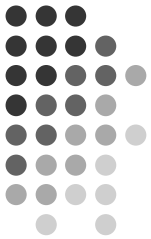
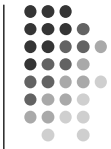
Computational Goals
Correctness & Efficiency

6B

Efficiency

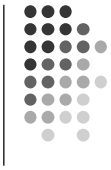


Efficiency



- A computer program should be totally correct, but it should also
  - execute as quickly as possible (time-efficiency)
  - use memory wisely (storage-efficiency)
- How do we compare programs (or algorithms in general) with respect to execution time?
  - various computers run at different speeds due to different processors
  - compilers optimize code before execution
  - the same algorithm can be written differently depending on the programming paradigm

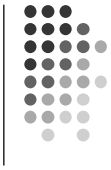
Example: Linear Search



- Input: Vector A of n unique integers.
  - The integers are not in any specific order.
- Output: The index of a specific integer (called the target) or 0 if the integer is not found.
- Algorithm:

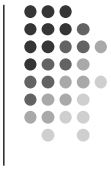
  1. Set index = 1.
  2. While index ≤ N and A[index] ≠ target, do the following:
     a. Add 1 to index
  3. If index ≤ N, output index;
     Otherwise, output 0.

Some questions



- What is the maximum number of data values we need to examine? (i.e. worst case)
- What is the minimum number of data values we need to examine? (i.e. best case)
- How many storage cells do we need to store information for use in this algorithm?

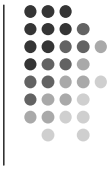
Linear Search



- We measure time efficiency by counting the number of operations performed by the algorithm.
- But what is an operation?

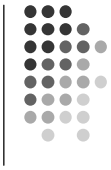
1. Set index = 1.
2. While index ≤ N and A[index] ≠ target,
   do the following:
   a. Add 1 to index
3. If index ≤ N, output index;
   Otherwise, output 0.

Counting Operations



- We typically count operations that are a function of the amount of data (n) that we have to process.
- Abstraction:
  - We don't count individual operators (+, -, ...).
  - We count more general operations:
    - assignments (← or "Set ... equal to ...")
    - comparisons (<, >, ≤, ≥, =, ≠)
- For linear search, we might count operations as:
  # ops = 1 (step 1) + N+1 (while comparison) +
  N (step 2a) + 2 (step 3) = 2N + 4
  (note: this is the worst case scenario)

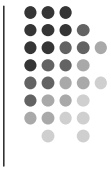
Example: Binary Search

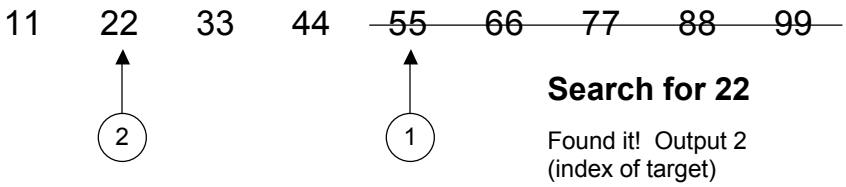


- Input: Vector A of n unique integers.
  - The integers must be sorted in increasing order.
- Output: The index of a specific integer (called the target) or 0 if the integer is not found.
- (Recursive) Algorithm:

  1. If number of integers to be examined is 0, output 0 and stop.
  2. Set mid = subscript of middle element of A.
  3. If A[mid] = target, output mid and stop.
  4. If A[mid] > target, perform binary search on A[1..mid-1]
     Otherwise, perform binary search on A[mid+1..n]

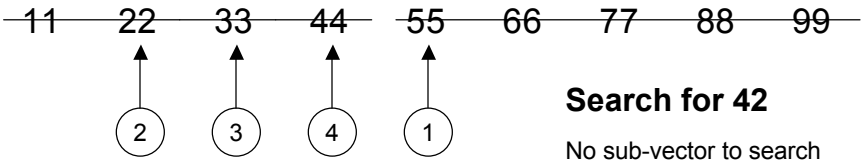
Binary Search





Search for 22

Found it! Output 2
(index of target)



Search for 42

No sub-vector to search
Output 0 (not found)

Counting Operations Again

- For binary search, consider the worst-case scenario (target is not in vector)
- How many times can we split the vector in half before we the vector becomes empty?
- 9 --> 4 --> 2 --> 1 --> 0
- In general, we can split the vector in half approximately $\lfloor \log_2 n \rfloor + 1$ times before it becomes empty.
- Recall the log function:

  $\log_a b = c$ is equivalent to **$a^c = b$**

  Examples: $\log_2 128 = 7$ $\log_2 n = 5$ implies $n = 32$

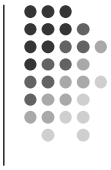
Comparing Algorithms

- Assume an algorithm requires N data values to process. If each operation takes 1 μs* to execute, how many μs will it take to run the algorithm on 100 data values if the algorithm has the following number of computations?

  * 1 microsecond (μs) = 1 millionth of a second

Number of Computations	Execution Time
N	100 μs
$N \cdot \log_2 N$	665 μs
N^2	10,000 μs
N^3	1,000,000 μs = 1 sec

Comparing Algorithms

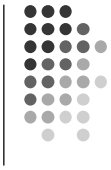


- Assume an algorithm requires N data values to process. If each operation takes 1 μs to execute, how many μs will it take to run the algorithm on 100 data values if the algorithm has the following number of computations?

Number of Computations	Execution Time
2^N	$> 10^{30}$ μs
N!	$> 10^{160}$ μs
N^N	$> 10^{201}$ μs

  - The number of protons in the known universe is $< 10^{79}$
  - The number of microseconds since the Big Bang is $< 10^{24}$.

Order of Complexity

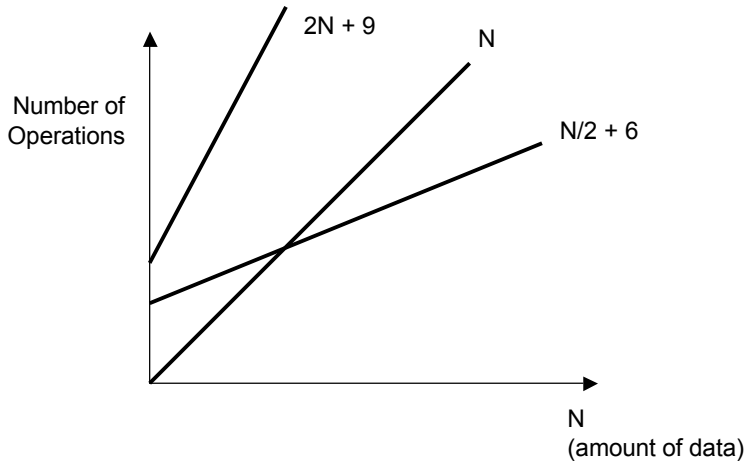


- For very large n, we express the number of operations as the (time) order of complexity.
- Order of complexity for worst-case behavior is often expressed using Big-O notation:

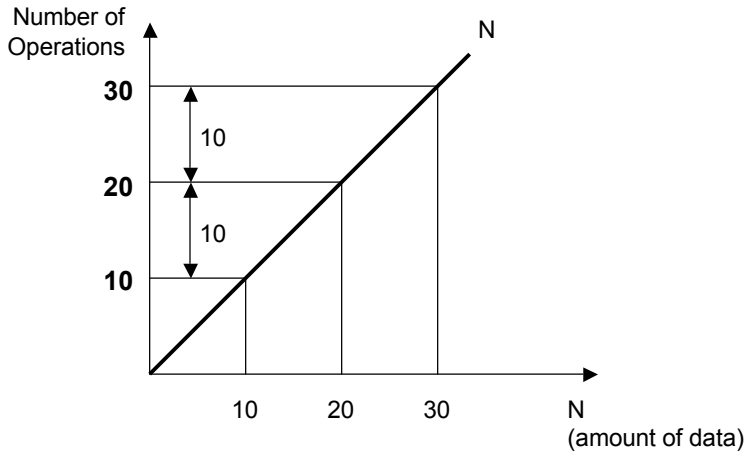
Number of operations	Order of Complexity
N	O(N)
N/2 + 6	O(N)
2N + 9	O(N)

Usually doesn't matter what the constants are... we are only concerned about the highest power of n.

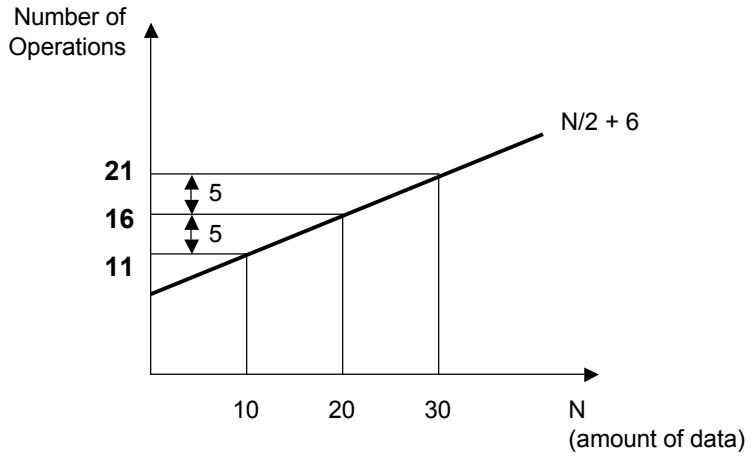
O(N) ("Linear")


2N + 9
N
N/2 + 6
Number of Operations
N
(amount of data)

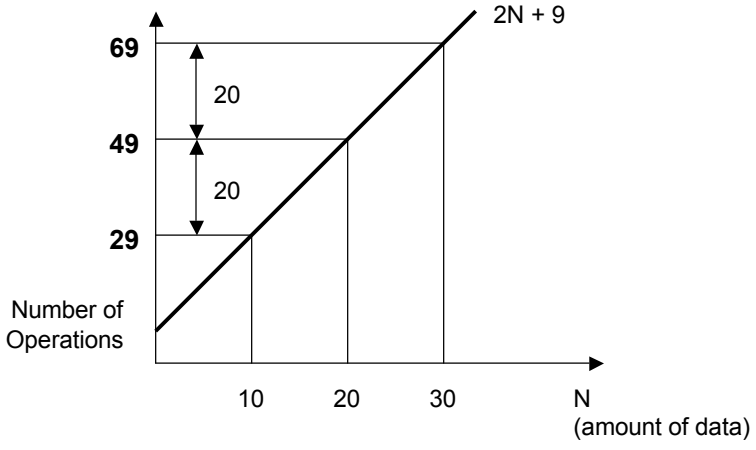
O(N)


Number of Operations
N
30
20
10
10
10
10
20
30
N
(amount of data)

O(N)


Number of Operations
N/2 + 6
21
16
11
5
5
10
20
30
N
(amount of data)

O(N)

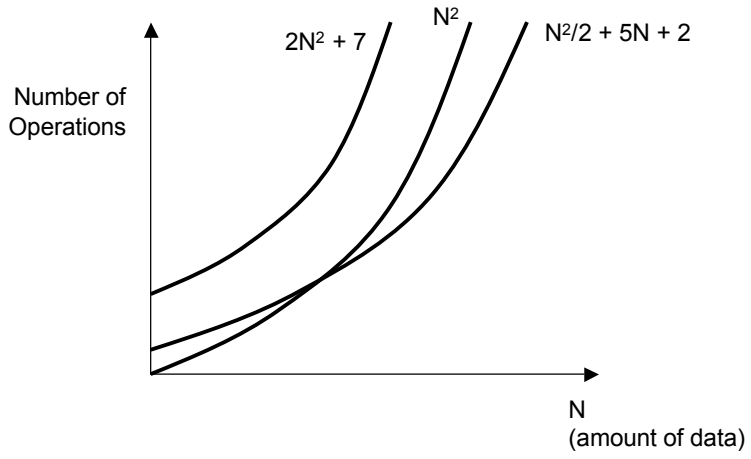

2N + 9
69
49
29
20
20
Number of Operations
10
20
30
N
(amount of data)

Order of Complexity

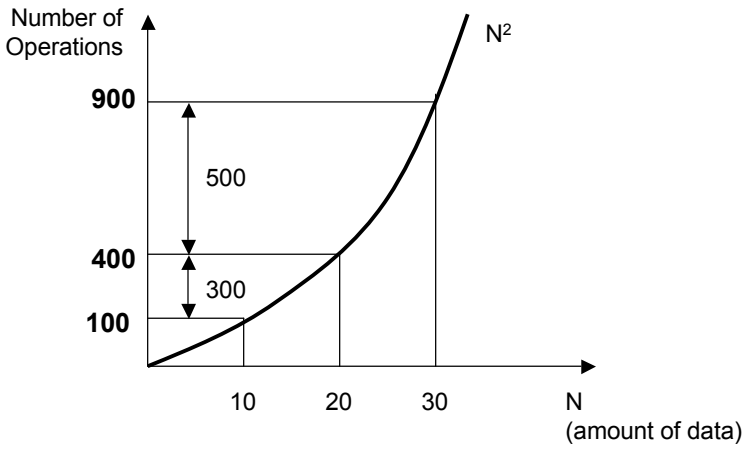
Number of operations	Order of Complexity
N^2	$O(N^2)$
$2N^2 + 7$	$O(N^2)$
$N^2/2 + 5N + 2$	$O(N^2)$

Usually doesn't matter what the constants are... we are only concerned about the highest power of n.

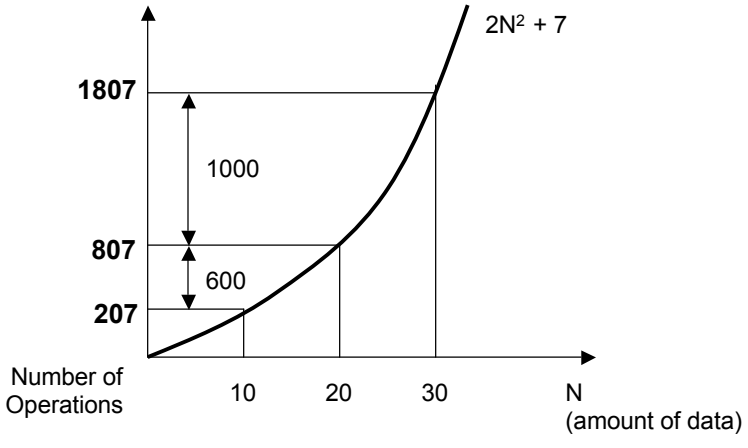
$O(N^2)$ ("Quadratic")



$O(N^2)$


Number of Operations
N^2
900
500
400
300
100
10
20
30
N
(amount of data)

$O(N^2)$

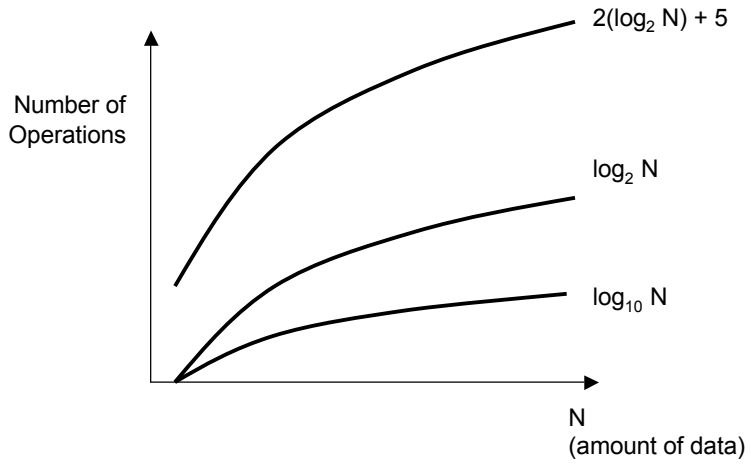

$2N^2 + 7$
1807
1000
807
600
207
Number of Operations
10
20
30
N
(amount of data)

Order of Complexity

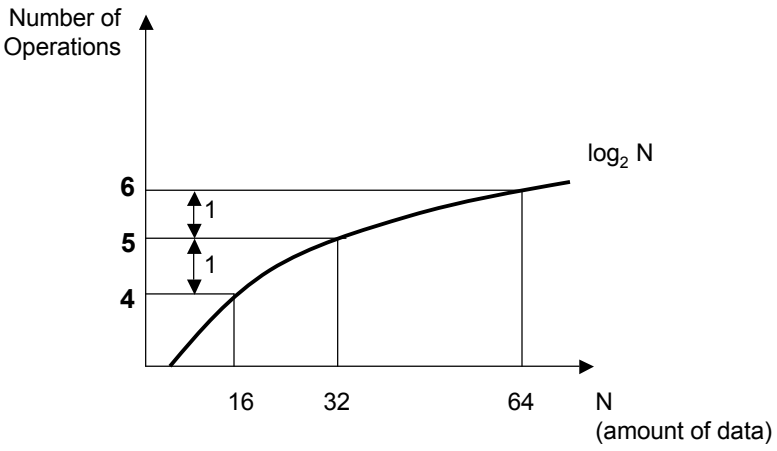
Number of operations	Order of Complexity
$\log_2 N$	O(log N)
$\log_{10} N$	O(log N)
$2(\log_2 N) + 5$	O(log N)

The logarithm base is not written in big O notation since all that matters is that the function is logarithmic.

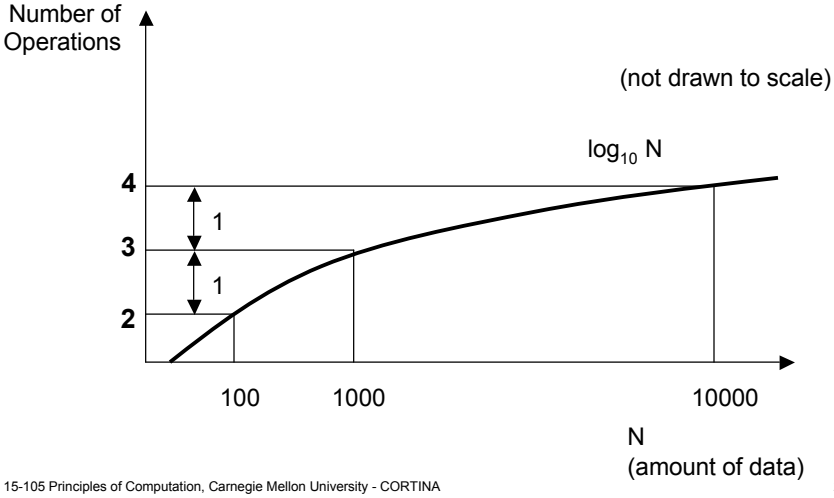
O(log N) ("Logarithmic")



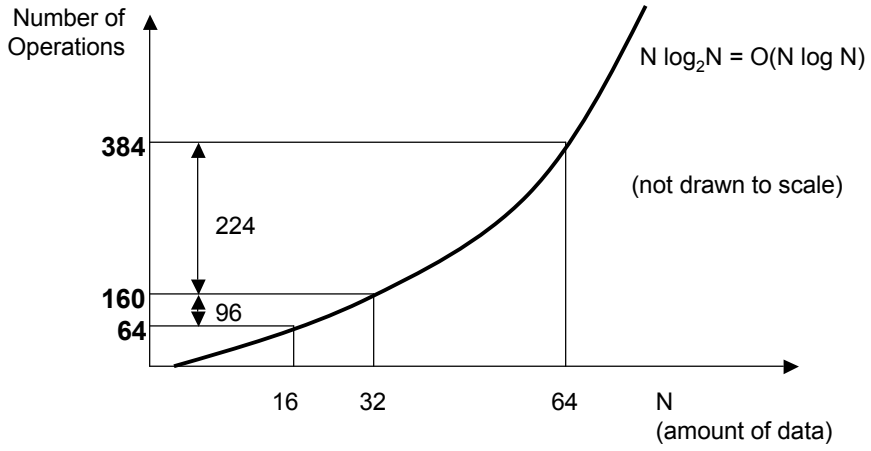
O(log N)


Number of Operations
log2 N
6
5
4
1
1
16
32
64
N
(amount of data)

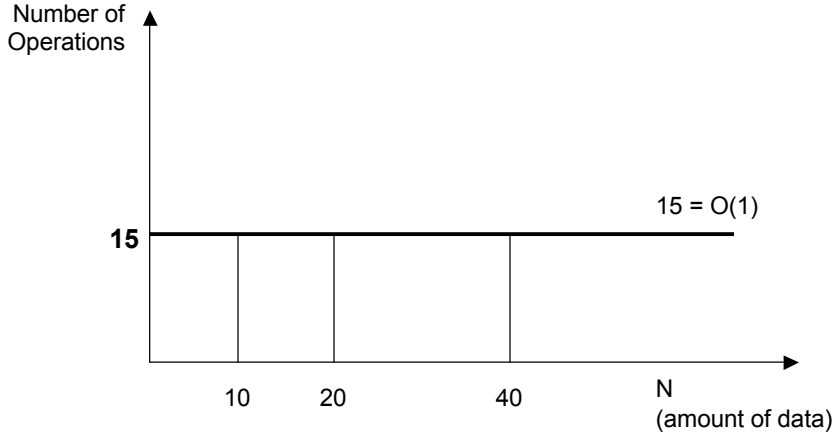
O(log N)


Number of Operations
(not drawn to scale)
log10 N
4
3
2
1
1
100
1000
10000
N
(amount of data)

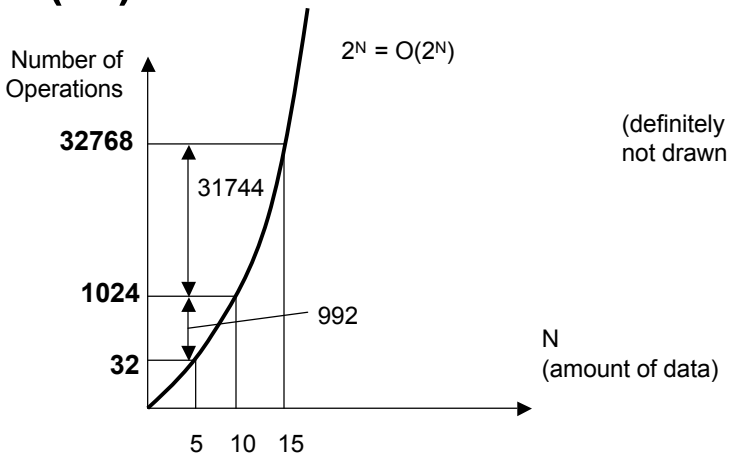
O(N log N)



O(1) ("Constant")

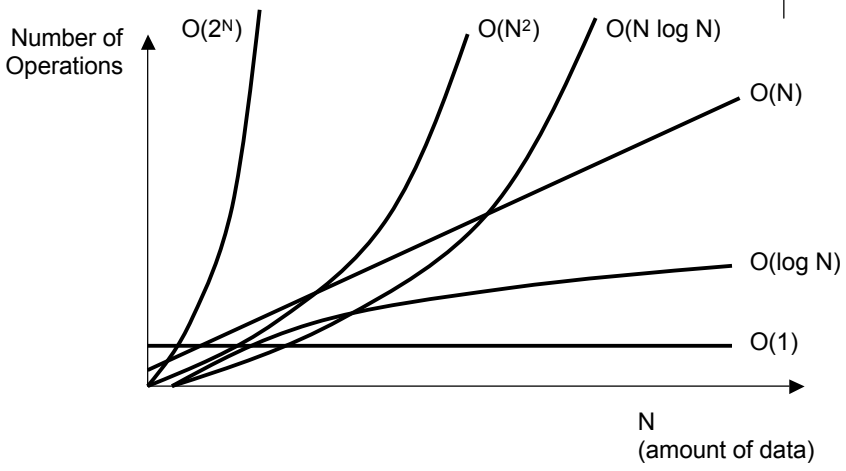


$O(2^N)$

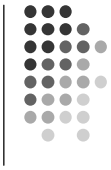


(definitely not drawn to scale)

Comparing Big O Functions

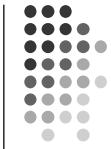


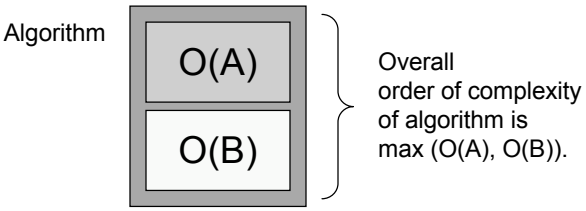
Searches



- Linear Search
  - Count comparisons as operations.
  - Worst Case: O(N)
- Binary Search
  - Count comparisons as operations.
  - Worst Case: O(log N)
- Which algorithm is better?
  - In general, for large values of N: log N < N so binary search is better for large N.
  - BUT, binary search requires a sorted vector.
  - What about if the target is in the first vector cell?

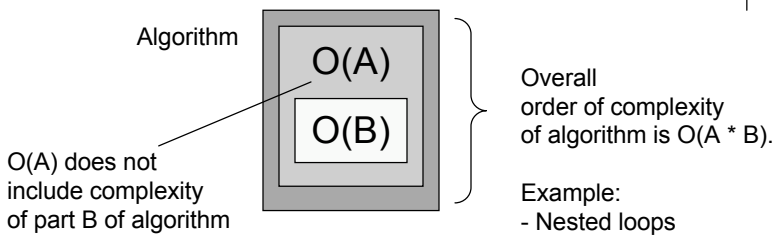
Order of Complexity





- Examples:
  - O(log N) + O(N) = O(N)
  - O(N log N) + O(N) = O(N log N)
  - O(N log N) + $O(N^2)$ = $O(N^2)$
  - $O(2^N)$ + $O(N^2)$ = $O(2^N)$

Order of Complexity



- Examples:
  - O(log N) * O(N) = O(N log N)
  - O(N log N) * O(N) = O(N^2 log N)
  - O(N) * O(1) = O(N)

Nested Loops Example 1

- What is the (worst-case) order of complexity of the following algorithm?

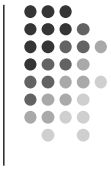
1. Set i = 1 } 1 = O(1)
2. While i ≤ N do the following:
   a. Set j = 1
   b. While j ≤ N do the following:
      i. Output i * j
      ii. Add 1 to j } 2N = O(N)
   c. Add 1 to i

3N = O(N) (count step b as one op.)

Count assignment and I/O instructions as operations

O(1) + O(N*N) = O(N^2)

Nested Loops Example 2



- What is the (worst-case) order of complexity of the following algorithm?
  1. Set i = 1 } 1 = O(1)
  2. While i ≤ N do the following:
     a. Set j = 1
     b. While **j < N** do the following:
        i. Output i * j
        ii. **Multiply j by 2**
     c. Add 1 to i

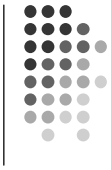
Count assignment and I/O instructions as operations

3N = O(N) (count step b as one op.)

2(⌈$\log_2 N$⌉) = O(log N)

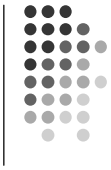
O(1) + O(N*logN) = O(NlogN)

Bubble Sort



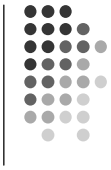
- What is the worst-case order of complexity of bubble sort on N data values?
- Count the comparisons as operations.
  - First pass through: N-1 comparisons
  - Second pass through: N-2 comparisons
  - ...
  - Last (N-1st) pass through: 1 comparison
- Order of complexity:
  (N-1)+(N-2)+...+1 = N(N-1)/2 = (N^2-N)/2 = O(N^2)
  - We call bubble sort a "quadratic sort".

Recursive Algorithms



- We use recurrence relations to determine the order of complexity of recursive algorithms.
- Example (Towers of Hanoi algorithm):
  - Let C(N) = the number of operations needed for N discs (NOTE: C(1) = 1)
  - In general, the number of operations is:
    the number required to move N-1 discs to the middle peg +
    1 (to move the largest disc) +
    the number required to move N-1 discs to the final peg
  - Thus, C(N) = C(N-1) + 1 + C(N-1) = 2C(N-1) + 1
  - Solution: $C(N) = 2^N - 1 = O(2^N)$

Merge Sort



- What is the order of complexity of merge sort?
  - Let C(N) = total number of operations performed for merge sort on N data values.
  - Operations include comparisons in the merge process.
  - C(1) = 0
  - Maximum number of comparisons to merge 2 arrays on size N/2 into one array of size N: N-1 comparisons.
  - C(N) = C(N/2) + C(N/2) + N - 1 = 2C(N/2) + N - 1
  - Solution: $C(N) = N\log_2 N - N + 1 = O(N \log N)$.
- Order of complexity usually doesn't take into account the extra overhead of handling recursion on the computer.

Complexity...

- "Simplicity does not precede complexity, but follows it."
- "Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it."

Alan Perlis
First head of the CS dept at CMU
Recipient of the ACM Turing Award in 1966.