CS213/293 Data Structure and Algorithms 2024

Lecture 1: Why should you study data structures?

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-07-31

Next course in programming

Is CS101 and SSL not enough to be a programmer?

In CS101, you learned to walk.

In this course, you will learn to dance.

What is data?

Things are not data, but information about them is data.

Example 1.1

Age of people, height of trees, price of stocks, and number of likes.

Data is big!

We are living in the age of big data!

*Image is from the Internet.

Exercise 1.1

1. *Estimate the number of messages exchanged for status level in Whatsapp.*
2. *How much text data was used to train ChatGPT?*

We need to work on data

We process data to solve our problems.

Example 1.2

1. *Predict the weather*
2. *Find a webpage*
3. *Recognize fingerprint*

Disorganized data will need a lot of time to process.

Exercise 1.2

How much time do we need to find an element in an array?

Problems

Definition 1.1

A *problem* *is a pair of an input specification and an output specification.*

Example 1.3

The problem of *search* *consists of the following specifications*

- *Input specification: an array S of elements and an element e*
- *Output specification: position of e in S if it exists. If it is not found, return -1.*

Output specifications refer to the variables in the input specifications

Exercise 1.3

According to the specification, what should happen if e occurs multiple times in S?

Algorithms

Definition 1.2

An algorithm solves a given problem.

- Input $\in$ Input specifications
- Output $\in$ Output specifications

$$\text{Input} \longrightarrow \boxed{\text{Algorithms}} \longrightarrow \text{Output}$$

Note: There can be many algorithms to solve a problem.

Exercise 1.4

1. *What is an algorithm?*
2. *How is it different from a program?*

Commentary: An algorithm is a step-by-step process that processes a small amount of data in each step and eventually computes the output. The formal definition of the algorithm will be presented to you in CS310. It took the genius of Alan Turing to give the precise definition of an algorithm.

Example: an algorithm for search

Example 1.4

```
int search( int* S, int n, int e) {
  // n is the length of the array S
  // We are looking for element e in S
  for( int i=0; i < n; i++ ) {
    if( S[i] == e ) {
      return i;
    }
  }
  return -1; // Not found
}
```

Exercise 1.5

What is the run time of the above algorithm if e is not in S?

Commentary: Answer: We count memory accesses, arithmetic operations (including comparisons), assignments, and jumps. The loop in the program will iterate n times. In each iteration, there will be one memory access `S[i]`, three arithmetic operations `i<n`, `S[i] == e` and `i++`, and two jumps. At the initialization, there is an assignment `i=0`. For the loop exit, there will be one more comparison and jump. $Time = nT_{Read} + (3n+2)T_{Arith} + (2n+1)T_{jump} + T_{return}$ Give this program to `https://godbolt.org/` and see the assembly. Check if the above analysis is faithful!

Data needs structure

Storing data as a pile of stuff, will not work. We need structure.

Example 1.5

Store files in the order of the year. How do we store data at IIT Bombay Hospital?

Structured data helps us solve problems faster

We can exploit the structure to design efficient algorithms to solve our problems.

The goal of this course!

Example: search on well-structured data

Example 1.6

Let us consider the problem of search consisting of the following specifications

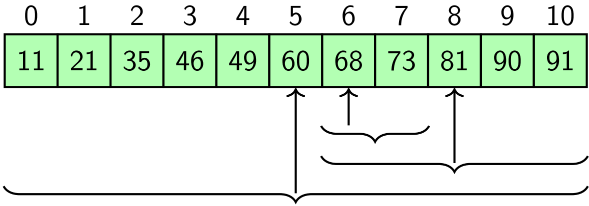
- *Input specification: a non-decreasing array S and an element e*
- *Output specification: Position of e in S. If not found, return -1.*

Example: search on well-structured data

Let us see how we can exploit the structured data!

Let us try to search 68 in the following array.

- Look at the middle point of the array.
- Since the value at the middle point is less than 68, we search only in the upper half.
- We have halved our search space.
- We recursively half the space.



A better search

Example 1.7

```
int BinarySearch(int* S, int n, int e){
  // S is a sorted array
  int first = 0, last = n;
  int mid = (first + last) / 2;
  while (first < last) {
    if (S[mid] == e) return mid;
    if (S[mid] > e) {
      last = mid;
    } else {
      first = mid + 1;
    }
    mid = (first + last) / 2;
  }
  return -1;
```

Commentary: Answer: There will be k iterations. In each iteration, the function will follow the same path. In each iteration, there will be

- a memory access `S[mid]`,(why only one)
- five arithmetic operations `first < last`, `S[mid] == e`, `S[i] > e`, `first+last`, and `../2`,
- one assignment `last = mid`, (Why?)
- three jumps because of two ifs and a loop exit,

For loop exit, there will be one additional comparison and a jump at the loop head. In the initialization section, we have two assignments and two arithmetic operations.
$Time = kT_{Read} + (6k+5)T_{Arith} + (3k+1)T_{jump} + T_{return}$

Exercise 1.6

Let $n = 2^{k-1}$. How much time will it take to run the above algorithm if `S[0] > e`?

Topic 1.1

Big-O notation

How much resource does an algorithm need?

There can be many algorithms to solve a problem.

Some are good and some are bad.

Good algorithms are efficient in

- time and
- space.

Our method of measuring time is cumbersome and machine-dependent.

We need approximate counting that is machine-independent.

Commentary: Sometimes there is a trade-off between time and space. For example, the inefficient linear search only needed one extra integer, but the binary search used three extra integers. The difference between two integers may be a minor issue, but it illustrates the trade-off.

Input size

An algorithm may have different running times for different inputs.

How do we think about comparing algorithms?

We define the rough size of the input, usually in terms of important parameters of input.

Example 1.8

In the problem of search, we say that the number of elements in the array is the input size.

Please note that the size of individual elements is not considered. *(Why?)*

Commentary: Ideally, the number of bits in the binary representation of the input is the size, which is too detailed and cumbersome to handle. In the case of search, we assume that elements are drawn from the space of size 2^{32} and can be represented using 32 bits. Therefore, the type of the element was `int`.

Best/Average/Worst case

For a given size of inputs, we may further make the following distinction.

1. Best case: Shortest running time for some input.
2. Worst case: Worst running time for some input.
3. Average case: Average running time on all the inputs of the given size.

Exercise 1.7

How can we modify almost any algorithm to have a good best-case running time?

Example: Best/Average/Worst case

Example 1.9

```
int BinarySearch(int* S, int n, int e){
  // S is a sorted array
  int first = 0, last = n;
  int mid = (first + last) / 2;
  while (first < last) {
    if (S[mid] == e) return mid;
    if (S[mid] > e) {
      last = mid;
    } else {
      first = mid + 1;
    }
    mid = (first + last) / 2;
  }
  return -1;
}
```

In BinarySearch, let $n = 2^{k-1}$.

1. Best case: $e == S[n/2]$
   $T_{Read} + 6T_{Arith} + T_{return}$,
2. Worst case: $e \notin S$
   We have seen the worst case.
3. The average case is roughly equal to the worst case because most often the loop will iterate k times. (Why?)

Commentary: Analyzing the average case is usually involved. For some important algorithms, we will do a detailed average time analysis.

Asymptotic behavior

For short inputs, an algorithm may use a shortcut for better running time.

To avoid such false comparisons, we look at the behavior of the algorithms in limit.

Ignore hardware-specific details

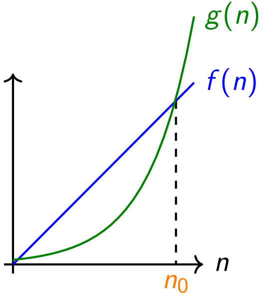
- Round numbers $100000000000001 \approx 100000000000000$
- Ignore coefficients $3kT_{Arith} \approx k$

Big-O notation: approximate measure

Definition 1.3

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in O(g(n))$ if there are c and n_0 such that

$$f(n) \leq cg(n) \qquad \text{for all } n \geq n_0.$$



- In limit, $cg(n)$ will dominate $f(n)$
- We say $f(n)$ is $O(g(n))$

Exercise 1.8

Which of the following are the true statements?

- $5n + 8 \in O(n)$
- $5n + 8 \in O(n^2)$
- $5n^2 + 8 \in O(n)$
- $n^2 + n \in O(n^2)$
- $50000000000000000000000n^2 \in O(n^2)$
- $50n^2 logn + 60n^2 \in O(n^2 logn)$

Example: Big-O of the worst case of `BinarySearch`

Example 1.10

In `BinarySearch`, let $n = 2^{k-1}$.

1. Worst case: $e \notin S$

   $kT_{Read} + (6k+5)T_{Arith} + (3k+1)T_{jump} + T_{return} \in O(k)$

   Since $k = log\ n + 1$, therefore $k \in O(log\ n)$

   We may also say `BinarySearch` is $O(log\ n)$.

   Therefore, the worst-case running time of `BinarySearch` is $O(log\ n)$.

Exercise 1.9

Prove that $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.

What does Big O says?

Expresses the approximate number of operations executed by the program as a function of input size

Hierarchy of algorithms

- $O(log\ n)$ algorithm is better than $O(n)$
- We say $O(log\ n) < O(n) < O(n^2) < O(2^n)$

May hide large constants!!

Complexity of a problem

The complexity of a problem is the complexity of the best-known algorithm for the problem.

Exercise 1.10

What is the complexity of the following problem?

- *sorting an array* $O(n^2)$ ✗
- *matrix multiplication* $O(n^3)$ ✗

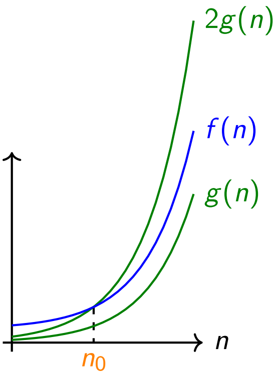
Best algorithm is still not known

Exercise 1.11

What is the best-known complexity for the above problems?

Commentary: A discussion on the latest developments in matrix multiplication algorithms. https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication

Θ-Notation



Definition 1.4 (Tight bound)

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in \Theta(g(n))$ if there are c_1, c_2, and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \text{for all } n \geq n_0.$$

There are more variations of the above definition. Please look at the end.

Exercise 1.12

a. Does the worst-case complexity of `BinarySearch` *belong to $\Theta(\log n)$?*

b. If yes, give c_1, c_2, and n_0 for the application of the above definition on `BinarySearch`*.*

Names of complexity classes

- Constant: $O(1)$
- Logarithmic: $O(logn)$
- Linear: $O(n)$
- Quadratic: $O(n^2)$
- Polynomial : $O(n^k)$ for some given k
- Exponential : $O(2^n)$

Topic 1.2

Tutorial Problems

Problem: Compute the exact running time of insertion sort.

Exercise 1.13

The following is the code for insertion sort. Compute the exact worst-case running time of the code in terms of n and the cost of doing various machine operations.

```
for( int j = 1; j < n; j++ ) {
  int key = A[j];
  int i = j-1;
  while( i >= 0 ) {
    if( A[i] > key ) {
      A[i+1] = A[i];
    }else{
      break;
    }
    i--;
  }
  A[i+1] = key;
}
```

Problem: additions and multiplication

Exercise 1.14

What is the time complexity of binary addition and multiplication? How much time does it take to do unary addition?

Problem: hierarchy of complexity

Exercise 1.15
Given $f(n) = a_0n^0 + ... + a_dn^d$ *and* $g(n) = b_0n^0 + ... + b_en^e$ *with* $d > e$ *and* $a_d > 0$ *(Why?), show that* $f(n) \notin O(g(n))$.

Topic 1.3

Problems

Order of functions

Exercise 1.16

- *If $f(n) \leq F(n)$ and $G(n) \geq g(n)$ (in order sense) then show that $\frac{f(n)}{G(n)} \leq \frac{F(n)}{g(n)}$.*
- *Is $f(n)$ the same order as $f(n)|sin(n)|$?*

Commentary: Source Milind notes

Exercise: an important complexity class!

Exercise 1.17

Prove that $O(\log(n!)) = O(n \log n)$. *Hint: Stirling's approximation*

Topic 1.4

Extra slides: More on complexity

Ω notation

Definition 1.5 (Lower bound)

Let f and g be functions $\mathbb{N} \rightarrow \mathbb{N}$. We say $f(n) \in \Omega(g(n))$ if there are c and n_0 such that

$$cg(n) \leq f(n) \qquad \text{for all } n \geq n_0.$$

Small-o,ω notation

Definition 1.6 (Strict Upper bound)

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in o(g(n))$ if for each c, there is n_0 such that

$$f(n) \leq cg(n) \qquad \textit{for all } n \geq n_0.$$

Definition 1.7 (Strict Lower bound)

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in \omega(g(n))$ if for each c, there is n_0 such that

$$cg(n) \leq f(n) \qquad \textit{for all } n \geq n_0.$$

Exercise 1.18

a. Prove that $f \in O(g)$ implies $f \in o(g)$.

b. Show that $f \in o(g)$ does not imply $f \in O(g)$.

Size of functions

We can define a partial order over functions using the above notations

- $f(n) \in O(g(n))$ implies $f(n) \leq g(n)$
- $f(n) \in o(g(n))$ implies $f(n) < g(n)$
- $f(n) \in \Omega(g(n))$ implies $f(n) \geq g(n)$
- $f(n) \in \omega(g(n))$ implies $f(n) > g(n)$
- $f(n) \in \Theta(g(n))$ implies $f(n) = g(n)$

Exercise 1.19

Show that the partial order is well-defined.

Commentary: Why do we need to prove that the definition is well-defined?

End of Lecture 1