Chapter 7

INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS ANALYSIS

Reference books:

- A Practical Introduction to Data Structures and Algorithm Analysis by Clifford A. Shaffer
- *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie
- *Programming in C (3rd Edition)* by Stephen G. Kochan.
- Data Structures and Algorithm Analysis in C by Mark Allen Weiss

Data Structures and Algorithms

- The *Heart* of Computer Science
  - Data structures
  - Algorithm analysis
  - Study of important algorithms
  - Algorithm design techniques
- Why DS and Algorithm are important
  - Some problems are difficult to solve and good solutions are known
  - Some “solutions” don’t always work
  - Some simple algorithms don’t scale well
  - Data structures and algorithms make good tools

The Need for Data Structures

Data structures organize data

$\Rightarrow$ more efficient programs.

More powerful computers $\Rightarrow$ more complex applications.

More complex applications demand more calculations.

Complex computing tasks are unlike our everyday experience.

Organizing Data

Any organization for a collection of records can be searched, processed in any order, or modified.

The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

A solution is said to be efficient if it solves the problem within its resource constraints.

- Space
- Time

- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Some Questions to Ask

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

A data structure requires:

- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

Data Structure Philosophy (cont)

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:

- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight

Costs and Benefits

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

Any data structure requires:

- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

Costs and Benefits (cont)

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:

- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight

Abstract Data Types

Abstract Data Type (ADT): a definition for a data type solely in terms of a set of values and a set of operations on that data type.

Each ADT operation is defined by its inputs and outputs.

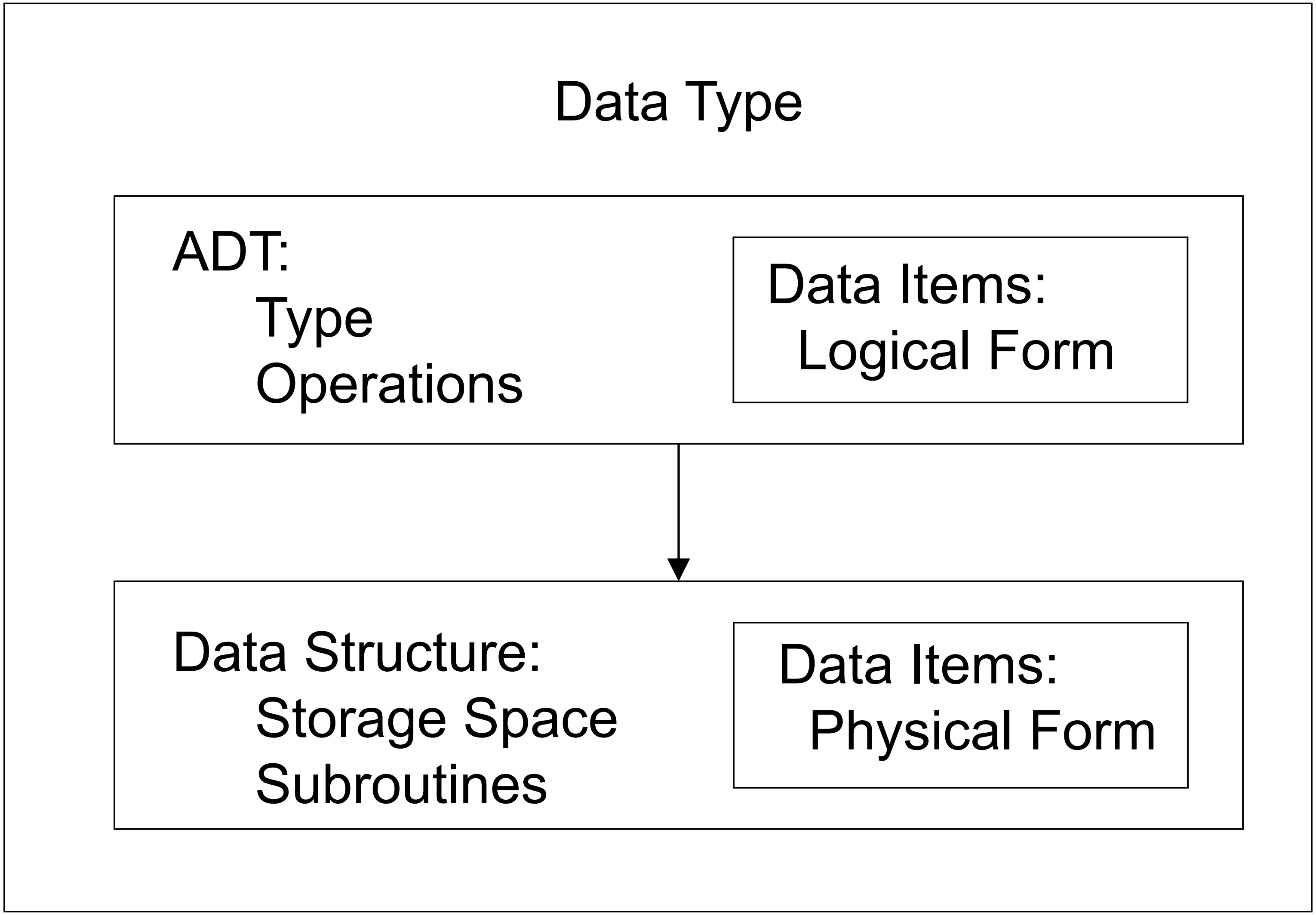
Encapsulation: Hide implementation details.

Data Structure

- A data structure is the physical implementation of an ADT.
  - Each operation associated with the ADT is implemented by one or more subroutines in the implementation.
- Data structure usually refers to an organization for data in main memory.
- File structure is an organization for data on peripheral storage, such as a disk drive.

Logical vs. Physical Form

- Data items have both a logical and a physical form.
- Logical form: definition of the data item within an ADT.
  - Ex: Integers in mathematical sense: +, -
- Physical form: implementation of the data item within a data structure.
  - Ex: 16/32 bit integers, overflow.


Data Type
ADT:
Type
Operations
Data Items:
Logical Form
Data Structure:
Storage Space
Subroutines
Data Items:
Physical Form

Problems

- Problem: a task to be performed.
  - Best thought of as inputs and matching outputs.
  - Problem definition should include constraints on the resources that may be consumed by any acceptable solution.

Problems (cont)

- Problems ⇔ mathematical functions
  - A function is a matching between inputs (the domain) and outputs (the range).
  - An input to a function may be single number, or a collection of information.
  - The values making up an input are called the parameters of the function.
  - A particular input must always result in the same output every time the function is computed.

Algorithms and Programs

Algorithm: a method or a process followed to solve a problem.

- A recipe.

An algorithm takes the input to a problem (function) and transforms it to the output.

- A mapping of input to output.

A problem can have many algorithms.

Algorithm Properties

An algorithm possesses the following properties:

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.

A computer program is an instance, or concrete representation, for an algorithm in some programming language.

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY | ParisTech INSTITUT DES SCIENCES ET TECHNOLOGIES PARIS INSTITUTE OF TECHNOLOGY

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals.

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Algorithm Efficiency (cont)

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm's cost?

How to Measure Efficiency?

1. Empirical comparison (run programs)
2. Asymptotic Algorithm Analysis

Critical resources: running time, space

Factors affecting running time:

- speed of CPU, bus, peripheral hardware
- programming language, quality of code etc.

For most algorithms, running time depends on "size" of the input.

Running time is expressed as T(*n*) for some function T on input size *n*.

Theoretical Analysis

- Uses a **high-level description** of the algorithm instead of an implementation
- Characterizes running time as a function of the **input size**, *n*.
- Takes into account all possible inputs, often analyzing the **worst case**
- Allows us to evaluate the speed of an algorithm **independent** of the hardware/software environment

Example: Find largest value

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm *arrayMax*(A, n)
Input array A of n integers
Output maximum element of A

currentMax $\leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $A[i] >$ *currentMax* **then**
    currentMax $\leftarrow A[i]$
return *currentMax*

Instruction cost

- Questions
  - Can a program be asymptotically faster on one type of CPU vs another?
  - Do all CPU instructions take equally long?
- The Random Access Machine (RAM) Model
  - Memory cells are numbered and accessing any cell in memory takes unit time.

Primitive Operations

- Basic computations performed by an algorithm
- Largely independent from any programming language
- Exact definition not important
  - constant number of machine cycles per statement)
  - Assumed to take a constant amount of time in the RAM model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input

Algorithm $arrayMax(A, n)$	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2 + n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return $currentMax$	1
Total	$7n - 1$

Estimating Running Time

- Algorithm arrayMax executes 7n – 1 primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let T(n) be worst-case time of arrayMax. Then

$$a\,(7n-1) \le T(n) \le b\,(7n-1)$$

- Hence, the running time T(n) is bounded by two linear functions

Summary on Runtime

- We can count the number of RAM-equivalent statements executed as a function of input size
- All remaining implementation dependencies amount to multiplicative constants, which we will ignore
  - (But watch out for statements that take more than constant time)
- Linear, quadratic, etc. runtime is an *intrinsic property of an algorithm*

Best, Worst, Average Cases

Not all inputs of a given size take the same time to run.

Sequential search for *K* in an array of *n* integers:

- Begin at first element in array and look at each element in turn until *K* is found

Best case: The first element is K

Worst case: The last element is K

Average case: Go halfway through the array

Which Analysis to Use?

While average time appears to be the fairest measure, it may be difficult to determine.

When is the worst case time important?
- Real-time applications e.g. air traffic control system

Faster Computer or Algorithm?

What happens when we buy a computer 10 times faster?

T(n)	n	n'	Change	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}\, n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	-----

Remember Math?

Comparaisons asymptotiques

- On a deux fonctions f et g ($g > 0$) définies sur $[a, +\infty[$, on veut les comparer au voisinage de $+\infty$.
  - f est négligeable devant g (au voisinage de $+\infty$) si :

$$\frac{f(x)}{g(x)} \xrightarrow[x \to +\infty]{} 0, \text{ on écrit } f(x) = o_{+\infty}(g(x)).$$

  - f est un grand O de g si :

$$\exists M > 0,\ \forall x \in [a, +\infty[,\ \left|\frac{f(x)}{g(x)}\right| \leq M, \text{ on écrit } f(x) = O_{+\infty}(g(x)).$$

  - f est un grand oméga de g si :

$$\exists M > 0,\ \forall x \in [a, +\infty[,\ \left|\frac{f(x)}{g(x)}\right| \geq M, \text{ on écrit } f(x) = \Omega_{+\infty}(g(x)).$$

Asymptotic Analysis: Big-Oh

- Definition:
  - For T(n) a non-negatively valued function, T(n) is in the set O(f(n)) if there exist two positive constants c and n_0 such that $T(n) \leq cf(n)$ for all $n > n_0$.
- Usage:
  - The algorithm is in O(n^2) in [best, average, worst] case.
- Meaning:
  - For all data sets big enough (i.e., $n>n_0$), the algorithm always executes in less than $cf(n)$ steps in [best, average, worst] case.

Big-oh Notation (cont)

Big-oh notation indicates an upper bound.

Example: If $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$.

Wish tightest upper bound:
While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

Big-Oh Examples

Example 1: Finding value X in an array (average cost).

$T(n) = c_s n/2$.

For all values of $n > 1$, $c_s n/2 \leq c_s n$.

Therefore, by the definition, $T(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Big-Oh Examples

Example 2: $T(n) = c_1n^2 + c_2n$ in average case.

$c_1n^2 + c_2n \le c_1n^2 + c_2n^2 \le (c_1 + c_2)n^2$ for all $n > 1$.

$T(n) \le cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $T(n)$ is in $O(n^2)$ by the definition.

Example 3: $T(n) = c$. We say this is in $O(1)$.

A Common Misunderstanding

"The best case for my algorithm is n=1 because that is the fastest." WRONG!

Big-oh refers to a growth rate as n grows to ∞.

Best case is defined as which input of size n is cheapest among all inputs of size n.

Big-Omega

Definition: For T(n) a non-negatively valued function, T(n) is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $T(n) \geq cg(n)$ for all $n > n_0$.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than $cg(n)$ steps.

Lower bound.

Big-Omega Example

$\mathrm{T}(n) = c_1 n^2 + c_2 n$.

$c_1 n^2 + c_2 n \geq c_1 n^2$ for all $n > 1$.
$\mathrm{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.

Therefore, $\mathrm{T}(n)$ is in $\Omega(n^2)$ by the definition.

We want the greatest lower bound.

Theta Notation

When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

Definition: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

A Common Misunderstanding

Confusing worst case with upper bound.

Upper bound refers to a growth rate.

Worst case refers to the worst input from among the choices for possible inputs of a given size.

Simplifying Rules

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Running Time Examples (1)

Example 1: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$.

Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

$\Theta(n)$.

Running Time Examples (2)

Example 3:

```
sum = 0;
for (i=1; i<=n; j++)
   for (j=1; j<=i; i++)
      sum++;
for (k=0; k<n; k++)
   A[k] = k;
```

$\Theta(n^2)$.

Running Time Examples (3)

Example 4:

```
sum1 = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=i; j++)
      sum2++;
```

$\Theta(n^2)$.

Running Time Examples (4)

Example 5:

```
sum1 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=k; j++)
      sum2++;
```

$T(n)=\sum_{i=0}^{\log n} n + \sum_{i=0}^{\log n} 2^i$

$\Theta(n \log n) + \Theta(n).$

Binary Search

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

How many elements are examined in worst case?

Binary Search

```
// Return position of element in sorted
// array of size n with value K.
int binary(int array[], int n, int K) {
  int l = -1;
  int r = n; // l, r are beyond array bounds
  while (l+1 != r) { // Stop when l, r meet
    int i = (l+r)/2; // Check middle
    if (K < array[i]) r = i;   // Left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;   // Right half
  }
  return n; // Search value not in array
}
```

$T(n) = T(n/2)+1$ for $n>1$; $T(1)=1$

$T(n) = \log n$

Other Control Statements

`while` loop: Analyze like a `for` loop.

`if` statement: Take greater complexity of `then`/`else` clauses.

`switch` statement: Take complexity of most expensive case.

Subroutine call: Complexity of the subroutine.

Analyzing Problems

Upper bound: Upper bound of best known algorithm.

Lower bound: Lower bound for every possible algorithm.

Space/Time Tradeoff Principle

One can often reduce time if one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  - Boolean flags
- Table lookup
  - Factorials

Disk-based Space/Time Tradeoff Principle: The smaller you make the disk storage requirements, the faster your program will run.

Analyzing Problems: Example

Common misunderstanding: No distinction between upper/lower bound when you know the exact running time.

Example of imperfect knowledge: Sorting

1. Cost of I/O: $\Omega(n)$.
2. Bubble or insertion sort: $O(n^2)$.
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$.
4. We prove later that sorting is $\Omega(n \log n)$.

Multiple Parameters

Compute the rank ordering for all *C* pixel values in a picture of *P* pixels.

```
for (i=0; i<C; i++)   // Initialize count
  count[i] = 0;
for (i=0; i<P; i++)   // Look at all pixels
  count[value(i)]++; // Increment count
sort(count);          // Sort pixel counts
```

If we use *P* as the measure, then time is $\Theta(P \log P)$.

More accurate is $\Theta(P + C \log C)$.

Space Complexity

Space complexity can also be analyzed with asymptotic complexity analysis.

Time: Algorithm

Space: Data Structure