Asymptotic analysis provides basic vocabulary for discussing the design and analysis of algorithms.

Often referred to as "big O"

"Sweet spot" for high level reasoning about algorithms

- Coarse enough to suppress computer architecture / programming language / compiler-level details
- Sharp enough to make useful comparisons between different algorithms, especially on large sets of inputs

The point of asymptotic analysis is to suppress $\underbrace{\text{constant factors}}_{\text{TOO SYSTEM DEPENDENT}}$ and $\underbrace{\text{lower order terms}}_{\text{IRRELEVANT FOR LARGE NUMBER OF INPUTS}}$.

EXAMPLE:

From previous work on merge sort analysis the estimated runtime of the algorithm was $5n\lg(n) + 5n$.

SUPPRESS CONSTANT FACTORS: $5n\lg n + 5n \rightarrow n\lg(n) + n$

REMOVE LOWER ORDER TERMS: $n\lg(n) + n \rightarrow n\lg(n)$

RUNNING TIME IS: $O(n) = n\lg n$, O of $n \lg n$, where

n = input size (length of input arrays)

EXAMPLE; ONE LOOP

Does arbitrary array, A, of length n contain the target value, t?

```
for i=1 to n:
  if A[i] == t:
    return True
return False
```

In the worst case t may be in the last position or not in the array. The entire array will be scanned and compared once or n times before ending.

Therefore, $O(n) = n$ or linear runtime

Two Loops in Sequence

Given: arrays A and B, both containing n elements
a target value, t,

Does either array A or array B contain t?

```
for i=1 to n:
  if A[i] == t:
    return True

for i=1 to n:
  if B[i] == t:
    return True

return False
```

Similar to the one loop example t could be in the last position or not in one of the arrays. Therefore, there will be n comparisons in the worst-case per loop.

Since there are two loops in sequence, there are $n + n$ comparisons or $2n$.

Suppressing constant factors $2n \rightarrow n$

$\therefore$ $O(n) = n$ for two loops in sequence.

Two Nested Loops

Given arrays A and B, both with n elements

Do arrays A and B have a value in common?

```
for i=1 to n:
  for j=1 to n:
    if A[i] == B[j]:
      return True
return False
```

For every iteration of the outer loop the inner loop does n iterations.

Therefore, $n \text{ loops} \cdot n \frac{\text{iterations}}{\text{loop}} = n^2 \text{ iterations}$

$O(n) = n^2$ for two nested loops

Two Nested Loops: Part Deux

Given an array A of length n

Does array A have duplicate elements?

```
for i=1 to n-1:
  for j=i+1 to n:
    if A[i] == A[j]:
      return True
return False
```

For every iteration of the outer loop, the inner loop does one less iteration. That is on the outer loops k^{th} iteration the inner loop does $n-k$ iterations, effectively halving the number of comparisons compared to the previous example.

$n \text{ Loops} \cdot \frac{n}{2} \frac{\text{iterations}}{\text{Loop}} = \frac{n^2}{2} \text{ iterations}$ $O(n) = n^2$ since the $\frac{1}{2}$ is suppressed

BIG-O FORMAL DEFINITION

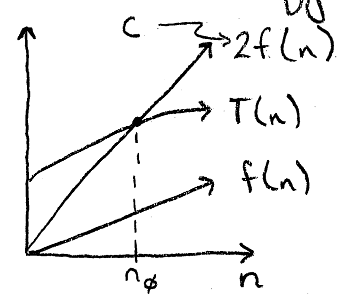
Let $T(n)$, a function on an input quantity of n, $n = 1, 2, \ldots$

Usually n is selected such that it will represent the worst case running time or input quantity to the algorithm.

When is $T(n) = O(f(n))$?

This occurs for all sufficiently large n, $n_\emptyset \le n$, where $n_\emptyset$ is the minimum input quantity to satisfy the above requirement, and $T(n)$ is bounded above by a constant multiple of $f(n)$.

Here's a picture to clarify -



$T(n)$ is bounded by $f(n)$ and a constant times $f(n)$, $2f(n)$, so $T(n) = O(f(n))$

$T(n) = O(f(n))$ if and only if there exists constants c and $n_\emptyset$ both greater than zero; $c, n_\emptyset > \emptyset$; such that $T(n) \le c \cdot f(n)$ for all $n \ge n_\emptyset$, where c and $n_\emptyset$ are independent of the number of elements put into the function.

BIG OMEGA, Ω, and BIG THETA, Θ

Big Omega and Big Theta are both notations to describe an algorithms runtime.

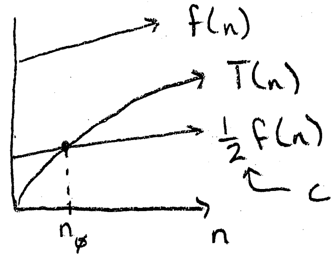
Ω represents the best time an algorithm can run, whereas big O is concerned with worst case runtime.

Θ represents both big-O and big-Ω, a bound on the upper and lower runtimes for an algorithm

(con't on next sheet)

BIG-Ω AND BIG Θ (con't)

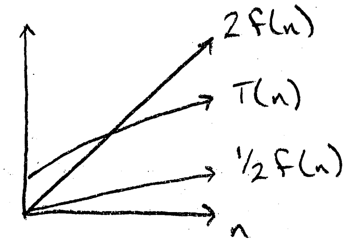
BIG Ω PICTURE



$T(n)$ is bounded by a constant times $f(n)$ on the low side, $\frac{1}{2} f(n)$, so $T(n) = \Omega(f(n))$

$T(n) = \Omega(f(n))$ if and only if there exists constants c and $n_\emptyset$ where $c, n_\emptyset > \emptyset$; such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_\emptyset$, where c and $n_\emptyset$ are independent of the number of elements put into the function, n.

BIG Θ PICTURE



$T(n)$ is bounded by both $2f(n)$ and $\frac{1}{2} f(n)$, so $T(n) = \Theta(f(n))$

While Big Ω provides the best case in the real world we prepare for the worst case (factor of safety)

Additionally Big Θ provides for a more exact definition on the bounds of an algorithms runtime, but in practice we do not generally need the tighter tolerance because we are only concerned with reducing the upper bound, big O, of the runtime.