Computer Science 3A - CSC3A10

Lecture 3: Analysis Tools

Academy of Computer Science and Software Engineering
University of Johannesburg

- Analysis of Algorithms

1. Analysis Functions

2. Experimental Studies
   - Experimental Studies

3. Runtime Analysis
   - Runtime Analysis
   - Asymptotic notation
   - Big-Omega and Big-Theta
   - Exercises

Analysis of Algorithms

- A data structure is a systematic way of organizing and accessing data
- An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.
- These, along with input and output all impact the runtime.

Analysis of Algorithms

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- That is why we focus on the worst case running time (which is easier to analyze and crucial to applications such a games, finance and robotics).

Analysis of Algorithms II

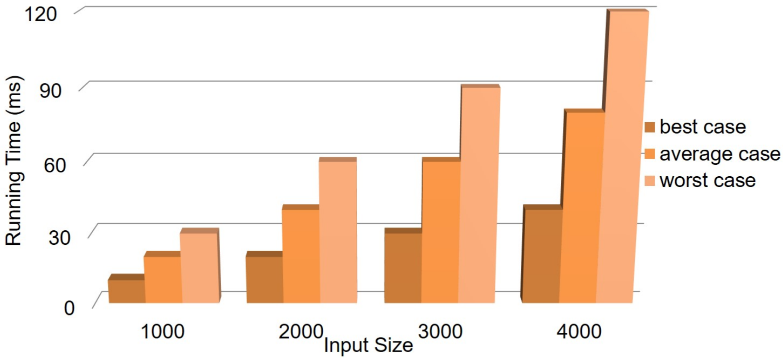


Figure: Comparing the various analysis cases

Analysis Functions

Seven functions used in analysis

The Constant function

$$f(n) = c$$

Data structure run in times proportional to a constant function

The Logarithm Function

$$f(n) = log_b(n) \ iff \ b^x = n$$

Data structure run in times proportional to a logarithm function

Seven functions used in analysis II

The Linear Function

$$f(n) = n$$

Algorithms run in times proportional to a linear function

The N-log-N Function

$$f(n) = nlog(n)$$

Algorithms run in times proportional to a n-log-n function

Seven functions used in analysis II

The Quadratic Function

$$f(n) = n^2$$

Less practical if algorithms run in times proportional to a quadratic function

The Cubic Function

$$f(n) = n^3$$

Less practical if algorithms run in times proportional to a cubic function

Seven functions used in analysis III

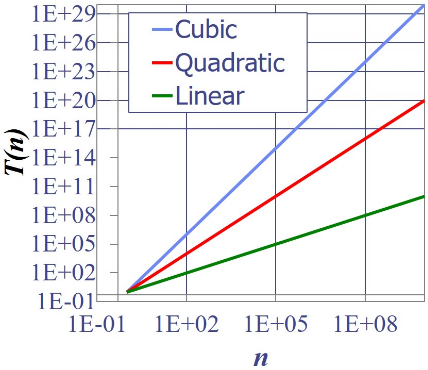
The Exponential Function

$$f(n) = b^n$$

Infeasible for algorithms to run in times proportional to a exponential function (exception if smallest amount of data)

Seven functions used in analysis IV

In a log-log chart, the slope of the line corresponds to the growth rate



Seven functions used in analysis V

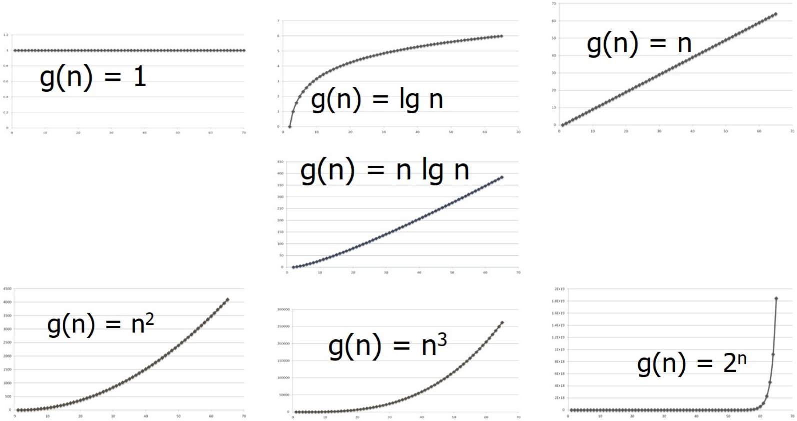


Figure: Functions Graphed using a "normal" scale

Experimental Studies

Experimental Studies

One way is to implement it:

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like *System.currentTimeMillis()* to get an accurate measure of the actual running time
- Plot the results

Experimental Studies II

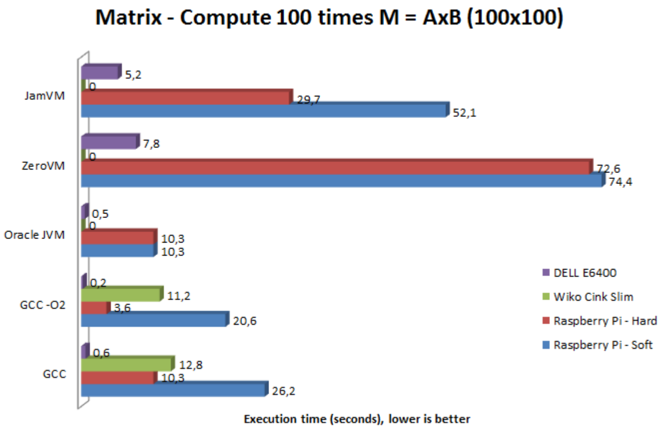


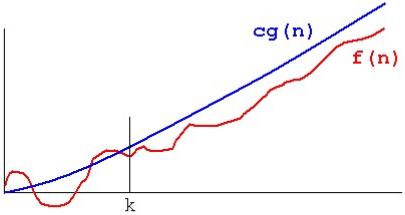
Figure: The plot results from an experimental study

Experimental Studies III

There are however limitations:

- Have to fully implement and execute an algorithm to study its run time experimentally
- Difficult to compare experimental run times of two algorithms unless the experiments are performed in the same hardware and software environments
- Experiments are usually done on a limited set of data, hence run times of inputs not included are never tested (and this information may be vital).

Runtime Analysis



Method of analysis of the runtime

Is it necessary to get the exact runtime? What about another approach that gives an estimate instead?

Runtime analysis:

- Uses a high-level description of the algorithm 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

Primative Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Assumed to take a constant amount of time

Primitive Operation Examples

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

Counting Primitive Operations

```
1 public int arrayMax(A, n)
2   currentMax = A[0];
3   for (int i = 1; i<n; i
        ++){
4     if (A[i] > currentMax
          )
5       currentMax = A[i];
6   }
7   return currentMax
```

A function that calculates the maximum in an array

```
1 # operations
2 2 %because currentMax is
      not declared here
3 2+3(n-1)=3n-1 % because
      the loop starts at 1
4 (n-1)(2)=2n-2
5 (n-1)(2)=2n-2
6
7 1
```

Primitive counting

Total: $7n - 2$

Estimating Running Time

Algorithm arrayMax executes 7n - 2 primitive operations in the worst case. If we 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:

$$a(7n - 2) \leq T(n) \leq b(7n - 2)$$

Hence, the running time T(n) is bounded by two linear functions

Growth Rate of Running Time

Changing the hardware/ software environment

- Affects T(n) by a constant factor, but
- Does not alter the growth rate of T(n)

The linear growth rate of the running time T(n) is an intrinsic property of algorithm arrayMax

Counting Primitive Operations II

```
public int binSearch(A, x, l, h)
  lo = l
  hi = h
  while (lo < hi) {
    mid = (lo + hi)/2;
    if (A[mid] < x)
      lo = mid + 1;
    else if (A[mid] == x)
      return mid;
    else
      hi = mid - 1;
  }
  return FAIL
```

A function that determines the position of x in A if found, otherwise fail.

```
# operations
1 % lo not declared
1 %hi not declared
logn % because lo or hi is changed
3logn
2logn
2logn
2logn
1

2logn

1
```

Total: $12logn + 4$

Why Growth Matters

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Big-Oh notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c > 0$ and integer constant $n_0 \geq 1$ such that:
  - $f(n) \leq cg(n) for\ n \geq n_0$
  - i.e. has asymptotic upper bounds
- $f(n)$ is big-Oh of $g(n)$ or $f(n)$ is order of $g(n)$
- The function 8n-2 is $O(n)$
  - By definition we need to find $c > 0$ and $n_0 \geq 1$ such that $8n - 2 \leq cn$ for all $n \geq n_0$
  - Possible choice $c = 8$ and $n_0 = 1$, any real number ≥ 8 will work for c and any integer ≥ 1 will work for n_0

Big-Oh notation Examples

Example: 2n + 10 is O(n)

- $2n + 10 \leq cn$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- Pick c = 3 and $n_0 = 10$

Big-Oh notation Examples II

Example: the function n^2 is not O(n)

- $n^2 \leq cn$
- $n \leq c$
- The above inequality cannot be satisfied since c must be a constant

Big-Oh notation Examples III

Example: 7n-2 is O(n)

- need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq c \cdot n$ *for* $n \geq n_0$
- this is true for $c = 7$ and $n_0 = 1$

Big-Oh notation Examples IV

Example: $3n^3 + 20n^2 + 5$ is $O(n^3)$

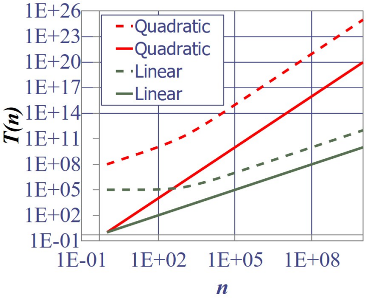
- $3n^3 + 20n^2 + 5$ need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3 for\ n \geq n_0$
- this is true for $c = 4$ and $n_0 = 21$

Example: $3logn + 5$ is $O(logn)$

- need $c > 0$ and $n_0 \geq 1$ such that $3logn + 5 \leq c \cdot logn\ for\ n \geq n_0$
- this is true for $c = 8$ and $n_0 = 2$

Constant Factors

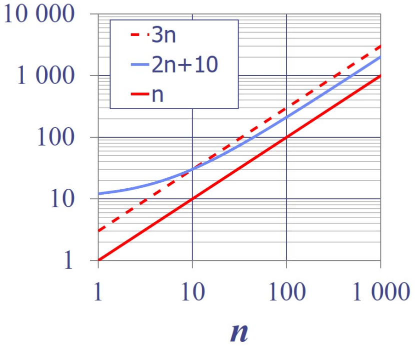
- The growth rate is not affected by constant factors or lower-order terms
- For Example
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Big-Oh notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that
$f(n) \leq cg(n)$ *for* $n \geq n_0$
Example: 2n + 10 is O(n)

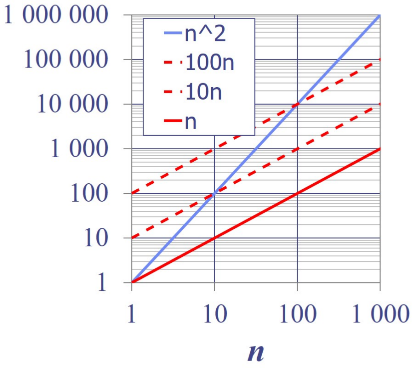
- $2n + 10 \leq cn$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$



Big-Oh notation II

Example: the function n^2 is not $O(n)$

- $n^2 \leq cn$
- $n \leq c$
- The above inequality cannot be satisfied since c must be a constant



Aysmptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation

Aysmptotic Algorithm Analysis II

- Example:
  - We determine that algorithm arrayMax executes at most $7n - 2$ primitive operations
  - We say that algorithm arrayMax "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Big-Oh Rules

If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,

- Drop lower-order terms
- Drop constant factors

Use the smallest possible class of functions

- Say "$2n$ is $O(n)$" instead of "$2n$ is $O(2n)$"

Use the simplest expression of the class

- Say "3n + 5 is O(n)" instead of "3n + 5 is O(3n)"

Prefix Averages (Quadratic)

```
1  int[] prefixAverages1(int[] X, n)
2  //Input array X of n integers
3  //Output array A of prefix averages
4    int[] A = new int[n];
5    for(int i=0; i < n; i++){
6      s = X[0];
7      for (int j = 1; j<i; j++)
8        s = s + X[j];
9      A[i] = s / (i + 1);
10   }
11   return A;
```

Prefix Average1

```
1
2
3
4  n+2 %because array
       allocation is n
5  2 + 3n
6  2n
7  2n + 3n^2
8  3n^2
9  4n
10
11 1
```

Runtime Analysis

$(6n^2 + 12n + 5)$ - $O(n^2)$

Prefix Averages (Quadratic) II

- The running time of prefixAverages1 is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1)/2$
- Thus, algorithm prefixAverages1 runs in $O(n^2)$ time

Prefix Averages (Linear)

```
1  int[] prefixAverages2(X, n)
2  //Input array X of n integers
3  //Output array A of prefix averages
      of X
4    int[] A = new int[n];
5    int s = 0;
6    for(int i = 0 ; i< n − 1;i++){
7      s = s + X[i];
8      A[i] = s / (i + 1);
9    }
10   return A;
```

1	
2	
3	
4	n+2
5	2
6	2+4(n−1) = 4n−2
7	3(n−1) = 3n−3
8	4(n−1) = 4n−4
9	
10	1

Runtime Analysis (12n-4)

Algorithm prefixAverages2 runs in $O(n)$ time!

Big-Omega and Big-Theta

big-Omega

- at least or asymptotic lower bound
- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n) for\ n \ \geq n_0$

big-Theta

- between or asymptotically tight bound
- $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)\ for\ n \geq n_0$

Big-Omega and Big-Theta II

Example: $5n^2$ is $\Omega(n^2)$:

- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ *for* $n \geq n_0$ let $c = 1$ and $n_0 = 1$

Example: $5n^2$ is $\Theta(n^2)$

- $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$ Let $c = 6$ and $n_0 = 1$

Reinforcement exercises:

- R-4.6 to R-4.21
- R-4.28 to R-4.30

Creativity exercises:

- C-4.33
- C-4.38
- C-4.42
- C-4.43
- C-4.47