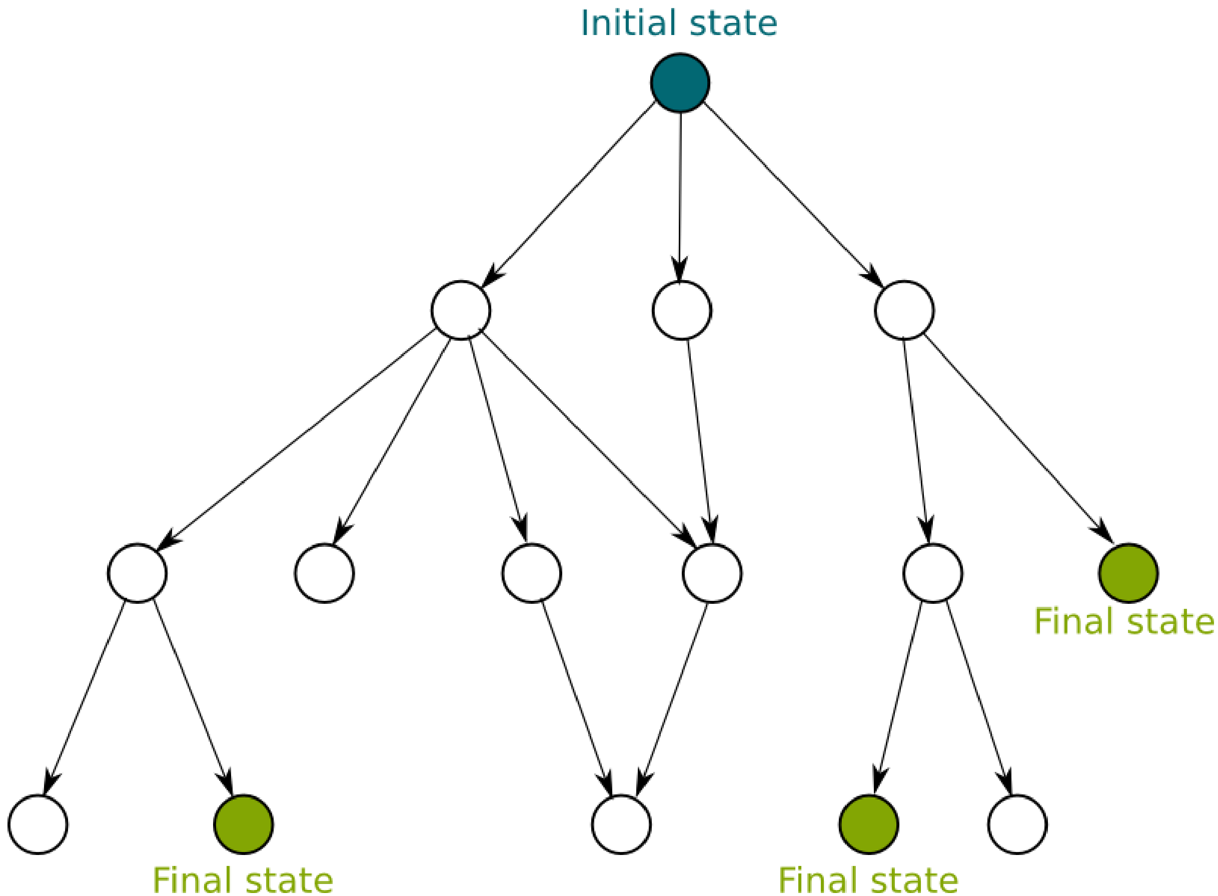
Turing Machines



Turing Machines and Algorithms

- **Church-Turing Thesis:** Every algorithm can be realized as a Turing Machine.
- A multitape-TM is equivalent to a single-tape TM.
- A TM can simulate a computer.
- A computer with an *infinite tape* can simulate a TM.
- Turing Machines are more powerful than modern day computers!!
- What about Nondeterministic Turing Machines?

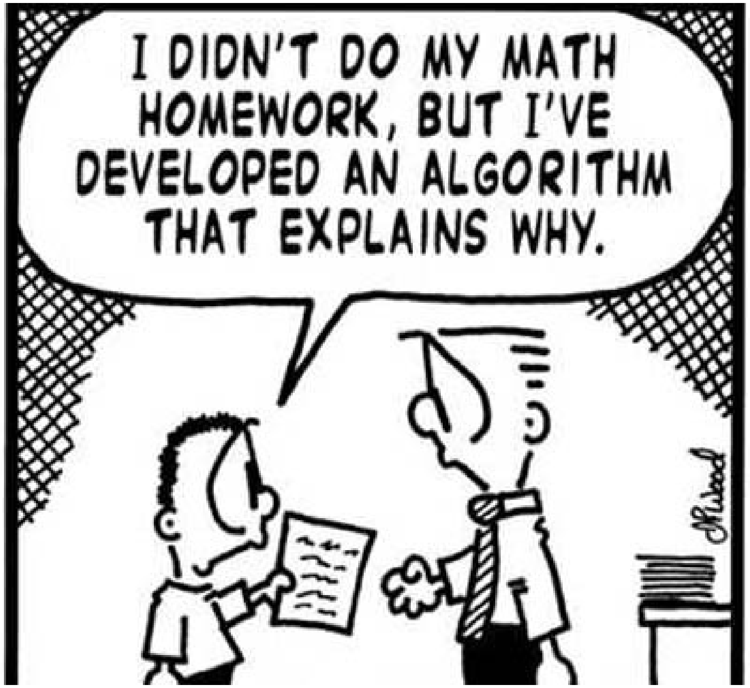
Non-determinism: The Power of Guessing



An NTM is equivalent in power to a DTM; but what about the amount of time?

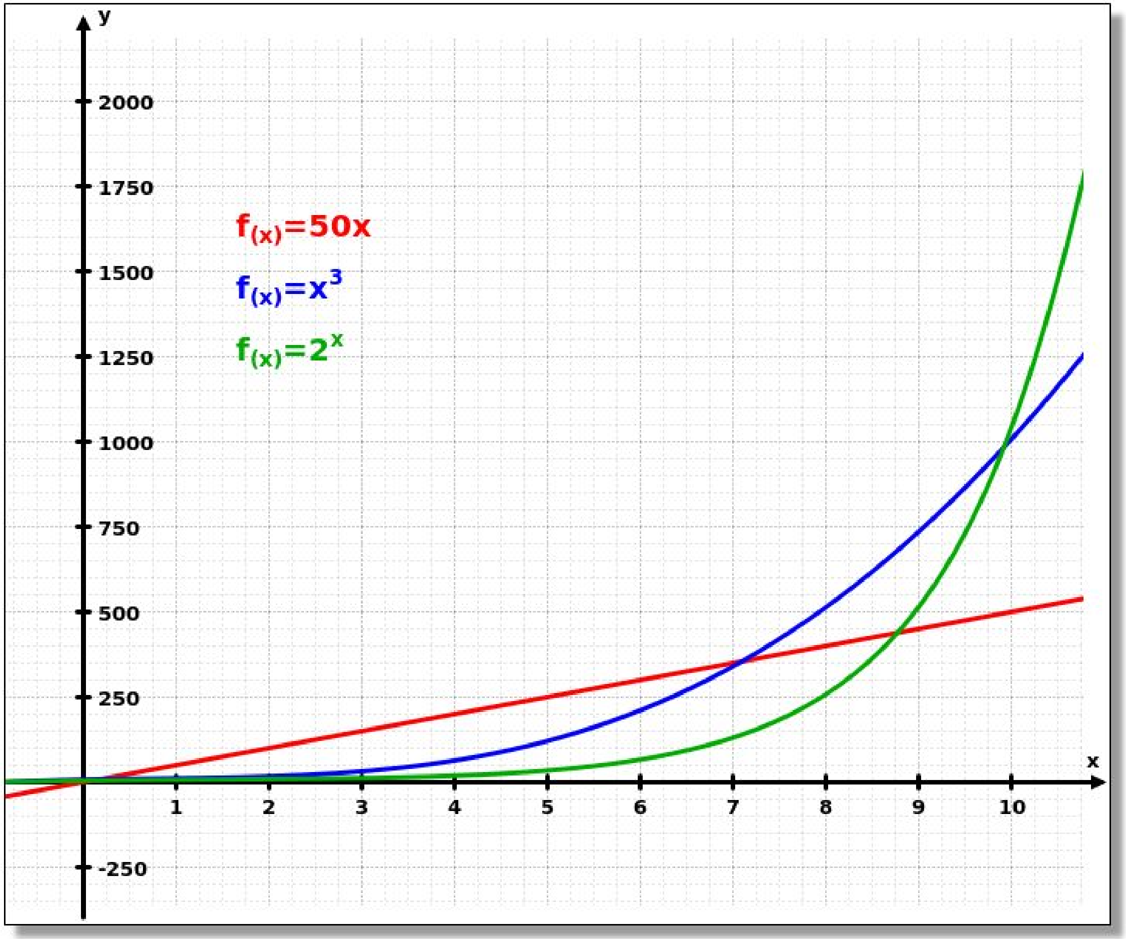
Moving towards Solutions

- Problems
- Algorithms
- Programming
- Translation
- Research in Systems



Can a problem be solved in “good-enough” time?

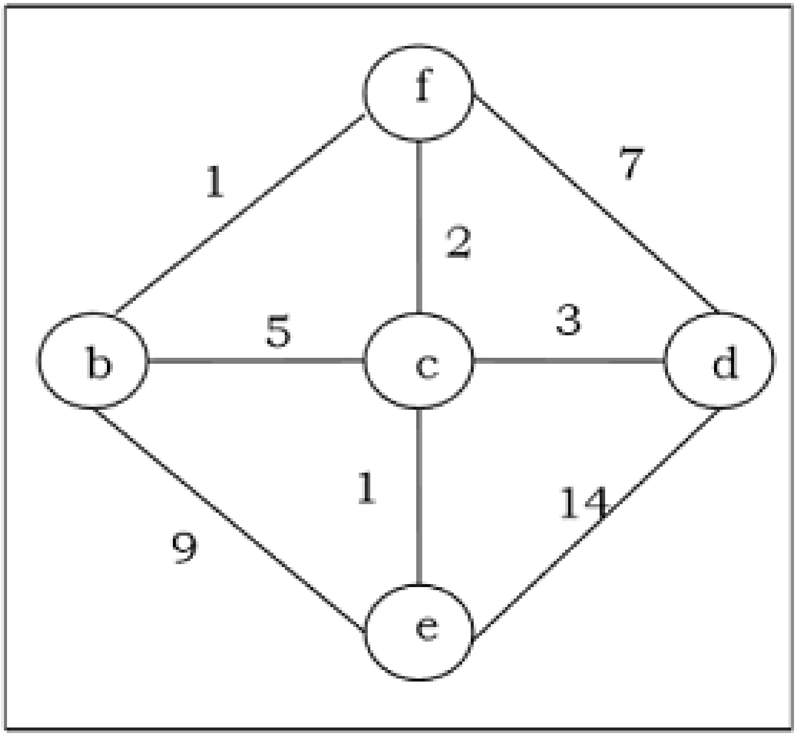
Linear vs Polynomial vs Exponential

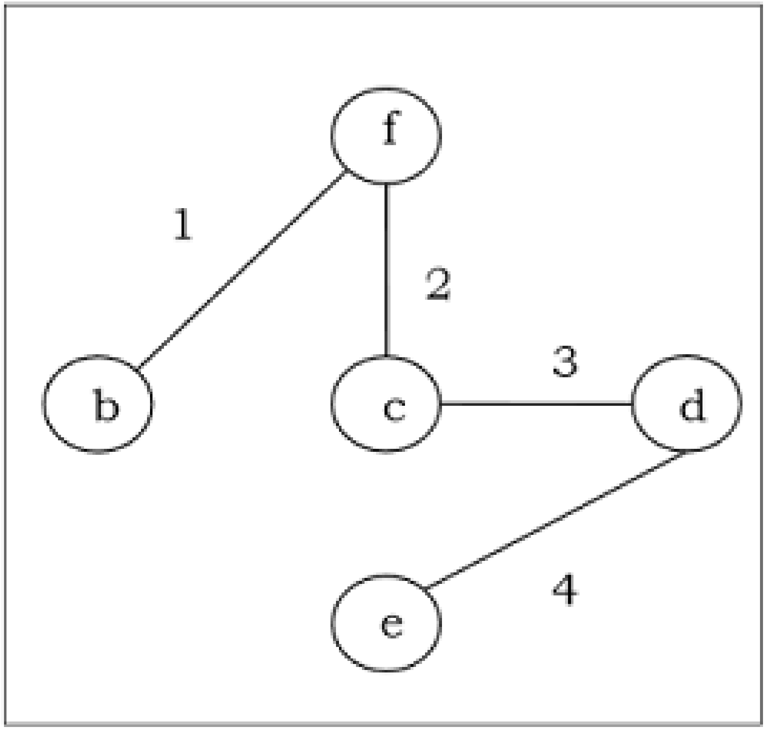


The *P* class of problems

- Problems that can be solved in polynomial time by a Deterministic Turing Machine
- All practical problems that we write algorithms for
- Example: Minimum Spanning Tree

The Minimum Spanning Tree Problem

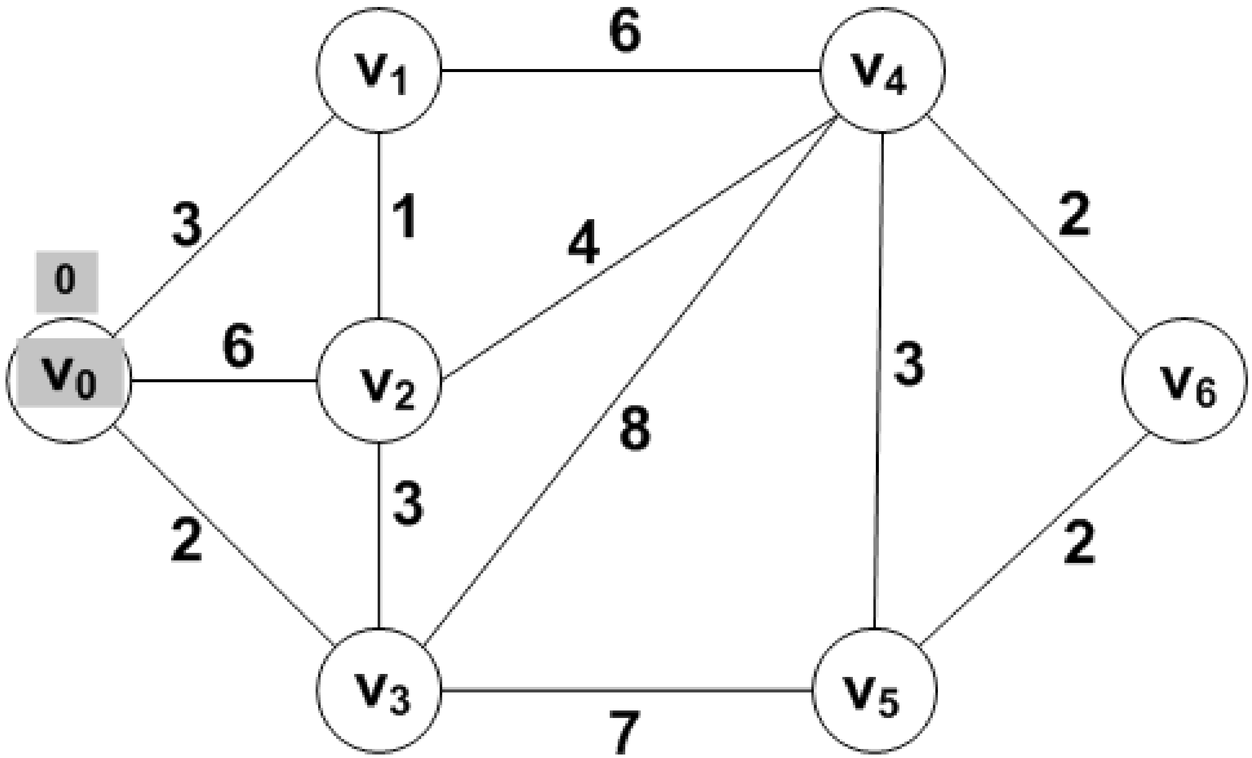




The *NP* class of problems

- Problems that can be solved in polynomial time by a Nondeterministic Turing Machine
  - A given solution can be ***checked*** in polynomial time by a Deterministic Turing Machine
- Even though the power of an NTM is equivalent to that of a DTM, the time requirements of NP may not be in the “good-enough” zone
- Example: Travelling Salesman Problem (decision version)

The Travelling Salesman Problem



Is *P* = *NP*?

- A problem Q is ***NP-Complete*** if:
  - Q is in *NP*
  - All problems in *NP* can be reduced (in polynomial time) to Q
- A problem R is ***NP-Hard*** if:
  - All problems in *NP* can be reduced (in polynomial time) to R
  - It's not known whether R is in *NP*
- Thus, if even a single NP-Complete problem can be solved by an algorithm in polynomial time, then *P* = *NP.*
- It seems that P != NP; however, there is no proof yet!

Comparing Algorithms

- Given two algorithms A_1 and A_2 that solve the same problem, find out which one is more efficient.

Approach 1: Implement and Test

- Will the results be affected by:
  - Testcase?
  - Programming language?
  - Programmer?
  - Machine?

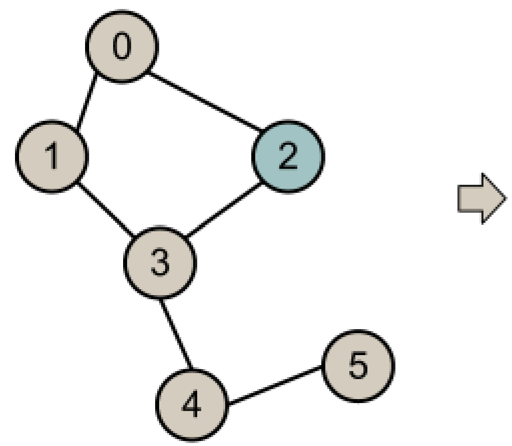
Approach 2: Be lazy

- Don’t implement
- Formalize the running time
- For big inputs, ignore everything except the fastest growing term
- Drop the constants
- Compare the asymptotic complexity

Example: Breadth First Search

Breadth First Search

```
queue.enqueue(first);

while q not empty

    cur = queue.dequeue();

    visited.add(cur);

    for each neighbour n of cur

        if visited.absent(n)

            queue.enqueue(n)
```



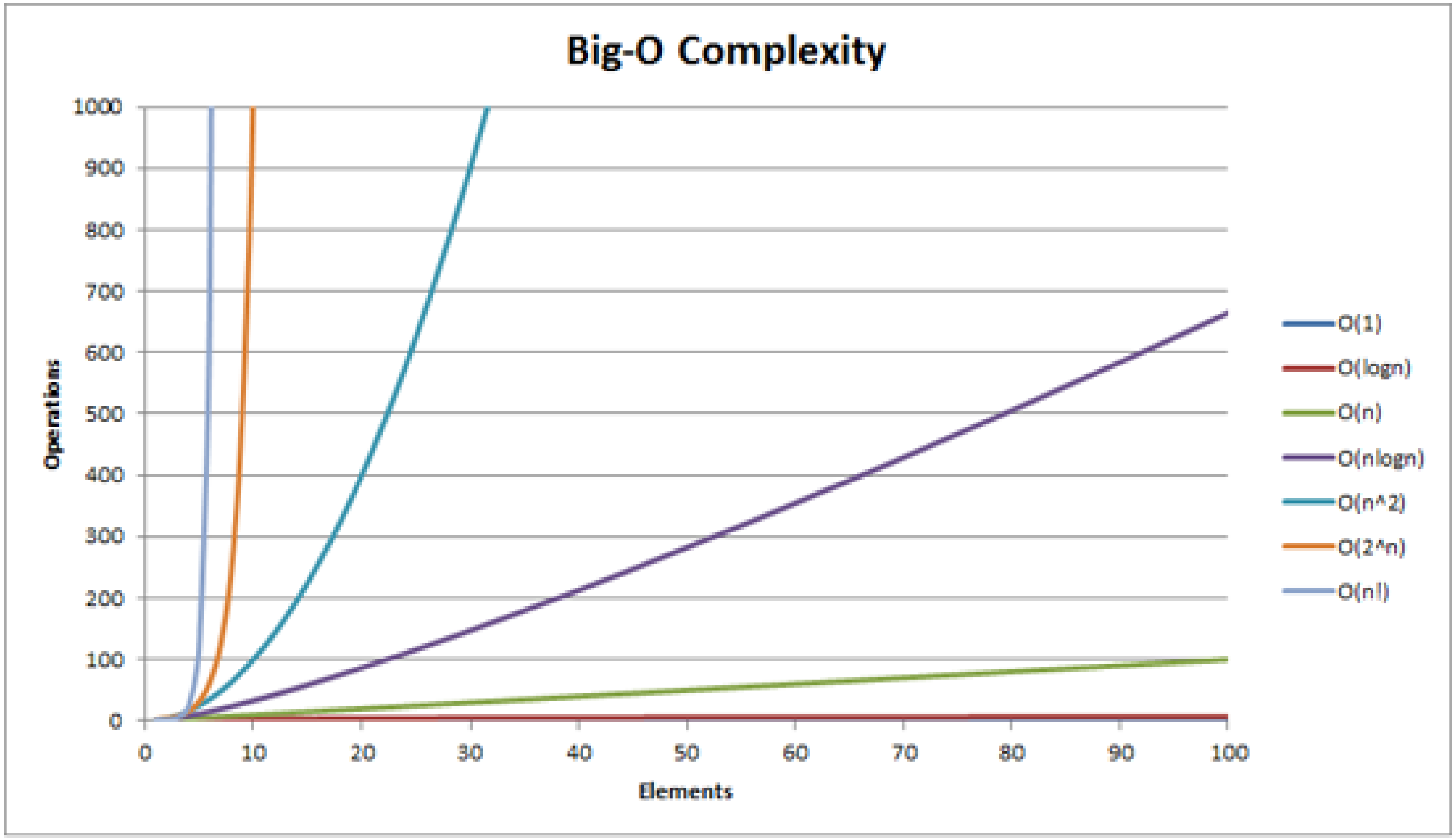

```
1. q = {}
2. q = {2}
3. q = {0, 3}
4. q = {1}
5. q = {4}
6. q = {5}
```

Using a queue

Complexity:

```
  V * (O(1) + (V * numNeighbors) + O(1))
= V + E + V
= 2V + E
= O(V+E) <= O(V²)
```

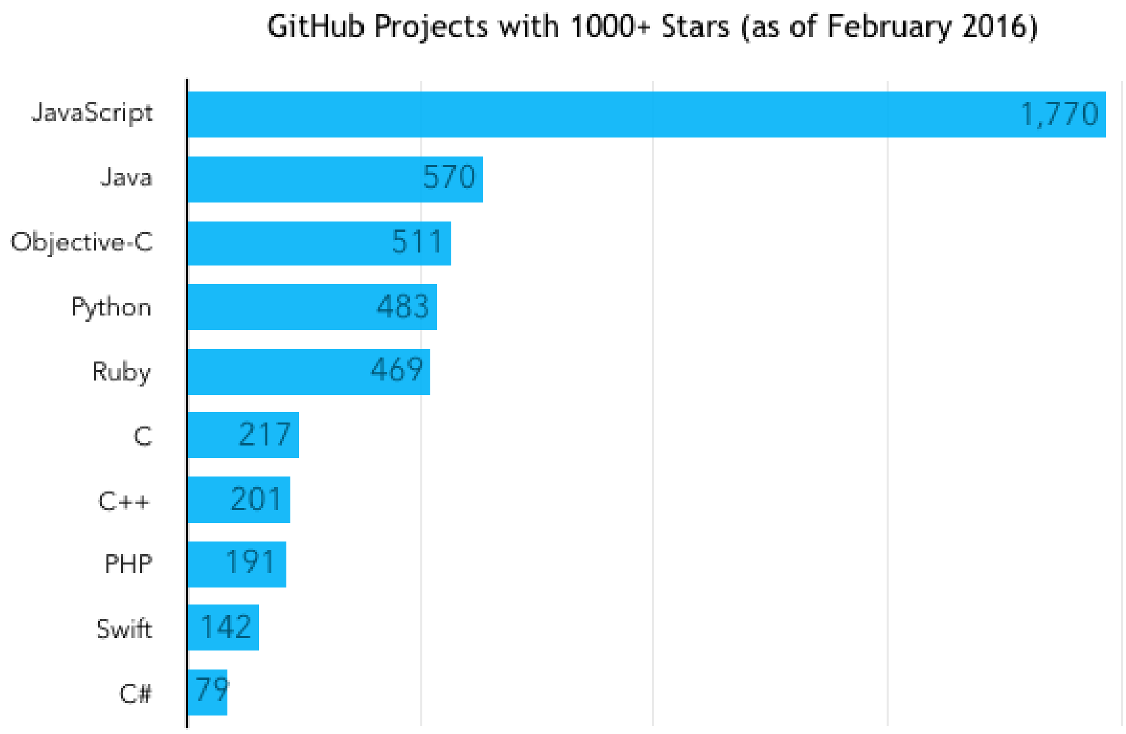
Big-Oh



Big-O Complexity
1000
900
800
700
600
500
400
300
200
100
0
Operations
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Hands on

- Problems
- Algorithms
- Programming
- Translation
- Research in Systems



Is the following C string-copy function correct?

```
char* strcpy_p1(char* dst, char* src) {
    while (src != '\0') {
        *dst++ = *src++;
    }
    return dst;
}
```

What about this one?

```
char* strcpy_p2(char* dst, char* src) {
    char* addr = dst;
    while (src != ‘\0’) {
        *dst++ = *src++;
    }
    return addr;
}
```

And this one?

```
char* strcpy_p3(char* dst, char* src) {
    assert ((dst != NULL) && (src != NULL));
    char* addr = dst;
    while (src != '\0') {
        *dst++ = *src++;
    }
    return addr;
}
```

Why is this one better?

```
char* strcpy_p4(char* dst, const char* src) {
    assert ((dst != NULL) && (src != NULL));
    char* addr = dst;
    while (src != ‘\0’) {
        *dst++ = *src++;
    }
    return addr;
}
```

Is this equivalent to the previous one?

```
char* strcpy_p5(char* dst, const char* src) {
    int i;
    assert ((dst != NULL) && (src != NULL));
    for (i=0; src[i]!='\0'; ++i) {
        dst[i] = src[i];
    }
    return dst;
}
```

Is one of the two better than the other?

```
char* strcpy_p4(char* dst, const char* src) {
    assert ((dst != NULL) && (src != NULL));
    char* addr = dst;
    while (src != '\0') {
        *dst++ = *src++;
    }
    return addr;
}
```

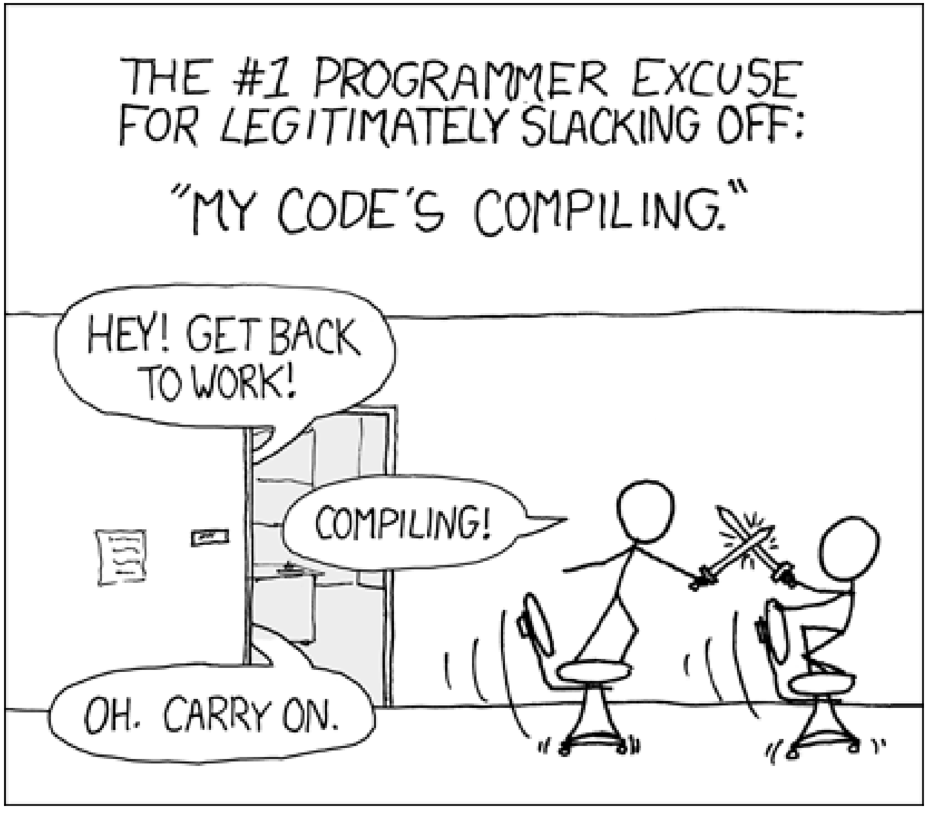
```
char* strcpy_p5(char* dst, const char* src) {
    int i;
    assert ((dst != NULL) && (src != NULL));
    for (i=0; src[i]!='\0'; ++i) {
        dst[i] = src[i];
    }
    return dst;
}
```

A few tricky PL concepts for GATE

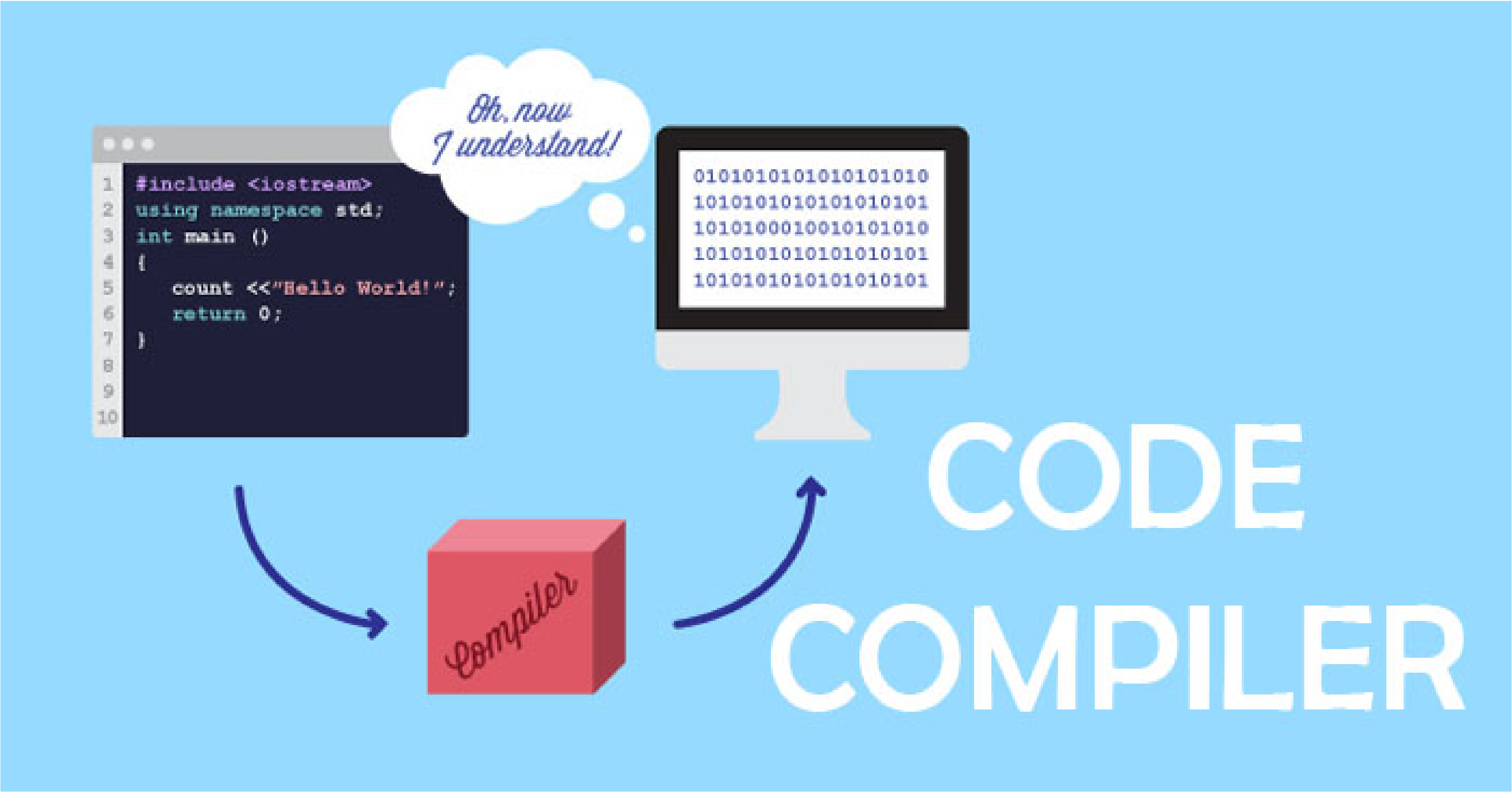
- Semantics of pointer arithmetic
- Memory management – The Stack and the Heap
- Understanding recursive programs
- Operations on various data structures

C Theriyadhu

- Problems
- Algorithms
- Programming
- Translation
- Research in Systems



Source to binary



Why do we *need* a compiler?

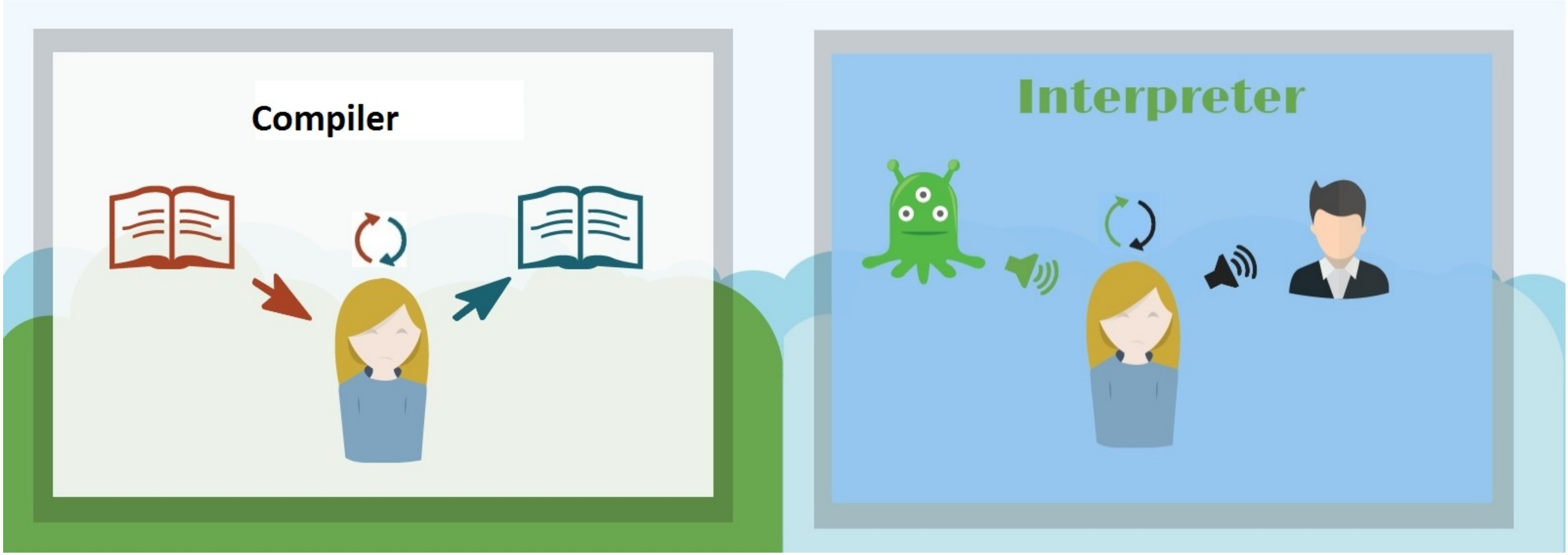


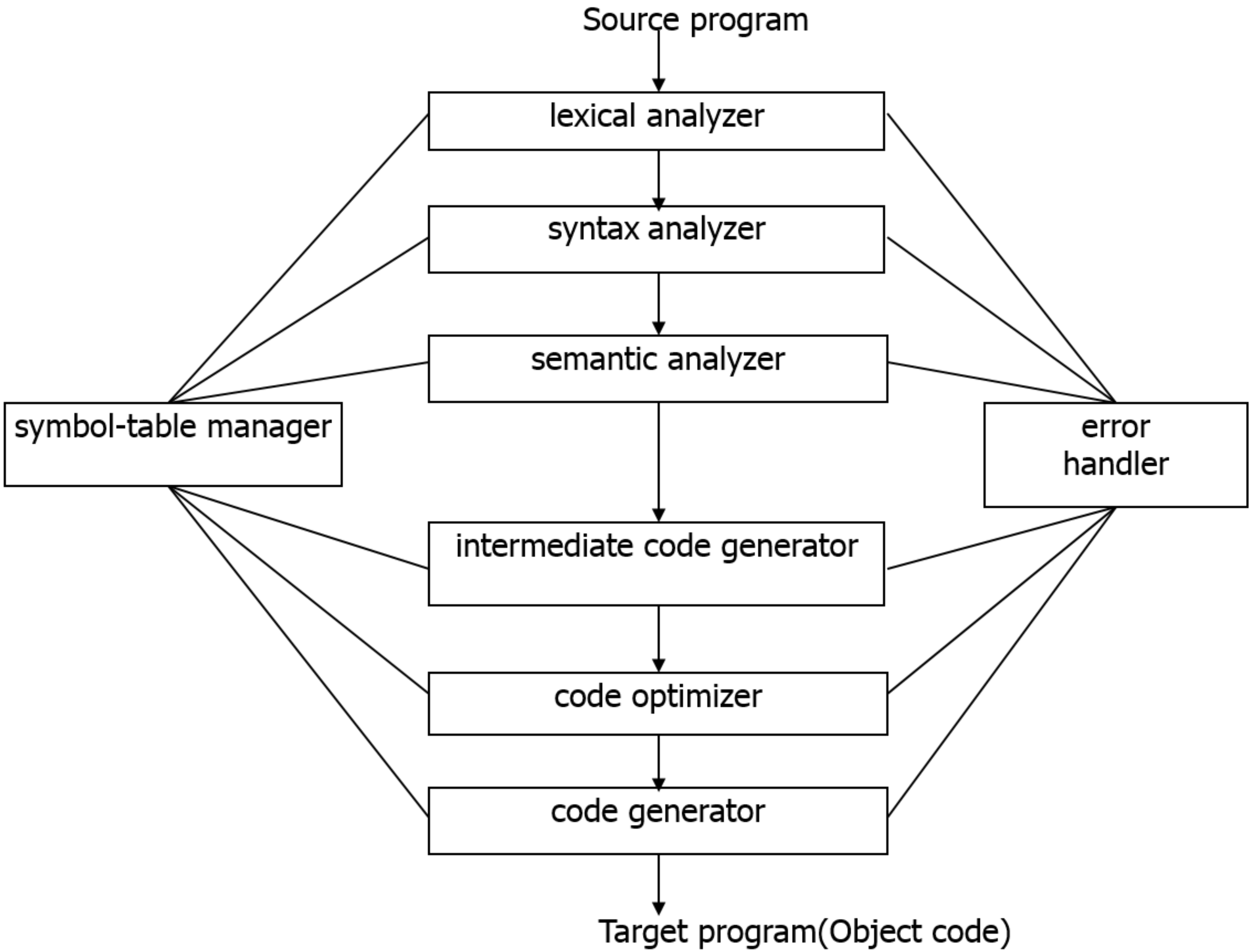
Image source: https://stackoverflow.com/questions/2377273/how-does-an-interpreter-compiler-work

Why do we *need* a compiler?

	A COMPILER	AN INTERPRETER
Input	... takes an entire program as its input.	... takes a single line of code, or instruction, as its input.
Output	... generates intermediate object code.	... does not generate any intermediate object code.
Speed	... executes faster.	... executes slower.
Memory	... requires more memory in order to create object code.	... requires less memory (doesn't create object code).
Workload	... doesn't need to compile every single time, just once.	... has to convert high-level languages to low-level programs at execution.
Errors	... displays errors once the entire program is checked.	... displays errors when each instruction is run.

Image source: https://www.upwork.com

Phases of a typical compiler



Loop invariant code motion

```
for (i=0; i<N; ++i) {
    for (j=0; j<N; ++j) {
        c = N*(N-1);
        d = c+a[i];
        a[i] = d;
    }
}
```

```
c = N*(N-1);
for (i=0; i<N; ++i) {
    for (j=0; j<N; ++j) {
        d = c+a[i];
        a[i] = d;
    }
}
```

Correct loop invariant code motion

```
c = 100;
...
for (i=0; i<N; ++i) {
    for (j=0; j<N; ++j) {
        c = N*(N-1);
        d = c+a[i];
        a[i] = d;
    }
}
...
if (c > 50) {
    printf("CEE");
}
```

```
c = 100;
...
if (N > 0) {
    c = N*(N-1);
}
for (i=0; i<N; ++i) {
    for (j=0; j<N; ++j) {
        d = c+a[i];
        a[i] = d;
    }
}
...
if (c > 50) {
    printf("CEE");
}
```

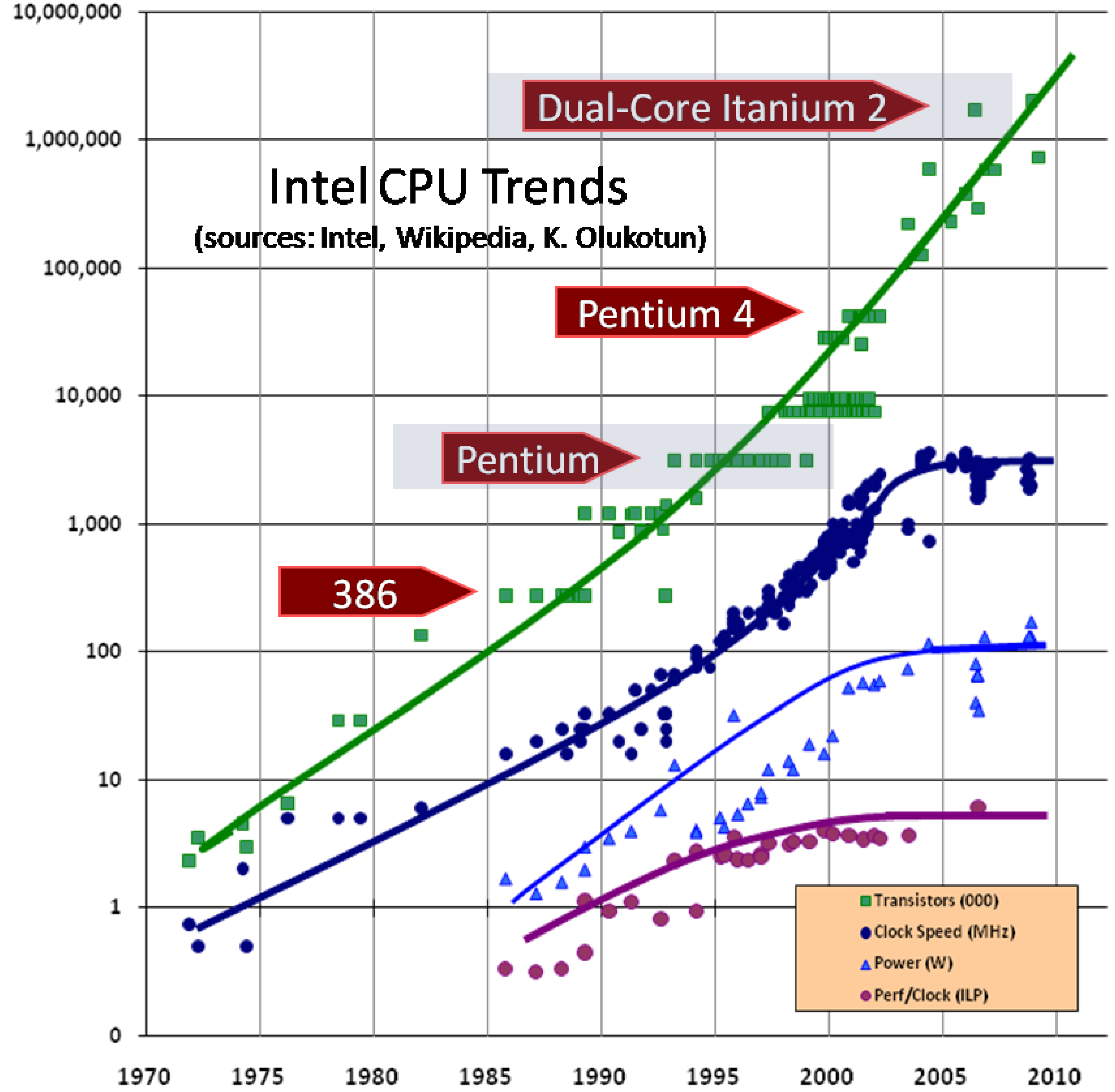
A few tricky Compilers concepts for GATE

- Program equivalence
- Program correctness
- Code optimization

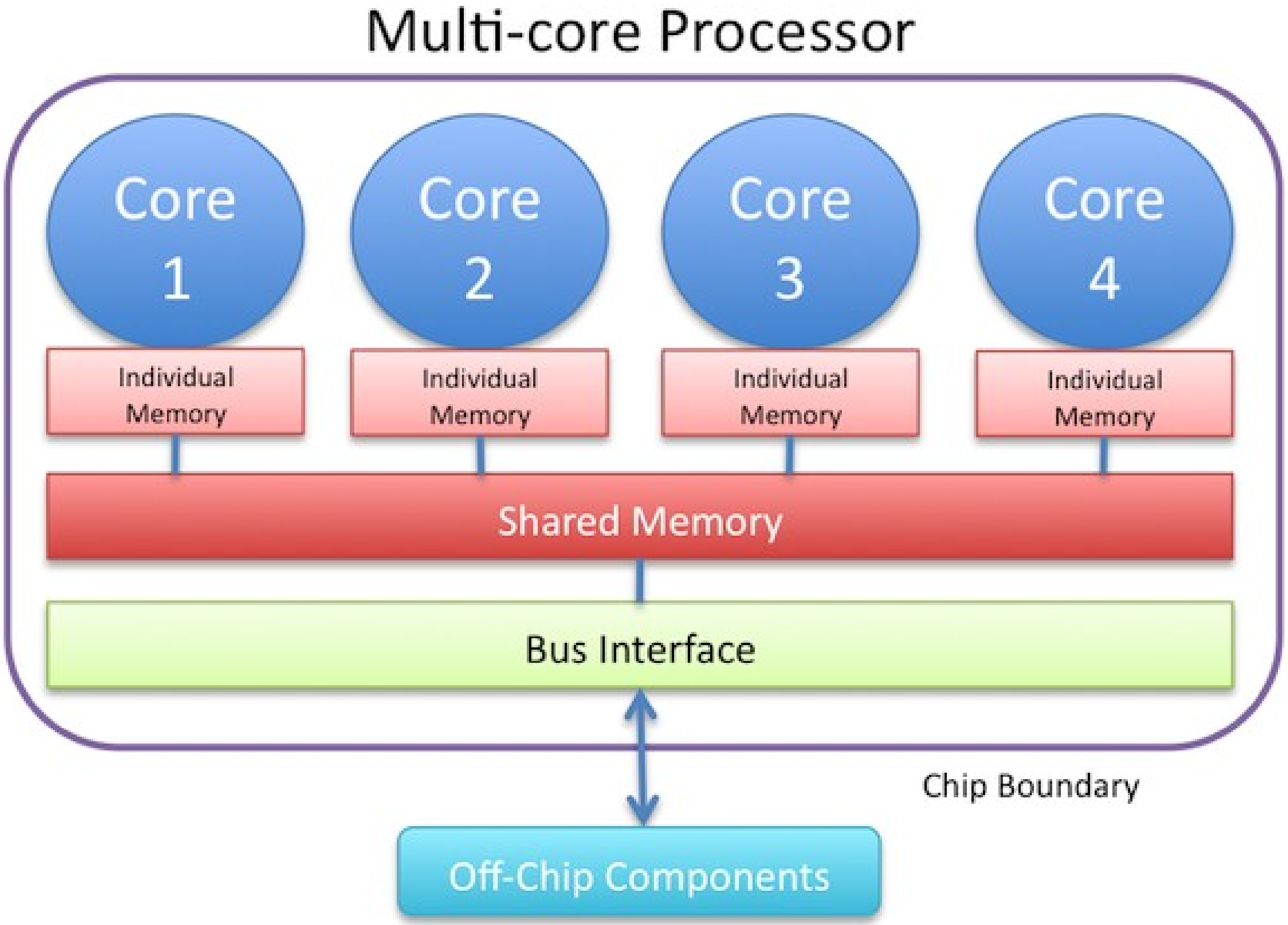
Who created this stupid computer?

- Problems
- Algorithms
- Programming
- Translation
- Research in Systems

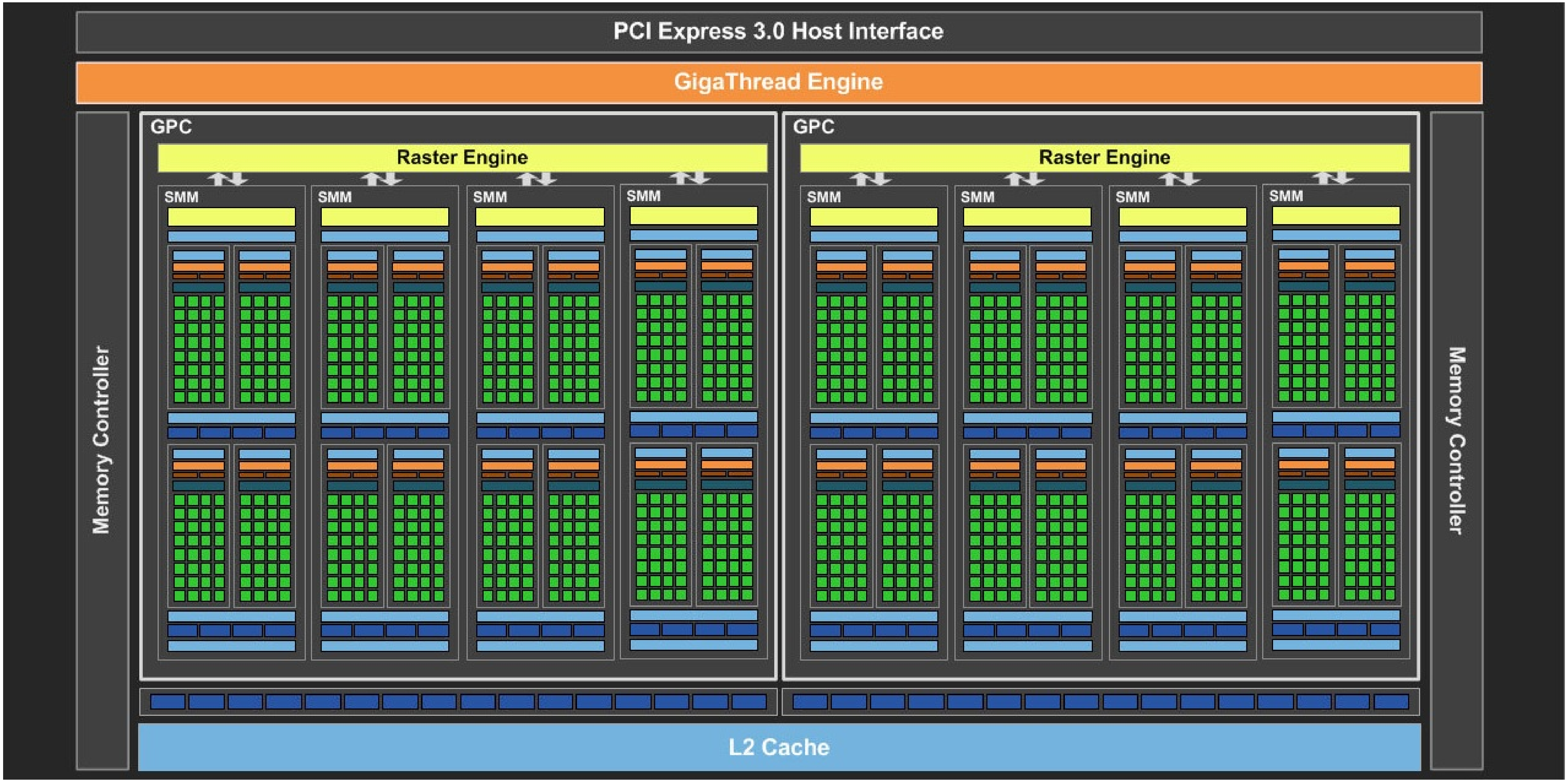
The Free Lunch is Over



Multicore systems



Manycore systems



Final words

- Clearing GATE opens up a plethora of opportunities; take it seriously
- Stay away from All-in-One guides; your textbooks should be your best friends
- Join a good test series; assessing yourself apriori boosts the confidence significantly
- Practice previous-year papers
- Join Post-GATE guidance groups on Facebook
- Apply wisely at various institutes
- Know the difference between industry and research programmes
- Low score: Look out for winter admissions

Stay Hungry, Stay Foolish, Stay Connected

www.cse.iitm.ac.in/~manas

github.com/manasthakur
gist.github.com/manasthakur

manasthakur17@gmail.com

manasthakur.wordpress.com

linkedin.com/in/manasthakur

manasthakur.github.io/docs/year4sct.pdf