Analysis of Algorithms

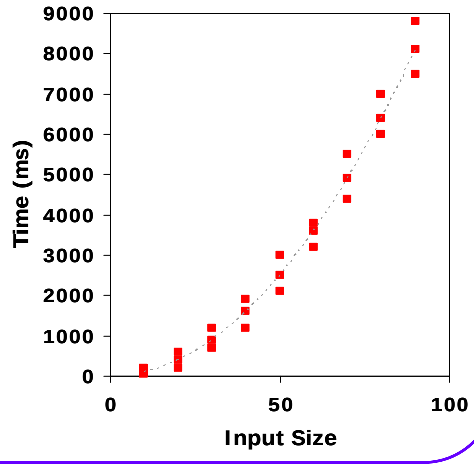
The Big-O Notation

Introduction

- There are many ways (algorithms) to do the same job,
- How can we quantify and compare performance of different algorithms given:
    - Different machines, processors, architectures?
    - Different data size, orderings?
    - Different computer languages?
    - Different compilers?

Introduction: Experimental Studies

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



Introduction: 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

Introduction: Analysis of Algorithms

- An alternative to experimental studies, is to use the theoretical Analysis of Algorithms approach
- It allows us to evaluate the speed of an algorithm independent of the hardware/software environment
  1. Use the *size* of the input, n, rather than the input itself
  2. Write the outline of the algorithm (i.e. use pseudo-code)
  3. Count number of primitive operations in the pseudo-code:
     - One primitive operation = one time unit
     - This count, t, is proportional to the actual execution time
  4. Express the resulting count as a function, $t = f(n)$, for the worst-case input, e.g. $t = f(n) = 21\ n^2 + 6\ n$.
  5. Find the big-O function, $g(n)$, corresponding to $f(n)$.

Analysis of Algorithms: Pseudo-code

- It is a high-level description of an algorithm
- More structured than normal English
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: Find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
  Input array ***A*** of ***n*** integers
  Output max. element of ***A***

  currentMax ← ***A***[0]
  for ***i*** ← 1 **to** ***n*** − 1 **do**
    if ***A***[***i***] > ***currentMax*** **then**
      currentMax ← ***A***[***i***]
  return ***currentMax***

Analysis of Algorithms: Pseudo-code Writing Details

- Flow control:
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Expressions:
  - ← Assignment (like = in Java)
  - = Equality testing (like == in Java)
  - n^2 Superscripts and other mathematical formatting allowed
- Method declaration:
  - **Algorithm** ***method*** (***arg*** [, ***arg***…])
    - **Input** …
    - **Output** …
- Method call:
  - ***var.method*** (***arg*** [, ***arg***…])
- Return value:
  - **return** ***expression***

Analysis of Algorithms: Primitive Operations

The set of primitive operations include:

- Input of a scalar value
- Output of a scalar value
- Returning from a method
- Indexing into an array
- Following an object reference
- Accessing value of a variable, array element, or field of an object
- Assignment to a variable, array element, or field of an object
- Calling a method (not counting argument evaluation and execution of the method body)
- A single arithmetic or logical operation

Analysis of Algorithms: Counting Primitive Operations

- During the analysis of the algorithm, identify the number of primitive operations in each step:
  - For a conditional, count the number of primitive operations on the executed branch
  - For a loop, count the number of primitive operations in the loop body times the number of iterations
  - For a method, count the number of primitive operations in the method's body

Analysis of Algorithms: Counting Primitive Operations

Example:

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by the algorithm, as a function of the input size

Algorithm $arrayMax(A, n)$	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n + 1$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return $currentMax$	1
Total:	$8n - 2$

Analysis of Algorithms: Average-Case and Worst-Case

- We are interested in counting the primitive operations for the average-case input of an algorithm
  - Difficult to find because it requires:
    - Defining a probability distribution on the set of inputs, which is difficult to obtain
    - Calculating expected counts based on a given input distribution, which involves sophisticated probability theory.
- But counting the primitive operations for the worst-case input is much easier and gives stronger results
  - Requires to identify the worst-case input

Analysis of Algorithms: Estimating Running Time

- Suppose an algorithm executes $8n - 2$ 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 the worst-case time, Then

$$a\,(8n - 2) \le T(n) \le b\,(8n - 2)$$

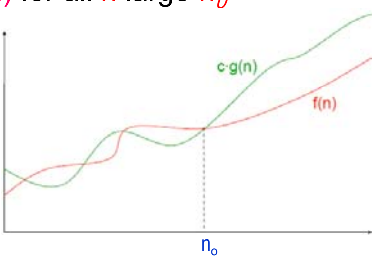
- Hence, the running time $T(n)$ is bounded by two linear functions.

Analysis of Algorithms: 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 $T(n)$
- Thus, the growth rate is not affected by:
  - Constant factors or
  - Lower-order terms
- Examples:
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function

Analysis of Algorithms: The Big-O Notation

- A function $f(n)$ is $O(g(n))$ if there exist constants c, and $n_0>0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- What does this mean in English?
  - $c \cdot g(n)$ is an upper bound of $f(n)$ for all n large n_0
- Examples:
  - f(n) = $3n^2+2n+1$ is $O(n^2)$
  - f(n) = 2n is $O(n)$
  - f(n) = $1000n^3$ is $O(n^3)$



Analysis of Algorithms: More Big-O Examples

- $f(n) = 7n - 2$
  - $7n - 2$ is $O(n)$
  - Find $c > 0$ 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 $n_0 = 1$
- $f(n) = 3n^3 + 20n^2 + 5$
  - $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - Find $c > 0$ 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 $n_0 = 21$
- $f(n) = 3 \log n + 5$
  - $3 \log n + 5$ is $O(\log n)$
  - Find $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
  - This is true for $c = 8$ and $n_0 = 2$

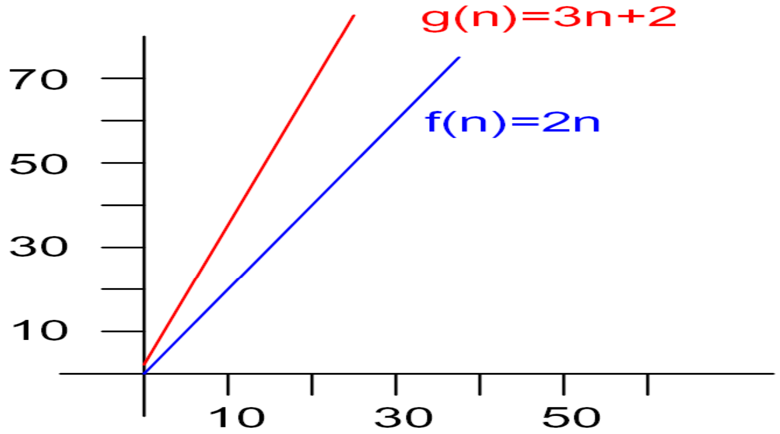
Analysis of Algorithms: Big-O Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"
- Note:
  Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations of an algorithm

Analysis of Algorithms:
Commonly Seen Time Bounds

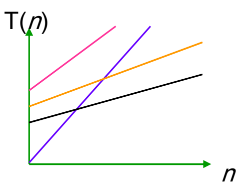
Name	Big-O	Performance
Constant	O(1)	Excellent
Logarithmic	O(log n)	Excellent
Linear	O(n)	Good
n log n	O(n log n)	Pretty Good
Quadratic	$O(n^2)$	OK
Qubic	$O(n^3)$	Maybe OK
Exponential	$O(2^n)$	Too Slow

Analysis of Algorithms:
Linear Performance

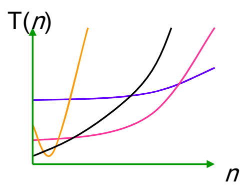


Analysis of Algorithms: Linear vs. Quadratic Performance

Examples:

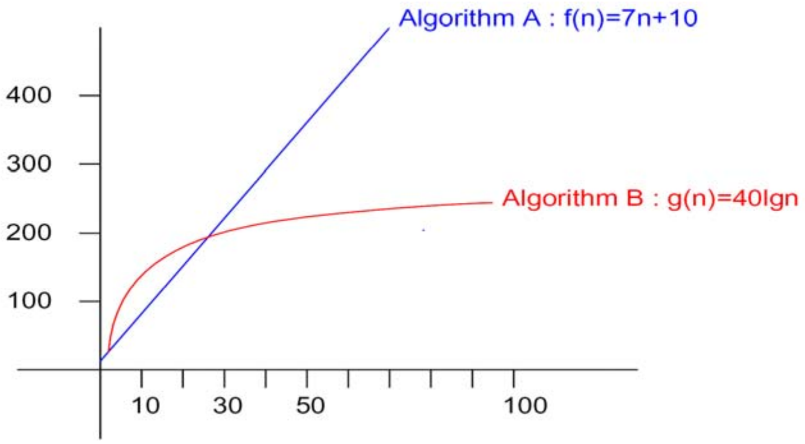


O(n)
linear

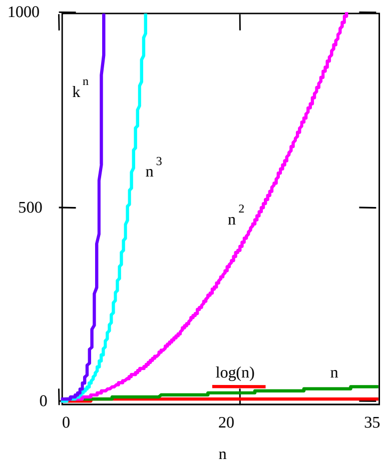


O(n^2)
quadratic

Analysis of Algorithms: Linear vs. Logarithmic Performance



Analysis of Algorithms:
Comparison of All Functions



Analysis of Algorithms
Performance types

Intuition

scale of strength ↓

Adjective	O-notation
constant	O(1)
logarithmic	O(logn)
linear	O(n)
nlogn	O(nlogn)
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(2^n)$, $O(10^n)$, etc.

Analysis of Algorithms:
Running time Comparisons

Running time for algorithm

f(n)	n=256	n=1024	n=1,048,576
1	1μsec	1μsec	1μsec
$\log_2 n$	8μsec	10μsec	20μsec
n	256μsec	1.02ms	1.05sec
$n \log_2 n$	2.05ms	10.2ms	21sec
n^2	65.5ms	1.05sec	1.8wks
n^3	16.8sec	17.9min	36,559yrs
2^n	$3.7\text{x}10^{63}$yrs	$5.7\text{x}10^{294}$yrs	$2.1\text{x}10^{315639}$yrs

Analysis of Algorithms:
Problem-Size Examples

Largest problem that can be solved if Time <= T at
1μsec per step

f(n)	T=1min	T=1hr	T=1wk	T=1yr
n	$6\text{x}10^7$	$3.6\text{x}10^9$	$6\text{x}10^{11}$	$3.2\text{x}10^{13}$
nlogn	$2.8\text{x}10^6$	$1.3\text{x}10^8$	$1.8\text{x}10^{10}$	$8\text{x}10^{11}$
n^2	$7.8\text{x}10^3$	$6\text{x}10^4$	$7.8\text{x}10^5$	$5.6\text{x}10^6$
n^3	$3.9\text{x}10^2$	$1.5\text{x}10^3$	$8.5\text{x}10^3$	$3.2\text{x}10^4$
2^n	25	31	39	44

Example Algorithms
Sequential Search

Given a list of n items and the key of an Item, (target), Find out if the item is in the list.

Algorithm:

```
Assume the list is in an array of size n,
static boolean sequentialSearch (int[] a, int target)
{
  for (int i = 0; i < a.length; i++)          // 2n +1
  {
    if (a[i] == target) return true;          // n
  }
  return false;                               // 1
}
```

Position	Key
1	Mohammad
2	Ahmad
3	Abdullah
4	Yaser
5	Tareq
6	Mustafa
7	Ali
8	Fareed

Sequential Search Performance

- Worst case: Target was not in the list

  f(n) = 2 + 3n ➔ O(n)

- Best case: Target was first in the list

  f(n) = 2+3*1 ➔ O(1)

- Average case:

  Depends on the probability, *p*, of the number of times the loop is executed before the target was found

  f(n) = 2+*p**3n ➔ O(n/2)

Example Algorithms
Sequential Search with Sentinel

- Can we make sequential search faster?

  Note that within the loop there are:
  - One assignment statement
  - Two conditional statements
- We can move one conditional statement to outside the loop
  - At the cost of one additional space, a large list can be searched faster
  - The code to the right uses a sentinel to search faster!
  - If the target is not in the list

$f(n) = 7 + 2n \rightarrow O(n)$

Assume the list is in an array of size n,

```
static boolean fastSequentialSearch (int[] a, int target) {
  int i = 0;                                     // 1
  int n = a.length;                              // 2
  a[n] = target;          // add a sentinel ...  // 2
  while (a[i] != target)  i++;                   // 2n
  if (i != n) return true;                       // 2
  return false;                                  // 1
}
```

Example Algorithms
Binary Search

Given a sorted list of n items and the key of an Item, (e.g.: target_key=Zahed), Find out if the item is in the list.

Algorithm:

Assume the list is in a sorted array of size n,

```
static boolean BinarySearch (int[] a, int target) {
  int low = 0;
  int high = a.length - 1;        // = n-1 = 7
  while (low <= high) {           // iteration 1
    int mid = (low + high)/2;     // = 7/2 = 3
    if (a[mid] < target)
      low = mid+1;                // = 4
    else
      if (a[mid] > target)
        high = mid - 1;
      else return true;
  }
  return false;
}
```

Position	Key
0	Abdullah
1	Ahmad
2	Ali
3	Fareed
4	Mohammad
5	Mustafa
6	Tareq
7	Yaser

Example Algorithms
Binary Search

Given a sorted list of n items and the key of an Item, (e.g.: target_key=Zahed), Find out if the item is in the list.

Algorithm:

```
Assume the list is in a sorted array of size n,
static boolean BinarySearch (int[] a, int target) {
  int low = 0;
  int high = a.length - 1;           // = n-1 = 7
  while (low <= high) {              // iteration 2
    int mid = (low + high)/2;        // = 11/2 = 5
    if (a[mid] < target)
      low = mid+1;                   // = 6
    else
      if (a[mid] > target)
        high = mid - 1;
      else return true;
  }
  return false;
}
```

Position	Key
0	Abdullah
1	Ahmad
2	Ali
3	Fareed
4	Mohammad
5	Mustafa
6	Tareq
7	Yaser

Example Algorithms
Binary Search

Given a sorted list of n items and the key of an Item, (e.g.: target_key=Zahed), Find out if the item is in the list.

Algorithm:

```
Assume the list is in a sorted array of size n,
static boolean BinarySearch (int[] a, int target) {
  int low = 0;
  int high = a.length - 1;           // = n-1 = 7
  while (low <= high) {              // iteration 3
    int mid = (low + high)/2;        // = 13/2 = 6
    if (a[mid] < target)
      low = mid+1;                   // = 7
    else
      if (a[mid] > target)
        high = mid - 1;
      else return true;
  }
  return false;
}
```

Position	Key
0	Abdullah
1	Ahmad
2	Ali
3	Fareed
4	Mohammad
5	Mustafa
6	Tareq
7	Yaser

Example Algorithms
Binary Search

Given a sorted list of n items and the key of an Item, (e.g.: target_key=Zahed), Find out if the item is in the list.

Algorithm:

```
Assume the list is in a sorted array of size n,
static boolean BinarySearch (int[] a, int target) {
  int low = 0;
  int high = a.length - 1;          // = n-1 = 7
  while (low <= high) {             // iteration 4
    int mid = (low + high)/2;       // = 14/2 = 7
    if (a[mid] < target)
      low = mid+1;                  // = 8
    else
      if (a[mid] > target)
        high = mid - 1;
      else return true;
  }
  return false;
}
```

Position	Key
0	Abdullah
1	Ahmad
2	Ali
3	Fareed
4	Mohammad
5	Mustafa
6	Tareq
7	Yaser

Example Algorithms
Binary Search

Given a sorted list of n items and the key of an Item, (e.g.: target_key=Zahed), Find out if the item is in the list.

Algorithm:

```
Assume the list is in a sorted array of size n,
static boolean BinarySearch (int[] a, int target) {
  int low = 0;
  int high = a.length - 1;          // = n-1 = 7
  while (low <= high) {             // No iteration
    int mid = (low + high)/2;       // = 14/2 = 7
    if (a[mid] < target)
      low = mid+1;                  // = 8
    else
      if (a[mid] > target)
        high = mid - 1;
      else return true;
  }
  return false;
}
```

Position	Key
0	Abdullah
1	Ahmad
2	Ali
3	Fareed
4	Mohammad
5	Mustafa
6	Tareq
7	Yaser

Binary Search Performance

- Worst case:

  Target was not in the list,

  The loop is executed k times:

  at ½, ¼, $^1/_8$, … , $^1/_{2^k}$.

  such that: $2^k >= n$

  Then: $k >= \log_2 n$ ➔ $O(\log_2 n)$

Example Algorithms
Matrix Multiplication

Given two matrices A and B of size nxn each, Find C = A x B

Algorithm:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

for all i and j from 1 to n

```
{
   int A[n][n], B[n][n], C[n][n];
   for (i=0; i < n; i++)
      for (j=0; j < n; j++)
      {
         C[i][j]=0;
         for (k=0; k < n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
}
```

This Algorithm has *Time Complexity* of $O(n^3)$

Example Algorithms
Sparse Matrix

The Sparse matrix structure:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 36 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 17 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Row	Column	Value
2	3	36
4	6	17
6	1	-1

Space complexity is: $O(n^2)$ | $O(3k)$; k is # of non-zero entries

Time complexity for printing is: $O(n^2)$ | $O(kn^2)$; $O(k)$ for searching table.

Time complexity for inserting a non-zero value is: $O(1)$ | $O(k)$.