Algorithm Analysis

CS 201

Introduction

- Once a correct algorithm is designed for a given problem, an important step is to determine how much in the way of resources (such as time and space) the algorithm will require
  You will learn how to estimate the time required by the algorithm

- The idea is not to find the exact computational time required by the algorithm since this time is to be affected by other factors
  - The programming language using which it is implemented
  - The machine on which it is run

- Instead, we want to identify how the required time for the algorithm grows as a function of its input size
  You will learn asymptotic notations to express the growth rate of the algorithm

Mathematical background

- Asymptotic notations are used to express the growth rate of a function
- They allow to establish a relative order among functions
- You will learn three asymptotic notations: **big-Oh, big-omega, big-theta**

Compare $f(N) = N$, $g(N) = 1000\ N$ and $h(N) = N^2$

- We do not want to claim that $g(N) < h(N)$
  - Since you can find points for which g(N) is less than h(N) and vice versa
- Instead, we want to compare their relative rates of growth as a function of N
  - The growth rates of f(N) and g(N) are the same
  - The growth rates of both of these functions are less than that of h(N)

Asymptotic notations: Big-Oh

- This notation is used to express an upper-bound on a function
- $T(N) = O(f(N))$
  - $f(N)$ is an upper-bound on $T(N)$
  - The growth rate of $T(N)$ is less than or equal to that of $f(N)$
  - $T(N)$ grows at a rate no faster than $f(N)$

Definition (Big-Oh)

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c \cdot f(N)$ for all $N \geq n_0$

As an exercise, show that

- $2N^2 = O(N^2)$ $\rightarrow$ N^2 is an upper-bound on $2N^2$
- $100N^2 = O(N^3)$ $\rightarrow$ N^3 is an upper-bound on $100N^2$
- $N^2 \neq O(10000N)$ $\rightarrow$ But $10000N$ is **not** an upper-bound on N^2

Asymptotic notations: Big-omega

- This notation is used to express a lower-bound on a function
- $T(N) = \Omega(g(N))$
  - $g(N)$ is a lower-bound on $T(N)$
  - The growth rate of $T(N)$ is greater than or equal to that of $g(N)$
  - $T(N)$ grows at a rate no slower than $g(N)$

Definition (Big-omega)

$T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c \cdot g(N)$ for all $N \geq n_0$

As an exercise, show that

- $2N^2 = \Omega(N^2)$ $\rightarrow$ N^2 is also a lower-bound on $2N^2$
- $100N^2 = \Omega(N)$ $\rightarrow$ N is a lower-bound on $100N^2$
- $N \neq \Omega(N^2)$ $\rightarrow$ But N^2 is **not** a lower-bound on N

Asymptotic notations: Big-theta

- This notation is used to express a tight-bound on a function
- $T(N) = \Theta(h(N))$
  - $h(N)$ is a tight-bound on $T(N)$
  - The growth rate of $T(N)$ is equal to that of $h(N)$

Definition (Big-theta)

$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$

- $N^2 = O(N^3)$ but $N^2 \neq \Omega(N^3)$ $\rightarrow$ N^2 grows slower than N^3, thus $N^2 \neq \Theta(N^3)$
- $N^3 \neq O(N^2)$ but $N^3 = \Omega(N^2)$ $\rightarrow$ N^3 grows faster than N^2, thus $N^3 \neq \Theta(N^2)$
- $2N^2 = O(N^2)$ and $2N^2 = \Omega(N^2)$ $\rightarrow$ $2N^2$ and N^2 grows at the same rate, and thus, $2N^2 = \Theta(N^2)$

Asymptotic notations

Rule 1: If $T_1(N) = O(\ f(N)\ )$ and $T_2(N) = O(\ g(N)\ )$

- $T_1(N) + T_2(N) = O(\ f(N) + g(N)\ )$
- $T_1(N) \times T_2(N) = O(\ f(N) \times g(N)\ )$

Rule 2: If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(\ N^k\ )$

Rule 3: $\log^m (N) = O(\ N\ )$ for any constant m $\longrightarrow$ Logarithms grow very slowly!

If $g(\ N\ ) = 2\ N^2$,

$g(\ N\ ) = O(\ N^4\ )$, $g(\ N\ ) = O(\ N^3\ )$, $g(\ N\ ) = O(\ N^2\ )$ are all technically correct.

But the last one is the BEST ANSWER.

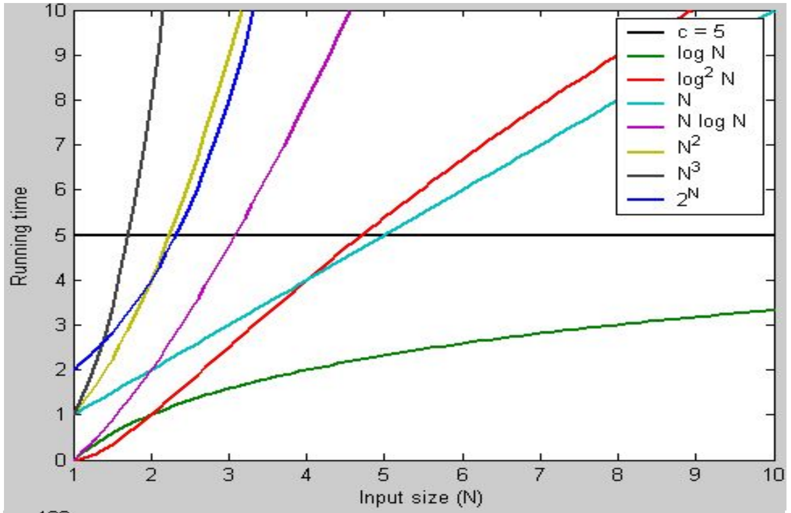
If $h(\ N\ ) = 2\ N^2 + 100\ N$

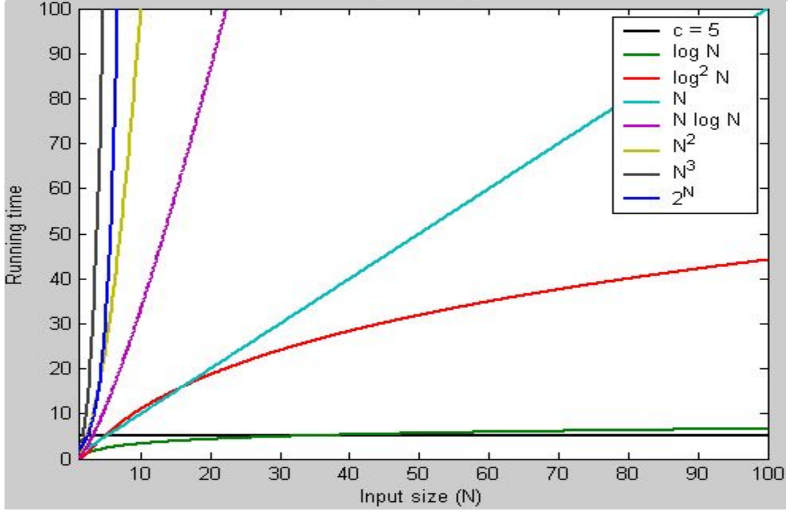
Do NOT say $h(N) = O\ (\ 2\ N^2 + 100\ N\ )$ or $h(N) = O\ (\ 2\ N^2\ )$ or $h(N) = (\ N^2 + N\ )$.

The correct form is $h(N) = O\ (\ N^2\ )$.

Typical growth rates

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





Algorithm analysis

- If two programs are expected to take similar times, probably the best way to decide which is faster is to code them both up and run them!

- On the other hand, we would like to eliminate the bad algorithmic ideas early by algorithm analysis
    - Asymptotic notations are not affected by the programming language. Thus, there is no need to code an algorithm for finding its time complexity (you can analyze the time complexity of even a pseudocode). However, after coding the algorithm, if it runs much more slowly than the algorithm analysis suggests, there may be an implementation inefficiency.

Although using big-theta would be more precise, big-Oh answers are typically given.

How to find the time complexity of a given program?

- We assume that simple instructions (such as addition, comparison, and assignment) take exactly one unit time
    - Unlike the case with real computers (for example, I/O operations take more time compared to comparison and arithmetic operators)
    - Although different instructions can take different amounts of time (these would correspond to constants in asymptotic notations), one may ignore this difference in the analysis
- Obviously, we cannot have this assumption for complex operations such as matrix inversion, list insertion, and sorting
- We assume infinite memory
- We do not include the time required to read the input

General rules

Rule 1 – loop: The running time of *a loop* is at most the running time of the statements inside the loop (including tests) times the number of iterations.

Rule 2 – nested loops: Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the number of iterations performed by all of the loops.

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

General rules

Rule 3 – if / else: The running time is never more than that of the test plus the larger of the running times of S1 and S2.

```
if ( test )
   S1
else
   S2
```

Rule 4 – consecutive statements: Just add their running times.

```
for ( i = 0; i < n; i++ )
   arr[ i ] = i;
for ( i = 0; i < n; i++ )
   for ( j = i; j < n; j++ )
      arr[ i ] += arr[ j ];
```

Simple example: What is the time complexity of the following function?

```
int partialSumOfSquares( int* arr, int n ){

   int result = 0;

   for ( int i = 0; i < n; i++ )
      if ( arr[ i ] < 0)
         result += arr[i] * arr[i];

   return result;
}
```

More examples: What are the time complexities of the following code fragments?

→
```
for ( i = 0; i < N * N; i++ )
    k++;
```

→
```
for ( i = 0; i < N * N; i += 40 )
    k++;
```

→
```
for ( i = 0; i < N * N; i += N )
    k++;
```

→
```
for ( i = 10; i < N * N / 3; i += 2 )
    k++;
```

→
```
for ( i = 0; i < 1000000000; i++ )
    k++;
```

→
```
for ( i = 1; i < N; i *= 2 )
    k++;
```

→
```
for ( i = N; i < 10; i /= 4 )
    k++;
```

→
```
for ( i = 0; i < N / 100; i++ )
    for ( j = M; j > 40; j -= 4 )
        k++;
```

→
```
for ( i = 1; i < N ; i *= 5 )
    for ( j = M / 2; j < M; j++ )
        k++;
```

General rules

Rule 5 – function calls: Find the running time of each function call. Be careful about their analyses if these are recursive functions. The analysis is trivial for some recursions, but could be hard for some others.

```
long factorial( int n ) {
    if ( n <= 1 )
        return 1;

    return n * factorial(n - 1);
}
```

Recurrence relation:

$T(n) = T(n - 1) + \Theta(1)$

$T(1) = \Theta(1)$

Solving it with the substitution method

$T(n) = T(n - 1) + \Theta(1)$

$T(n) = T(n - 2) + \Theta(1) + \Theta(1)$

$T(n) = T(n - 3) + \Theta(1) + \Theta(1) + \Theta(1)$

...

$T(n) = T(n - k) + k\ \Theta(1)$

$T(n) = T(1) + (n - 1)\ \Theta(1)$

$T(n) = n\ \Theta(1)$

$T(n) = \Theta(n)$

Example: What is the time complexity of the following power functions?

```
long iterativePower( long x, long n ) {
   long result = 1;
   for ( int i = 1; i <= n; i++ )
      result = result * x;
   return result;
}
```

```
long recursivePower( long x, long n ) {
   if ( n == 0 )
      return 1;

   if ( n == 1 )
      return x;

   if ( n % 2 == 0 )
      return recursivePower( x * x, n / 2 );

   return recursivePower( x * x, n / 2) * x;
}
```

Recurrence relation:

$T(n) = T(n / 2) + \Theta(1)$

$T(1) = \Theta(1)$

Solving it with the substitution method

$T(n) = T(n / 2) + \Theta(1)$

$T(n) = T(n / 4) + \Theta(1) + \Theta(1)$

$T(n) = T(n / 8) + \Theta(1) + \Theta(1) + \Theta(1)$

…

$T(n) = T(n / 2^k) + k\ \Theta(1)$

$T(n) = T(1) + \log n\ \Theta(1)$

$T(n) = \Theta(1) + \Theta(\log n)$

$T(n) = \Theta(\log n)$

Example: What is the time complexity of the following Fibonacci functions?

```
int iterativeFib( int n ) {
   int previous = 1, current = 1, next = 1;

   for ( int i = 3; i <= n; i++ ) {
      next = current + previous;
      previous = current;
      current = next;
   }
   return next;
}
```

```
int recursiveFib( int n ) {
   if ( n <= 2 )
      return 1;
   return recursiveFib(n - 1) + recursiveFib(n - 2);
}
```

Recurrence relation:

$T(n) = T(n - 1) + T(n - 2) + \Theta(1)$

$T(1) = \Theta(1)$

By induction, it is possible to show that

$T(n) < (5 / 3)^n$ and

$T(n) \geq (3 / 2)^n$

Thus, the running time of recursiveFib grows exponentially !!!

Worst-case, best-case, and average-case analyses

- Typically, the input size is the main consideration

- Sometimes, the input size to be considered may differ from one run to another
    - **Worst-case analysis** represents a guarantee on any possible input
    - **Average-case analysis** often reflects the typical behavior
    - **Best-case analysis** is often of a little interest

- Generally, it is focused on the worst-case analysis
    - It provides a bound on all inputs
    - Average-case bounds are much more difficult to compute

Example: Find the worst case, best case, and average case upper-bounds for the **sequential search algorithm**

```
int sequentialSearch( int* arr, int N, int value ){

     for ( int i = 0; i < N; i++)
          if ( arr[i] == value )
               return i;
     return -1;
}
```

- For a successful search:
  - **Worst-case:** N iterations are necessary when the value is the last array item → $O(N)$
  - **Best-case:** Only 1 iteration is necessary when the value is the first array item → $O(1)$
  - **Average-case:** (N + 1) / 2 iterations are necessary on the average. For calculating this average, one needs to consider all possibilities → $O(N)$
- For an unsuccessful search:
  - **Worst-case**, **best-case**, and **average-case** are all the same → $O(N)$

Example: Find the worst case, best case, and average case upper-bounds for the **binary search algorithm**

```
int binarySearch( int* arr, int N, int value ){
   int low = 0, high = N - 1;
   while ( low <= high ) {
      int mid = (low + high) / 2;
      if ( arr[ mid ] < value )
         low = mid + 1;
      else if ( arr[ mid ] > value )
         high = mid - 1;
      else
         return mid;
   }
   return -1;
}
```

- For a successful search:
  - **Worst-case** → $O(\log N)$, **best-case** → $O(1)$, and **average-case** → $O(\log N)$
- For an unsuccessful search:
  - **Worst-case**, **best-case**, and **average-case** are all the same → $O(\log N)$