CS 2009 – Design an analysis of Algorithm

Week-2

What is the specification of algorithm?

We can specify an algorithm by 3 types:

- Using Natural language
- Pseudocode
- Flowchart.

Analysis of an Algorithm

What is analysis of an algorithm?

- Algorithm analysis is an important part of a computational complexity theory.
- There two types of analysis:

  1.priori

  2.posteriori

- Priori Analysis is the theoretical estimation of resources required.

- On the other hand posterior Analysis done after implement the algorithm on a target machine. In a posteriori analysis, we analyze actual statistics about the algorithms consumption of time and space, while it is executing.

Strengthening the Informal Definition

- **Important Features:**
  - **Finiteness:** Algorithm should end in finite amount of steps
  - **Definiteness :** each instruction should be clear
  - **Input:** valid input clearly specified
  - **Output:** single/multiple valid output
  - **Effectiveness:** steps are sufficiently simple and basic
  - **Generality:** the algorithm *should* be applicable to all problems of a similar form

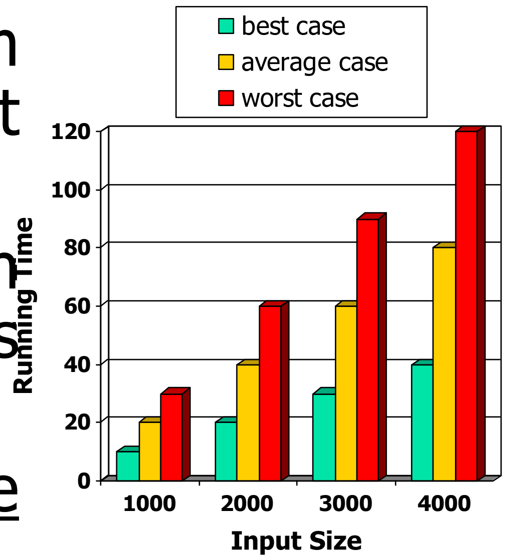
Algorithm analysis

Objective:

- Performance analysis is the criteria for judging the algorithm.
- It have direct relationship to efficiency.
- When we solve a problem ,there may be more then one algorithm to solve a problem, through analysis we find the run time of an algorithms and we choose the best algorithm which takes lesser run time.

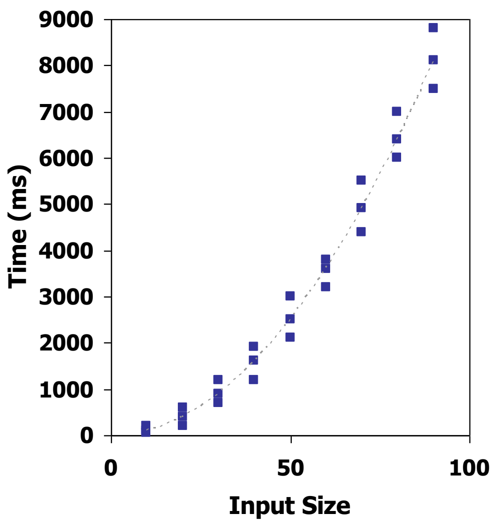
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like (tic; toc; in Matlab) to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

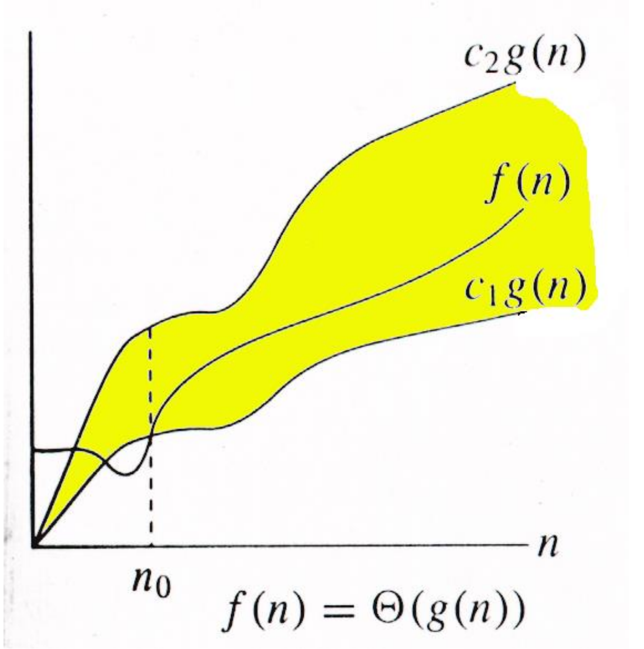
- Uses a high-level description of the algorithm (Pseudo code) instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Asymptotic Notations

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n) :$
$\exists$ **positive constants** c_1, c_2, **and** n_0,
such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Θ-notation

Math:

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

Engineering:

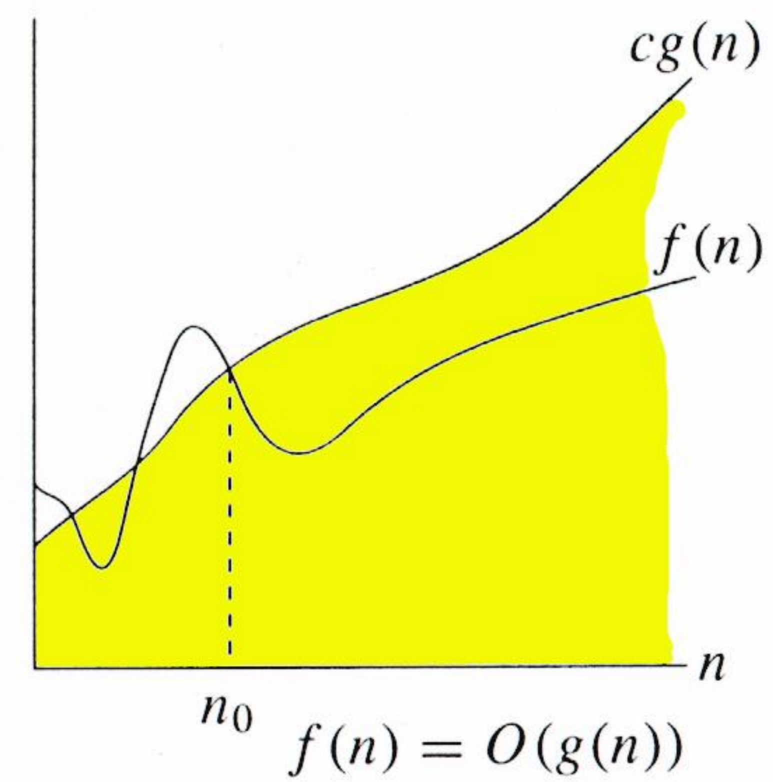
- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n)$:
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ $\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.



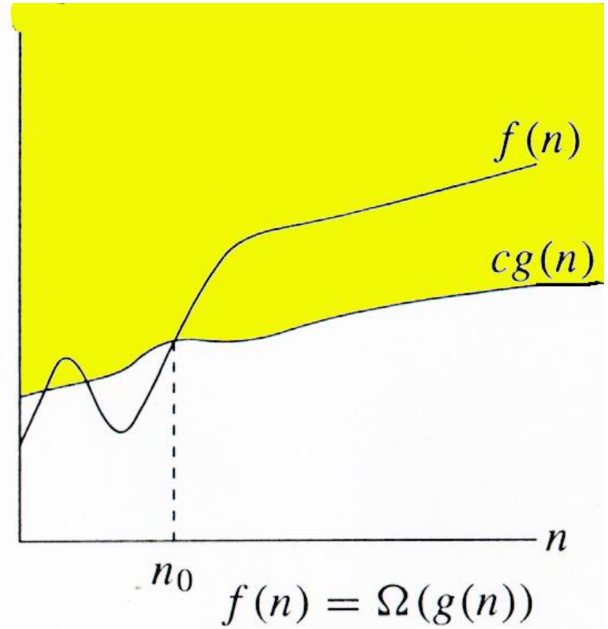
$g(n)$ is an *asymptotic upper bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.
$\Theta(g(n)) \subset O(g(n))$.

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n)$:
$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$

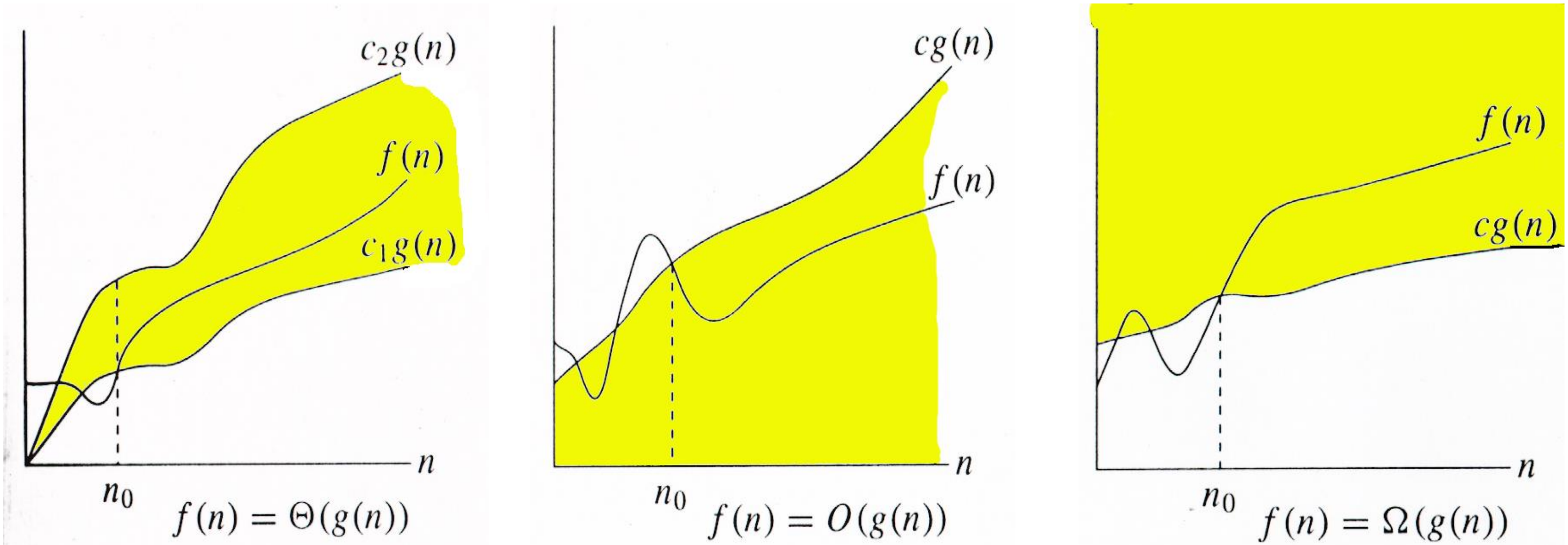


Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.
$\Theta(g(n)) \subset \Omega(g(n))$.

Relations Between Θ, *O*, Ω



Relations Between Θ, Ω, O

Theorem : For any two functions $g(n)$ and $f(n)$,

$f(n) = \Theta(g(n))$ iff

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

- I.e., $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Θ-notation

Math:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

Ω O

Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic Notation

- O notation: asymptotic "less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality":
  - f(n)= Θ (g(n)) implies: f(n) "=" g(n)

=> $(3n+2)=\Omega(n^2)$?????

- **No**

=> $f(n)= \Omega(n)$

if f(n) is lower bounded by 'n' then it can be lower bounded by any g(n) which is lower bounded by 'n'

i-e $(3n+2)=\Omega(\log n)$

$(3n+2)=\Omega(\log(\log(n)))$

But we always go for the closest lower bound or tighter lower bound

- If $(3n+2)=O(n)$

=> $(3n+2)=O(n^2)$

=> $(3n+2)=O(n^3)$

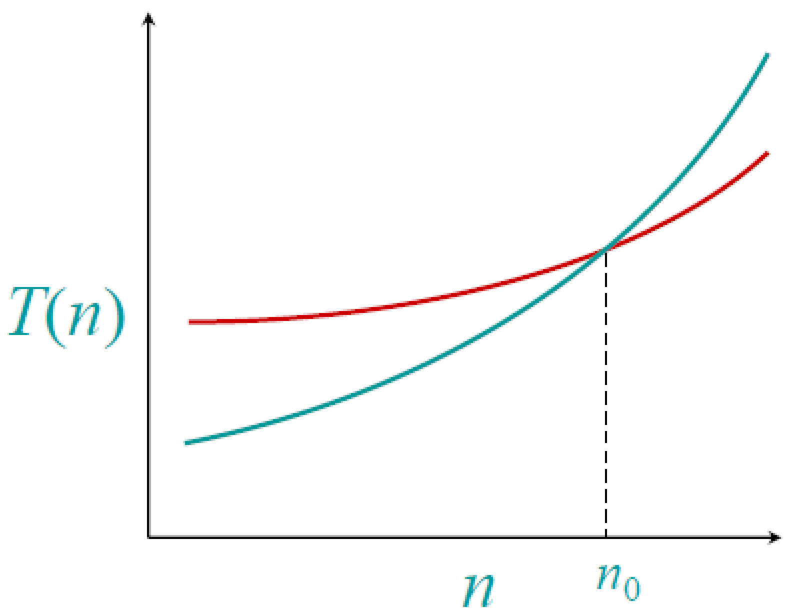
=> $(3n+2)=O(n^4)$ so on…

But we always go for the least upper bound or tighter upper bound

- $3n+2=\Theta(n)$
- $3n^2+2n+1=\Theta(n^2)$
- $6n^3+n^2=\Theta(n^3)$

Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

Examples of algorithms

- Here is a list of common asymptotic running times:
- $\Theta(1)$: Constant time; can't beat it!
- $\Theta(\log n)$: Inserting into a balanced binary tree; time to find an item in a sorted array of length n using binary search.
- $\Theta(n)$: About the fastest that an algorithm can run.
- $\Theta(n \log n)$: Best sorting algorithms.
- $\Theta(n^2)$, $\Theta(n^3)$: Polynomial time. These running times are acceptable when the exponent of n issmall or n is not to large, e.g., $n \leq 1000$.
- $\Theta(2^n)$, $\Theta(3^n)$: Exponential time. Acceptable only if n is small, e.g., $n \leq 50$.
- $\Theta(n!)$: Acceptable only for really small n, e.g. $n \leq 20$.

Best worst and average time complexity

- Best case: The algorithm take as min time as it can.
  - Searching item in array
  - Found first item as key
- Worst case: The algorithm take max time as it can
  - Searching item in array
  - Found the last item/ did not found item
- Average case: The algorithm takes average time
  - Found in middle of array (Just example)

What is the relationship between Big O, Θ, Ω and best, worst, and average case of an algorithm?

- The O and Ω notations do only describe the bounds of a function that describes the asymptotic behavior of the actual behavior of the algorithm. Here's an example
  - Ω describes the **lower bound**:
    - f(*n*) ∈ Ω(g(*n*)) means the asymptotic behavior of f(*n*) is **not less than** g(*n*)·*k* for some positive *k*, so f(*n*) is **always at least as much as** g(*n*)·*k*.
  - O describes the **upper bound**:
    - f(*n*) ∈ O(g(*n*)) means the asymptotic behavior of f(*n*) is **not more than** g(*n*)·*k* for some positive *k*, so f(*n*) is **always at most as much as** g(*n*)·*k*.

We are usually interested in the **worst case** complexity

Best worst average vs Notations

- These two can be applied on both the best case and the worst case for binary search:
- best case: first element you look at is the one you are looking for

Compare with finding max in array

  - Ω(1): you need *at least* one lookup
  - O(1): you need *at most* one lookup
- worst case: element is not present
  - Ω(log *n*): you need *at least* log *n* steps until you can say that the element you are looking for is not present
  - O(log *n*): you need *at most* log *n* steps until you can say that the element you are looking for is not present
- But often we do only want to know the upper bound or tight bound as the lower bound has not much practical information.

Estimating Running Time

- For example any Algorithm 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

  $$a\,(7n + 1) \leq T(n) \leq b(7n + 1)$$
- Hence, the running time $T(n)$ is bounded by two linear functions

Simple Example (1)

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N)
{
      int s=0;
      for (int i=0; i< N; i++)
         s = s + A[i];
      return s;
}
```

How should we analyse this?

Example of **Basic** Operations:

- Arithmetic operations: *, /, %, +, -
- Assignment statements
- Simple conditional tests: if (x < 12) ...
- method call (Note: the execution time of the method itself may depend on the value of parameter and it may not be constant)
- a method's return statement
- Memory Access
- We consider an operation such as ++ , += , and *= as consisting of two basic operations.
- **Note**: To simplify complexity analysis we will not consider memory access (fetch or store) operations.

Simple Complexity Analysis: Loops

- We start by considering how to count operations in **for**-loops.
  - We use integer division throughout.

- First of all, we should know the number of iterations of the loop; say it is **x**.
  - Then the loop condition is executed **x + 1** times.
  - Each of the statements in the loop body is executed **x** times.
  - The loop-index update statement is executed **x** times.

Simple Example (2)

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N){
    int s=0;         ← 1

    for (int i=0; i< N; i++)
        s = s + A[i];
    return s;
}
```

1,2,8: Once

3: N+1 times

4,5,6,7: Once per each iteration of for loop, N iteration

Total: 5N + 4

The *complexity function* of the algorithm is : $f(N) = 5N + 4$

Example explanation

- Estimated running time for different values of N for 5N+3:
  - N = 10 => 53 steps
  - N = 100 => 503 steps
  - N = 1,000 => 5003 steps
  - N = 1,000,000 => 5,000,003 steps
- As N grows, the number of steps grow in linear proportion to N for this function “Sum”

What Dominates in Previous Example?

- What about the +3 and 5 in 5N+3?
  - – As N gets large, the +3 becomes insignificant
  - – 5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance
- Asymptotic Complexity: As N gets large, concentrate on the highest order term:
  - Drop lower order terms such as +3
  - Drop the constant coefficient of the highest order term i.e. N
  - The 5N+3 time bound is said to "grow asymptotically" like N

Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)
- The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

Simple Complexity Analysis: Loops (with <)

- In the following for-loop:

```
for (int i = k; i < n; i = i + m){
        statement1;
        statement2;
 }
```

The number of iterations is: **(n – k) / m**

- The initialization statement, **i = k**, is executed **one** time.
- The condition, **i < n**, is executed **(n – k) / m + 1** times.
- The update statement, **i = i + m**, is executed **(n – k) / m** times.
- Each of **statement1** and **statement2** is executed **(n – k) / m** times.

Simple Complexity Analysis: Loop Example

- Find the exact number of basic operations in the following program fragment:

```
double x, y;
x = 2.5 ; y = 3.0;
for(int i = 0; i < n; i++){
    a[i] = x * y;
    x = 2.5 * x;
    y = y + a[i];
}
```

- There are 2 assignments outside the loop => 2 operations.
- The **for** loop actually comprises
- an assignment (i = 0) => 1 operation
- a test (i < n) => n + 1 operations
- an increment (i++) => n operations
- the loop body that has three ***assignments***, two ***multiplications***, and an ***addition*** => 6 n operations

Thus the total number of basic operations is 6 * n + n + (n + 1) + 3
= **8n + 4**

Running Time Calculations

- Simple for loop

```
int Sum (int N)          {
/* 1 */   int sum = 0;
/* 2 */   for (int i =1; i <= N; i++)
/* 3 */        sum = sum + i * i * i;
/* 4 */        return sum ;              }
```

Q: What is the running time?

Line 1 & 4 → 2 units of time → 2

Line 2 → 1 unit (initialize) + (N +1) tests + N Increments → $2N + 2$

Line 3 → 4 units (1 add, 2 muls., 1 assign) * N executions → $4N$

Total → $6N + 4$

A: O(N)

Be careful to differentiate between

- Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.

- Efficiency: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger.

Standard Analysis Techniques

- Constant time statements
- Analyzing Loops
- Analyzing Nested Loops
- Analyzing Conditional Statements

Constant time example

- Simplest case: O(1) time statements
- Assignment statements of simple data types
- int x = y;
- Arithmetic operations:
- x = 5 * y + 4 - z;
- Array referencing:
- A[j] = 5;

Analyze Loop

- Any loop has two parts:
- – How many iterations are performed?
- – How many steps per iteration?
  - **int sum = 0,j;**
  - **for (j=0; j < N; j++)**
  - **sum = sum +j;**
- – Loop executes N times (0..N-1)
- – 4 = O(1) steps per iteration
- • Total time is N * O(1) = O(N*1) = O(N)

Nested and consecutive loops

- For a sequence of statements, compute their complexity functions individually and add them up

```
for (j=0; j < N; j++)
  for (k =0; k < j; k++)
    sum = sum + j*k;
for (l=0; l < N; l++)
  sum = sum -l;
cout<<"Sum="<<sum;
```

$O(N^2)$

$O(N)$

$O(1)$

- Total cost is O(N^2) + O(N) +O(1) = O(N^2)
- Or N square

Nested loop

```
for(i=0;i<n;i++)
{
   for(j=0;j<n;j++)
    {
      for(k=0;k<n;k++)
          {    }
      }
  }
```

$O(n^3)$

```
for(i=0;i<n;i++)
    { }
for(j=0;j<n;j++)
     {    }
for(k=0;k<n;k++)
            {    }
```

$O(n)$

Control Structures

- if (condition)
  - statement1;
- else
  - statement2;
- where statement1 runs in O(N) time and statement2 runs in O(N^2) time?

- **Dominant Term would decide here.**

Why it is important

- Suppose a program has run time O(n!) and the run time for
  - n = 10 is 1 second
  - For n = 12, the run time is 2 minutes
  - For n = 14, the run time is 6 hours
  - For n = 16, the run time is 2 months
  - For n = 18, the run time is 50 years
  - For n = 20, the run time is 200 centuries

Task

```
for (i=0; i<n; i++)
{for (j=0; j<n;j++)
{
Sequence of statements….
}
}
```

Statements with Function call

```
for(i=0;i<n;i++)
    {  g(n);  }
```

Function of Growth rate

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

Functions in order of increasing growth rate

Classes of Complexities

- Constant: $O(c)$,
- Logarithmic: $O(\log_c n)$,
- Linear: $O(n)$,
- Quadratic: $O(n^2)$,
- Cubic: $O(n^3)$,
- Polynomial: $O(n^c)$
- Exponential: $O(c^n)$

Home Task

- Write an algorithm to sort the data in array. Bring it handwritten on A4.