Introduction to Computers and Programming

Prof. I. K. Lundqvist

Lecture 10
April 8 2004

Today

- How to determine Big-O
- Compare data structures and algorithms
- Sorting algorithms

How to determine Big-O

- Partition algorithm into known pieces
- Identify relationship between pieces
  - Sequential code (+)
  - Nested code (*)
- Drop constants
- Only keep the most dominant factors

Does Big-O tell the whole story?

- $T_x(n) = T_y(n) = O(\lg n)$
- $T_1(n)=50+3n+(10+5+15)n = 50+33n$

```
setup of algorithm            -- takes 50 time units
read n elements into array    -- 3 units/element
for i in 1..n loop
       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 in 1..n loop
       do operation1 on A[i]   -- takes 10 units
       do operation2 on A[i]   -- takes  5 units
```

Data structure	Traversal	Search	Insert
Unsorted L List	N		
Sorted L List	N		
Unsorted Array	N		
Sorted Array	N		
Binary Tree	N		
BST	N		
F&B BST	N		

Searching

- **Linear** (sequential) search
  - Checks every element of a list until a match is found
  - Can be used to search an unordered list
- **Binary** search
  - Searches a set of **sorted** data for a particular data
  - Considerable faster than a linear search
  - Can be implemented using recursion or iteration

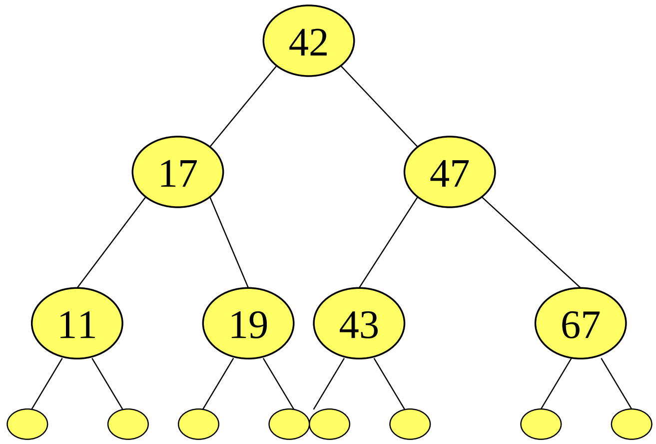
Linear Search

- If data distributed randomly
  - **Average** case:
    - N/2 comparisons needed
  - **Best** case:
    - values is equal to first element tested
  - **Worst** case:
    - value not in list → N comparisons needed

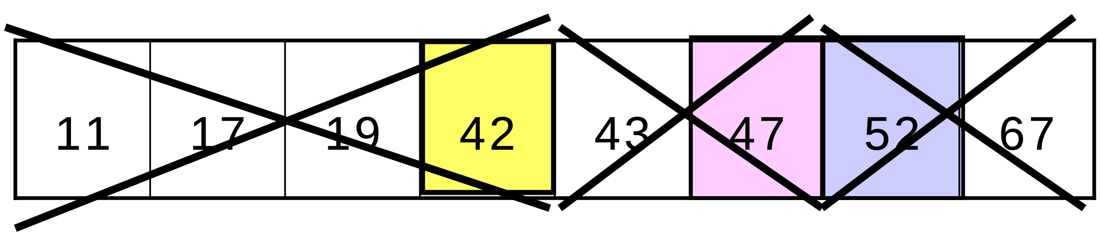
Linear search is O(N)

Data structure	Traversal	Search	Insert
Unsorted L List	N	N	
Sorted L List	N	N	
Unsorted Array	N	N	
Sorted Array	N		
Binary Tree	N	N	
BST	N	N	
F&B BST	N		

Full and Balanced Binary Search Tree



Binary Search



50 not found

3 comparisons

3 = Log (8)

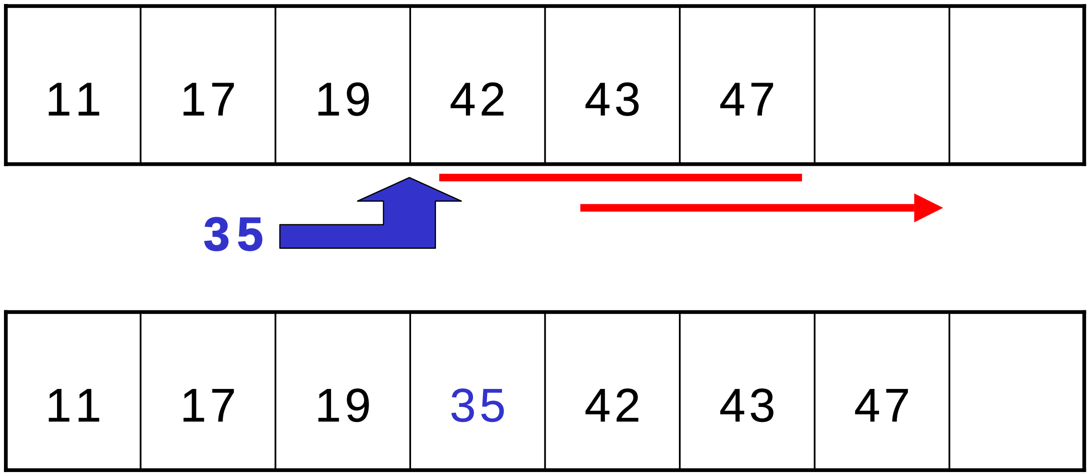
Binary Search

- Can be performed on
  - Sorted arrays
  - Full and balanced BSTs
- Compares and cuts half the work
  - We cut work in ½ each time
  - How many times can we cut in half?

Binary search is **O(Log N)**

Data structure	Traversal	Search	Insert
Unsorted L List	N	N	PSET
Sorted L List	N	N	PSET
Unsorted Array	N	N	1
Sorted Array	N	Log N	
Binary Tree	N	N	1
BST	N	N	
F&B BST	N	Log N	

Insertion/Shuffling Elements



Shuffle is **O(N)**

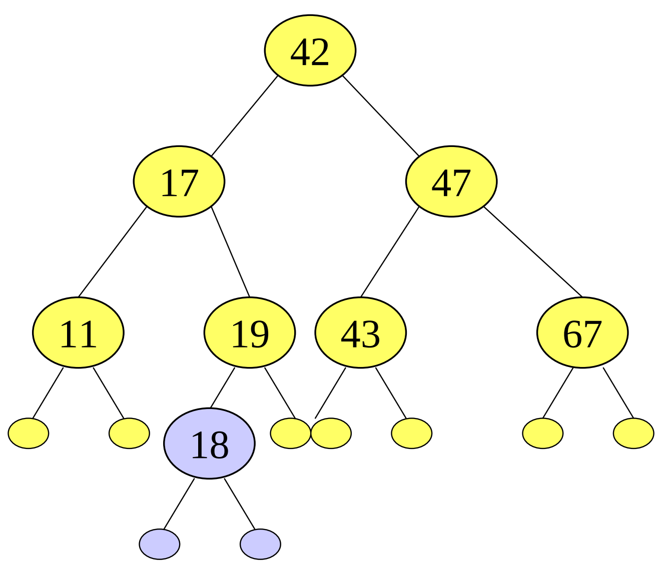
Insertion to a Sorted Array

- Sorted Array
  - Finding the right spot – O(Log N)
  - Performing the shuffle – O(N)
  - Performing the insertion - O(1)

  - Total work: O(Log N + N + 1) = O(N)

Data structure	Traversal	Search	Insert
Unsorted L List	N	N	PSET
Sorted L List	N	N	PSET
Unsorted Array	N	N	1
Sorted Array	N	Log N	N
Binary Tree	N	N	1
BST	N	N	
F&B BST	N	Log N	

Insertion into a F&B BST



Insertion into a F&B BST

- Finding the right spot – O(Log N)
- Performing the insertion – O(1)

- Total work: O(Log N + 1) = O(Log N)

Data structure	Traversal	Search	Insert
Unsorted L List	N	N	PSET
Sorted L List	N	N	PSET
Unsorted Array	N	N	1
Sorted Array	N	Log N	N
Binary Tree	N	N	1
BST	N	N	N
F&B BST	N	Log N	Log N

Sorting Algorithms

In the Worst Case

- **Insertion** sort
- Bubble sort
- Selection sort
- ...

} $O(N^2)$ or worse

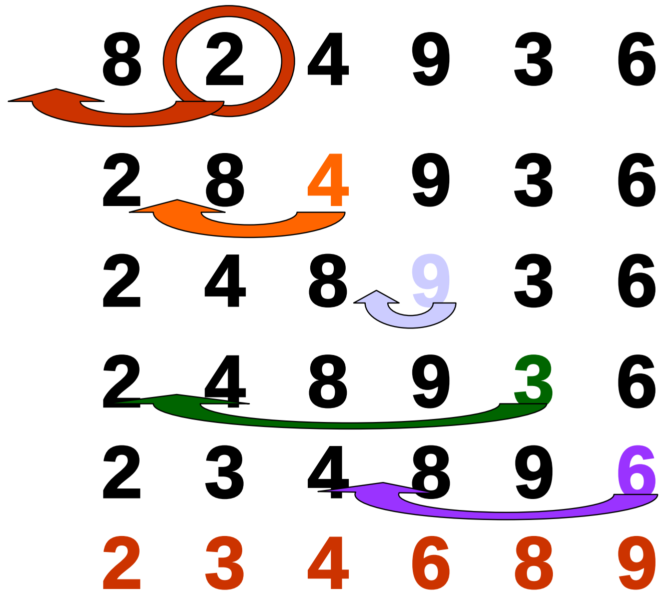
- **Merge** sort
- Heap sort
- Quick sort
- ...

} O(N Log N) or better

Insertion Sort

- Sorted array/list is built one item at a time
  - Simple to implement
  - Efficient on small data sets
  - Efficient on already almost ordered data sets
  - Minimal memory requirements

Insertion Sort



Insertion Sort

Statement	Work
`InsertionSort(A, n)`	$T(n)$
for `j` **in** `2..n` **do**	$c_1 n$
`key:= A[j]`	$c_2(n-1)$
`i  := j-1`	$c_3(n-1)$
while `i > 0` **and** `A[i] > key`	$c_4 X$
`A[i+1]:= A[i]`	$c_5(X-(n-1))$
`i:= i-1`	$c_6(X-(n-1))$
`A[i+1]:= key`	$c_7(n-1)$

$X = x_2 + x_3 + \ldots + x_n$ where x_i is number of while expression evaluations for the i^{th} for loop iteration

Insertion Sort Analysis

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 X + c_5(X - (n-1)) + c_6(X - (n-1)) + c_7(n-1)$$

$$= c_8 X + c_9 n + c_{10}$$

Running time

- **Best** case:
  - inner loop never executed - Linear Function
- **Worst** case:
  - inner loop always executed - X is a quadratic function in n
- **Average** case:
  - all permutations equally likely

Insertion Sort – $O(N^2)$

- Assume you are sorting 250,000,000 item

  $N = 250{,}000{,}000$ $N^2 = 6.25 * 10^{16}$

  Assume you can do 1 operation/nanosecond

  → $6.25 * 10^7$ seconds

  = **1.98 years**

Merge Sort

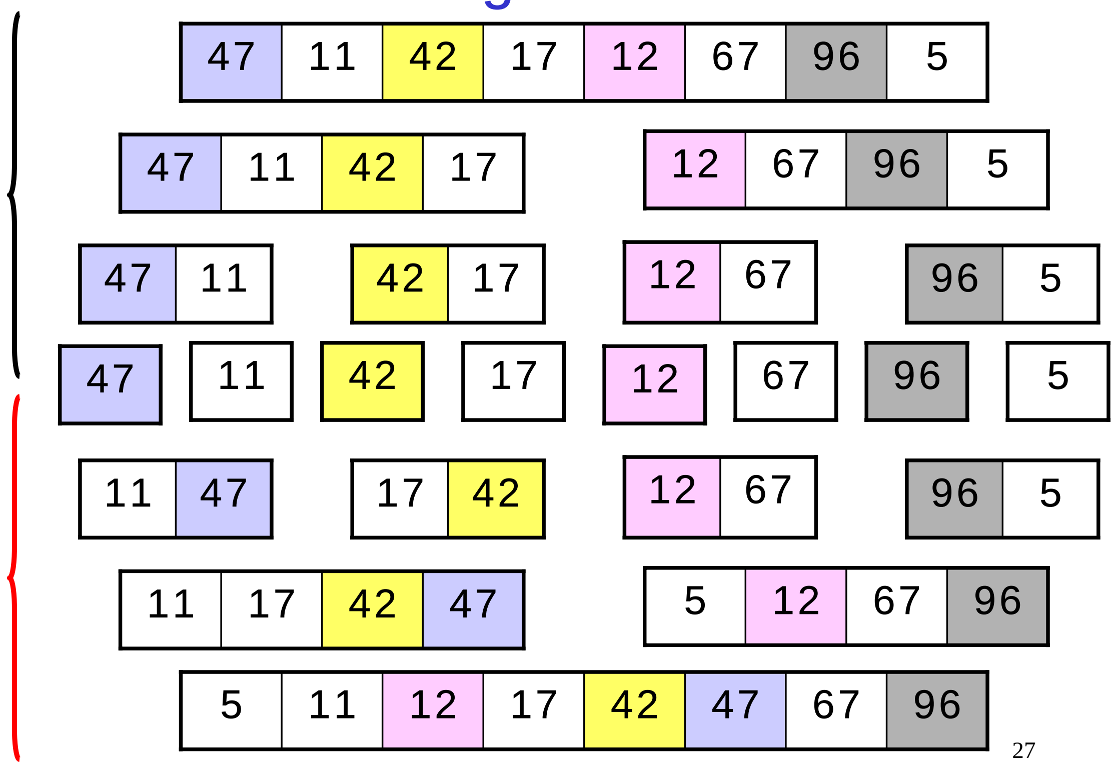
MergeSort A[1..n]

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

Divide and Conquer

- It is an algorithmic design paradigm that contains the following steps
  – **Divide**: Break the problem into smaller sub-problems
  – **Recur**: Solve each of the sub-problems recursively
  – **Conquer**: Combine the solutions of each of the sub-problems to form the solution of the problem

Merge Sort



Merge Sort – O(N * Log N)

- Assume you are sorting 250,000,000 item

  N = 250,000,000
  N*Log N = 250,000,000 * 28

  Assume you can do 1 operation/nanosecond

  → **7.25 seconds**

Merge Sort Analysis

Statement	Work
`MergeSort(A, left, right)`	T(n)
`   if (left < right)`	O(1)
`      mid := (left + right) / 2;`	O(1)
`      MergeSort(A, left, mid);`	T(n/2)
`      MergeSort(A, mid+1, right);`	T(n/2)
`      Merge(A, left, mid, right);`	O(n)

$$T(n) = \begin{cases} O(1) & \text{when } n = 1, \\ 2T(n/2) + O(n) & \text{when } n > 1 \end{cases}$$

Recurrence Equation