Week 6: Analysis of Algorithms

CS-250 Data Structure and Algorithms

DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

Terminologies

Algorithm analysis

- Algorithm: A clearly specified finite set of instructions a computer follows to solve a problem.

- Algorithm analysis: a process of determining the amount of time, resource, etc. required when executing an algorithm.

Algorithm analysis

- Problems can be solved in multiple ways
  - iteration, recursion, vector, set,
  - Other factors
  - working environment (e.g., resource constraints), performance metrics
  - Precise answer, estimation e.g., sorting, searching

Algorithm analysis

- Evaluate/compare Alternatives
  - Development time, cost, hardware required to run it, how general it is.
  - Correctness
  - Complexity
    - Resources the algorithm requires (e.g., time, memory used, energy consumption, response time)

Algorithm analysis

- Determine how efficiently it solves the problem
  - Mainly Time complexity and space complexity
  - we need measure that tells us how efficient it is for any input
    - Independent of the computer/software techniques

  - **Brainstorm Count Algorithm**

Time Complexity

- Is the algorithm “**fast enough**” for my needs
- How much longer will the algorithm take if I increase the amount of data it must process
- Given a set of algorithms that accomplish the same thing, which is the right one to choose

Why is running time important

How long to wait for search results

Running time depends on

- Running time depends on the input in more complicated ways rather its size.

- Larger input leads to larger running time usually

- It is the rate of growth (with input size) that matters.

Algorithm Analysis

- Empirical Approach (Time with Stopwatch)
  - Time the implementation of an algorithm Real time results
  - Dependent on Hardware, other activity (i.e., subject to variation)
  - Development time, cost, verification, tied to specific environment

- Analytical/ Theoretical Approach
  - Inspect the pseudocode
  - Can analyze without implementation No dependency on hardware
  - No dependency on software techniques No dependency on programming language
  - A formula that relates input size to the running time of the algorithm satisfies this requirement.

Asymptotic Algorithm Analysis

- Actual (wall-clock) time of a program is affected by:
  - size of the input
  - programming language
  - programming tricks
  - compiler
  - CPU speed
  - multiprogramming level (other users)
- Instead of wall-clock time, look at the pattern of the program's behavior as the problem size increases.
  - This is called **asymptotic analysis**.

- That is, look at the shape of the function that gives the running time on inputs of size , with more emphasis on what happens as gets big.

Understanding of analysis

Algorithm Analysis

- The efficiency of any algorithmic solution to a problem is a measure of the:
  - Time efficiency: How much time it takes to complete.
  - Space efficiency: How much space it occupies.
- Solution:

Algorithm Analysis

- The efficiency of any algorithmic solution to a problem is a measure of the:
  - Time efficiency: How much time it takes to complete.
  - Space efficiency: How much space it occupies.
- Solution:

We want a method in which we can compare our designed algorithms WITHOUT executing them. The method should be independent of hardware/compiler/operating system so that we can actually know which algorithm performs better than others.

Algorithm Analysis

- The efficiency of any algorithmic solution to a problem is a measure of the:
  - Time efficiency: How much time it takes to complete.
  - Space efficiency: How much space it occupies.
- Solution:

We want a method in which we can compare our designed algorithms WITHOUT executing them. The method should be independent of hardware/compiler/operating system so that we can actually know which algorithm performs better than others.

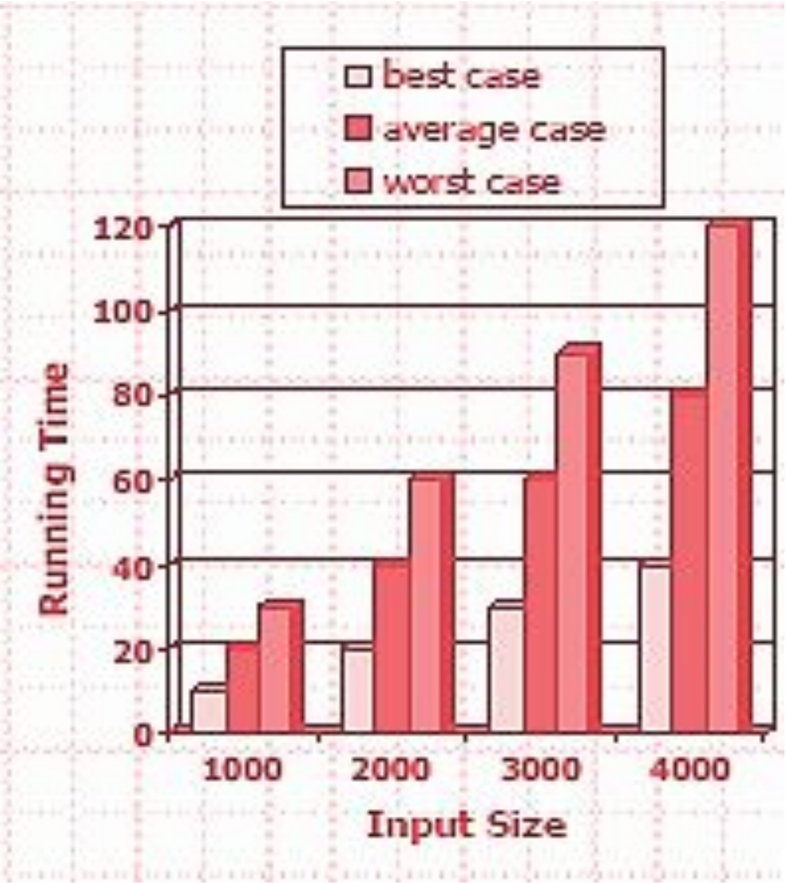
Preferably, we should analyze them mathematically
That is Analysis of algorithm

Algorithm Analysis

- Estimate the performance of an algorithm through
  - The number of operations required to process an input or input set
  - Process an input of certain size
- Require a function expressing relation between
  - n & t called
  - time complexity function **T(n)**
    - where n defines input size and t is running time for that input.
- For calculating **T(n)** we need to compute the total number of program steps
  - can be the number of executable statements

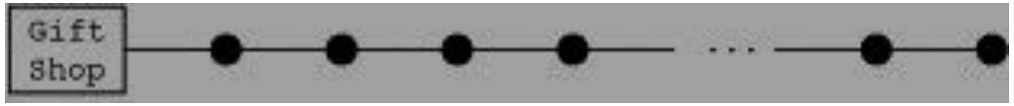
Running time (T) as a function of input (N)

- Most algorithms transform input into output.
- The running time 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.
  - what would happen if an autopilot algorithm ran drastically slower for some unforeseen, untested inputs?



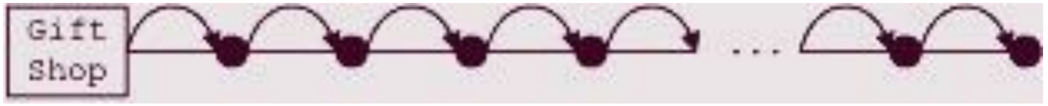
Real world example

- Problem
  - 50 packages delivered to 50 different houses
  - 50 houses one mile apart, in the same area



Gift shop and each dot representing a house

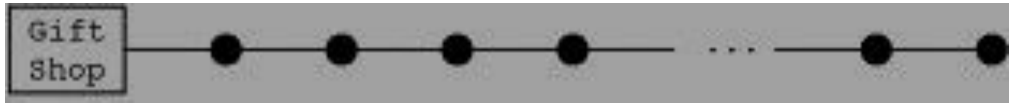
- Solution-1
  - Driver picks up all 50 packages
  - Drives one mile to first house, delivers first package
  - Drives another mile, delivers second package
  - Drives another mile, delivers third package, and so on
  - Distance driven to deliver packages
    - 1+1+1+… +1 = 50 miles
- Total distance traveled: 50 + 50 = 100 miles



Package delivering scheme

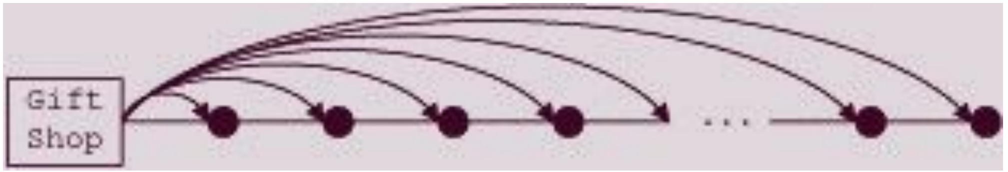
Real world example

- Problem:
  - 50 packages delivered to 50 different houses
  - 50 houses one mile apart, in the same area



Gift shop and each dot representing a house

- Solution-2



Package delivering scheme

  - Driver picks up first package, drives one mile to the first house, delivers package, returns to the shop
  - Driver picks up second package, drives two miles, delivers second package, returns to the shop
- Total distance traveled: 2 * (1+2+3+…+50) = 2550 miles

Comparison of solutions

- Problem: n packages to deliver to n houses, each one mile apart
- Solution 1: total distance traveled
  - 1+1+1+… +n = 2n miles
  - Function of n
- Solution 2: total distance traveled
  - $2 * (1+2+3+\ldots+n) = 2*(n(n+1) / 2) = n^2+n$
  - Function of n^2

n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

Calculating T(N)

- A program step is the syntactically / semantically meaningful segments of a program
  - A step DOES NOT correspond to a definite time unit
  - A step count is telling us how run time for a program changes with change in data size

- Calculate the total number of steps/executable statements in a program
  - Find the frequency of each statement and sum them up
  - Don’t count comments and declarations

A code example

- Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                          //Line 1

cin >> num1 >> num2;                                  //Line 2

if (num1 >= num2)                                     //Line 3
    max = num1;                                       //Line 4
else                                                  //Line 5
    max = num2;                                       //Line 6

cout << "The maximum number is: " << max << endl;    //Line 7
```

A code example

- Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                    //Line 1

cin >> num1 >> num2;                            //Line 2

if (num1 >= num2)                               //Line 3
    max = num1;                                 //Line 4
else                                            //Line 5
    max = num2;                                 //Line 6

cout << "The maximum number is: " << max << endl;   //Line 7
```

1 operation

A code example

- Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                 //Line 1

cin >> num1 >> num2;                         //Line 2

if (num1 >= num2)                            //Line 3
    max = num1;                              //Line 4
else                                         //Line 5
    max = num2;                              //Line 6

cout << "The maximum number is: " << max << endl;   //Line 7
```

1 operation

2 operations

A code example

- Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                  //Line 1

cin >> num1 >> num2;                          //Line 2

if (num1 >= num2)                             //Line 3
    max = num1;                               //Line 4
else                                          //Line 5
    max = num2;                               //Line 6

cout << "The maximum number is: " << max << endl;   //Line 7
```

1 operation

2 operations

1 operation

A code example

- Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                    //Line 1

cin >> num1 >> num2;                            //Line 2

if (num1 >= num2)                               //Line 3
    max = num1;                                 //Line 4
else                                            //Line 5
    max = num2;                                 //Line 6

cout << "The maximum number is: " << max << endl;   //Line 7
```

Only one of them will be executed

1 operation

2 operations

1 operation

1 operation

A code example

- Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                       //Line 1

cin >> num1 >> num2;                               //Line 2

if (num1 >= num2)                                  //Line 3
    max = num1;                                    //Line 4
else                                               //Line 5
    max = num2;                                    //Line 6

cout << "The maximum number is: " << max << endl;  //Line 7
```

1 operation

2 operations

1 operation

1 operation

3 operations

A code example

- Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                         //Line 1

cin >> num1 >> num2;                                 //Line 2

if (num1 >= num2)                                    //Line 3
    max = num1;                                      //Line 4
else                                                 //Line 5
    max = num2;                                      //Line 6

cout << "The maximum number is: " << max << endl;    //Line 7
```

1 operation

2 operations

1 operation

1 operation

3 operations

Total = 8 operations

Another code example

- Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl;  //Line 1

count = 0;                                                  //Line 2
sum = 0;                                                    //Line 3

cin >> num;                                                 //Line 4

while (num != -1)                                           //Line 5
{
    sum = sum + num;                                        //Line 6
    count++;                                                //Line 7
    cin >> num;                                             //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;       //Line 9

if (count != 0)                                             //Line 10
    average = sum / count;                                  //Line 11
else                                                        //Line 12
    average = 0;                                            //Line 13

cout << "The average is: " << average << endl;              //Line 14
```

Another code example

- Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl;  //Line 1

count = 0;                                                 //Line 2
sum = 0;                                                   //Line 3

cin >> num;                                                //Line 4

while (num != -1)                                          //Line 5
{
    sum = sum + num;                                       //Line 6
    count++;                                               //Line 7
    cin >> num;                                            //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;      //Line 9

if (count != 0)                                            //Line 10
    average = sum / count;                                 //Line 11
else                                                       //Line 12
    average = 0;                                           //Line 13

cout << "The average is: " << average << endl;             //Line 14
```

2 operations

1 operation

1 operation

1 operation

Another code example

- Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl;   //Line 1

count = 0;                                                  //Line 2
sum = 0;                                                    //Line 3

cin >> num;                                                 //Line 4

while (num != -1)                                           //Line 5
{
    sum = sum + num;                                        //Line 6
    count++;                                                //Line 7
    cin >> num;                                             //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;       //Line 9

if (count != 0)                                             //Line 10
    average = sum / count;                                  //Line 11
else                                                        //Line 12
    average = 0;                                            //Line 13

cout << "The average is: " << average << endl;              //Line 14
```

N times the condition is TRUE
+ 1 time the condition is FALSE

2 operations
1 operation
1 operation
1 operation
N+1 operations
2N operations
N operations
N operations

Another code example

- Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl;  //Line 1

count = 0;                                                  //Line 2
sum = 0;                                                    //Line 3

cin >> num;                                                 //Line 4

while (num != -1)                                           //Line 5
{
    sum = sum + num;                                        //Line 6
    count++;                                                //Line 7
    cin >> num;                                             //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;       //Line 9

if (count != 0)                                             //Line 10
    average = sum / count;                                  //Line 11
else                                                        //Line 12
    average = 0;                                            //Line 13

cout << "The average is: " << average << endl;              //Line 14
```

N times the condition is TRUE
+ 1 time the condition is FALSE

Executed while the cond. is TRUE

2 operations

1 operation

1 operation

1 operation

N+1 operations

2N operations

N operations

N operations

3 operations

Another code example

- Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl;   //Line 1

count = 0;                                                   //Line 2
sum = 0;                                                     //Line 3

cin >> num;                                                  //Line 4

while (num != -1)                                            //Line 5
{
    sum = sum + num;                                         //Line 6
    count++;                                                 //Line 7
    cin >> num;                                              //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;       //Line 9

if (count != 0)                                              //Line 10
    average = sum / count;                                   //Line 11
else                                                         //Line 12
    average = 0;                                             //Line 13

cout << "The average is: " << average << endl;              //Line 14
```

N times the condition is TRUE + 1 time the condition is FALSE

Executed while the cond. is TRUE

Only one of them will be executed, take the max: 2

2 operations
1 operation
1 operation
1 operation
N+1 operations
2N operations
N operations
N operations
3 operations
1 operation
2 operations
1 operation

Another code example

- Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl;   //Line 1

count = 0;                                                   //Line 2
sum = 0;                                                     //Line 3

cin >> num;                                                  //Line 4

while (num != -1)                                            //Line 5
{
    sum = sum + num;                                         //Line 6
    count++;                                                 //Line 7
    cin >> num;                                              //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;        //Line 9

if (count != 0)                                              //Line 10
    average = sum / count;                                   //Line 11
else                                                         //Line 12
    average = 0;                                             //Line 13

cout << "The average is: " << average << endl;               //Line 14
```

N times the condition is TRUE
+ 1 time the condition is FALSE

Executed while the cond. is TRUE

Only one of them will be executed, take the max: 2

2 operations
1 operation
1 operation
1 operation
N+1 operations
2N operations
N operations
N operations
3 operations
1 operation
2 operations
1 operation
3 operation

Another code example

- If the while loop executes N times then: 2+1+1+1+5*N + 1 + 3 + 1 + (2) + 3 = 5N+(15)

```
cout << "Enter positive integers ending with -1" << endl;   //Line 1

count = 0;                                                  //Line 2
sum = 0;                                                    //Line 3

cin >> num;                                                 //Line 4

while (num != -1)                                           //Line 5
{
    sum = sum + num;                                        //Line 6
    count++;                                                //Line 7
    cin >> num;                                             //Line 8
}

cout << "The sum of the numbers is: " << sum << endl;       //Line 9

if (count != 0)                                             //Line 10
    average = sum / count;                                  //Line 11
else                                                        //Line 12
    average = 0;                                            //Line 13

cout << "The average is: " << average << endl;              //Line 14
```

2 operations

1 operation

1 operation

1 operation

N+1 operations

2N operations

N operations

N operations

3 operations

1 operation

2 operations

1 operation

3 operation

N times the condition is TRUE + 1 time the condition is FALSE

Executed while the cond. is TRUE

Only one of them will be executed, take the max: 2

Analysis of ‘FOR’ loop

```
1:  int sum = 0
2:  for (i=0;i<n;++i)
3:       sum++;
4:  cout << sum
```

You can see….1 assignment step i=0
1 conditional step i<n
will run n +1 time
1 increment step ++i
will run n time
Statement 2 executes n times
Statement 3 executes 1 time
T(n) = ???

The condition always executes one more time than the loop itself

Analysis of Nested ‘FOR’ loop

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

Rule of thumb

Simple programs can be analysed by counting the nested loops of the program.

A single loop over n items yields f(n) = n.

A loop within a loop yields f(n) = n^2.

A loop within a loop within a loop yields f(n) = n^3.

The statement sum++ executes n*m times

So in a nested for loop if the loop variables are independent then:

*The total number of times a statement executes = outer loop times * inner loop times*

Analysis of loops: < OR <=

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

- No. of Iterations: (n – k) / m times.
- i = k, is executed 1time.
- i < n, is executed (n – k) / m + 1times.
- i = i + m, is executed (n – k) / m times.
- Body of loop is executed (n – k) / m times

Analysis of loops: < OR <=

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

- No. of Iterations: (n – k) / m times.
- i = k, is executed 1time.
- i < n, is executed (n – k) / m + 1times.
- i = i + m, is executed (n – k) / m times.
- Body of loop is executed (n – k) / m times

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

- No. of iterations is: (n – k) / m+1 times
- i = k, is executed 1 time.
- i <= n, is executed (n – k) / m+2 times.
- i = i + m, is executed (n – k) / m+1 times.
- Body of loop is executed (n–k)/m+1times

Loops with logarithmic iterations

- In the following for-loop: (with <)
  - The number of iterations is: (Logm(n / k))

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

Loops with logarithmic iterations

- In the following for-loop: (with <)
  - The number of iterations is: (Logm(n / k))

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

- In the following for-loop: (with <=)
  - The number of iterations is: (Logm (n / k) + 1)

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

Algorithm analysis

- Sequential search algorithm

```
for(i=0; i<n; i++)          // at most  1 + (n+1) + n
    if(a[i] = =  searchKey)    // at most n (worst case)
        return true;           // 0 or 1 time
```

Total ops = 3n + 2

  - n: represents list size
  - f(n): number of basic operations (3n + 2)
  - c: units of computer time to execute one operation
    - Depends on computer speed (varies)
  - cf(n): computer time to execute f(n) operations

Algorithm analysis

Various values of n, $2n$, n^2, and $n^2 + n$

n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

(n) and (2n) are close, so we magnify (n)
(n^2) and (n^2 + n) are close, so we magnify (n^2)

Algorithm analysis

Various values of n, $2n$, n^2, and $n^2 + n$

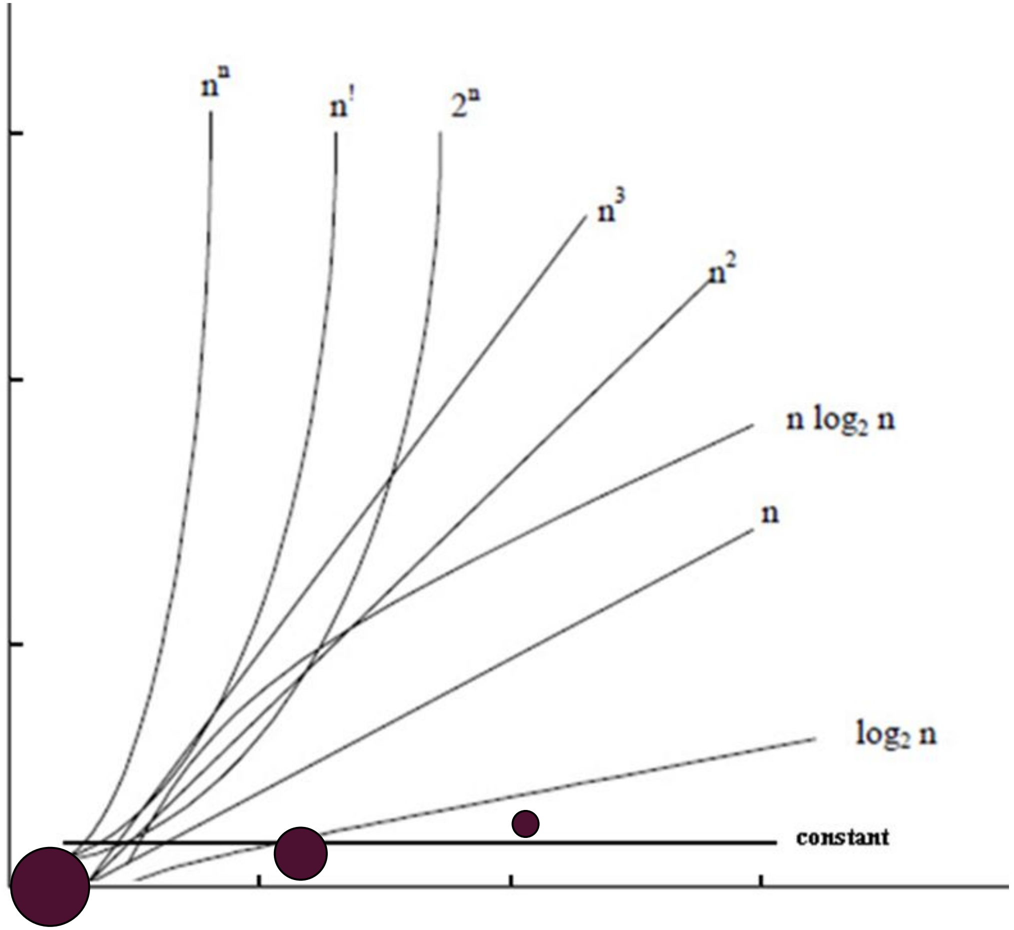
n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

When n becomes too large, n and n^2 becomes very different

Growth rates

WHICH GROWTH RATE IS BETTER???

n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296



This is certainly what I would like my program to be!!! (but this is just wishful thinking)

Growth rates

> *Asymptotic complexity studies the efficiency of an algorithm as the input size becomes large*

- Notation useful in describing algorithm behavior
  - Shows how a function f(n) grows as n increases without bound
- Asymptotic Analysis
  - Study of the function f as n becomes larger and larger without bound
  - Examples of functions
    - g(n)=n2 (no linear term)
    - f(n)=n2 + 4n + 20
  - As n becomes larger and larger
    - Term 4n + 20 in f(n) becomes insignificant
    - Term n2 becomes dominant term

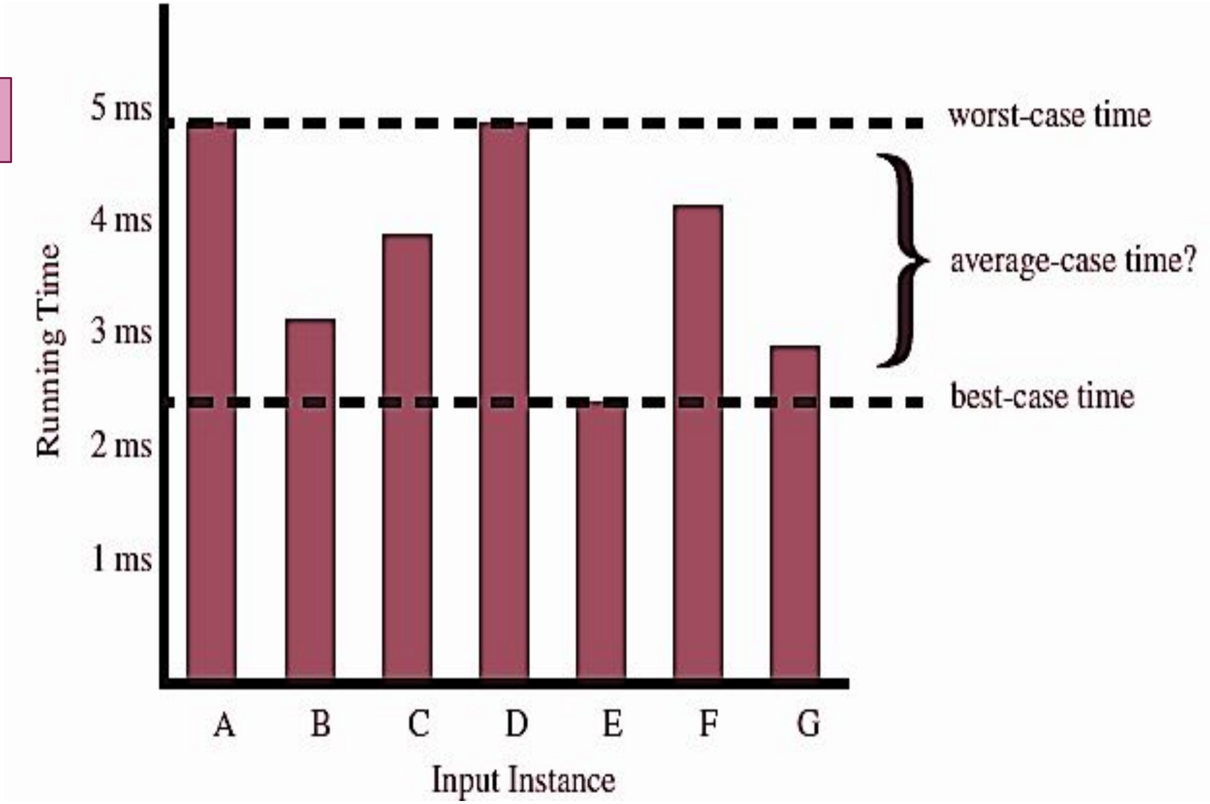
n	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	100	160
50	2500	2720
100	10,000	10,420
1000	1,000,000	1,004,020
10,000	100,000,000	100,040,020

Growth rate of n^2 and $n^2 + 4n + 20n$

Asymptotic analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh (O) notation

Lower Bound < = Average Time < = Upper Bound



- There are three types of analysis:
  - Worst case
    - Maximum number of steps
  - Best case
    - Minimum number of steps
  - Average case
    - Average number of steps

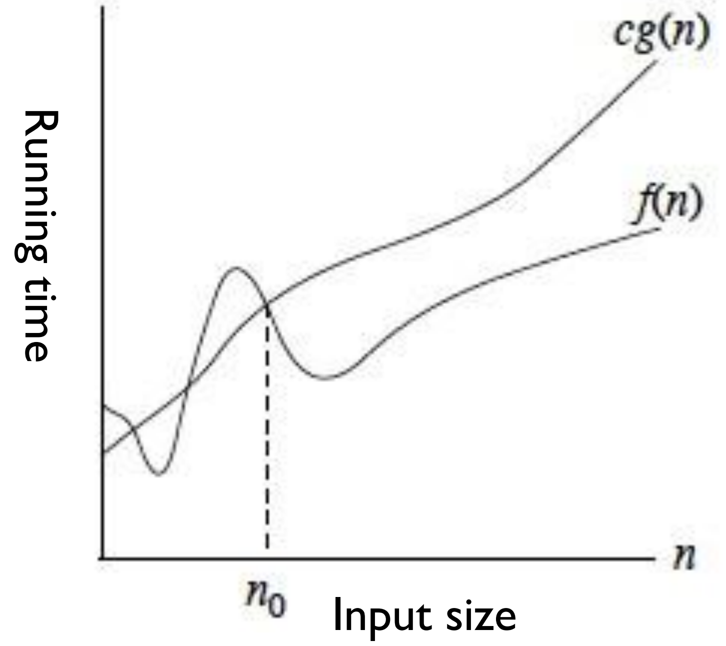
Big-O

f(n) is O(g(n)) if there exist positive numbers c & N such that f(n)<=cg(n) for all n>=N

g(n) is called the upper bound on f(n)

OR

f(n) grows at the most as large as g(n)



T(n) = O(f(n)) if there are positive constants c and N such that T(n) <= c f(n) where n >= N (or n_0)

This says that function T(n) grows at a rate no faster than f(n); thus f(n) is an upper bound on T(n).

Big-O

- $T(n) = 8n+2$
  - we need to find a real constant $c>0$ and an integer constant $n_0 \geq 1$ such that $8n+2 \leq cn$ for every integer $n \geq n_0$.
  - Rule of thumb for finding c:
    Add up all coefficients in T(n)
  - Once you have found c, now put value of c in equation ($8n+2 \leq cn$) for different values of n to find n_0
- For $T(n) = 8n + 2$, $c=10$ (8+2) & $n_0=1$ (n_0 is break point)
- So $T(n) = 8n+2$ O(n) //ignoring the coefficients

Big-O

Example: $T(n) = n^2 + 3n + 4$

Big-O

Example: $T(n) = n^2 + 3n + 4$

$n^2 + 3n + 4 <= 8\ n^2$

for all

$c = 8,\ n_0=1$

so we can say that $T(n)$ is $O(n^2)$

OR

$T(n)$ is in the order of n^2.

$T(n)$ is bounded above by a + real multiple of n^2

Big-O

Example: $T(n) = 3\log n + 2$

Big-O

Example: $T(n) = 3\log n + 2$

$3\log n + 2 <= O(\log n)$

for all

$c=5, n_0=2$

Note that logn is zero for n = 1.

That is why we use $n_0 = 2$.

Big-O

- Big-O offers an equation to describe how the time of a procedure changes relative to its input.
  - It describes the trend.

- It does not define exactly how long it takes, as a procedure with a larger big-O time than another procedure could be faster on specific inputs.

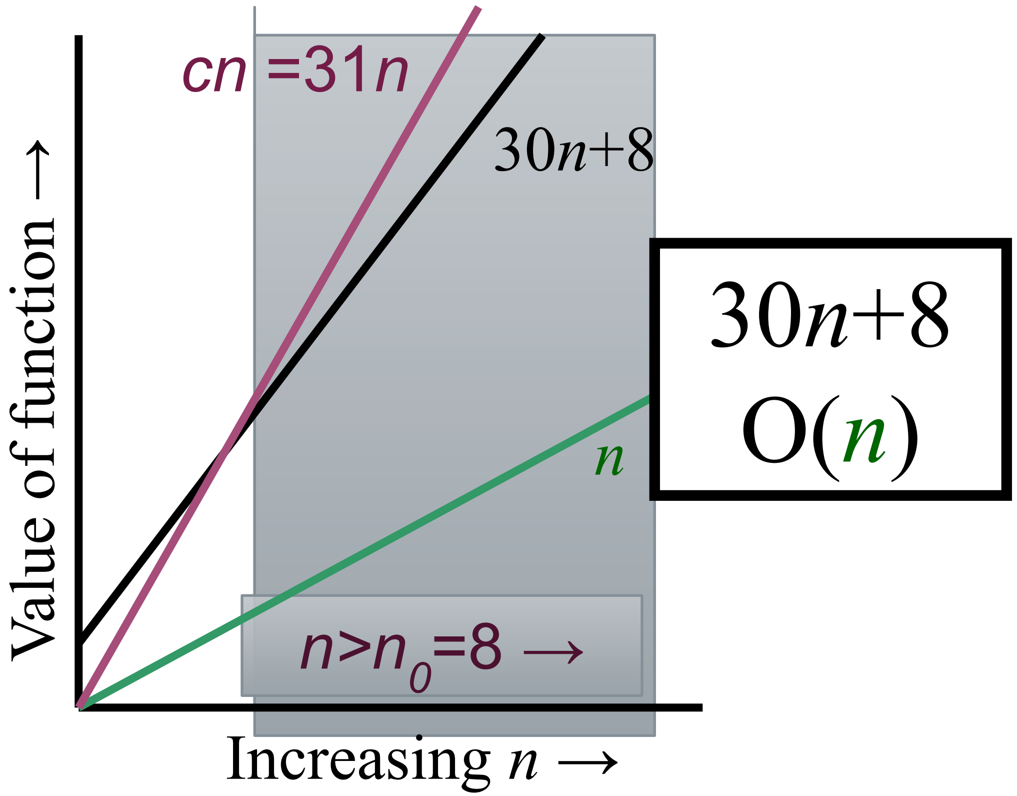
- A function's Big-O notation is determined by how it responds to different inputs.
  - How much slower is it if we give it a list of 1000 things to work on instead of a list of 1 thing?

Graphical example

- Show that 30n+8 is O(n).
  - Let c=31, n_0=8. Assume n>n_0=8.

- Note 30n+8 isn't less than n anywhere (n>0).
- It isn't even less than 31n everywhere.

- But it is less than 31n everywhere to the right of n=8.



Big-O rules

- If f(n) is a polynomial of degree d, then **f(n) is O(nd)**
  - Drop lower-order terms, Drop constant factors
  - Example: T(n)=5n2+n O(n2)
- If f(n) is logkn where k is any constant, then **f(n) is O(n)**
  - This tells us that logarithms grow very slowly.
- If f(n)=c where c is any constant, then **f(n) is O(1)**
- Use the smallest and possible class of functions
  - “2n +3 is O(n)” instead of “O(n^2)” ,
  - $5n^2$+n is O(n^2) rather than O(n) or O(n^3)

Big-O and growth rates

- The big-Oh notation gives an upper bound on the growth rate.
  - The statement “f(n) is O(g(n))” means that the growth rate of f(n) is no more than the growth rate of g(n).
- We can use the big-Oh notation to rank functions according to their growth rate.
- Seven functions are ordered by increasing growth rate in the sequence below:

constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 4.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Growth rates

```
sum++;
```

O(1)

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

O(n)

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

$O(n^2)$

Growth rates

Function $g(n)$	Growth rate of $f(n)$
$g(n) = 1$	The growth rate is constant and so does not depend on n, the size of the problem.
$g(n) = \log_2 n$	The growth rate is a function of $\log_2 n$. Because a logarithm function grows slowly, the growth rate of the function f is also slow.
$g(n) = n$	The growth rate is linear. The growth rate of f is directly proportional to the size of the problem.
$g(n) = n\log_2 n$	The growth rate is faster than the linear algorithm.
$g(n) = n^2$	The growth rate of such functions increases rapidly with the size of the problem. The growth rate is quadrupled when the problem size is doubled.
$g(n) = 2^n$	The growth rate is exponential. The growth rate is squared when the problem size is doubled.

Growth rates

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	0	1	1	2
2	1	2	4	8	4
8	3	24	64	512	256
64	6	384	4096	262,144	About 5 years
128	7	896	16,384	2,097,152	Approx 6 billion years, 600,000 times more than age of univ.

(If one operation takes 10^{-11} seconds)

Growth rates

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	0	1	1	2
2	1	2	4	8	4
8	3	24	64	512	256
64	6	384	4096	262,144	About 5 years
128	7	896	16,38 4	2,097,1 52	Approx 6 billion years, 600,000 times more than age of univ.

(If one operation takes 10^{-11} seconds)

Complexity and Tractability

	$T(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01μs	.03μs	.1μs	1μs	10μs	10s	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84h	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1s
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121d	18m
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1y	13d
100	.1μs	.66μs	10μs	1ms	100ms	3171y	4×10^{13}y
10^3	1μs	9.96μs	1ms	1s	16.67m	3.17×10^{13}y	32×10^{283}y
10^4	10μs	130μs	100ms	16.67m	115.7d	3.17×10^{23}y	
10^5	100μs	1.66ms	10s	11.57d	3171y	3.17×10^{33}y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^7y	3.17×10^{43}y	

Assume the computer does 1 billion ops per sec.

Limitations

- Big-O notation cannot compare algorithms in the same complexity class.
- Big-O notation only gives sensible comparisons of algorithms in different complexity classes when n is large

- Consider two algorithms for same task:
  - Linear: $f(n) = 1000\,n$
  - Quadratic: $f'(n) = n^2/1000$

- The quadratic one is faster for $n < 1000000$.

Limitations

- Big-Oh is an estimate tool for algorithm analysis.

- It ignores the costs of memory access, data movements, memory allocation, etc. => hard to have a precise analysis.

- Ex: 2nlogn vs. 1000n.
    - Which is faster? => it depends on n

Summary

- In this lecture, we have been discussed:
  - Complexities of algorithms in terms of time and space
  - Asymptotic analysis
  - Big-Oh and its growth rate