CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

Outline

Algorithm Analysis

Algorithms

An **algorithm** is a step-by-step procedure defined for a mathematical model of computing.

It can use variables, conditional expressions, looping, and sequences of steps.

It can be expressed in pseudocode, flowcharts, or languages such as Python, C++ or even English.

Algorithm Analysis

Program Performance

A program will take a certain amount of time to run, and it will use other resources as well, such as space (memory).

This depends on:

- the size of the input to the program (how many data items are being processed), and
- the model of the computer, etc.

It is important to use an algorithm which will guarantee to the user that for reasonably sized inputs the program will run "quickly".

Algorithm analysis

Algorithm analysis allows us to characterize algorithms according to how much time and memory they require to accomplish the taks.

Algorithm Analysis

Program Performance

A program will take a certain amount of time to run, and it will use other resources as well, such as space (memory).

This depends on:

- the size of the input to the program (how many data items are being processed), and
- the model of the computer, etc.

It is important to use an algorithm which will guarantee to the user that for reasonably sized inputs the program will run "quickly".

Algorithm analysis

Algorithm analysis allows us to characterize algorithms according to how much time and memory they require to accomplish the taks.

Algorithm Analysis

Asymptotic Analysis

Asymptotic analysis is a way to compare algorithms to decide which one will have a faster running time for most inputs.

This is done by determining the **running time** function $T(n)$, depending on the input size n, and then determining the behavior of this function as n becomes large.

$T(n)$ of an algorithm on a particular input n is the number of primitive operations or "steps".

Algorithm Analysis

Asymptotic Analysis

Asymptotic analysis is a way to compare algorithms to decide which one will have a faster running time for most inputs.

This is done by determining the **running time** function $T(n)$, depending on the input size n, and then determining the behavior of this function as n becomes large.

$T(n)$ of an algorithm on a particular input n is the number of primitive operations or "steps".

Algorithm Analysis

Asymptotic Analysis

Asymptotic analysis is a way to compare algorithms to decide which one will have a faster running time for most inputs.

This is done by determining the **running time** function $T(n)$, depending on the input size n, and then determining the behavior of this function as n becomes large.

$T(n)$ of an algorithm on a particular input n is the number of primitive operations or "steps".

Algorithm Analysis

Some Math

Suppose that for one algorithm the running time is $T_1(n) = 4n + 2$ while for a second the running time is $T_2(n) = 3n^2$.

Which function grows faster when n grows?

$T_1(2) = 10$, $T_2(2) = 12$ - not so bad yet, ... but

$T_1(10) = 42$, $T_2(10) = 300$ and

$T_1(100) = 402$, $T_2(100) = 30,000$ - this is already serious

Let's take a look at their ratio:

$$r(x) = \frac{T_1(x)}{T_2(x)} = \frac{4x+2}{3x^2} = \frac{4x}{3x^2} + \frac{2}{3x^2} = \frac{4}{3x} + \frac{2}{3x^2}$$

$r(x) \to 0$, as $x \to \infty$. Graph of $y = r(x)$ has an asymptote $y = 0$.

Algorithm Analysis

Some Math

Suppose that for one algorithm the running time is $T_1(n) = 4n + 2$ while for a second the running time is $T_2(n) = 3n^2$.

Which function grows faster when n grows?

$T_1(2) = 10,\ T_2(2) = 12$ - not so bad yet, ... but

$T_1(10) = 42,\ T_2(10) = 300$ and

$T_1(100) = 402,\ T_2(100) = 30,000$ - this is already serious

Let's take a look at their ratio:

$$r(x) = \frac{T_1(x)}{T_2(x)} = \frac{4x+2}{3x^2} = \frac{4x}{3x^2} + \frac{2}{3x^2} = \frac{4}{3x} + \frac{2}{3x^2}$$

$r(x) \to 0$, as $x \to \infty$. Graph of $y = r(x)$ has an asymptote $y = 0$.

Algorithm Analysis

Some Math

Suppose that for one algorithm the running time is $T_1(n) = 4n + 2$ while for a second the running time is $T_2(n) = 3n^2$.

Which function grows faster when n grows?

$T_1(2) = 10,\ T_2(2) = 12$ - not so bad yet, ... but

$T_1(10) = 42,\ T_2(10) = 300$ and

$T_1(100) = 402,\ T_2(100) = 30,000$ - this is already serious

Let's take a look at their ratio:

$$r(x) = \frac{T_1(x)}{T_2(x)} = \frac{4x+2}{3x^2} = \frac{4x}{3x^2} + \frac{2}{3x^2} = \frac{4}{3x} + \frac{2}{3x^2}$$

$r(x) \to 0$, as $x \to \infty$. Graph of $y = r(x)$ has an asymptote $y = 0$.

Algorithm Analysis

Theta (Θ) Notation

Let's use even more simplifying abstraction: it is the **rate of growth** or the **order of growth** of the running time that really interests us.

We therefore consider only the leading term of a formula, since the lower-order terms are relatively insignificant for large n, and ignore the leading coefficient.

In that last slide, we would say that
$T_1(n) = 4n + 2$, a linear function, is a $\Theta(n)$ function, and
$T_2(n) = 3n^2$, a quadratic function, is $\Theta(n^2)$.

Formally, a function $T(n)$ is $\Theta(f(n))$ if there are constants c_1, c_2 and n_0 such that for all $n > n_0$, $T(n) < c_1 f(n)$ and $T(n) > c_2 f(n)$.

Algorithm Analysis

Theta (Θ) Notation

Let's use even more simplifying abstraction: it is the **rate of growth** or the **order of growth** of the running time that really interests us.

We therefore consider only the leading term of a formula, since the lower-order terms are relatively insignificant for large n, and ignore the leading coefficient.

In that last slide, we would say that
$T_1(n) = 4n + 2$, a linear function, is a $\Theta(n)$ function, and
$T_2(n) = 3n^2$, a quadratic function, is $\Theta(n^2)$.

Formally, a function $T(n)$ is $\Theta(f(n))$ if there are constants c_1, c_2 and n_0 such that for all $n > n_0$, $T(n) < c_1 f(n)$ and $T(n) > c_2 f(n)$.

Algorithm Analysis

Theta (Θ) Notation

Let's use even more simplifying abstraction: it is the **rate of growth** or the **order of growth** of the running time that really interests us.

We therefore consider only the leading term of a formula, since the lower-order terms are relatively insignificant for large n, and ignore the leading coefficient.

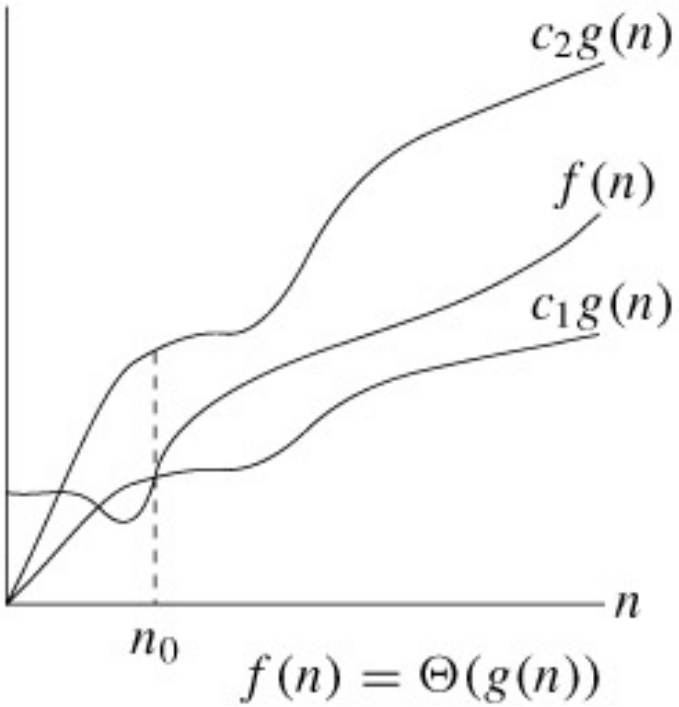
In that last slide, we would say that
$T_1(n) = 4n + 2$, a linear function, is a $\Theta(n)$ function, and
$T_2(n) = 3n^2$, a quadratic function, is $\Theta(n^2)$.

Formally, a function $T(n)$ is $\Theta(f(n))$ if there are constants c_1, c_2 and n_0 such that for all $n > n_0$, $T(n) < c_1 f(n)$ and $T(n) > c_2 f(n)$.

Algorithm Analysis

Theta (Θ) Notation

A function $f(n)$ is $\Theta(g(n))$ if there are constants c_1, c_2 and n_0 such that for all $n > n_0$, $c_2 g(n) < f(n) < c_1 g(n)$



Algorithm Analysis

Some Facts About Θ Notation

- A $\Theta(n^2)$ running time will dominate (be eventually slower than) a $\Theta(n)$ running time.
- A $\Theta(1)$ running time is a constant (times 1) so it does not grow as n increases. (This is the fastest kind of algorithm.)
- A $\Theta(log\ n)$ algorithm is faster than a $\Theta(n)$ algorithm.

Algorithm Analysis

Big O Notation

There is a less precise way to classify functions called "big-O" or simply O-notation.

A function $T(n)$ is $O(n^2)$, for example, if it is eventually less than cn^2 for some c.

This limits how large the running time is for large n, (i.e.how slow the algorithm is), but it gives less information: there is no lower bound on the growth of the function $T(n)$ – it could actually be faster than cn for some c.

Thus, $T_2(n) = 3n^2$ is $O(n^2)$, but so is $T_1(n) = 4n + 2$. (Any $O(n)$ function is automatically $O(n^2)$, since it can be bounded above by a quadratic function.)

Algorithm Analysis

Big O Notation

There is a less precise way to classify functions called "big-O" or simply O-notation.

A function $T(n)$ is $O(n^2)$, for example, if it is eventually less than cn^2 for some c.

This limits how large the running time is for large n, (i.e.how slow the algorithm is), but it gives less information: there is no lower bound on the growth of the function $T(n)$ – it could actually be faster than cn for some c.

Thus, $T_2(n) = 3n^2$ is $O(n^2)$, but so is $T_1(n) = 4n + 2$. (Any $O(n)$ function is automatically $O(n^2)$, since it can be bounded above by a quadratic function.)

Algorithm Analysis

Big O Notation

There is a less precise way to classify functions called "big-O" or simply O-notation.

A function $T(n)$ is $O(n^2)$, for example, if it is eventually less than cn^2 for some c.

This limits how large the running time is for large n, (i.e.how slow the algorithm is), but it gives less information: there is no lower bound on the growth of the function $T(n)$ – it could actually be faster than cn for some c.

Thus, $T_2(n) = 3n^2$ is $O(n^2)$, but so is $T_1(n) = 4n + 2$. (Any $O(n)$ function is automatically $O(n^2)$, since it can be bounded above by a quadratic function.)

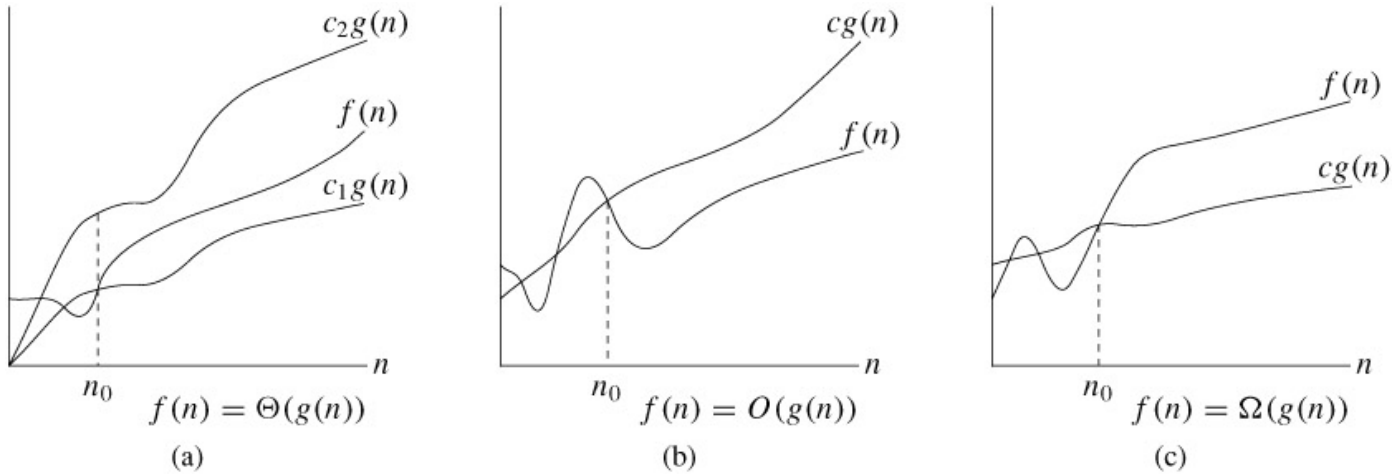
Algorithm Analysis

Big O, Θ and Ω Notations

A function $f(n)$ is $\Theta(g(n))$ if there are constants c_1, c_2 and n_0 such that for all $n > n_0$, $c_2 g(n) < f(n) < c_1 g(n)$

A function $f(n)$ is $O(g(n))$ if there are constant c and n_0 such that for all $n > n_0$, $f(n) < cg(n)$

A function $f(n)$ is $\Omega(g(n))$ if there are constants c and n_0 such that for all $n > n_0$, $f(n) > cg(n)$



Algorithm Analysis

There are three types of analysis, possibly giving different running time functions:

- Best case is the fastest possible time. We don't use this, since all it tells us is what will happen if you are lucky. This gives no guarantee to the customer.
- Average case is how fast the algorithm runs on average, over different sets of inputs. It gives a good idea of how much time the program will use over a long period of time.
- Worst case is how fast the algorithm is guaranteed to run.

  To get the running time, count each single step as 1 time unit. Then, using math, calculate how many time units it takes the entire algorithm to run if the size of the input is n.

Algorithm Analysis

First Example: Linear Search

```
def search(items, target):
    i = 0
    while i < len(items):
        if items[i] == target:
            return i
        i += 1
    return -1
```

Algorithm Analysis

Analysis of Linear Search (worst case)

```
def search(items, target):
    i = 0                    1 step
    while i < len(items):    2 steps (comparison, len)
    worst case:  n iterations (no target)
        if items[i] == target:          2 steps
            return i                never executed
        i += 1               1 step
    return -1                1 step
```

running time: $T(n) = 5n + 2$.
The algorithm is $\Theta(n)$.

Algorithm Analysis

Analysis of Linear Search (worst case)

```
def search(items, target):
    i = 0                    1 step
    while i < len(items):    2 steps (comparison, len)
    worst case:  n iterations (no target)
        if items[i] == target:          2 steps
            return i                never executed
        i += 1               1 step
    return -1                1 step
```

running time: $T(n) = 5n + 2$.
The algorithm is $\Theta(n)$.

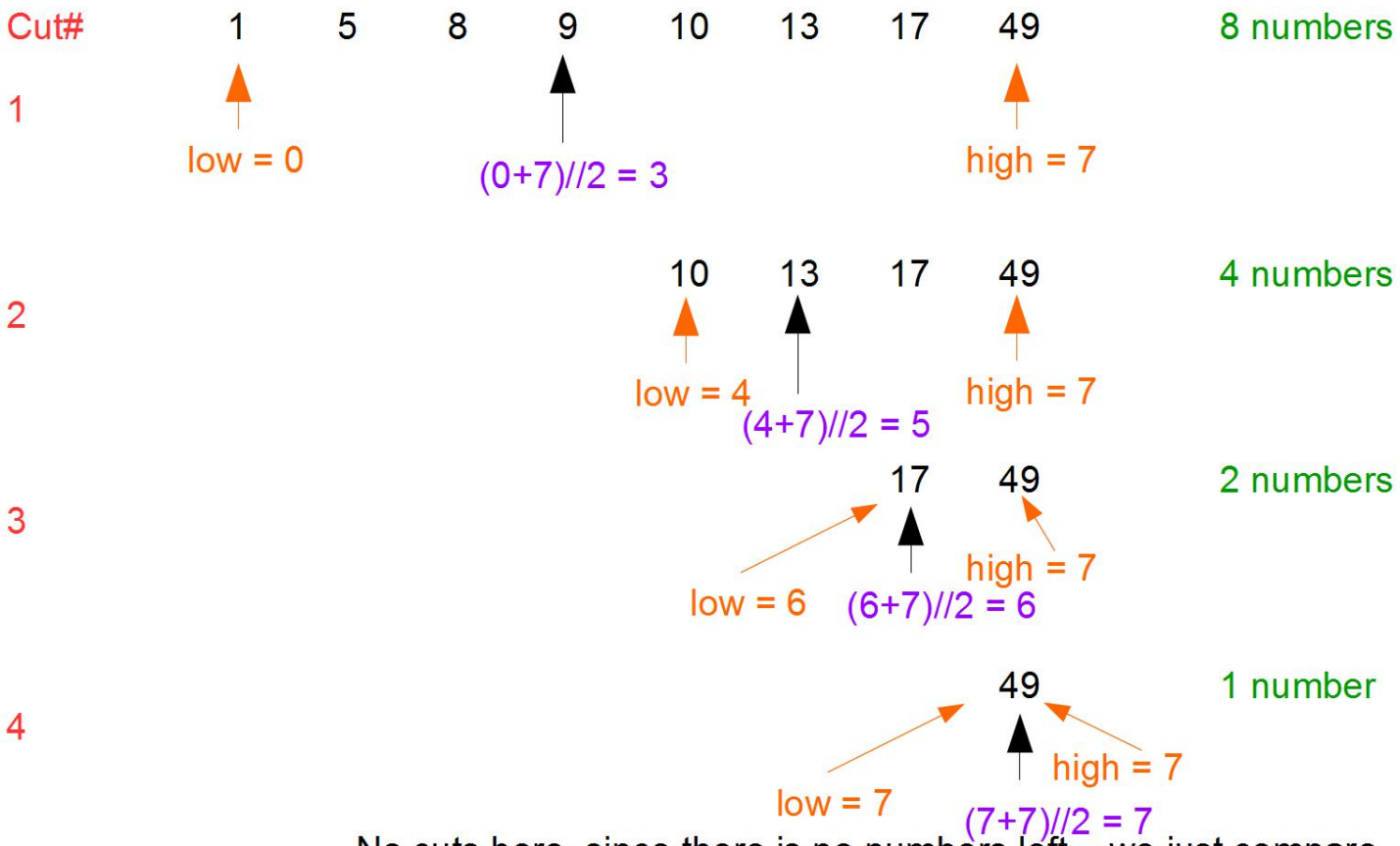
Algorithm Analysis

Second Example: Binary Search (Precondition: items list is sorted)

```
def search(items, target):
    low = 0
    high = len(items) - 1
    while low <= high:
        mid = (low + high) // 2
        item = items[mid]
        if target == item :
            return mid
        elif target < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

Binary Search Example

2) Now, let's see how the binary search algorithm provided in our book works: let's take a list of 8 integer numbers: 1, 5, 8, 9, 10, 13, 17, 49, *and let's try to find 26*
(note that they are already sorted as Binary Search has this requirement)



No cuts here, since there is no numbers left – we just compare

STOP

Binary Search Example

With binary search we can see the following pattern:

List size	# of cuts	
1	0	
2	1	$2^1=2$
4	2	$2^2=4$
8	3	$2^3=8$
16	4	$2^4=16$

Do you see
$2^{\#\text{of cuts}} = n$ dependency?

If we re-write it in logarithmic form:
$\log_2 n = \#$ of cuts

Algorithm Analysis

Analysis of Binary Search (worst case)

```
def search(items, target):
 low = 0                          1 step
 high = len(items) - 1            2 steps
 while low <= high:               1 step; log2 n iterations
    mid = (low + high) // 2       2 steps
    item = items[mid]             1 step
    if target == item :           1 step
       return mid
    elif target < item:
       high = mid - 1             1 or 0 steps
    else:
       low = mid + 1              1 or 0 steps
 return -1                        1 step
```

Algorithm Analysis

Analysis of Binary Search (worst case): $T(n) = 6(log_2\ n) + 4$

```
def search(items, target):
 low = 0                          1 step
 high = len(items) - 1            2 steps
 while low <= high:               1 step; log2 n iterations
    mid = (low + high) // 2       2 steps
    item = items[mid]             1 step
    if target == item :           1 step
       return mid
    elif target < item:
       high = mid - 1             1 or 0 steps
    else:
       low = mid + 1              1 or 0 steps
 return -1                        1 step
```

Algorithm Analysis

Analysis of Binary Search (worst case): $T(n) = 6(log_2\ n) + 4$

So this algorithm is $\Theta(log\ n)$.
(In Θ notation, we don't care what the logarithm base is, since changing from base a to base b is just multiplying by a constant, $log_b\ a$.)

Conclusion: binary search ($\Theta(log\ n)$) is faster than linear search ($\Theta(n)$)
but: binary search needs sorted array, whereas linear search does not require it.

In-class work

In-class work

see the handout