Algorithms and complexity

Lectures

Lectures

Pranab Roy

Algorithm : definitions

- **An algorithm is an effective methodAn algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function.**

- **Starting from an initial state and initial input (sometimes empty), the instructions describe a computation describe a computation that, when executed, will proceed through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state .**

An informal definition could be "a set of rules that precisely defines a sequence of operations "

- Algorithms are essential to the way computers process data.
- Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing.

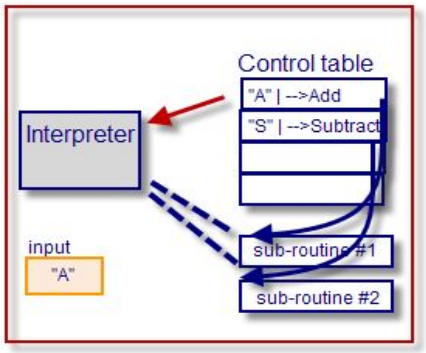
Algorithms : [contd.]

Algorithms can be expressed in many kinds of notation, including natural languagesnatural languages, pseudocodenatural languages, pseudocode, flowchartsnatural languages, pseudocode, flowcharts, programming languagesnatural languages, pseudocode, flowcharts, programming languages or control tables (processed by interpreters).

Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements.

Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

Control tables are tablesControl tables are tables that control the control flow or play a major part in program control.



pseudocode is an informal high-level description of the operating principle
of a computer program or other algorithm.
It uses the structural conventions of a programming language,
but is intended for human reading rather than machine reading.

Analysis of algorithms

An algorithm can be analyzed in terms of time efficiency or space utilization. The running time of an algorithm is influenced by several factors:

- Speed of the machine running the program
- Language in which the program was written. For example, programs written in assembly language generally run faster than those written in C or C++, which in turn tend to run faster than those written in Java.
- Efficiency of the compiler that created the program
- The size of the input: processing 1000 records will take more time than processing 10 records.
- Organization of the input: if the item we are searching for is at the top of the list, it will take less time to find it than if it is at the bottom.

Performance of a Program

- Program performance is the amount of computer memory and time needed to run a program.

- How is it determined?
  1. Analytically
     - performance analysis
  2. Experimentally
     - performance measurement

Measurement Criteria

- **Space**
  - amount of memory program occupies
  - usually measured in bytes, KB or MB
- **Time**
  - execution time
  - usually measured by the number of executions

Space complexity

- Space complexity is defined as the amount of memory a program needs to run to completion.

- Why is this of concern?
  - We could be running on a multi-user system where programs are allocated a specific amount of space.
  - We may not have sufficient memory on our computer.
  - There may be multiple solutions, each having different space requirements.
  - The space complexity may define an upper bound on the data that the program can handle.

Program space : components

- Program space = Instruction space + data space + stack space

- The **instruction space** is dependent on several factors.
  - the compiler that generates the machine code
  - the compiler options that were set at compilation time
  - the target computer

- **Data space**
  - very much dependent on the computer architecture and compiler
  - The magnitude of the data that a program works with is another factor

- **Environment Stack Space**
  - **Every time a function is called, the following data are saved on the stack.**
    1. **the return address**
    2. **the values of all local variables and value formal parameters**
    3. **the binding of all reference and const reference parameters**
  -

Time complexity

- Time complexity is the amount of computer time a program needs to run.

- Why do we care about time complexity?
  - Some computers require upper limits for program execution times.
  - Some programs require a real-time response.
  - If there are many solutions to a problem, typically we'd like to choose the quickest.

- How do we measure?
  1. Count a particular operation (operation counts)
  2. Count the number of steps (step counts)

Running time…

- **Obvious observations are:**

1. Running time of an algorithm increases with input size
2. But, ***distinct data for same input size***, may have different running times (true or false)[can you think of an example??]
3. Running time is also effected by hardware (processor, memory, clock rate, etc..)
4. Software environment in which algorithm is implemented

In addition to above, will it depend on data structure and algorithm??

Measuring running time…

1. ***Experimental study***
2. ***Analytical study***

- **Experimental study**: Measures the **actual** running time of an algorithm It requires–
- Writing programs by implementing different algorithms to solve the problem
- Running each program with data sets of varying size and composition (may use system utility function to measure the time)

Limitations…

- Process of coding every algorithm is cumbersome
- Experiments can be done on a limited set of data only
- Lots of system dependency (hardware and software)

Analytical Study

- Comparing implementations is very difficult.
- Implementations in different languages are different and not possible to check implementation.

- Can we get the idea of running time from an algorithm?
- **What constitutes the running time?**
- Running time = Total time taken by the processor to execute all steps

= t1+t2+t3…tn = n*t if every step stakes same time.

- Does each step take the same time?
- Can we say every step is a collection of Basic operations?

The RAM (Random Access Machine) model of computation

- The RAM (Random Access Machine) model of computation measures the run time of an algorithm by summing up the number of steps needed to execute the algorithm on a set of data. It operates by the following principles:
- Basic logical or arithmetic operations (+, *, =, if, call) are considered to be simple operations that take one time step.
- Loops and subroutines are complex operations composed of multiple time steps.
- All memory access takes exactly one time step.
- **This model encapsulates the core functionality of computers but does not mimic them completely**.

For example, an addition operation and a multiplication operation are both worth a single time step, however, in reality it will take a machine more operations to compute a product versus a sum.

Cont..

- The reason the RAM model makes these assumptions is because doing so allows a balance between simplicity and completely imitating underlying machine, resulting in a tool that is useful in practice.
- The exact analysis of algorithms is a difficult task.
- The algorithm analysis needs to be both machine and language independent.
- For example, if our computer becomes twice as fast after a recent update, the complexity of our algorithm still remains the same.

Does actual data in an input play a role in complexity?

- We would compare two algorithms for the same size of input
- But, for a same size of input, there can be infinitely many inputs.
- Does every input take same amount of time?
- If not, again difficult to compare
- So either we can compare best case, average case or worst case
- For average, define a distribution, it involves probability, expectations, etc. NOT Easy
- Worst and Best case?

Operation count

- Worst case count = maximum count
- Best case count = minimum count
- Average count

Step count

- The operation-count method omits accounting for the time spent on all but the chosen operation
- The step-count method count for all the time spent in all parts of the program
- A program step is loosely defined to be a syntactically or semantically meaningful segment of a program for which the execution time is independent of the instance characteristics.

Asymptotic analysis

- A primitive operation corresponds to a low level instruction whose execution time depends on the hardware or software environment
- Therefore, the number of primitive operations needed to execute the algorithm will give a high level estimate of the running time of that algorithm
- Let us also assume that the running time of each primitive operation is approximately same
- Hence, the no. of primitive operations will be directly proportional to the actual running time of algorithm

Best case, Worst case and Average Case

- As a programmer and for a user, worst-case or average-case analysis is more important than best-case
- It is more challenging to do average-case analysis in comparison to worst-case because it requires calculation over an input distribution
- Worst-case requires the ability to identify the worst-case input only
- So, we will specify running time in terms of worst-case

Asymptotic analysis

It requires–

- Writing programs in pseudo code (more structured way of writing algorithm) (better than natural language but less formal than PL)
- Characterize running time as a function of input size
- Evaluating running time by studying a high level description of the algorithm without actually implementing it or running experiments on it
- Evaluating relative performance independent of hardware and software environment

Asymptotic analysis

Example: **Findmax** (a, n) /* input an array of size n*/ Cmax = a[0]

```
For  i = 1 to n-1
If cmax < a[i]
    then cmax = a[i]
    Return cmax
```

If we examine above code, it has following operations:

- ***Assignment***
- ***addition***
- ***Comparison***
- ***return***

Asymptotic analysis

- Our objective is to analyze high level pseudo code
- Every pseudo code is collection of certain set of basic instructions or primitive operations which are independent of programming language.

For example:

1. ***Assigning a value to a variable***
2. ***Performing arithmetic operations***
3. ***Comparing two numbers***
4. ***Accessing an array through index***
5. ***Calling a function***
6. ***Returning from a function***
7. ***Reading/writing……***

Example…

Selection sort

```
1.for i = 0 to n-1
2.{min = a[i]
3.Index = i
4.for j=i+1 to n-1
5.{
6.  if a[i] > a[j]
7.    {min = a[j]; index = j };
8.}
9.  Swap(a[i],a[index]);
10.}
```

More Examples

```
int add(int a[],int n)
{ int sum; sum =0;

for (int i=0;i<n;i++)
sum = sum +a[i];

return sum;

}
```

```
int search(int a[],int n,int x)
{ int i, flag =0;
for ( i=0;i<n && !flag ;i++) if
(a[i] == x) flag =1;
if (i==n) return -1; else return i;
}
```

Need for Asymptotic Complexity

- While evaluating the running time of simple algorithm arraymax, we had counted exactly how many primitive operations will be executed corresponding to each line of code
- How important is this??
- We do not want to calculate exact number of operations even for worst case
- We want to capture “how the running time of an algorithm increases with the size of input in worst-case” or
- we want to know how our program will behave for very large inputs (how much maximum time it will take)
- We are not interested for small inputs

What is *asymptotic complexity?*

- In most real-world examples it is not so easy to calculate the complexity function (relationship between *t and n is very complex)*
- But not normally necessary, e.g. consider $t = f(n) = n^2 + 5n$; For all $n>5$, n^2 *is largest, and for very large n, the 5n term* is insignificant
- Therefore we can approximate *f(n) by the* n^2 *term only.* This is called *asymptotic complexity*
- An approximation of the computational complexity that holds for large *n*
- Used when it is difficult or unnecessary to determine true computational complexity

- Examine these functions:

  $4N + 51;$

  $N + 1234;$

  $N^2 + 135;$

  $3N^2 + 12$

  $5N^2 + 100N + 60$

  $N^3 - 100;$

  $N^3 + 1345;$

- Describes the behavior of the time or space complexity for large instance characteristic
- **Big Oh (O)** notation provides an **upper bound** for the function *f*
- **Omega (Ω)** notation provides a **lower-bound**
- **Theta (Θ)** notation is used when an algorithm can be **bounded both from above and below** by the same function

Growth rate of algorithms

- We often want to compare the performance of algorithms
- When doing so we generally want to know how they perform when the problem size (*n*) is large
- Since cost functions are complex, and may be difficult to compute, we approximate them using O notation

Asymptotic growth of functions

Asymptotic growth : **The rate of growth of a function**

Given a particular differentiable function f(n), all other differentiable functions fall into three classes:

- **growing with the same rate**
- **growing faster**
- **growing slower**

Big Oh notation

In mathematics, **big O notation** is used to describe the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions.

Let $f(x)$ and $g(x)$ be two functions defined on some subset of the real numbers. Then

$$f(x) = O(g(x)) \text{ as } x \to \infty$$

if and only if there is a positive constant M such that for all sufficiently large values of x, $f(x)$ is at most M multiplied by $g(x)$ in absolute value. That is, $f(x) = O(g(x))$ if and only if there exists a positive real number M and a real number x_0 such that

$$|f(x)| \leq M|g(x)| \text{ for all } x > x_0.$$

The “Big-Oh” notation or Big O-notation

- Let f(n) and g(n) be two functions.
- We say that f(n) is O(g(n)) if there exists a real constant c>0 and an integer constant n_0 >=1, such that f(n)<=cg(n) for every integer n>= n_0
- This definition is referred as “f(n) is Big Oh of g(n)” or “f(n) is of the order of g(n)”
- Note: Here g(n) is generally some well defined/well understood function of n

- Examine the behavior of f(n) =n, f(n) = 2n+2, f(n) = n -7, f(n) = 3n..etc
- We say that 7n-2 is O(n)
- To prove this, we have to find a real constant c>0 and an integer constant n_0 >=1 such that 7n-2 <=cn for every n >=n_0.
- We can see that c = 7 and n_0 =1 will fulfil this requirement
- **The big-Oh notation allows us to say that a function of n is less than or equal to another function upto a constant factor and in asymptotic sense they are same as n grows to infinity**

Cont…

- $3n+2 = O(n)$ as $3n+2 <= 4n$ for all $n>=2$
- $3n+3 = O(n)$ as $3n+3 <= 4n$ for all $n>=3$
- $110n+6 = O(n)$ as $110n+6 <= 101n$ for all $n>=6$
- $10n^2 + 4n +2 = O(n^2)$ as $10n^2 + 4n +2 <= 11n^2$ for all $n>=5$
- $6*2^n + n^2 = O(2^n)$ as $6*2^n + n^2 <= 7*2^n$ for $n>=4$
- $4n + 3 = O(n^2)$ Is this true?

Lower Bound

Ω notation: lower bound

- It provides an asymptotic lower bound

Def: For a given functions g(n) and f(n), we say g(n) is the lower bound of f(n) or $f(n) = \Omega(g(n))$ if there exists positive constants c and n_0 such that $0<=cg(n)<=f(n)$ for all $n >= n_0$

Suppose, $f(n) = n^2$ and g(n) = n then $f(n) = \Omega(g(n))$ where c =1 and $n_0>1$

Suppose, we have f(n) = 100 n - 101 and g(n) =n then c =1 and n>=2

Big Oh :Properties

Fastest growing function dominates a sum

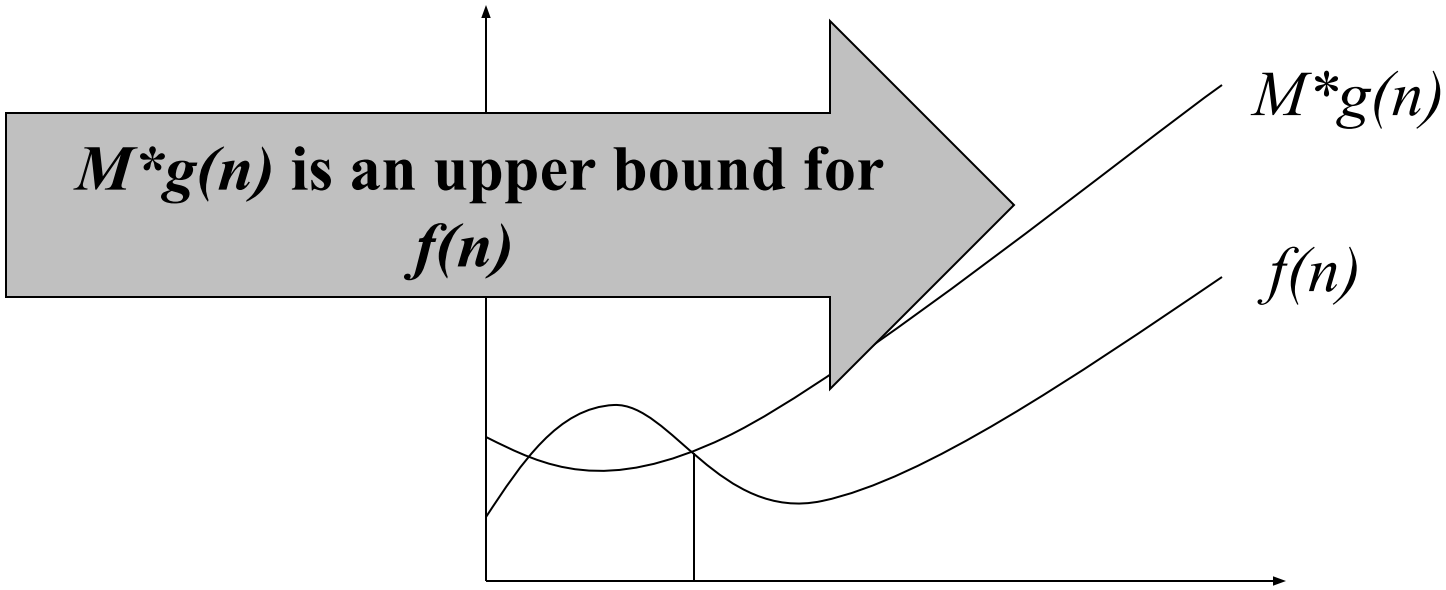
$O(f(n)+g(n))$ is $O(\max\{f(n), g(n)\})$

Product of upper bounds is upper bound for the product

If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$

***f* is *O(g)* is transitive**

If f is $O(g)$ and g is $O(h)$ then f is $O(h)$

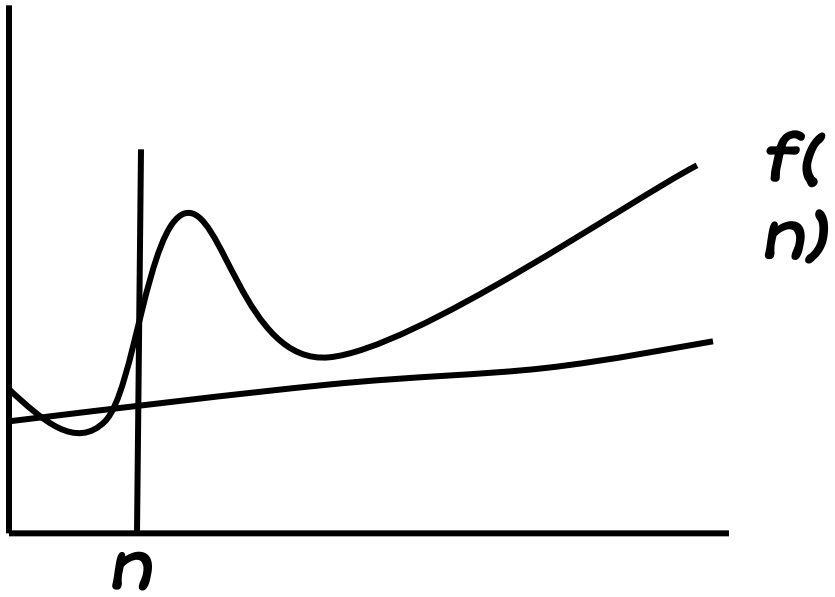


Big Omega notation

For non-negative functions, *f(n)* and *g(n)*, if there exists an integer n_0 and a constant *c* > 0 such that for all integers $n > n_0$, *f(n)* ≥ *cg(n)*, then *f(n)* is omega of *g(n)*. This is denoted as "*f(n)* = *Ω(g(n))*"

$f = \Omega(g)$ means that the function f dominates g in some limit,
In this context Ω is referred to as a lower bound.

- ***f(n) = Ω (g(n))***
  - **iff** ***∃ c, n_0 > 0 s.t. ∀ n ≥ n0 , 0 ≤ cg(n) ≤ f(n)***



f(n) *is eventually lower-bounded by* ***g(n)***

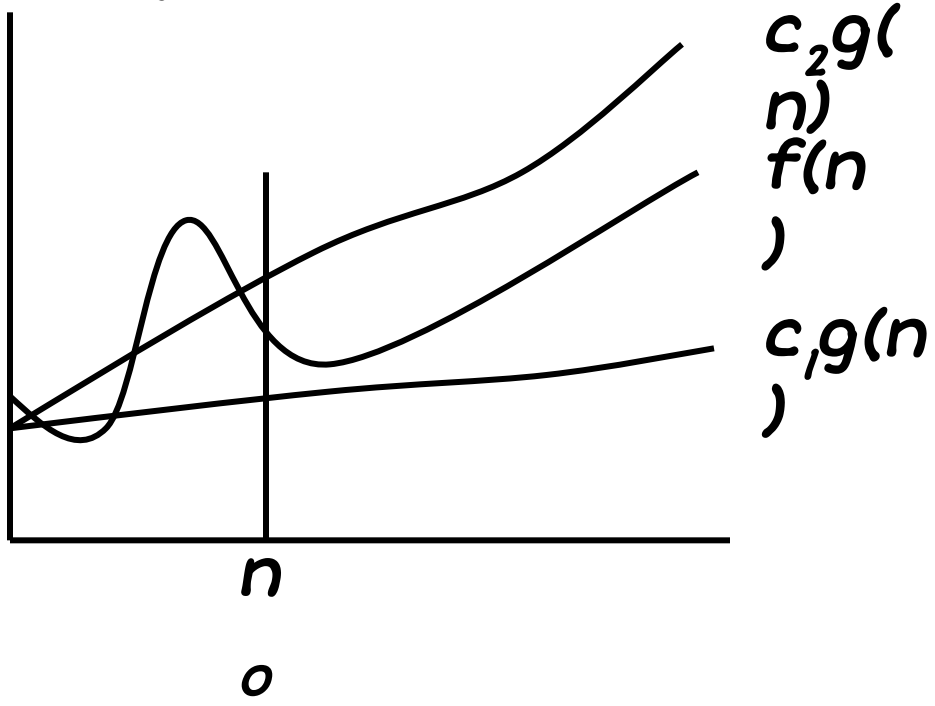
Big Theta notation

The Θ notation describes asymptotic tight bounds.

DEF. Big Theta $f(n)$ is $\Theta(g(n))$ iff $\exists$ positive real constants C_1 and C_2 and a positive integer n_0, such that

$$C_1 g(n) \leq f(n) \leq C_2 g(n) \quad \forall \; n \geq n_0$$

- The Θ notation is used when the function *f* can be bounded both from above and below by the same function *g*



f(n) has the same long-term rate of growth as ***g(n)***

Some relatives

Little-oh – f(n) is o(g(n)) if for any c>0 there is n_0 such that $f(n) < c(g(n))$ for $n > n_0$.

Little-omega
Little-theta

Worst vs Average case

Worst case complexity: provides absolute guarantees for time a program will run. The worst case complexity as a function of *n* is longest possible time for any input of size *n*.

Average case complexity: suitable if small function is repeated often or okay to take a long time –very rarely. The average case as a function of *n* is the avg. complexity over all possible inputs of that length.

Avg. case complexity analysis usually requires probability theory.

Insertion Sort

- The *insertion sort* uses a vector's partial ordering.
- On the *k*th pass, the *k*th item should be inserted into its place among the first *k* items in the vector.
- After the *k*th pass (*k* starting at 1), the first *k* items of the vector should be in sorted order.

This is like the way that people pick up playing cards and order them in their hands. When holding the first (k - 1) cards in order, a person will pick up the kth card and compare it with cards already held until its sorted spot is found.

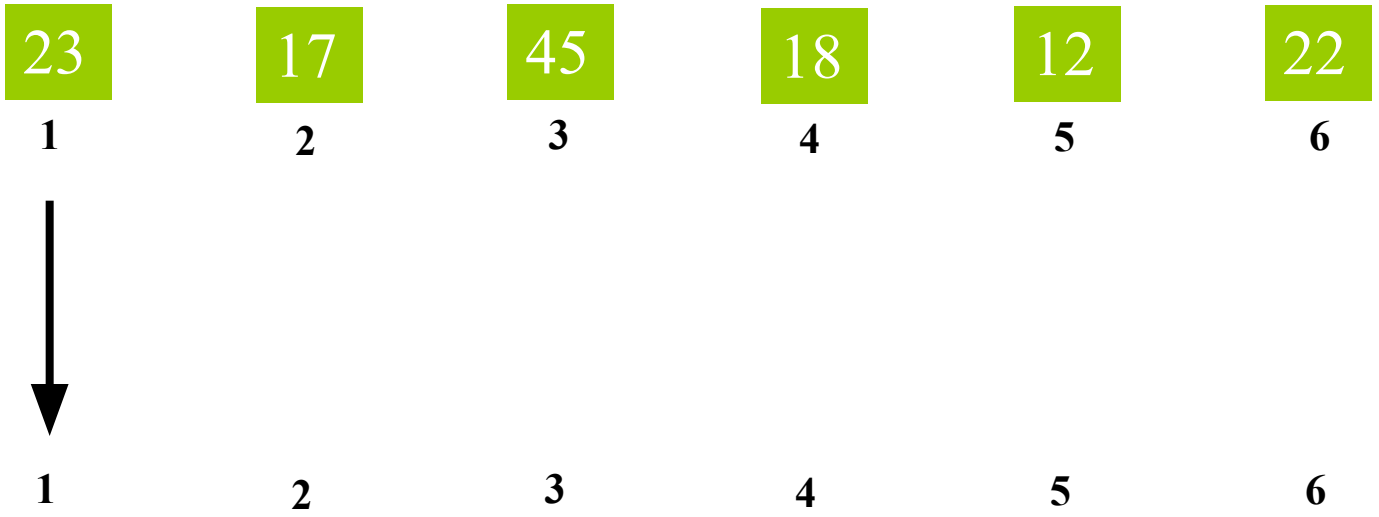
Insertion sort:example

Given some numbered cards.

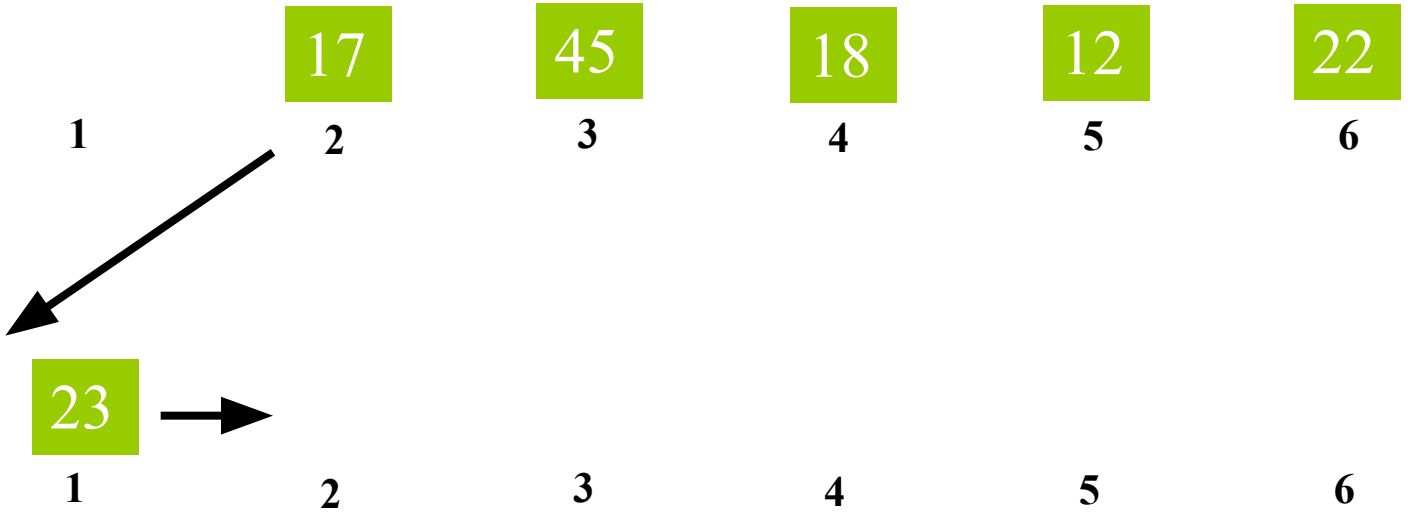
Our aim is to put them into nondecreasing order.

Insertion sort:example

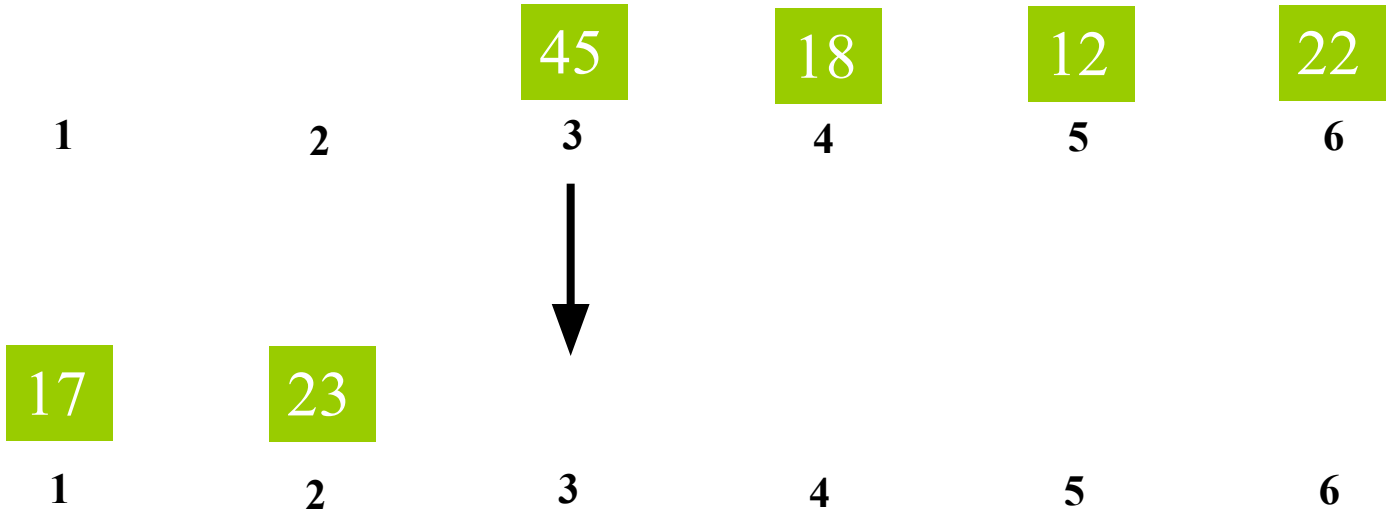
Insertion sort:example



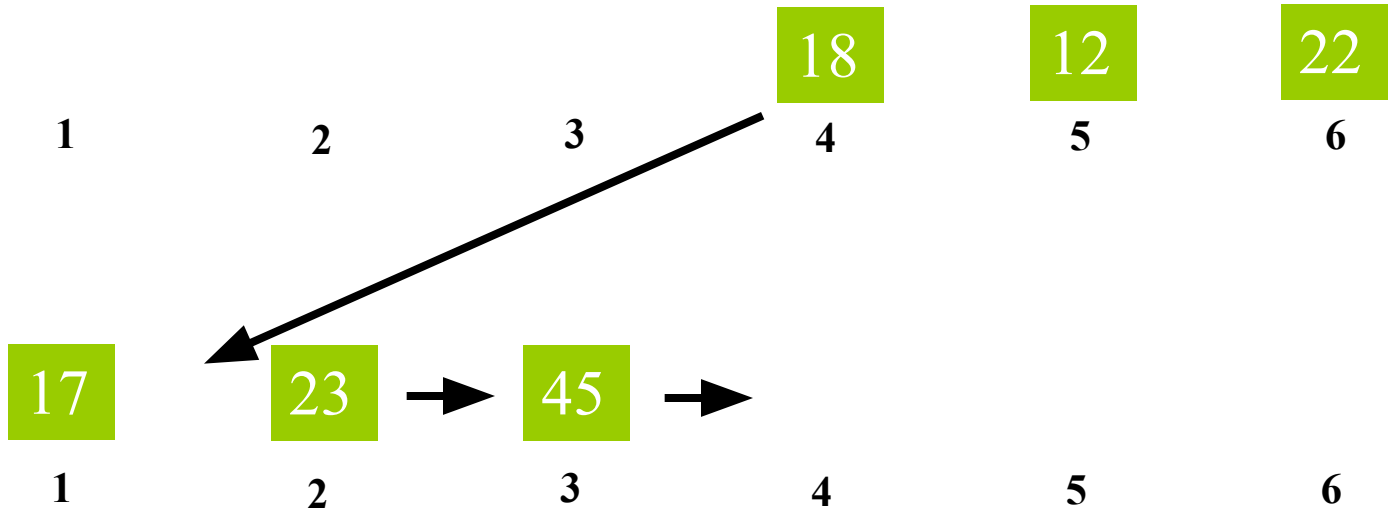
Insertion sort:example



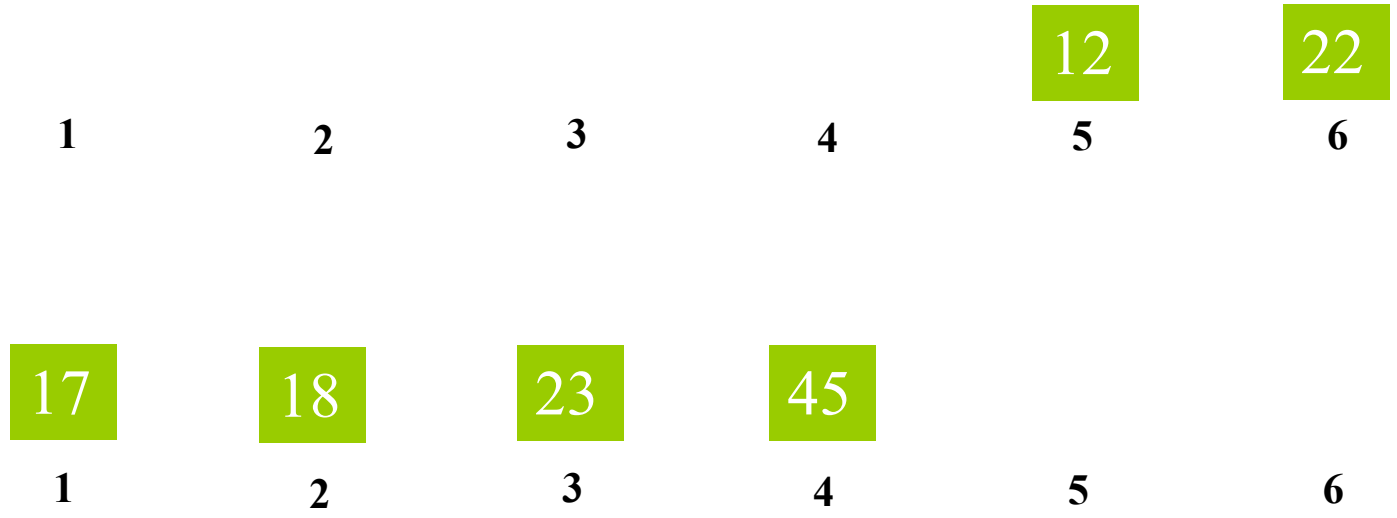
Insertion sort:example



Insertion sort:example



Insertion sort:example



Insertion sort:example

1	2	3	4	5	6
12	17	18	22	23	45
1	2	3	4	5	6

Insertion sort : Pseudocode

A is an array of numbers of length *n*, *B* is an empty array

```
B[1] = A[1]
for j = 2 to n
{
    i = j - 1
    while 0 < i and A[j]  <
 B[i]
        i = i - 1
    for k = j downto i + 2
        B[k] = B[k-1]
    B[i+1] = A[j]
}
```

Insertion of *j*th card

Finding the place to insert *A[j]*

Shifting a part of array *B*

Inserting *A[j]*

Analysis

	Insertion-Sort(A)	Cost	Times (*Iterations*)
1	*B[1] = A[1]*	*c*1	
2	for *j = 2* to *n* {	*c*2	*1*
3	*i = j - 1*	*c*3	*n*
4	while *0 < i* and *A[j] < B[i]*	*c*4	*n* - 1
5	*i = i - 1*	*c*5	$\sum_{j=2}^{n} t_j$
6	for *k = j* downto *i + 2*	*c*6	$\sum_{j=2}^{n} (t_j - 1)$
7	*B[k] = B[k-1]*	*c*7	$\sum_{j=2}^{n} t_j$
8	*B[i+1] = A[j]* }	*c*8	$\sum_{j=2}^{n} (t_j - 1)$
			n - 1

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} t_j + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Analysis (Best case)

- **Best Case**:

  Array already sorted, $t_j = 1$ for all j.

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5(0) + c_6(n-1) + c_7(0) + c_8(n-1)$$

$$= (c_2 + c_3 + c_4 + c_6 + c_8) * n + c_1 - c_3 - c_4 - c_6 - c_8$$

$$= an + b \qquad \text{(Linear in } n\text{)}$$

Analysis (Worst case)

We are usually interested in the worst-case running time

- **Worst Case**:

Array in reverse order,

$$t_j = j \text{ for all } j.$$

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{(n-1)n}{2}$$

$$+ c_6\left(\frac{n(n+1)}{2} - 1\right) + c_7\frac{(n-1)n}{2} + c_8(n-1)$$

$$= \left(c_4/2 + c_5/2 + c_6/2 + c_7/2\right)n^2 + \left(c_2 + c_3 + c_4/2 - c_5/2 + c_6/2 - c_7/2 + c_8\right)n$$

$$+ c_1 - c_3 - c_4 - c_6 - c_8 = an^2 + bn + c \quad \textit{(quadratic in n)}$$

Note that $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$

Insertion sort: Average case

- Is it closer to the best case (*n* comparisons)?
- The worst case (*n* * (*n*-1) / 2) comparisons?
- It turns out that when random data is sorted, insertion sort is usually closer to the worst case
  - Around *n* * (*n*-1) / 4 comparisons
  - Calculating the average number of comparisons more exactly would require us to state assumptions about what the "average" input data set looked like
  - This would, for example, necessitate discussion of how items were distributed over the array
- Exact calculation of the number of operations required to perform even simple algorithms can be challenging

- (for instance, assume that each initial order of elements has the same probability to occur)

Insertion sort runtimes

- Best case: $\Omega(n)$. It occurs when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.
- Average case: $\theta(n^2)$ since there is a wide variation with the running time.
- Worst case: $O(n^2)$ if the numbers were sorted in reverse order.

Complexity Bounding Functions

- There are 3 bounding functions to consider when talking about the complexity of an algorithm, the upper bound, the lower bound, and a tight bound:

- Upper bound: f(n) = O(g(n)) Lower bound: f(n) = Ω(g(n))
- Tight bound: f(n) = Θ(g(n))

- The upper bound, f(n) = Θ(g(n)), means that for some constant c and value n0, where positive, the product of the two will always lie on or above f(n).
- The lower bound, f(n) = Ω(g(n)), means that for some constant c and value n0, where positive, the product of the two will always lie on or below f(n).
- The tight bound, f(n) = Θ(g(n)), means that two constants c1 and c2 exist for the value n0, where positive, that represent an upper bound and lower bound.

- This means for an algorithm to be tightly bounded, f(n) = O(g(n)) and f(n) = Ω(g(n)) must both be true.

Asymptotic Examples

- $20\ n^3 + 10n + 5$
- $\log n + 5$
- $n^2 + 10$
- $5n-20$
- $5n +100$
- 4000
- $5n/2 + 100$

Developing an Algorithm

1. Identify the Inputs
2. Identify the Processes
3. Identify the Outputs
4. Develop a HIPO Chart
5. Identify modules

1.Identify the inputs

- What data do I need?
- How will I get the data?
- In what format will the data be?

2.Identify the Processes

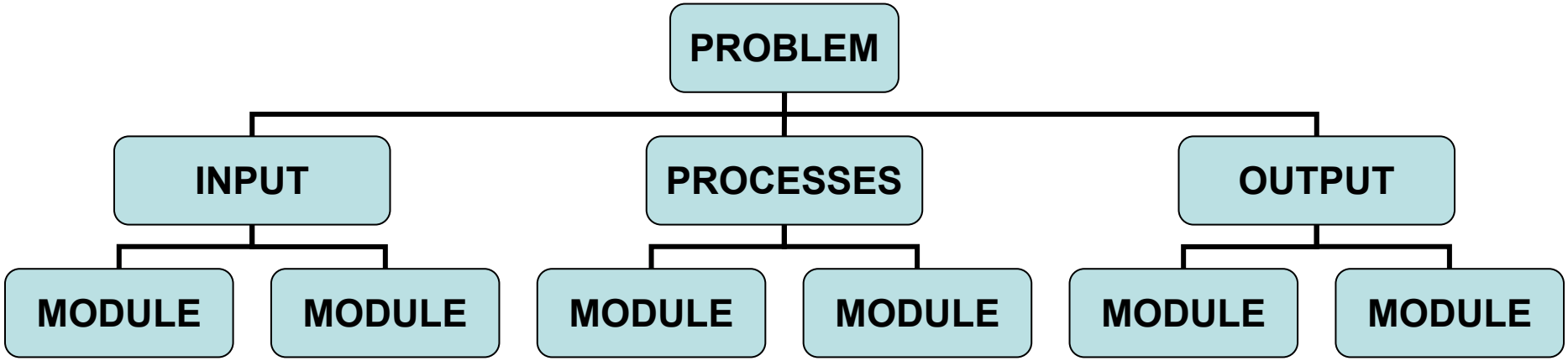
- How can one manipulate data to produce meaningful results?
- Data vs. Information

3.Identify the outputs

- What outputs do one needs to return to the user?
- What format should the outputs take?

4.Developing a HIPO Chart

Hierarchy of Inputs Processes & Outputs



5. Identify the relevant modules

- How can I break larger problems into smaller, more manageable pieces?
- What inputs do the modules need?
- What processes need to happen in the modules?
- What outputs are produced by the modules?

THANK YOU