ICS 103

Data Structures and Algorithms

International Institute of Information Technology

Hyderabad, India

The Need for Analysis

- From the previous lecture, we agree that efficiency of representation and efficiency of operation are both important.
- How to measure efficiency?
- What parameters are important in measuring efficiency?
- Need a standard notion.

The Need for Analysis

- What to measure? Several resources possible.
  - Time

The Need for Analysis

- What to measure? Several resources possible.
  - Time
  - Space

The Need for Analysis

- What to measure? Several resources possible.
  - Time
  - Space
  - Power

The Need for Analysis

- What to measure? Several resources possible.
  - Time
  - Space
  - Power
  - Physical facility cost
  - ....

The Need for Analysis

- What to measure? Several resources possible.
  - Time
  - Space
  - Power
  - Physical facility cost
  - ....
- Depending on the situation, one, or a combination of the above, assumes significance.
- In our discussion, let us focus on time.

The Need for Analysis

- We should first try to standardize our description of operations.
  - States what is allowed, how to describe an operation,
- Such a standard description is called an algorithm.
  - The word is attributed to an Arabian mathematician called al-Khowrazimi.
- An algorithm is a recipe for a solution and has input, output, definiteness, and finiteness.

How to Analyze?

- How do we measure the time taken?
- Most computers allow one to measure the time taken by a command to execute.
  - Use the time command on Unix/Linux based systems.
- A naïve approach is as follows:
  - Implement the algorithm on a given machine
  - Run it on a given input
  - Measure the time taken.

Several Pitfalls

- The naïve approach suffers from several pitfalls.
- For instance, say binary search on an array of 1 M entries on an Intel machine takes 0.1 microsecond.
  - Time taken on a given input may not hold a clue to time for some other input.
  - Size of input may affect the runtime
  - Machine model
  - System behaviour
- Need to be a bit more abstract.

A Three Step Approach

- We will first abstract out a machine model.

A Three Step Approach

- We will first abstract out a machine model.
- We will then abstract out a notion of measuring time.

A Three Step Approach

- We will first abstract out a machine model.
- We will then abstract out a notion of measuring time.
- We will then extend it to asymptotic behavior.

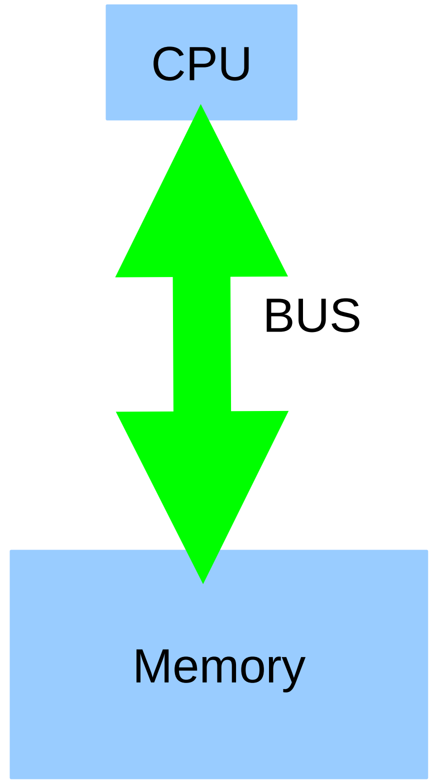
Step 1 – Abstracting the Machine

- What is a good machine model?
- How do you describe your computer?

Step 1 – Abstracting the Machine

- What is a good machine model?
- An abstract machine model should be able to generalize several existing models.
- A generally accepted model is the so called Random Access Machine (RAM) model.
- The RAM model is closely related to the von Neumann model of computation also.

The RAM Model



- A CPU and a memory connected by a bidirectional bus.
- Access to any cell of the memory possible, and has the same access time.

The RAM Model

- The CPU has
  - a limited set of registers
  - a program counter
  - supports program constructs such as
    - looping
    - recursion
    - jumping
    - branching

The RAM Model

- The CPU has a standard instruction set including:
  - Arithmetic operators : +, -, *, /
  - Logical operators : AND, OR, NOT
  - Conditional operators : =, <, >, <=, >=
  - Shift operators : <<, >>
  - Memory access operators : LOAD, STORE.

Today's Computers

- You will learn in CSO that today's computers are far from the kind we described in the abstraction.
- This abstraction however serves us well for now.
- Plus, better models are really complicated.

Step 2 : Measuring the Time Taken

- In reality, each of these operators take different number of machine cycles.
  - A LOAD typically takes more than time an ADD
- We will assume however that each takes the same number of cycles, or 1 unit of time.
  - For this reason, also called as the unit cost model.
- Measure the time taken also as a function of the input size.
- To measure input size, we count the number of bits required by the input.

Step 2 : Measuring the Time Taken

- In reality, each of these operators take different number of machine cycles.
  - A LOAD typically takes more than time an ADD
- We will assume however that each takes the same number of cycles, or 1 unit of time.
  - For this reason, also called as the unit cost model.
- Measure the time taken also as a function of the input size.
- To measure input size, we count the number of bits required by the input.

Step 2: Measuring the Time Taken

- Finally, the time taken is measured as a function of the input size.
  - n denotes the input size
  - T(n) denotes the time taken on input of size n.
- Several advantages in this approach.
  - Can know the time taken for any input.
  - Can compare different algorithms A and A' for the same problem using their time taken, T(n) and T'(n).
- Question: How to find T(n) for a given algorithm?

Step 2: Measuring the Time Taken

- Write the algorithm in reasonable pseudo-code
  - using only the operations provided on the RAM.
  - these are sometimes called as basic operations.
- Basic approach is to count the number of operations as a function of the input size.

An Example

```
Algorithm Sum-Integers(A)
    1.//A is an array of n integers.
    2. int i; sum = 0;
    3. for i = 1 to n do
        4. sum = sum + A[i];
    5. end-for
End Algorithm.
```

- The above example shows a program that adds n integers.
- We will count the time as a function of n.

An Example

- Line 1 is a comment and hence does not take any time.
- Line 2 declares two integers. If each takes a unit time, time for line 2 is 2 units.
- Line 3 starts a for loop running for n iterations. Let us assume that it takes 2 units to check the loop condition for every iteration.
  - Time for line 3 is 2n+1 units.
- Line 4 does 1 operation, hence takes 1 unit
  - For n iterations, line 4 takes n units.
- Line 5 takes no time as it indicates the end of the for loop.

```
Algorithm Sum-Integers(A)
    1.//A is an array of n integers.
    2. int i; sum = 0;
    3. for i = 1 to n do
        4. sum = sum + A[i];
    5. end-for
End Algorithm.
```

An Example

- Total time is the sum of the times for each line.
  - $T(n) = 0 + 2 + 2n+1 + n + 0 = 3n+3$.
- So the above algorithm has a time of 3n+3 units on an input of size n.
- Let us look at another example.

A Second Example

```
Algorithm MaximumSumContiguousSubsequence(A)
    1. // A is an array of n integers.
    2. int maxSum = 0;
    3. for i = 1 to n do
        4. int sum = 0
        5. for j = i to n do
            6. sum = sum + A[j];
        7. end-for
        8. if(sum > maxSum)
            9. maxSum = sum
    10. end-for
End Algorithm.
```

– What does the above program do?

A Second Example

- Let us count the time for every line.
- Line 1 – 0 time
- Line 2 – 1 unit
- Line 3 – 2n+1 units
- Line 4 – 1 unit for every iteration
- Line 5 – 2(n-i+1)+1
- Line 6 – 1 unit for every iteration
- Line 7 – 1 unit for every iteration
- Line 8 – 1 unit for every iteration
- Line 10 – no time

```
Algorithm MSCS (A)
    1. // A is an array of n integers.
    2. int maxSum = 0;
    3. for i = 1 to n do
        4. int sum = 0
        5. for j = i to n do
            6. sum = sum + A[j];
        7. end-for
        8. if(sum > maxSum)
            9. maxSum = sum
    10. end-for
End Algorithm.
```

A Second Example

- How to know the number of times line 9 is executed?
  - Depends on the input, and not just its size.
- No easy way to resolve the question.

A Second Example

- How to know the number of times line 9 is executed?
  - Depends on the input, and not just its size.
- No easy way to resolve the question.
- Accepted notion: Worst case behavior
  - Consider the situation when the input forces the algorithm to take the maximum possible amount of time.
  - In the present case, it amounts to saying that line 9 is executed in every iteration.
  - Sometimes referred to as worst-case analysis.

A Second Example

- Later, we will see other notions such as
  - best case, and
  - average case
- Advantage of worst-case analysis:
  - Removes any assumption on the nature of the input
  - need to consider only the size of the input.
  - Also, gives a fair basis for comparison.
  - Other notions are also important, but this notion is more prevalent.

A Second Example

- Time taken by the second program is
  - $T(n) = 1 + (2n+1) + (2n+1) + \Sigma_i (2(n-i+1)+1) + 3(2n+1)$
  - Simplifying yields T(n).
- So, the runtime of this program is said to be a quadratic function of the input size.
- If time permits, we shall see that there is a better solution for this problem.
  - Solution uses dynamic programming technique.

Step 2 – A Generalization

- We can propose a few rules for the second step.
- Simple Statement : unit time
  - includes arithmetic, logical, Boolean, ...
  - conditional statement :

```
if condition then
    Statement1
else
    Statement2
```

  - Time taken is the time to execute the condition + the maximum time taken between Statement1 and Statement2.

Step 2 – A Generalization

- Loop statement
  - for (loop init., condition, increment)   statement;
  - The time taken equals the product of the number of iterations and the time taken by the statement plus the time for loop condition and the increment evaluation.
  - What about nested loops?
    - Consider a nested product.

An Example

```
1.for i = 1 to n do
2.for j = 1 to n do
    3.C[i,j] = 0;
    4.for k = 1 to n do
        5.C[i,j] = C[i,j] + A[i,k].B[k,j]
    6.end-for
7.end-for
8.end-for
```

- Consider the matrix multiplication code.
- Matrix C = B . A, each of dimension nxn.

An Example

- Let us use the above example and the generalizations.
- Line 3
  - takes one unit time per iteration.
  - nested loop of n.n = n^2 iterations.
  - Total time for line 3 = n^2 units.
- Line 5
  - takes one unit time per iteration.
  - nested loop of n.n.n = n^3 iterations.
  - Total time for line 5 = n^3 units.

An Example

- Line 1 takes 2n+1 units of time.
- Line 2 takes 2n+1 units of time per iteration.
  - No. of iterations = n.
  - Total time for line 4 = n.(2n+1).
- Line 4 takes 2n+1 units of time per iteration.
  - No. of iterations = n^2.
  - Total time for line 4 = n^2.(2n+1).
- Lines 6, 7, 8, take no time.

An Example

- Total time taken by the program = $2n+1 + n(2n+1) + n^2 + n^2(2n+1) + n^3 = 3n^3+4n^2+3n+1$ units.
- So, matrix multiplication takes time proportional to the cube of the matrix dimensions.

Step 3 – Asymptotic Analysis

- Step 2 is useful but a bit too high on detail.
- Can we do away with some detail and focus on the big picture?
  - What is the big picture?
- Advantage with the big picture style
  - Hides unnecessary detail.
  - Good from a analytical view point.
- A word of caution: Even small detail is useful from a practical or empirical view point.

Step 3 – Asymptotic Analysis

- As part of the big picture, we will study the asymptotic behavior of the runtime.
- Asymptotic behavior tells us the behavior for large inputs, ignoring any aberrations for small inputs.
- A neat way to compare run-times of algorithms.
- Need a few definitions in this direction.

Step 3 – Asymptotic Analysis

- Consider our earlier examples and their runtimes
  - $3n+3$ for the sum of an array of integers
  - $3n^2/2 + 7n + ?$ for the maximum contiguous sum
  - sqrt(n) for the prime factorization of n
  - $3n^3+4n^2+3n+1$ for matrix multiplication
  - $\text{Log}^2\ n$ operations for the modular exponentiation
  - ....
- Need a way to simplify representing these runtimes further.
- Focus on how they grow with respect to n.

  Need not worry about the small detail.

Step 3 – Asymptotic Analysis

- Imagine the following definition. Take the higher order term, or the dominating term as the big picture.
- So, the first runtime is 3n, the second is $3n^2/2$, and so on.
- But, how to study "dominating" runtimes such as $3n^2/2$ and $n^2/2$.
    - Can we treat them as similar?
- Need a better definition that does not even care for constants.

Step 3 – Asymptotic Analysis

- Definition (Big-O) : Given two functions f, g : N -> N, we say that $f(n) \in O(g(n))$ if there exists two positive constants c; n_0 such that $f(n) \leq c. g(n)$ for all $n \geq n_0$.
- How to view this definition?
  - We are interested to see whether g(n) dominates f(n), but c.g(n) dominates f(n) for a positive constant c.
  - Also, beyond a certain fixed point n_0.
  - Leaves the order between f(n) and g(n) before n_0 completely unspecified.

Step 3 – Asymptotic Analysis

- A picture to illustrate the definition.
- The above definition lets us write f(n) = 1000n + 100 as belonging to O(g(n)) where g(n) = n.
  - What are c, and n_0 in this case?
  - So the growth rate of f(n) in this example is of the order of n, also called linear.

Step 3 : Asymptotic Analysis

- Another example:
  - $f(n) = 165n^2 + n^{1/3}$, and $g(n) = 0.01n^2$.
  - In this case also, it holds that $f(n) \in O(g(n))$.
  - What are c and n_0?
  - Here, we say that f(n) has a quadratic growth rate.
- More examples and general rules follow.

Step 3 : Asymptotic Analysis

- $\log^k n \in O(n)$ for any constant $k > 0$.
- If $f(n)$ is a polynomial of degree k, then $f(n) \in O(n^k)$.
- $\log(n^k) \in O(\log n)$ for any constant k.
- If $f(n)$ is a constant independent of n, then $f(n) \in O(1)$.
- Simple style: Write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.
  - Analogy to real analysis: Big-O like <.

Step 3 : Asymptotic Analysis

- As an example, our matrix multiplication program can be now analyzed as follows:
  - It has one addition that is nested in three for loops
  - It has one initialization that is nested in two for loops.
  - So, the total time is $O(n^3+n^2) = O(n^3)$.

Step 3 : Asymptotic Analysis

- The O-notation helps one to bound a function from the above.
- Sometimes, it is helpful to bound a function also from the below.
  - Need a notation for this.

Step 3 : Asymptotic Analysis

- Definition (Big-Omega): Given two functions f, g : N -> N, we say that $f(n) \in \Omega(g(n))$ if there exists two positive constants c; n_0 such that $f(n) \in c.\ g(n)$ for all $n \in n_0$.
- How to view this definition?
  - We are interested to see whether f(n) dominates not just g(n), but c.g(n) for a positive constant c.
  - Also, beyond a certain fixed point n_0.
  - Leaves the order between f(n) and g(n) before n_0 completely unspecified.

Step 3 : Asymptotic Analysis

- The notation conveys that f() grows at a faster rate than g(n) for large values of n.
- Example 1
  - $f(n) = n^2$ and $g(n) = 1000n+500$
  - It holds that $f(n) \in \Omega(g(n))$.
  - What are c, n_0 in this case?
- Example 2
  - $f(n) = n^{1/2}$ and $g(n) = \log_2 n$
  - $n^{1/2} \in \Omega(\log_2 n)$
  - What are c, n_0 in this case?

Step 3 : Asymptotic Analysis

- The Ω notation suggests that f(n) > g(n), analogically.
- In this case, it indicates that f(n) grows faster than g(n) asymptotically.
- To simplify notation, we also write $f(n) = \Omega(g(n))$ instead of $f(n) \in \Omega(g(n))$.
- What if f(n) and g(n) have similar growth rates?
  - One more definition.

Step 3: Asymptotic Analysis

- Definition (Big-Theta): Given two functions f, g : N -> N, we say that f(n) $\in \Theta$(g(n)) if it holds that f(n) $\in$ Ω(g(n)) and f(n) $\in$ O(g(n)).
- If f(n) Θ (g(n)), by definition we can say that f and g grow at the same rate.
- To simplify matters, we also write that f(n) = Θ(g(n)) instead of f(n) $\in$ Θ(g(n)).

Step 3: Asymptotic Analysis

- Example 1
  - $f(n) = 3n^2 - 3n + 5$, $g(n) = n^2$
  - $f(n) = \Theta(n^2)$.
  - What are c, n_0 for each of the two relations?
- Example 2

- The Θ notation suggests that f(n) = g(n) as far as asymptotic growth rates are concerned.

Step 3: Asymptotic Analysis

- We have notation for f(n) = g(n), f(n) < g(n), and f(n) > g(n).
  - Analogous to law of trichotomy for real numbers.
  - So does it holds that for any two functions f(n) and g(n), one of the above three holds?
  - The answer is NO.
  - There are functions f(n) and g(n) such that neither f(n) = O(g(n), f(n) = Ω(g(n)), and f(n) = Θ(g(n)).
  - HW: Find such a pair of functions.

Dealing with Recursive Programs

- So far, our programs are iterative in nature.
  - nested loops, etc.
- Several natural recursive programs
  - merge sort, quick sort, etc..
- How can we analyze such programs?

Recursive Programs

```
Algorithm FindMinimum(A)
    1. candidate1 = A(1);
    2. candidate2 = FindMinimum(A[2..n]);
    3. return min{candiate1, candidate2};
End Algorithm.
```

- Start with the above example.
- Line 1 and 3 are an O(1) time operation.
- How to represent the time taken for line 2?
  - recurrence relations to the rescue.

Recurrence Relations

- A recurrence relation is a way of specifying a function or a sequence where the value of the function at a given input is defined in terms of one or more function outputs at smaller input values.
- Imagine that T(n), the time taken by the above program for an input of size n, is a function.
- We can write a recurrence relation for T(n) as follows.

Recurrence Relations

- Notice that in Line 2, we are calling the same function recursively for an input of size n-1.
  - So, T(n-1) will be the time taken for that recursive call.
  - Using this, we can write T(n) as T(n –1) + O(1).
- To solve this recurrence relation, we need to know some initial values, say T(1) or so.
  - But this is the case when we need an exact solution.
  - Do we need an exact solution?

Recurrence Relations

- We actually need an asymptotic analysis.
  - This means that we may not need exact initial values.
  - Typically, we assume that initial values are all O(1)
  - Justified because of the fact on inputs of size O(1), the runtime is also O(1).
- We'll now propose a few solution strategies for solving recurrence relations.

Solving Recurrence Relations

- The Substitution Method
  - Try to guess a solution to the recurrence relation.
  - Verify whether our guess is correct. The verification is often done using mathematical induction.
  - we substitute the guessed value in to the recurrence and hence the name.
- An example follows.

The Substitution Method

- Consider for recurrence relation $T(n) = T(n-1) + O(1)$.
- Let us try to guess that the solution is $T(n) = O(n)$.
  - which means that $T(n) \leq c.n$ for some constant c.
- Verification proceeds as follows.
  - Let the above hold for all inputs up to n.
  - For n+1, $T(n+1) = T(n) + O(1)$ according to the recurrence relation.
  - Substituting for T(n), we need to show that $T(n+1) \leq cn + O(1) \leq c(n+1)$ for a large c.
  - Hence, our guess is correct.

Another Example

- Consider, $T(n) = 2T(n/2) + n$.
- Guess, $T(n) = O(n \log n)$
  - Meaning that there exists a positive constant c, such that $T(n) \leq c\, n \log n$.
- Verification proceeds as follows.
- Base case:
- Step: Need to verify that $T(n) \leq cn \log n$.
  - $T(n) = 2T(n/2) + n \leq 2c(n/2) \log (n/2) + n = cn (\log n - 1) + n = cn \log n - (c-1)n \leq cn \log n$ if $c > 1$.
  - hence, we showed that $T(n) = O(n \log n)$.

Another Solution Method

- The Recursion Tree Method
  - Imagine a tree where each internal node represents the cost of a subproblem.
  - The root represents the cost of the entire problem.
  - Leaf nodes represent boundary values.
  - Summing up the costs of all the problems at a level gives us the per-level cost.
  - Adding up all the per-level costs gives us the total cost of the entire problem.
  - One can use the recursion tree to arrive at a good guess and then verify the guess as in the substitution method.

The Recursion Tree Method – Example

- $T(n) = 2T(n/2) + O(n)$

A General Method

- There is a general way to solve recurrence relations.
- (Masters Theorem) Let $a \geq 1$, $b > 1$ be constants and $f(n)$ be a function. Let $T(n)$ be defined as $T(n) = aT(n/b) + f(n)$. Then $T(n)$ can be bounded as follows.
  - If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = O(n^{\log_b a})$.
  - If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$.
  - If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant > 0, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ all sufficiently large n then $T(n) = \Theta(f(n))$.

Master's Theorem

- Compares f(n) with $n^{\log_b a}$ and decides which term dominates the solution to T(n).
- If f(n) is smaller then $n^{\log_b a}$ by a significant margin, then $T(n) = O(n^{\log_b a})$.
- If f(n) has the same growth rate as $n^{\log_b a}$ then, $T(n) = \Theta(n^{\log_b a}.\log n)$.
- If f(n) has a higher growth rate than $n^{\log_b a}$ then, and certain conditions hold, then $T(n) = \Theta(f(n))$.

Examples to Master's Theorem

- Consider our earlier recurrence $T(n) = 2T(n/2) + n$.
  - Here, $f(n) = n$, $a = 2$, $b = 2$.
  - $n^{\log_b a} = n^{\log_2 2} = n$. This indicates that $f(n)$ and $n^{\log_b a}$ have the same growth rate.
  - So, case ii of the theorem applies.
  - The solution is therefore, $T(n) = \Theta(n \log n)$.

Another Example

- Consider $T(n) = 7T(n/2) + 8n^2$.
  - We have, $f(n) = 8n^2$, $a = 7$, and $b = 2$.
  - $n^{\log_b a} = n^{\log_2 7}$. Notice that $f(n) = O(n^{\log_2 7})$.
  - Hence, case i of the theorem applies.
  - Therefore, $T(n) = O(n^{\log_b a})$.
  - Note: There is a matrix multiplication algorithm with the above recurrence. Read about the four Russian's algorithm or Strassen's algorithm.

Master's Theorem

- Does not apply to all cases.
  - It may happen that none of the three conditions hold.
  - Or the recurrence relation is not in the given form.
- But, a useful tool in most cases.

Lab Session

- The lab shall be used to implement a few algorithms with orders $O(n)$, $O(\log n)$, $O(n^2)$, $O(n \log n)$, and $O(n^3)$ and try to time these algorithms and plot them on a common plot.