

ENPM 809X: Algorithms and Data Structures

E. Gunduzhan

Module 1: Introduction

Course Requirements

- Required textbook: *Introduction to Algorithms*, T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Third Edition, MIT Press. ISBN: 978-0-262-53305-8
- Pre-requisite: Basic programming skills
- Grading:
 - Biweekly homework assignments (30%)
 - Two programming projects (40%)
 - Final exam on May 18, 2023 (30%)

Please read the syllabus – by taking the course you are agreeing to the terms in the syllabus!

Communication

- Canvas site
- E-mail: egunduzh@umd.edu
- Teaching assistant: Rishabh Mukund rmukund@umd.edu
- Office hours (online):
 - Wednesdays 5pm – 6pm
 - Fridays 2pm – 3pm (with Rishabh)
 - By appointment

Code of Academic Integrity

The University of Maryland, College Park has a nationally recognized Code of Academic Integrity, administered by the Student Honor Council. This Code sets standards for academic integrity at Maryland for all undergraduate and graduate students. As a student you are responsible for upholding these standards for this course. It is very important for you to be aware of the consequences of cheating, fabrication, facilitation, and plagiarism. For more information on the Code of Academic Integrity of the Student Honor Council, please visit <http://shc.umd.edu/SHC/HonorPledgeInformation.aspx>.

All homeworks and programming projects are individual assignments. Academic dishonesty, including cheating, fabrication, plagiarism, copying solutions or code from another source (such as another student, internet, etc) will not be tolerated.

Algorithms

- A well-defined computational procedure (i.e. a sequence of steps) that will take a set of inputs and transform into a set of outputs.
- Algorithms typically solve a given problem, which specifies the desired relationship between the input(s) and output(s).
- Example: Sorting a sequence of numbers
 - Input: A sequence of numbers, $\langle a_1, a_2, \dots, a_n \rangle$
 - Desired output: A permutation of the given sequence, $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- We will see that there are many algorithms that can solve this problem with different running times

Some Interesting Problems We Will Discuss...

- Given a street map with known distances between intersections, how do you find the shortest path between two given points?
- Given two sequences $X = \langle x_1, x_2, \dots, x_n \rangle$ and $Y = \langle y_1, y_2, \dots, y_m \rangle$, how do you find the longest common subsequence of X and Y ?
For instance, a subsequence of $\langle A, B, C, D, E, F \rangle$ is $\langle B, E, F \rangle$
- Given a set of points on a plane, how do you find the “complex hull” of these points?
A complex hull is the smallest polygon that contains all given points

Hard Problems

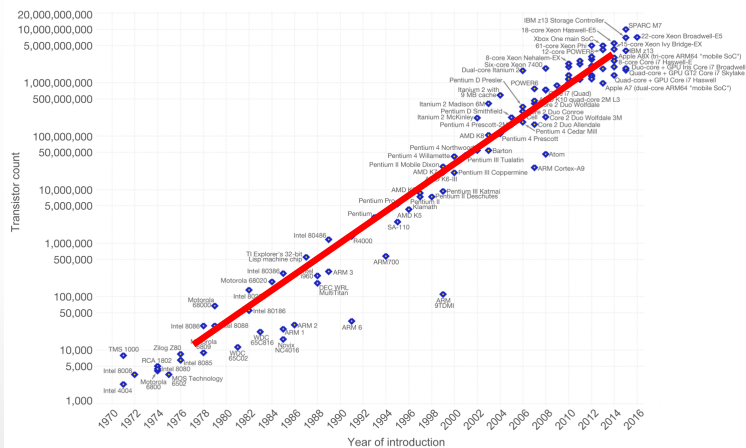
- Speed is the common measure of efficiency, i.e. how long does it take for the algorithm to produce the result? – especially as input size grows
- Most algorithms we will consider run in polynomial time, i.e.
 - For inputs of size n , their worst-case running time is $O(n^k)$ for some constant k
- All problems cannot be solved in polynomial time.
- For example, the “Halting Problem” cannot be solved by any computer, no matter how much time we allow.
 - Find a general algorithm that will determine whether an arbitrary computer program will finish running for an arbitrary input
- There are also problems that can be solved, but not in polynomial time, i.e. running time cannot be bounded by $O(n^k)$ for any constant k .

Hard Problems (cont'd)

- NP-complete problems:
 - No known polynomial-time algorithm exists for any NP-complete problem, but...
 - No one could prove that no polynomial-time algorithm can exist for them
 - If a polynomial-time algorithm exists for any one of them, then a polynomial-time algorithm exists for all of them
- Traveling Salesman problem (NP-complete):
 - A salesman has to visit N cities starting and ending at a given city.
 - The cities are connected via many paths, where each path has a "cost".
 - The salesman wants to minimize the total cost of his travel.

Moore's Law vs Data Growth

Moore's Law – The number of transistors on integrated circuit chips (1971-2016) Our World in Data
 Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

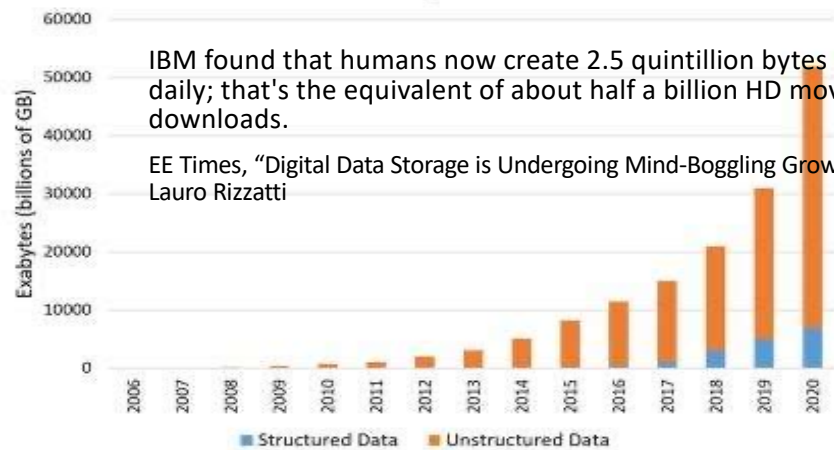


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
 The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic. Licensed under CC-BY-SA by the author Max Roser.

Figure taken from wikipedia.org

Number of transistors on chips doubles approximately every 2 years...

The Cambrian Explosion...of Data



IBM found that humans now create 2.5 quintillion bytes of data daily; that's the equivalent of about half a billion HD movie downloads.

EE Times, "Digital Data Storage is Undergoing Mind-Boggling Growth", Lauro Rizzatti

...but data is also growing exponentially.

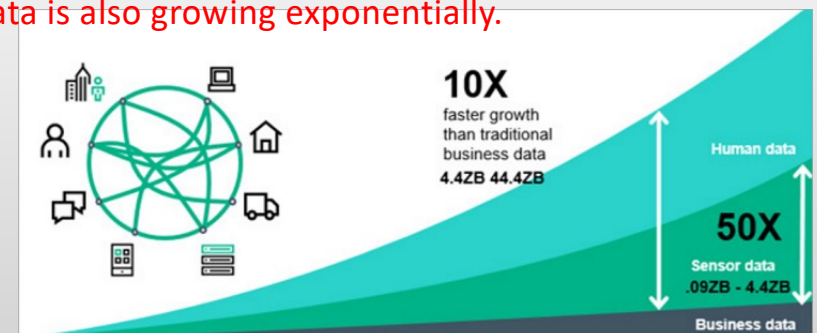


Figure taken from <https://insidebigdata.com/2017/02/16/the-exponential-growth-of-data/>

Importance of Algorithm Efficiency

Consider two algorithms that solve the same problem (e.g. sorting n items)

- First algorithm (A1) has a running time equal to c_1n^2
- Second algorithm (A2) has a running time equal to $c_2n\lg n$
- Two processors: P1 with 10 billion instructions/sec, P2 with 10 million instructions/sec
- Suppose A1 is implemented on P1 requiring $2n^2$ instructions, and A2 is implemented on P2 requiring $50n\lg n$ instructions

	Running Time ($n = 10$ million)	Running Time ($n = 100$ million)
A1 on P1	$2 \times 10^{14} / 10^{10} = 20000$ sec (~5.5 hours)	$2 \times 10^{16} / 10^{10} = 2000000$ sec (23 days)
A2 on P2	$50 \times 10^7 \times 23 / 10^7 = 1150$ sec (~19 min)	$50 \times 10^8 \times 27 / 10^7 = 13500$ sec (3.7 hours)

Data Structures

- Ways to store and organize data to facilitate access and modifications.
- Some examples:
 - Arrays
 - Linked lists
 - Hash tables
 - Binary trees
 - Red-black trees
- Need to select a data structure suitable to the given problem and algorithm

Topics to be Covered

- Week 1: Introduction to algorithms and data structures; Review of C programming
- Week 2: Growth of functions; Divide and conquer algorithms; Insertion sort and merge sort
- Week 3: Randomized algorithms; Heaps and heapsort
- Week 4: Priority queues; Quicksort; Sorting in linear time
- Week 5: Elementary data structures; Hash tables
- Week 6: Rooted trees; Binary trees; Red-black trees
- Week 7: Dynamic programming
- Week 8: Greedy algorithms; B-trees

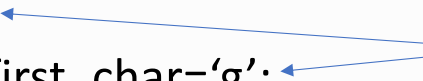
Spring Break

- Week 9: Advanced data structures
- Weeks 10-11: Graphs and elementary graph algorithms
- Weeks 11-12: Shortest paths
- Week 13: Maximum flow
- Week 14: Multithreaded algorithms
- Week 15: Review

Review of C

Variables

- Declaration of variables:

- `int x, y, z=5;`
 - `char letter1, first_char='g';`
 - `double x;`
- initial values
- 

- Constant variables:

- `const int number_pi = 3.1415;`

- Casting:

- Converts the type of variables
- `int n;`
- `float x = 102.365;`
- `n = (int)(x/32);`

Input/Output

- Input from keyboard:
 - `scanf("format string", &arg1, &arg2, ..., &argn);`
 - format string contains matching/excluding characters and expected types of the inputs - `%c, %d, %f, %s, ...`
- Output to screen:
 - `printf("format string", arg1, arg2, ..., argn);`
 - format string contains other characters to be printed and the types of the outputs - `%c, %d, %f, %s, ...`
- File input/output is similar but uses `fscanf, fprintf`
 - First get a file pointer by opening the file:
`fp = fopen(filename, "r");` or `fp = fopen(filename, "w");`
 - Close the file when done:
`fclose(fp)`

Condition Statements

- if-else:

```
if (expression with one or more conditions)
{
    statements-if-true;
}
else
{
    statements-if-false;
}
```

- switch:

```
switch (expression)
{
    case value-1:
        statements-1;
        break;
    case value-2:
        statements-2;
        break;
    default:
        statements-default;
}
```


Loops

```
for (statement; expression, statement)  
{  
    statements;  
}
```

```
while (expression)  
{  
    statements;  
}
```

```
do  
{  
    statements;  
} while (expression);
```

Command Line Arguments

```
int main (int argc; char *argv[])  
{  
...  
}
```

- argc is the total number of arguments
- argv[] is an array of strings (i.e. 2-D array of chars) that contains the arguments

Structures

```
struct car
{
    char make[40];
    char model[40]
    int year;
    int doors;
    float price;
}
```

Related data is stored and handled together

Functions

- Declaration:

```
double min(double x, double y);
```

- Definition:

```
double min(double x, double y)
{
    if (x <= y)
        return x;
    else
        return y;
}
```

- Call:

```
a = min(b, 2);
```

- Group of statements that perform a task
- Declaration tells the compiler the number and types of inputs and outputs
- Definition contains actual statements in the function
- Variables declared in a function are local – they don't affect variables with the same name outside that function
- Variables declared as inputs can change their values inside the function but this doesn't affect their values outside the function
 - But pointers can be passed as inputs and the function can modify the memory location pointed by these

Recursion

- A function may call itself
- Need to be careful that there is always a stopping condition; otherwise this will be an infinite loop of calls
- Example: Calculating the n^{th} Fibonacci number, i.e. $f(n) = f(n - 2) + f(n - 1)$

```
int Fibonacci (int n)
{
    if (n == 0)
        return (1);
    else if (n == 1)
        return (1);
    else
        return (fibonacci(n-2) + fibonacci(n-1));
}
```

Pointers

- Variables that store the memory location of another variable's value
- Two operators about pointers:
 - *p : the value at the address shown by p
 - &x : the address that stores x's value

```
int *p1, *p2, n;  
double *g, x;
```

```
p1 = &n;  
x = *g;
```

- NULL can be used to indicate an invalid pointer
 - if (p == NULL) ...
- A void pointer has no type, so can be assigned to any pointer

```
void *q;
```

Arrays

```
int a[10], b[25];  
double x[20];
```

```
int *p;
```

```
a[0] = 3;  
a[1] = 5;  
a[2] = 9;
```

```
p = a;
```

Equivalent to `p = &(a[0])`

We will have `*p = 3` after this statement

Also, `*(p+1) = 5`

Memory Allocation

```
void *malloc (size_t size);
```

Argument is the total size of the memory to be allocated.

Returns a pointer to the beginning of the allocated memory.

```
double *p;
```

```
int num_records = 100;
```

```
p = (double *)malloc (num_records * sizeof(double));
```

Allocated memory is released by “free” function.

```
free (p);
```