16.070

Introduction to Computers & Programming

Asymptotic analysis: upper/lower bounds, Θ notation
Binary, Insertion, and Merge sort

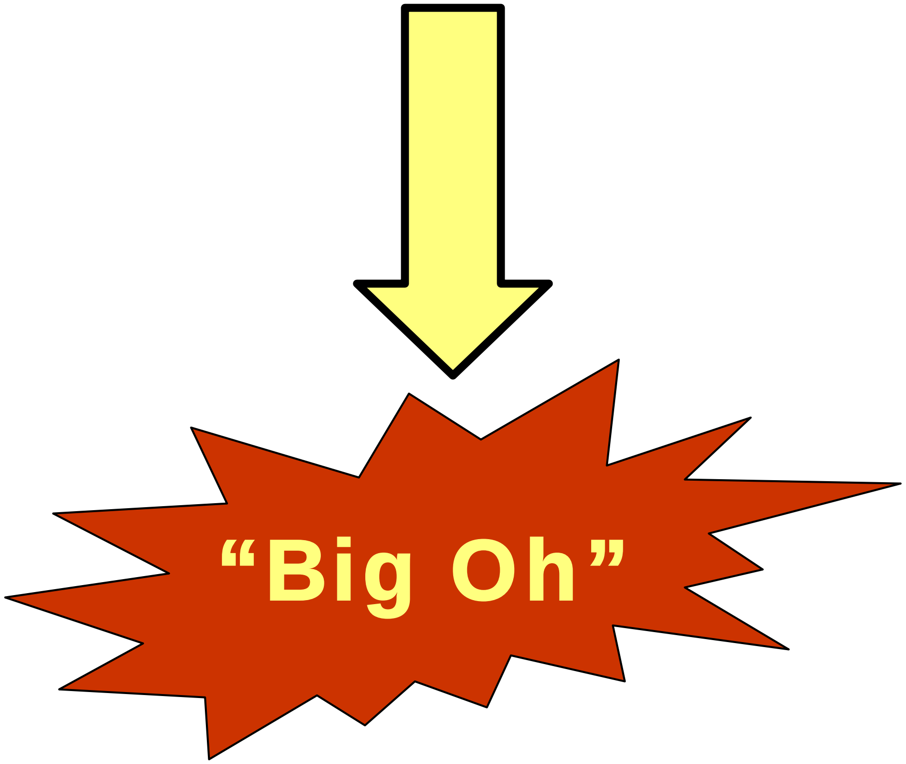
Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT

Complexity Analysis

- Complexity: rate at which storage or time grows as a function of the problem size
  - Growth depends on compiler, machine, …
- Asymptotic analysis
  - Describing the inherent complexity of a program, independent of machine and compiler.
  - A “proportionality” approach, expressing the complexity in terms of its relationship to some known function.

Asymptotic Analysis

- **Idea**: as problem size grows, the complexity can be described as a simple proportionality to some known function.



Asymptotic Analysis: Big-oh

Definition: **T**(n) = O(f(n)) -- *T of n is in Big Oh of f of n*
iff there are constants c and n_0 such that
T(n) ≤ cf(n) for all n ≥ n_0

Usage: The algorithm is in O(n^2) in [best, average, worst] case.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than $cf(n)$ steps in [best, average, worst] case.

Big oh is said to describe an “upper bound” on the complexity.

Big-oh Notation (cont)

Big-oh notation indicates an **upper bound** on the complexity

Example: If $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in $O(n^2)$.

Wish tightest upper bound:

While $\mathbf{T}(n) = 3n^2$ is in $O(n^{25})$, we prefer $O(n^2)$.

- $T(n) = O(1)$ constant growth
  - E.g., array access
- $T(n) = O(lg(n))$ logarithmic growth
  - E.g., binary search
- $T(n) = O(n)$ linear growth
  - E.g., looping over all elements in a 1-dimensional array
- $T(n) = O(n \log n)$ “n log n” growth
  - E.g., Merge sort
- $T(n) = O(n^k)$ polynomial growth
  - E.g., Selection sort is n^2, k seldom larger than 5
- $T(n) = O(2^n)$ exponential growth
  - E.g., basically useless for anything but very small problems

1 billion instructions/second

n	T(n)= n	T(n)= n lg(n)	T(n)= n^2	T(n)= n^3	T(n)= 2^n
5	0.005 μs	0.01 μs	0.03 μs	0.13 μs	0.03 μs
10	0.1 μs	0.03 μs	0.1 μs	1 μs	1 μs
20	0.02 μs	0.09 μs	0.4 μs	8 μs	1ms
50	0.05 μs	0.28 μs	2.5 μs	125 μs	13 days
100	0.1 μs	0.66 μs	10 μs	1 ms	$4x10^{13}$ years

Big-Oh Examples

Example 1: Finding value X in an array (average cost).

$\mathbf{T}(n) = c_s n/2$.

For all values of $n > 1$, $c_s n/2 <= c_s n$.

Therefore, by the definition, $\mathbf{T}(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Big-Oh Examples

Example 2: $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in average case.

$c_1 n^2 + c_2 n <= c_1 n^2 + c_2 n^2 <= (c_1 + c_2) n^2$ for all $n > 1$.

$\mathbf{T}(n) <= cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the definition.

Example 3: $\mathbf{T}(n) = c$. We say this is in $O(1)$.

Does Big-oh tell the whole story?

- $T_1(n) = T_2(n) = O(g(n))$
- $T_1(n) = 50 + 3n + (10 + 5 + 15)n = 50 + 33n$
  - ```
    Setup of algorithm              -- takes 50 time units
    read n elements into array       -- 3 units/element
    for i 1..n
           do operation1 on A[i]     -- takes 10 units
           do operation2 on A[i]     -- takes  5 units
           do operation3 on A[i]     -- takes 15 units
    ```
- $T_2(n) = 200 + 3n + (10 + 5)n = 200 + 18n$
  - ```
    Setup of algorithm             -- takes 200 time units
    read n elements into array       -- 3 units/element
    for i 1..n
           do operation1 on A[i]     -- takes 10 units
           do operation2 on A[i]     -- takes  5 units
    ```

Big-Omega

Definition: For **T**(n) a non-negatively valued function, **T**(n) is in the set Ω(g(n)) if there exist two positive constants c and n_0 such that **T**(n) >= $cg(n)$ for all $n > n_0$.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than $cg(n)$ steps.

Lower bound.

Big-Omega Example

T(n) = $c_1n^2 + c_2n$.

$c_1n^2 + c_2n >= c_1n^2$ for all $n > 1$.

T(n) >= cn^2 for $c = c_1$ and $n_0 = 1$.

Therefore, **T**(n) is in $\Omega(n^2)$ by the definition.

We want the greatest lower bound.

Theta Notation

When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

Definition: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

Tight bound.

A Common Misunderstanding

Confusing worst case with upper bound.

Upper bound refers to a growth rate.

Worst case refers to the worst input from among the choices for possible inputs of a given size.

Simplifying Rules

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.

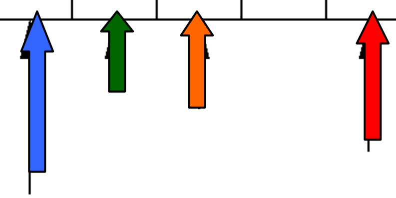
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.

3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.

4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Binary Search

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83



How many elements are examined in worst case?

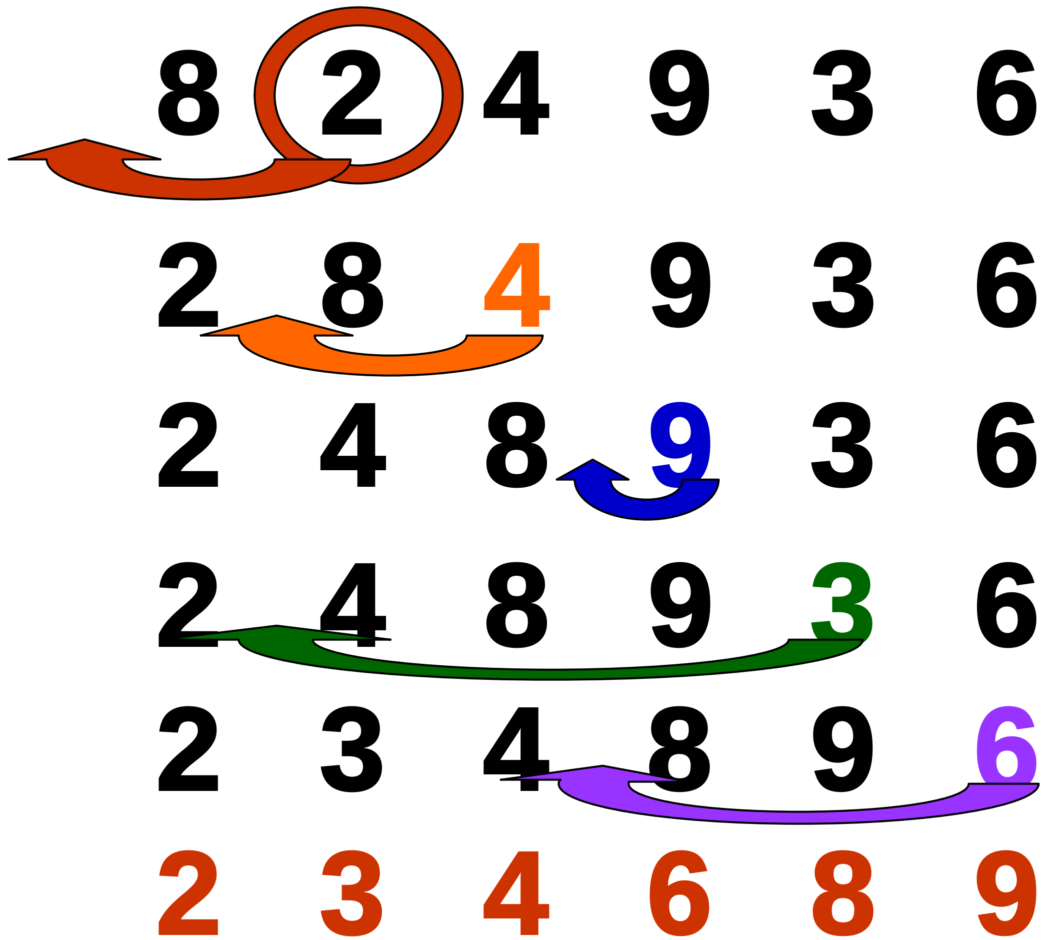
Binary Search

```
Procedure Binary_Search (Input_Array, Number_To_Search,
                                          Return_Index);

Begin
   Set Return_Index to –1;
   Set Current_Index to (Upper_Bound - Lower_Bound + 1) /2.

      Loop
         if the lower_bound > upper_bound
                Exit;
          end if
         if ( Input_Array(Current_Index) = Number_to_Search)
            then
              Return_Index := Current_Index
              Exit;
          end if
         if ( Input_Array(Current_Index) < Number_to_Search)
           then
              Lower_Bound := Current_Index +1
             else
              Upper_Bound := Current_Index – 1
         end if
     end loop
end Binary_Search;
```

Insertion sort

```
InsertionSort(A, n)              -- A[1..n]
  for j 2..n
    do key := A[j]
       i := j-1
       while i > 0 and A[i] > key
         A[i+1] := A[i]
         i := i-1
       A[i+1]:= key
```

Insertion sort



Insertion sort

- Running time
  - Depends on the input
    - An already sorted sequence is easier to sort
  - Worst case: input reverse sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

  - Average case: all permutations equally likely

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Merge sort

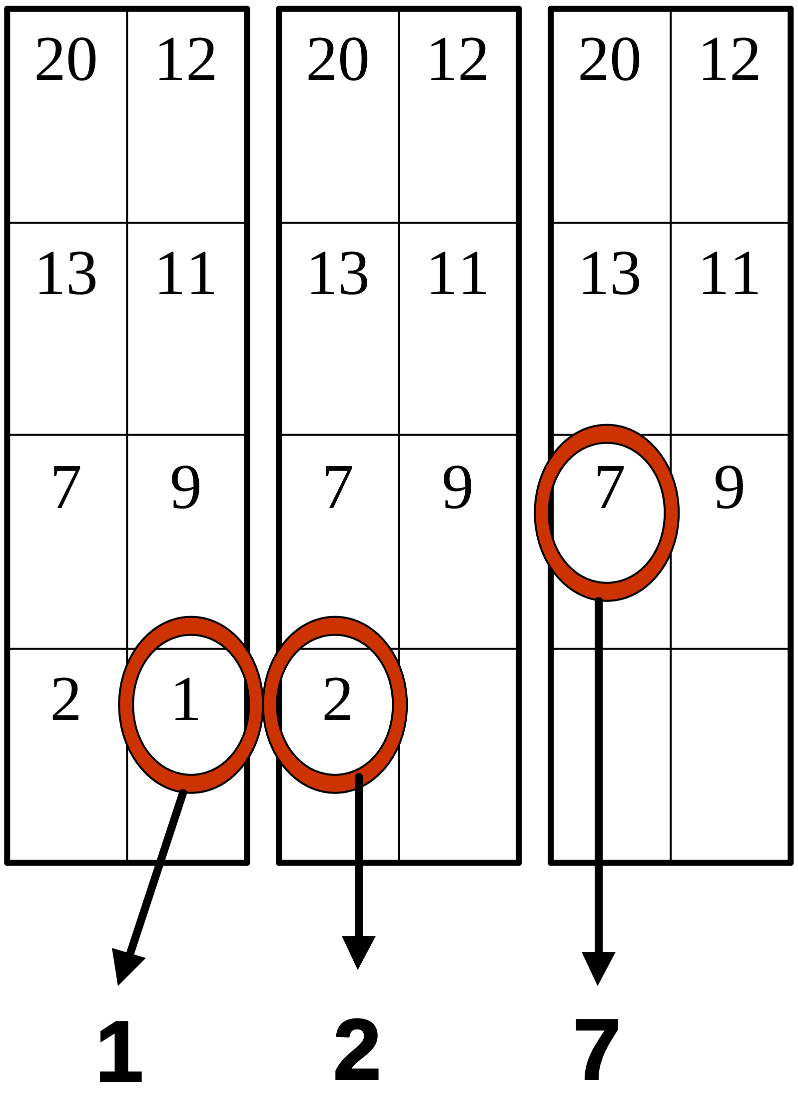
- MergeSort A[1..n]
  1. If the input sequence has only one element, return

  2. Partition the input sequence into two halves

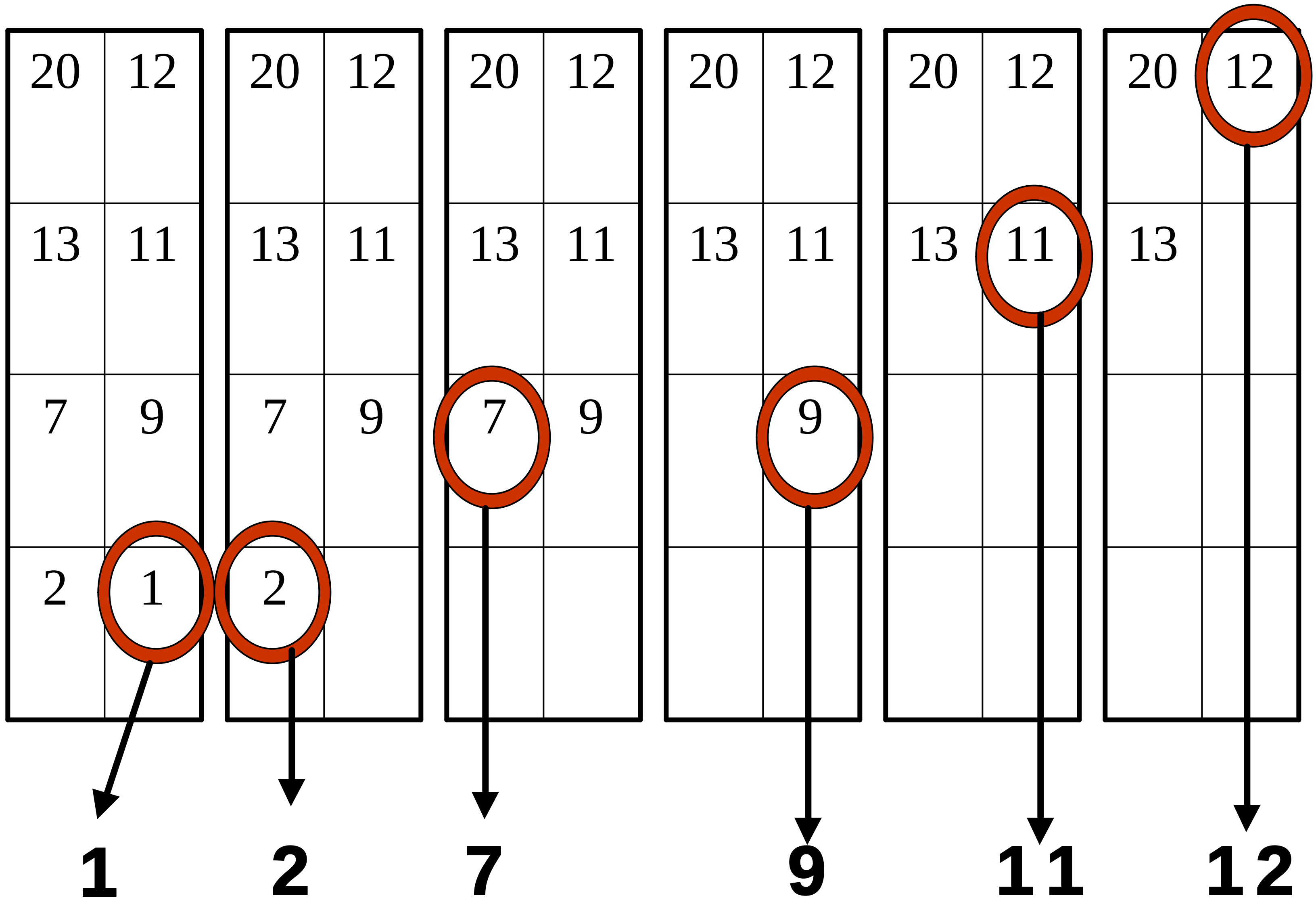
  3. Sort the two subsequences using the same algorithm

  4. Merge the two sorted subsequences to form the output sequence

Merge sort



Merge sort



Merge sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1 \\ 2T(n/2) + \Theta(n) \text{ if } n > 1 \end{cases}$$

...

$$\text{Total} = \Theta(n \lg n)$$

Space Bounds

- Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm

Space: Data Structure

Space/Time Tradeoff Principle

- One can often reduce time if one is willing to sacrifice space, or vice versa.
  - Encoding or packing information
    - Boolean flags
  - Table lookup
    - Factorial
- Disk-based Space/Time Tradeoff Principle: The smaller you make the disk storage requirements, the faster your program will run.