M3. Algorithm Complexity

Instructor: Manikandan Narayanan

Weeks 2-3

CS2700 (PDS) Moodle: https://courses.iitm.ac.in/course/view.php?id=4892

Acknowledgment of Sources

- Slides based on content from related
  - Courses:
    - IITM – Profs. **Rupesh**/Krishna(S)/Prashanth/Kartik's PDS (Thy/Lab) offerings (slides, quizzes, notes, lab assignments, etc. for instance from Rupesh's Jul 2019 offering - www.cse.iitm.ac.in/~rupesh/teaching/pds/jul19/)
    - *Most slides are based on Rupesh Nasre's slides – we thank him and acknowledge by marking **[RN]** in the bottom right of these slides.*
    - *O(n) Kadane's algorithm for MSS and its proof available at:* https://www.math.umd.edu/~immortal/CMSC351/notes/maxcontiguoussum.pdf
  - Books:
    - **Main textbook:** *"Data Structures and Algorithm Analysis in C++"* by ***Weiss*** (content, figures, slides, exercises/questions, etc.). – cited as [WeissBook]
    - Additional/optional book: *"Practice of Programming"* by *Kernighan and Pike* (style of programming, programming exercises/questions, etc.) – cited as [KPBook]

Outline for Module M3

- M3 Algorithm Complexity
  - **M3.1 Introduction (to worst-case, asymptotic complexity)**
  - M3.2 Background on growth rate of functions (big-Oh and cousins)
  - M3.3 Maximum Subarray Sum (MSS - cubic, quadratic and linear time) and Binary Search (logarithmic time) algorithms

Algorithms

- For the same problem, there could be multiple algorithms.
- An algorithm is a clearly specified sequence of simple instructions that solve a given problem.
  - An algorithm, by definition, terminates.
  - Otherwise, the sequence of instructions constitutes a *procedure*.
- The algorithm should be so clear to you that you should be able to make a machine understand it.
  - This is called **programming**.

[RN]

Algorithm Efficiency

- For the same problem, there could be multiple algorithms.
- We prefer the ones that run *fast*.
  - I don't want an algorithm that takes a year to sort!
  - By the way, there are computations that run for months!
  - Operating systems on servers may run for years.

    [rupesh@aqua ~]$ uptime
    17:15:41 up 228 days, 3:50, 155 users, load average: 0.21, 0.22, 0.26
- We would like to compare algorithms based on their speeds.
  - Mathematical model to capture algorithm efficiency.

[RN]

Misconceptions

- Program P1 takes 10 seconds, P2 takes 20 seconds, so I would choose P1.
  - Execution time is input-dependent.
  - Execution time is hardware-dependent.
  - Execution time is machine-load dependent.
  - Execution time is run-dependent too!
  - Other factors play a role; for instance:
  - whether the program is running in hostel or in DCF
  - Or whether in Chennai or Kashmir
  - Or whether in May or December!

```
$ time ls >/dev/null

real    0m0.002s
user    0m0.000s
sys     0m0.001s

// on another machine
$ time ls >/dev/null

real    0m0.005s
user    0m0.000s
sys     0m0.004s
```

[RN]

Examples

```
a = a + b;
b = a – b;
a = a – b;
```

Irrespective of the values of a and b, this program would take time proportional to three instructions.

```
for (ii = 0; ii < N; ++ii)
        a[ii] = 0;
```

Proportional to N.

```
for (ii = 0; ii < N; ++ii)
        for (jj = 0; jj < M; ++jj)
                mat[ii][jj] = ii + jj;
```

Proportional to N*M.

```
int fun(int n) {
        return (n == 0 ? 1 : 4 * fun(n / 3));
}
```

?

[RN]

Examples

```
a = a + b;
b = a – b;
a = a – b;
```

```
a[ii] = 0;
```

```
x = y;
if (x > 0)
        y = x + 1;
else
        z = x + 1;
```

```
for (ii = 0; ii < 1000; ++ii)
        a[ii] = 0;
```

- **All of these are equally efficient!**
- They all perform constant-time operations.
- We denote those as O(1).

[RN]

Examples

```
a[0] = 0;
a[1] = 0;
a[2] = 0;
...
a[n – 5] = 0;
```

```
int fact(int n) {
        if (n > 0) return n * fact(n – 1);
        return 1;
}
```

```
for (ii = 0; ii < 2*n; ++ii)
        a[ii] = 0;
```

- **All of these are equally efficient!**

- They all perform linear-time operation *(linear in n)*.
- We denote those as O(n).

[RN]

A note on input size

Input size, usually denoted by n or N, technically refers to the number of bits used to represent the input.

In many scenarios, it is simply the number of data points (of a primitive data type) provided as input.

Examples:
1. If an algorithm taken a vector of 10 integers, then $N = 10$.

2. If an algorithm takes a positive integer "a" to check if it is prime, then $N = 1$ (and not "a"). Technically, $N = \log_2 a$ bits.

A note on worst-case analysis

An algorithm can perform different number of operations, and therefore take different time, on different input instances.

E.g., binary search when key is right in the middle position of the array, vs., key in some other locations.

We are interested in an algorithm's running time for a worst-case or adversarial input.

We are also typically interested in the worst-case running time for large values of the input size N (also known as asymptotic running time).

So, (running time) complexity here refers to worst-case (over inputs) and asymptotic (wrt N) running time taken by an algorithm.

Outline for Module M3

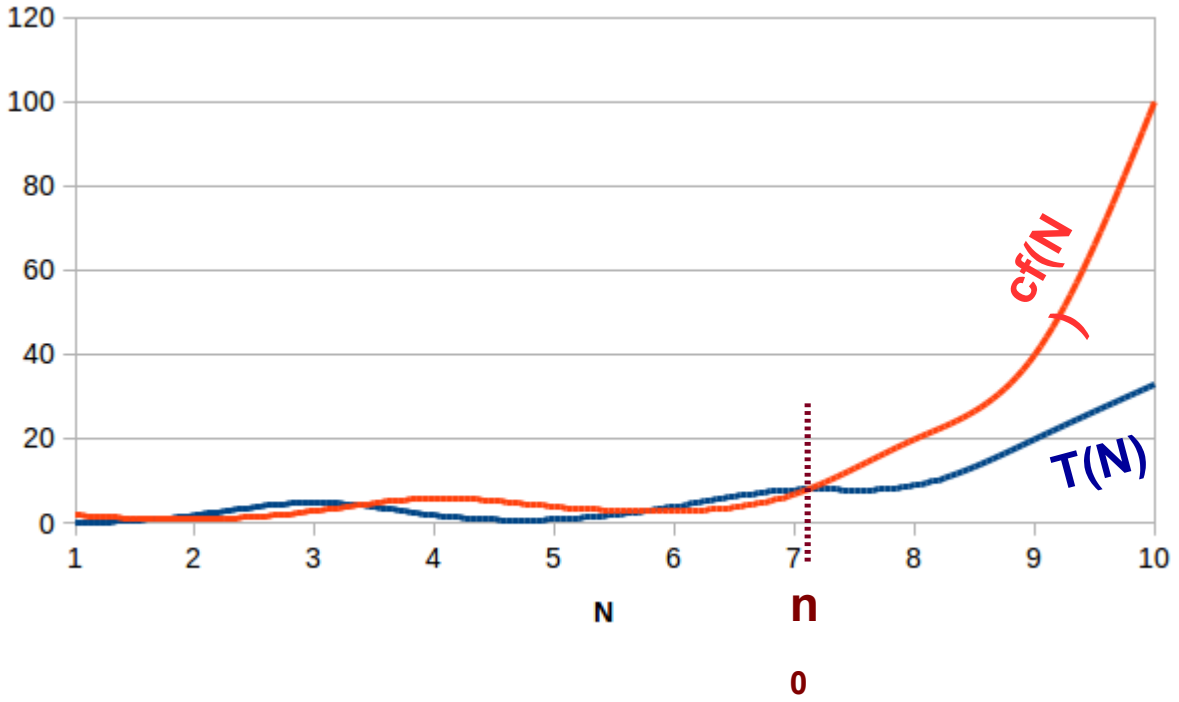
- M3 Algorithm Complexity
  - M3.1 Introduction (to worst-case, asymptotic complexity)
  - **M3.2 Background on growth rate of functions (big-Oh and cousins)**
  - M3.3 Maximum Subarray Sum (MSS - cubic, quadratic and linear time) and Binary Search (logarithmic time) algorithms

Definition

- T(N) = O(1) if $T(N) \leq c$ when $N \geq n_0$, for some positive c and n_0.
- T(N) = O(N) if $T(N) \leq cN$ when $N \geq n_0$, for some positive c and n_0.
- In general,
- **T(N) = O(f(N))** if there exist positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.
- Complexity captures the rate of growth of a function.

[RN]

Big O



- In general,
- **T(N) = O(f(N))** if there exist positive constants c and n_0 such that T(N) ≤ cf(N) when N ≥ n_0.
- The complexity is upper-bounded by c*f(N).

[RN]

Examples

```
a = a + b;
b = a – b;
a = a – b;
```

O(1)

Irrespective of the values of a and b, this program would take time proportional to three instructions.

```
for (ii = 0; ii < N; ++ii)
        a[ii] = 0;
```

O(N)

Proportional to N.

```
for (ii = 0; ii < N; ++ii)
        for (jj = 0; jj < M; ++jj)
                mat[ii][jj] = ii + jj;
```

O(N*M)

Proportional to N*M.

```
int fun(int n) {
        return (n <= 1 ? 1 : 4 * fun(n / 3));
}
```

?

[RN]

Solving for Time Complexity

$T(n) = c1 + T(n/3)$ and $T(1) = 1$

$= c1 + [c1 + T(n/9)]$

$= 2*c1 + T(n/3^2)$

$= 3*c1 + T(n/3^3)$

$= k*c1 + T(n/3^k)$

If $n == 3^k$,

$T(n) = \log_3 n * c1 + T(1) = c1*\log_3 n + 1 = O(\log_3 n)$

```
int fun(int n) {
        return (n <= 1 ? 1 : 4 * fun(n / 3));
}
```

[RN]

Types/Classes of Complexities

(see [WeissBook] for simpler defns. using only big-Oh)

Symbol	Name	Bound	Inequality
O(...)	Big O	Upper	T(n) <= cf(n)*
Ω(...)	Big Omega	Lower	T(n) >= cf(n)*
Θ(...)	Theta	Upper and Lower	$c_1 f(n) <= T(n) <= c_2 f(n)$*
o(...)	Little O	Strictly Upper	T(n) < cf(n)**
ω(...)	Little Omega	Strictly Lower	T(n) > cf(n)**

* There exists positive c, n_0 s.t. inequality holds for all $n > n_0$.

** For all positive c, there exists a positive n_0 s.t. inequality holds for all $n > n_0$.

[RN]

Types of Complexities

To reemphasize, see [WeissBook] for simpler defns. of complexity types/classes using only big-Oh!!

Notes

- Θ means O and Ω. It is a stronger guarantee on the complexity.
- If T(n) is Θ(n), then T(n) is also $O(n^2)$, also O(nlogn), also $O(n^3)$, $O(n^{100})$, $O(2^n)$; but it is not O(logn) or O(1).
- Big O is also called Big Oh.
- Asymptotically, all these functions have the same growth rate, i.e., T(n) ~= T(n+const.) ~= T(n/2) ~= T(1000n) ~= T(nlog2) ~= $T(2^{\log n})$ (here ~= implies they are of the same growth rate)
- $\text{Log}_2(x)$, that is, *log to the base 2* is sometimes written as lg(x).
- If T(n) = O(f(n)) then f(n) = Ω(T(n)).

[RN]

Complexity Arithmetic

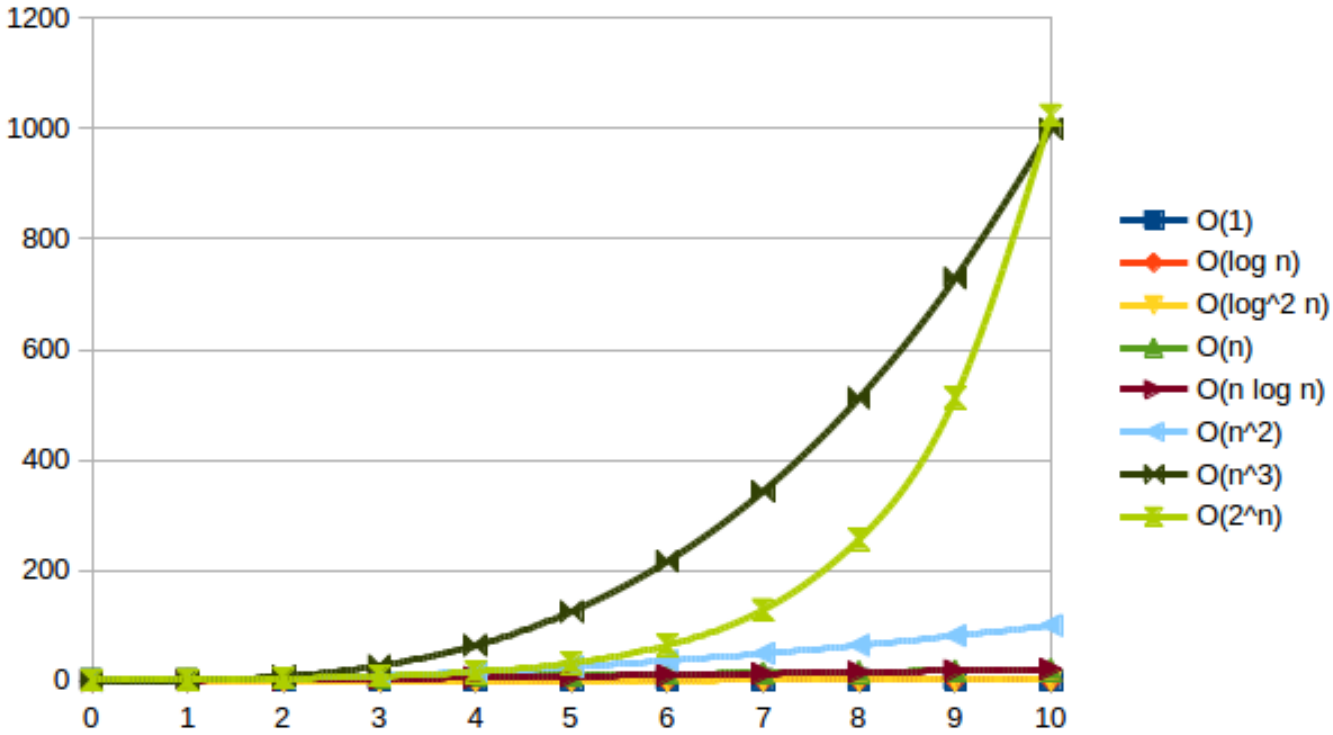
- If T1(n) = O(f(n)) and T2(n) = O(g(n)), then
  - T1(n) + T2(n) = **max**(O(f(n), O(g(n)))
  - T1(n) * T2(n) = O(f(n) * g(n))

- **Classwork**:
  - Write a C code that requires the use of T1(n) + T2(n).
  - Write a C code that requires the use of T1(n) * T2(n).

[RN]

Typical Complexities

Function	Name
c	Constant
Log N	Logarithmic
Log^2 N	Log-squared
N	Linear
N log N	Superlinear
N^2	Quadratic
N^3	Cubic
2^N	Exponential



Homework: Find which one grows faster: nlogn or $n^{1.5}$.

[RN]

Complexity Comparison

- Given two complexity functions f(n) and g(n), we can determine relative growth rates using $lim_{n \to \infty} f(n) / g(n)$, using L'Hospital's rule.
- Four possible values:
  - The limit is zero, implies f(n) = o(g(n)).
  - The limit is $c \neq 0$, implies f(n) = Θ(g(n)).
  - The limit is ∞, implies g(n) = o(f(n)).
  - The limit oscillates, implies there is no relation.

[RN]

Facets of Efficiency

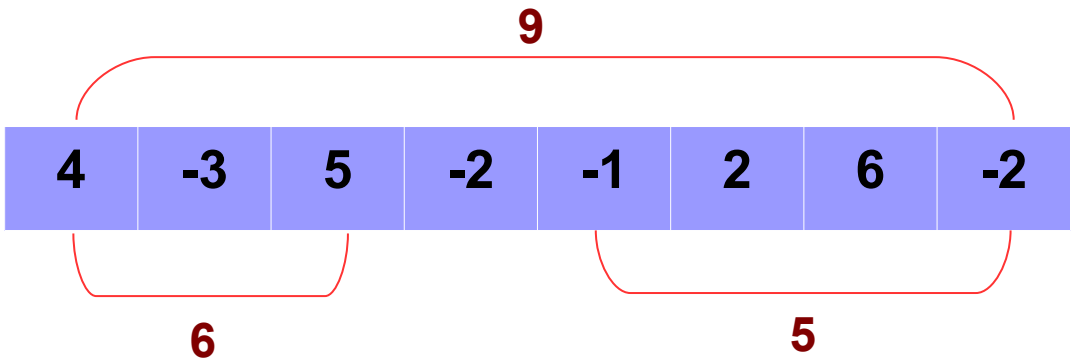
- An algorithm or its implementation may have various facets towards efficiency.
  - Time complexity (which we usually focus on)
  - Space complexity (considered in memory-critical systems such as embedded devices)
  - Energy complexity (e.g., your smartphones)
  - Security level (e.g., program with less versus more usage of pointers)
  - I/O complexity
  - ...

[RN]

Outline for Module M3

- M3 Algorithm Complexity
  - M3.1 Introduction (to worst-case, asymptotic complexity)
  - M3.2 Background on growth rate of functions (big-Oh and cousins)
  - **M3.3 Maximum Subarray Sum (MSS - cubic, quadratic and linear time) and Binary Search (logarithmic time) algorithms**

Max. Subsequence Sum



- **Problem Statement**
- Given an array of (positive, negative, zero) integer values, find the largest subsequence sum.
- A subsequence, better called as subarray, is a consecutive set of elements. If empty, its sum is zero.

[RN]

MSS: Algorithm 1

Exhaustive Algorithm

For each possible subsequence

Compute sum

If sum > current maxsum

current maxsum = sum

Return current maxsum

How many subsequences?

What is the complexity of this part?

Algorithm 1 takes $O(N^3)$ running time.

Source: mss1.cpp

[RN]

MSS: Algorithm 1

- Did we perform a tight mathematical analysis?
- To be precise, we need the following number of operations:

- $\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} O(1)$

$j - i + 1$

We will assume O(1) to be equal to constant 1. This would affect only the constant in BigOh.

[RN]

MSS: Algorithm 1

- Did we perform a tight mathematical analysis?
- To be precise, we need the following number of operations:

$$\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} (j - i + 1)$$

sum of first N-i integers
=
$(N - i)(N - i + 1) / 2$

[RN]

MSS: Algorithm 1

- Did we perform a tight mathematical analysis?
- To be precise, we need the following number of operations:

- $\sum_{i=0}^{N-1} (N - i)(N - i + 1) / 2$

$= (N^3 + 3N^2 + 2N) / 6$

$= O(N^3)$

The analysis is tight.
Is the algorithm tight?

[RN]

MSS: Algorithm 2

- **Observation:**
  - $\sum_{k=i}^{j} A[k] = A[j] + \sum_{k=i}^{j-1} A[k]$

For each starting position i

For each ending position j

Incrementally compute sum

If sum > maxsum

maxsum = sum

Return maxsum

Source: mss2.cpp

What is the complexity of this algorithm?

[RN]

MSS: Algorithm 3

- **Observation**: Discard fruitless subsequences early.
  - e.g., in {1, 2, -8, 4, -3, 5, -2, -1, 2, 6, -2}, we need not consider subsequences {-2} or {-2, -1} or even {-2, -1, 2} or {1, 2, -8, 4}.

For each position
  Add next element to sum
  If sum > maxsum
    Maxsum = sum
  Else if sum is negative
    sum = 0

Are you kidding?
This shouldn't work.
This is linear time algorithm!

Source: mss3.cpp

[RN]

A more careful look at the O(n) algo. for MSS, aka Kadane's algo.

Loop invariant: maintain

- MSS across all subarrays **ending at position *i*** (DP[i]), and
- MSS among all subarrays of a[0…i].

"Dynamic Programming" (DP) idea:

- Construct optimal solution of problem from optimal solution of its subproblems.
  - $DP[0] = 0$
  - $DP[i] = \max\{DP[i-1] + arr[i], arr[i]\}$

Why is the above true?

- Due to "Optimal substructure property", wherein optimal solution of a problem contains optimal solution to appropriately defined subproblems.
- Formally, prove the DP recurrence relation above, as shown next.

Kadane's algo. – pseudocode and correctness

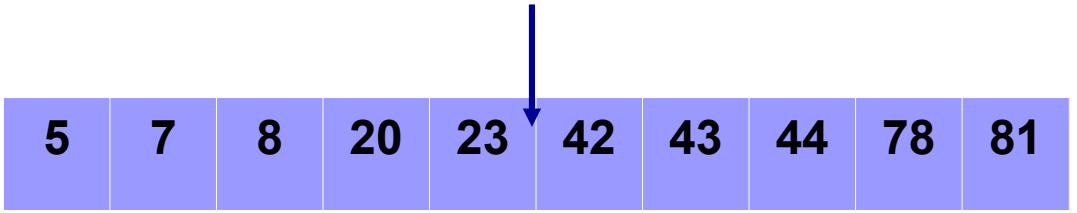
Pseudocode and correctness proof taken as is from:
https://www.math.umd.edu/~immortal/CMSC351/notes/maxcontiguoussum.pdf]

```
/* Kadane's Algo. – Equivalent to "MSS: Algorithm 3" shown in previous slides */
//PRE: A is a list of length n > 0.
maxoverall = A[0]
maxendingati = A[0]
for i = 1 to n-1
  maxendingati = max(maxendingati+A[i], A[i])
      maxoverall = max(maxoverall, maxendingati)
end
//POST: maxoverall is the maximum contiguous sum.
```

Proof (of recurrence relation; marked Theorem 4.2.1 in above link) shown in chalk-board.

Binary Search

- Searching in an array takes linear time O(N).
- If the array is sorted already, we can do better.
- We can cut the *search space* by half at every step.



Classwork: Write the code for binary search.
Source: **bsearch.cpp**

[RN]

Binary Search

- Constant amount of time required to
  - Find the mid element.
  - Check if it is the element to be searched.
  - Decide whether to go to the left or the right.
  - Cut the search space by half.
- $T(N) = T(N/2) + O(1)$
  - Thus, $T(N)$ is $O(\log N)$.

[RN]

Exercises

- Solve exercises at the end of Chapter 2 of Weiss's book.

[RN]