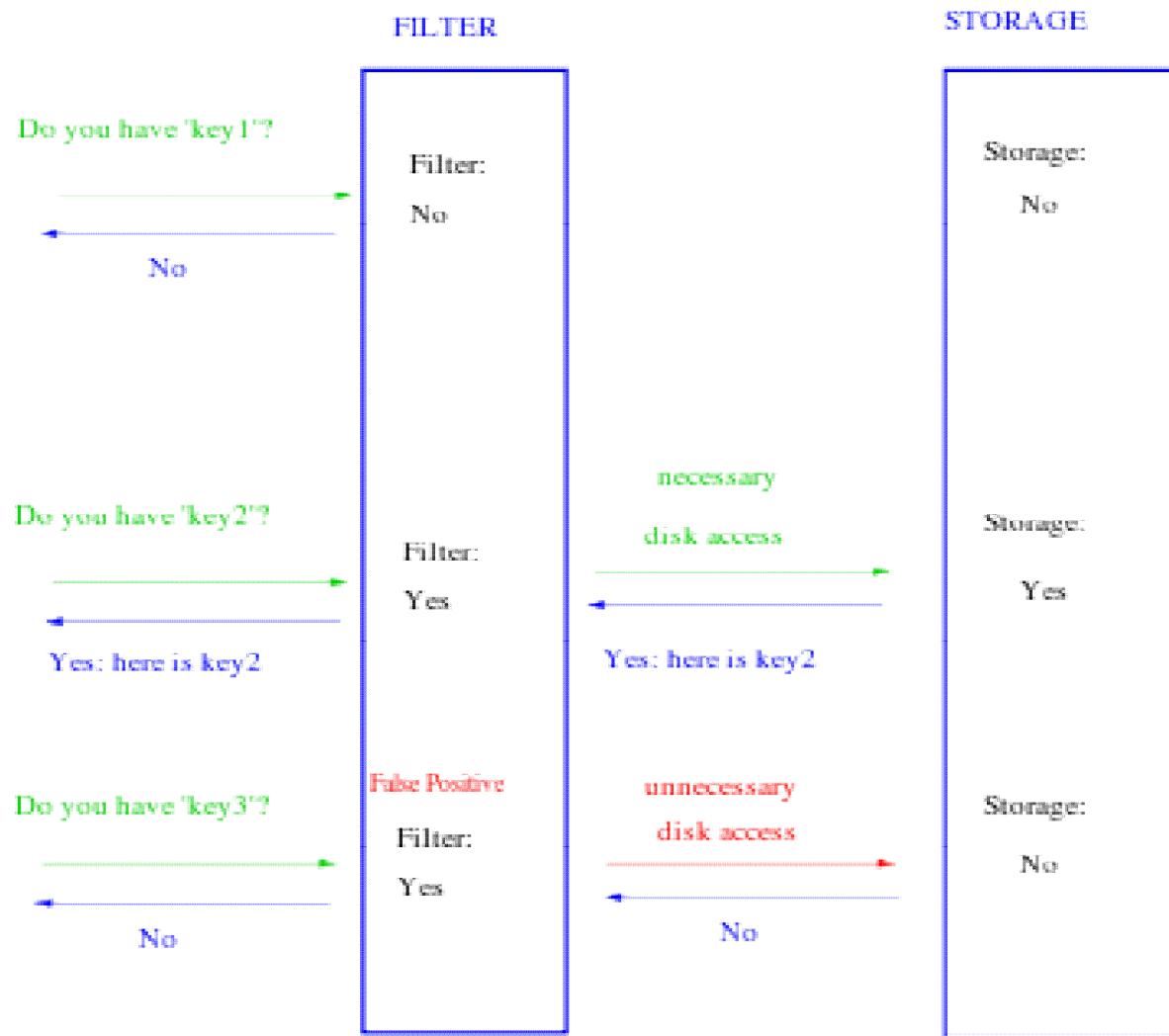


# Bloom filter

- A **Bloom filter** (1970) is a space-efficient probabilistic data structure to test whether an element is a member of a set:
  - Yes, maybe in the set.
    - False positive: could be mistaken.
    - Second slower lookup can correct this.
  - No, not in the set, 100% sure.
- Like a hash-table, with no deletion.
- Uses: blacklists in firewalls, do not fly lists.

# Bloom filter with 2<sup>nd</sup> lookup



# Bloom filter

1. Bloom filter used to speed up answers in a key-value storage system: databases, webservers
2. Values are stored on a disk which has slow access times.
3. Wastage of disk accesses on a false positive report.
4. Overall are much faster.
5. Need more memory.

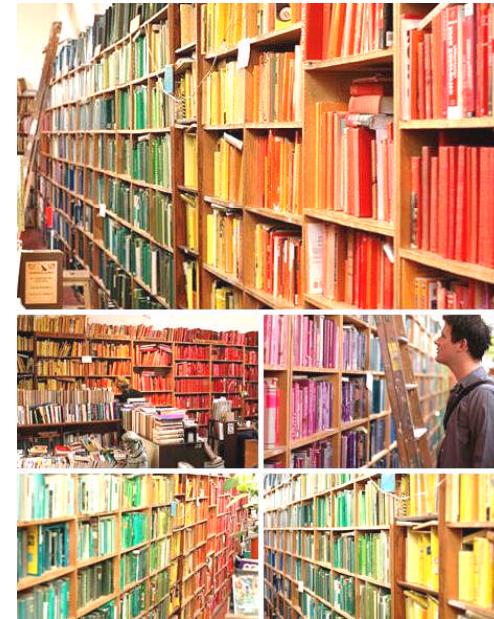
# Sorting

# Sorting

- 1. Input:** a list of  $n$  elements
- 2. Output:** the same list of  $n$  elements rearranges in ascending (non decreasing) or decending (non increasing) order.

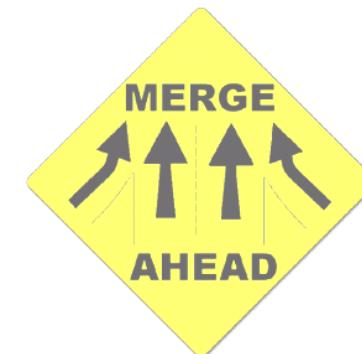
# Why sort?

1. Sorting is used by many applications
2. Sorting is initial step of many algorithms
3. Many techniques can be illustrated by studying sorting.
4. Sorted items are easier to search.



# Sorting

1. Selection sort
2. Insertion sort
3. Bubble sort
4. Merge sort
5. Heap sort
6. Quick sort
7. Radix sort



# Ideal sorting algorithm

1. Stable: Equal keys aren't reordered.
2. Operates in place, requiring  $O(1)$  extra space.
3. Worst-case  $O(n \cdot \log(n))$  key comparisons.
4. Worst-case  $O(n)$  swaps.
5. Adaptive: Speeds up to  $O(n)$  when data is nearly sorted or when there are few unique keys.

There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.



# Which is the best sort?

1. See how each algorithm operates.
2. There is no best sorting algorithm for all inputs.
3. Advantages and disadvantages of each algorithm.
4. Worse-case asymptotic behavior is not always the deciding factor in choosing an algorithm.
5. Show that the initial condition (input order and key distribution) affects performance as much as the algorithm choice.

# Comparison of running times

1. Searches
  1. Linear:  $n$  steps
  2. Binary:  $\log_2 n$  steps
  3. Binary search is about as fast as you can get
2. Sorts
  1. Bubble:  $n^2$  steps
  2. Insertion:  $n^2$  steps
3. There are other, more efficient, sorting techniques
  1. The fastest are heapsort, quicksort, and mergesort
  2. These each take  $n * \log_2 n$  steps
  3. In practice, quicksort is the fastest, followed by merge sort. qsort is part of standard C.

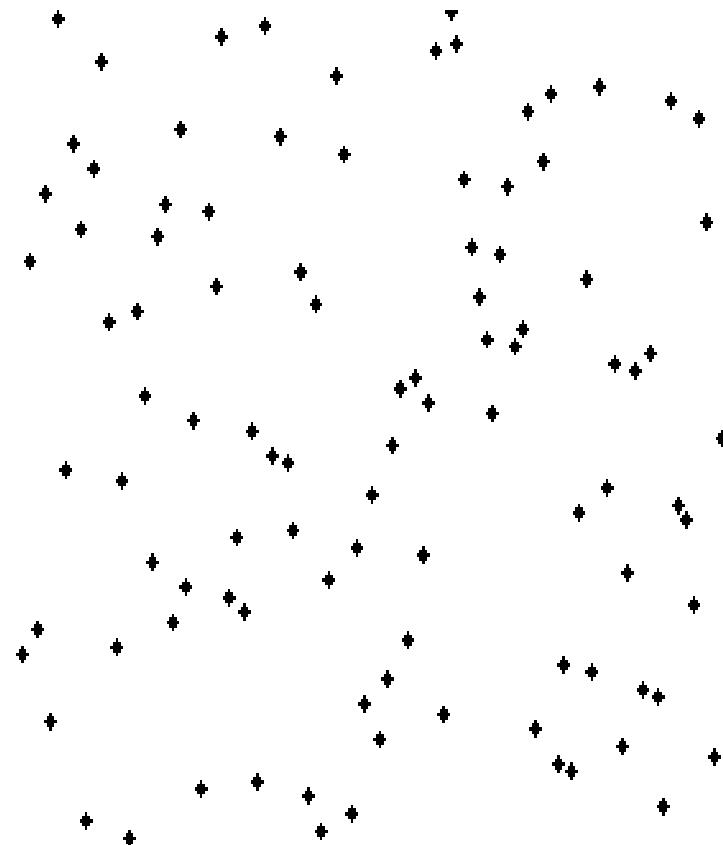
# Bubble sort

- A **bubble sort**, a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order.
- Note that the largest end gets sorted first, with smaller elements taking longer to move to their correct positions.

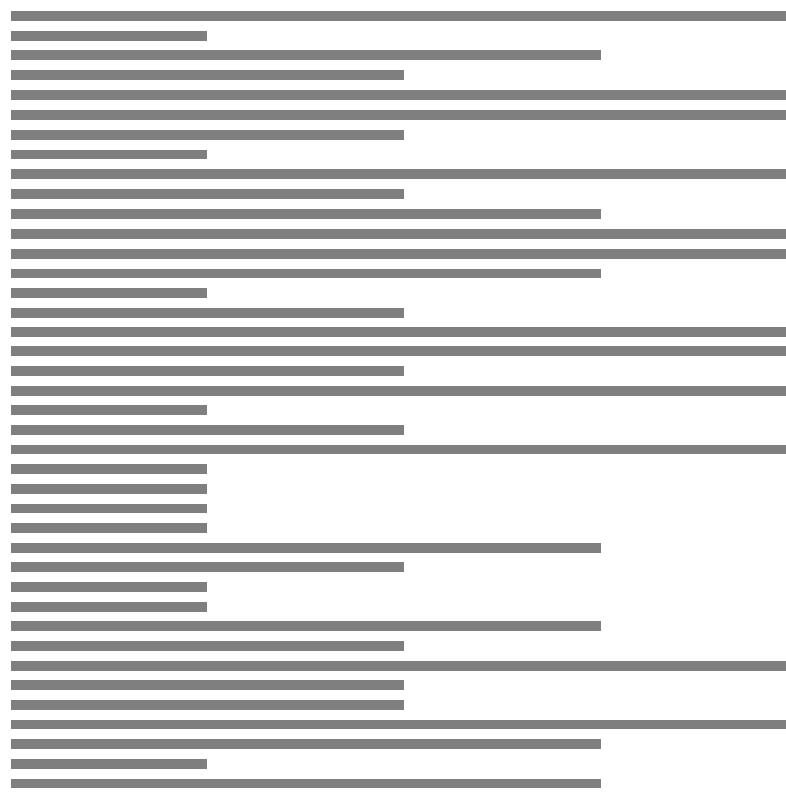
# Bubble sort complexity

- **Bubble sort** has worst-case and average complexity both  $O(n^2)$ , where  $n$  is the number of items being sorted.
- There exist many sorting algorithms with substantially better worst-case or average complexity of  $O(n \log n)$ .
- Even other  $O(n^2)$  sorting algorithms, such as insertion sort, tend to have better performance than **bubble sort**.
- Therefore, **bubble sort** is not a practical sorting algorithm when  $n$  is large.

# Bubble sort animation



# Bubble sort animation



# Bubble sort

- Bubble sort has many of the same properties as insertion sort, but has slightly higher overhead.
- In the case of nearly sorted data, bubble sort takes  $O(n)$  time, but requires at least 2 passes through the data (whereas insertion sort requires something more like 1 pass).
- \* Stable
- \*  $O(1)$  extra space
- \*  $O(n^2)$  comparisons and swaps
- \* Adaptive:  $O(n)$  when nearly sorted
- \* Ability to detect that the list is sorted

# Bubble sort

1. procedure bubbleSort( A : list of sortable items )
2.   repeat
3.     swapped = false
4.     for i = 1 to length(A) - 1 inclusive do:
5.       if A[i-1] > A[i] then
6.         swap( A[i-1], A[i] );
7.         swapped = true
8.     end if
9.     end for
10.  until not swapped
11. end procedure

# Bubble sorting example animation

6 5 3 1 8 7 2 4

# **INSERTION SORT**

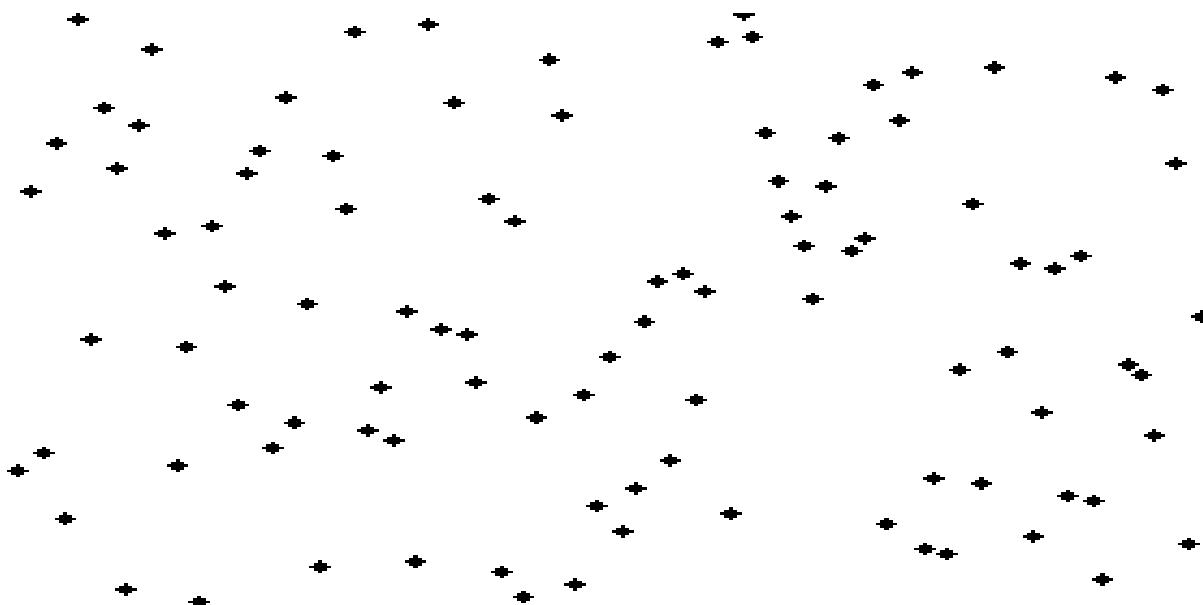
- Insertion sort keeps making the **left side** of the array sorted until the whole array is sorted.
- An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place.

- Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.
- Sorting is typically done in-place. The resulting array after  $k$  iterations has the property where the first  $k + 1$  entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result:

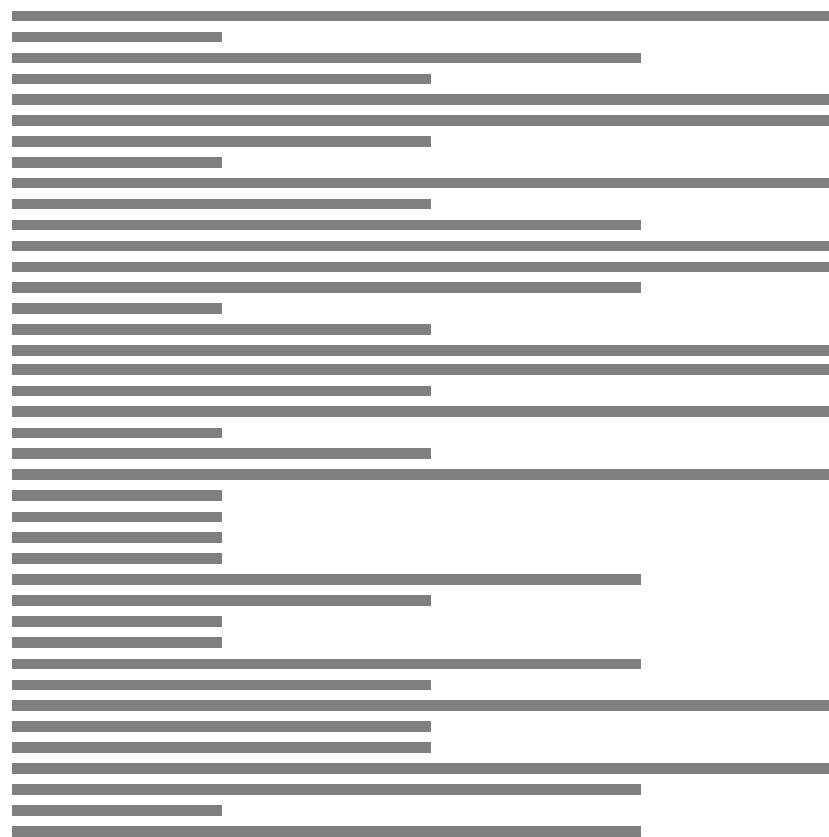
Sorted partial result		Unsorted data	
$\leq x$	$> x$	$x$	...

Sorted partial result		Unsorted data	
$\leq x$	$x$	$> x$	...

# Insertion sort animation



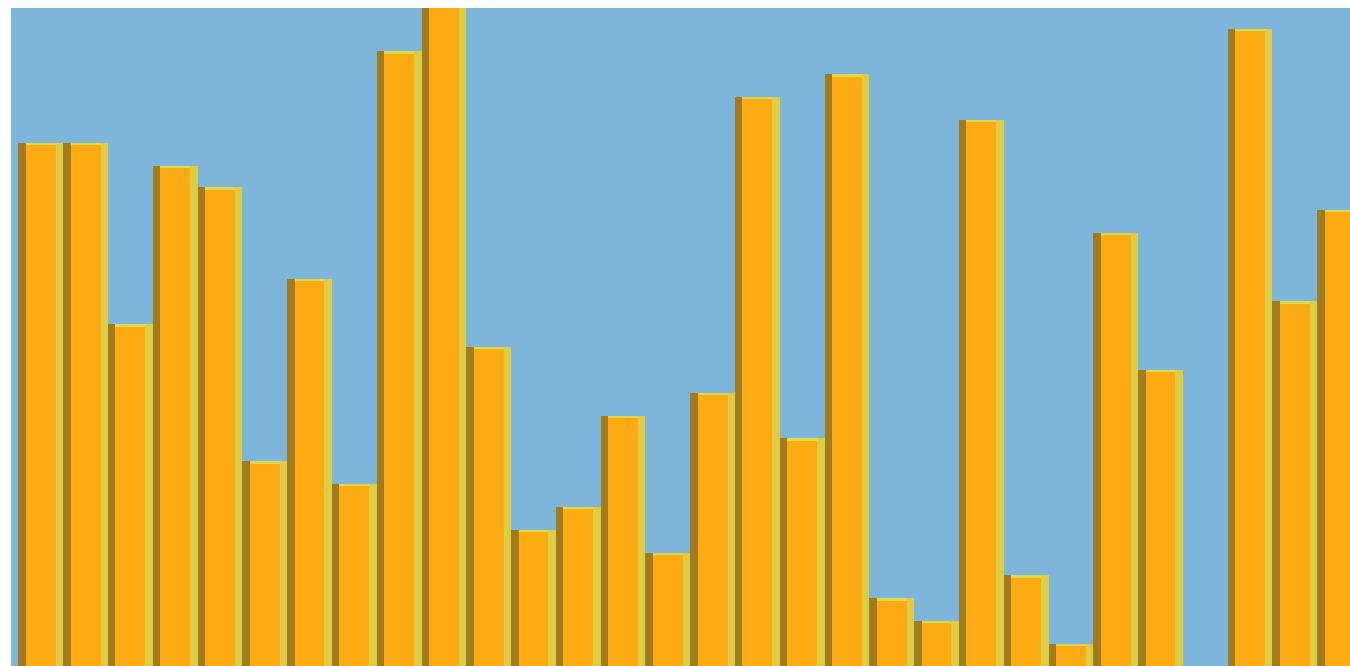
# Insertion sort animation



# Insertion sort

```
void insertSort(int a[], int length) {  
    int i, j, value;  
    for(i = 1; i < length; i++) {  
        value = a[i];  
        // find place for value  
        for (j = i - 1;  
             j >= 0 && a[j] > value;  
             j--) {  
            a[j + 1] = a[j]; // move up  
        }  
        a[j + 1] = value;  
    }  
}
```

Animation, each element left to right is put in its place



# Insertion sorting animation

6 5 3 1 8 7 2 4

# Properties of Insertion sort

- Stable
- $O(1)$  extra space
- $O(n^2)$  comparisons and swaps
- Adaptive:  $O(n)$  time when nearly sorted
- Very low overhead

# Insertion sort

- $A[i]$  is inserted in its proper position in the  $i$ th iteration in the sorted subarray  $A[1 \dots i-1]$
- In the  $i$ -th step, the elements from index  $i-1$  down to 1 are scanned, each time comparing  $A[i]$  with the element at the correct position.
- In each iteration an element is shifted one position up to a higher index.
- The process of comparison and shifting continues until:
  - Either an **element  $\leq A[i]$**  is found or
  - When all the sorted sequence so far is scanned.
- Then  $A[i]$  is inserted in its proper position.

# Insertion sort

- Although it is one of the elementary sorting algorithms with  $O(n^2)$  worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).
- For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

# Analysis

- Let  $a_0, \dots, a_{n-1}$  be the sequence to be sorted. At the beginning and after each iteration of the algorithm the sequence consists of two parts:
  - first part  $a_0, \dots, a_{i-1}$  is already sorted,
  - second part  $a_i, \dots, a_{n-1}$  is still unsorted ( $i$  in  $0, \dots, n$ ).
- **worst case** occurs when in every step the proper position for the element that is inserted is found at the beginning of the sorted part of the sequence.

The minimum # of element comparisons (best case) occurs when the array is already sorted in non-decreasing order.

In this case, the # of element comparisons is exactly  $n - 1$ , as each element  $A[i]$ ,  $2 \leq i \leq n$ , is compared with  $A[i - 1]$  only.

The maximum # of element comparisons (Worst case) occurs if the array is already sorted in decreasing order and all elements are distinct. In this case, the number is

$$\sum_{i=2}^n (i - 1) = \sum_{i=1}^{n-1} (i - 1) = n(n-1)/2$$

This is because each element  $A[i]$ ,  $2 \leq i \leq n$  is compared with each entry in subarray  $A[1 .. i-1]$

**→ Pros:** Relatively simple and easy to implement.  
**Cons:** Inefficient for large lists.

- **Best case:**  $O(n)$ . It occurs when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.
- Average case:  $\theta(n^2)$  since there is a wide variation with the running time.
- **Worst case:**  $O(n^2)$  if the numbers were sorted in reverse order.

# Selection sort

- How does it work:
  - first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.
- *How does it sort the list in a non-increasing order?*
- Selection sort is:
  - The simplest sorting techniques.
  - a good algorithm to sort a small number of elements
  - an incremental algorithm – induction method
- Selection sort is Inefficient for large lists.
- Its runtime is always quadratic

***Incremental algorithms*** → process the input elements one-by-one and maintain the solution for the elements processed so far.

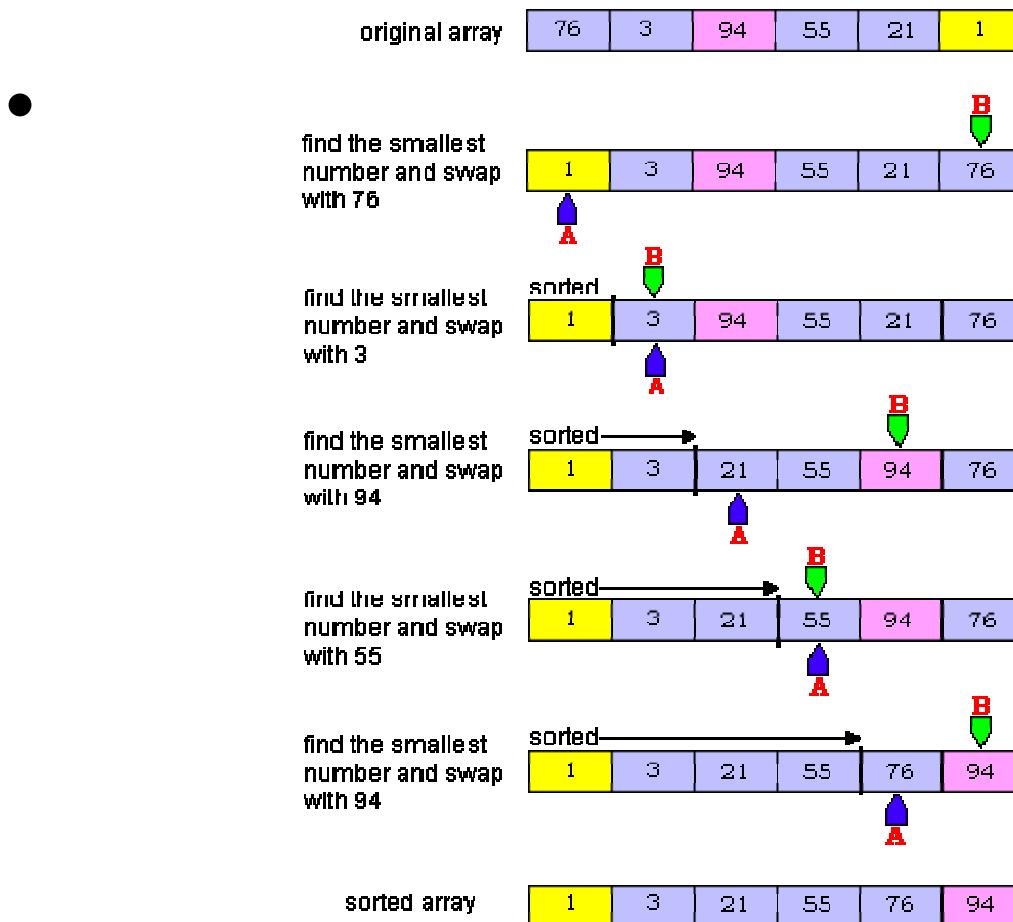
# Selection Sort Algorithm

**Input:** An array  $A[1..n]$  of  $n$  elements.

**Output:**  $A[1..n]$  sorted in nondecreasing order.

```
1. for  $i \leftarrow 1$  to  $n - 1$ 
2.    $k \leftarrow i$ 
3.   for  $j \leftarrow i + 1$  to  $n$  {Find the  $i$  th smallest element.}
4.     if  $A[j] < A[k]$  then  $k \leftarrow j$ 
5.   end for
6.   if  $k \neq i$  then interchange  $A[i]$  and  $A[k]$ 
7. end for
```

# Example



# Analysis of Algorithms

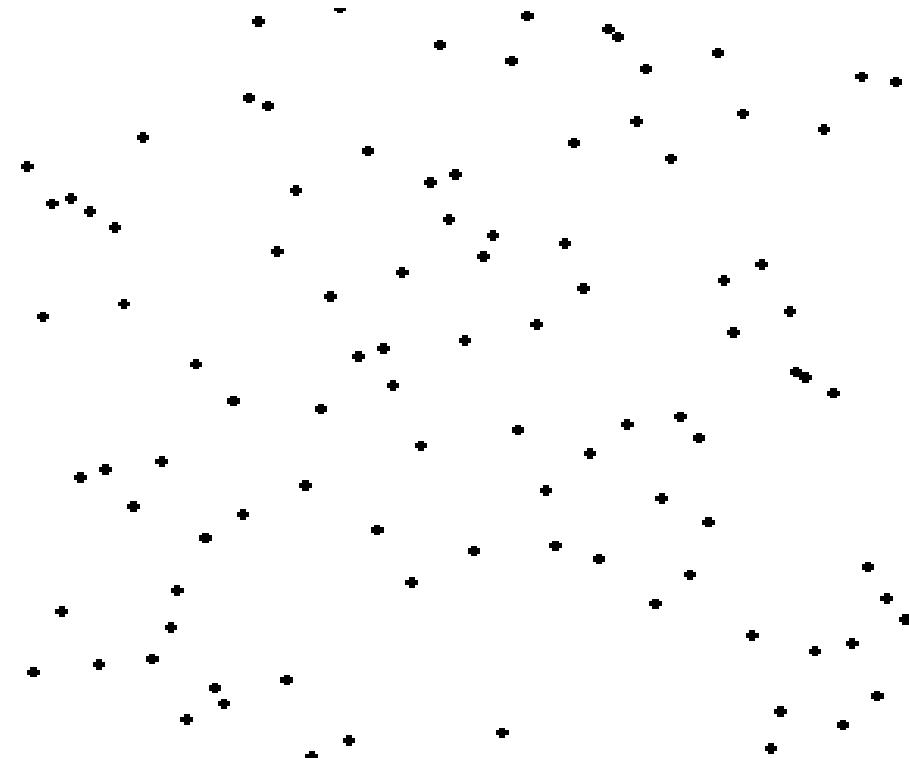
- $T(n)$  is the total number of accesses made from the beginning of selection\_sort until the end.
- selection\_sort itself simply calls swap and find\_min\_index as  $i$  goes from 1 to  $n-1$

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \text{find-min-element} + \text{swap} \\ &= n-1 + n-2 + n-3 + \dots + 1 = n(n-1)/2 \\ \text{Or } &= \sum (n - i) = n(n - 1) / 2 \rightarrow O(n^2) \end{aligned}$$

# Properties

- Not stable
- $O(1)$  extra space
- $\Theta(n^2)$  comparisons
- $\Theta(n)$  swaps
- Not adaptive

# Selection sort animation



# Selection sort animation



# Merge sort

- Merge sort (also commonly spelled mergesort) is an  $O(n \log n)$  comparison-based sorting algorithm.
- Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output.
- Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945

# How it works

- Conceptually, a merge sort works as follows
- 1. Divide the unsorted list into  $n$  sublists, each containing 1 element (a list of 1 element is considered sorted).
- 2. Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

# Merge sort animated example

6 5 3 1 8 7 2 4

# Algorithm: MERGE

```
// from wikipedia
function merge(left, right)
    var list result
    while length(left) > 0 or length(right) > 0
        if length(left) > 0 and length(right) > 0
            if first(left) <= first(right)
                append first(left) to result
                left = rest(left)
            else
                append first(right) to result
                right = rest(right)
            else if length(left) > 0
                append first(left) to result
                left = rest(left)
            else if length(right) > 0
                append first(right) to result
                right = rest(right)
        end while
    return result
```

# Merge Sort

divide-and-conquer

- **Divide:** divide the  $n$ -element sequence into two subproblems of  $n/2$  elements each.
- **Conquer:** sort the two subsequences recursively using merge sort. If the length of a sequence is 1, do nothing since it is already in order.
- **Combine:** merge the two sorted subsequences to produce the sorted answer.

## MERGE-SORT(A, $p,r$ )

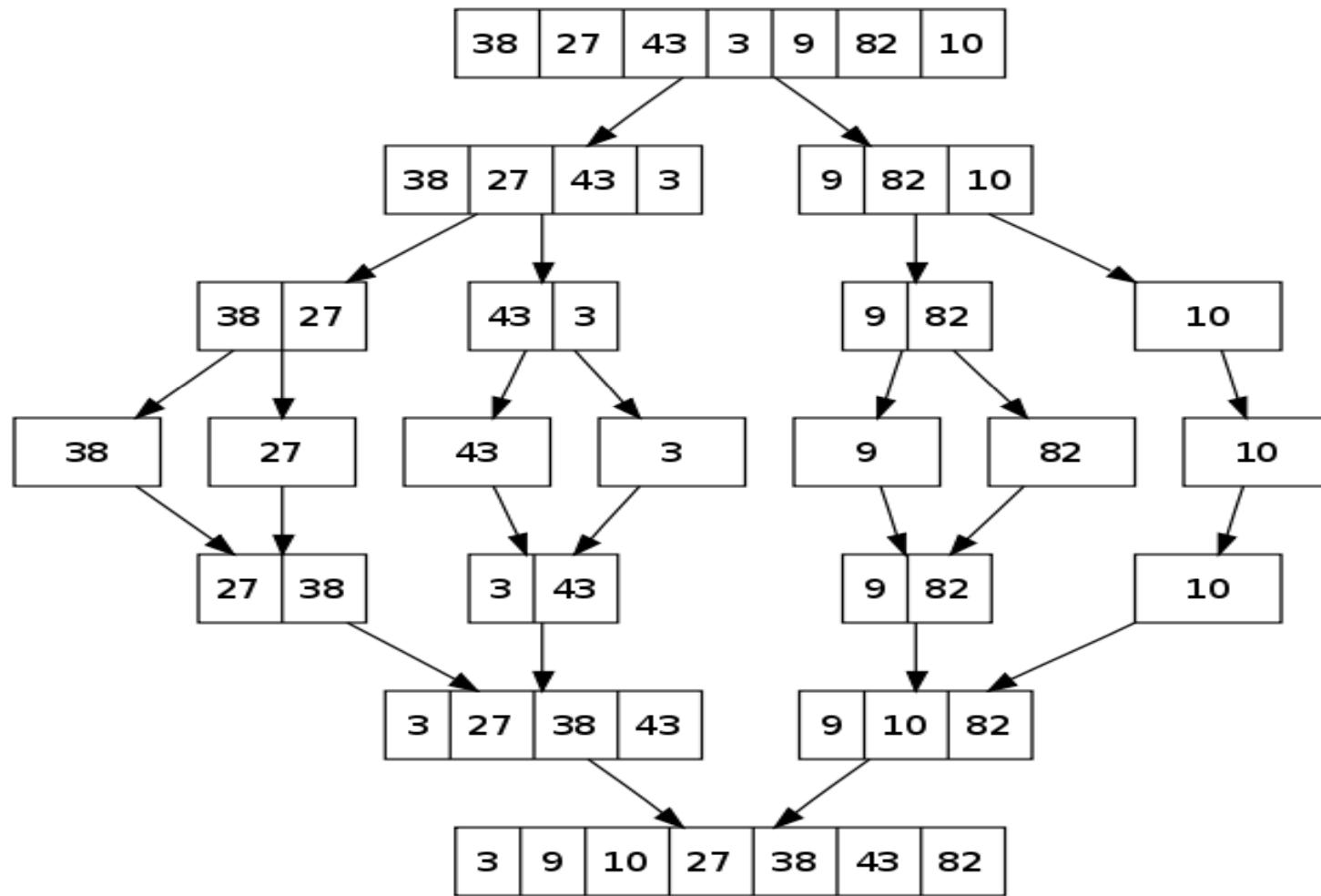
1. if  $lo < hi$
2. then  $mid \leftarrow \lfloor (lo+hi)/2 \rfloor$
3.       MERGE-SORT(A, $lo,mid$ )
4.       MERGE-SORT(A, $mid+1,hi$ )
5.       MERGE(A, $lo,mid,hi$ )

Call MERGE-SORT(A,1, $n$ ) (assume  $n$ =length of list A)

$$A = \{10, 5, 7, 6, 1, 4, 8, 3, 2, 9\}$$

# Merge sort EXAMPLE

from wikipedia



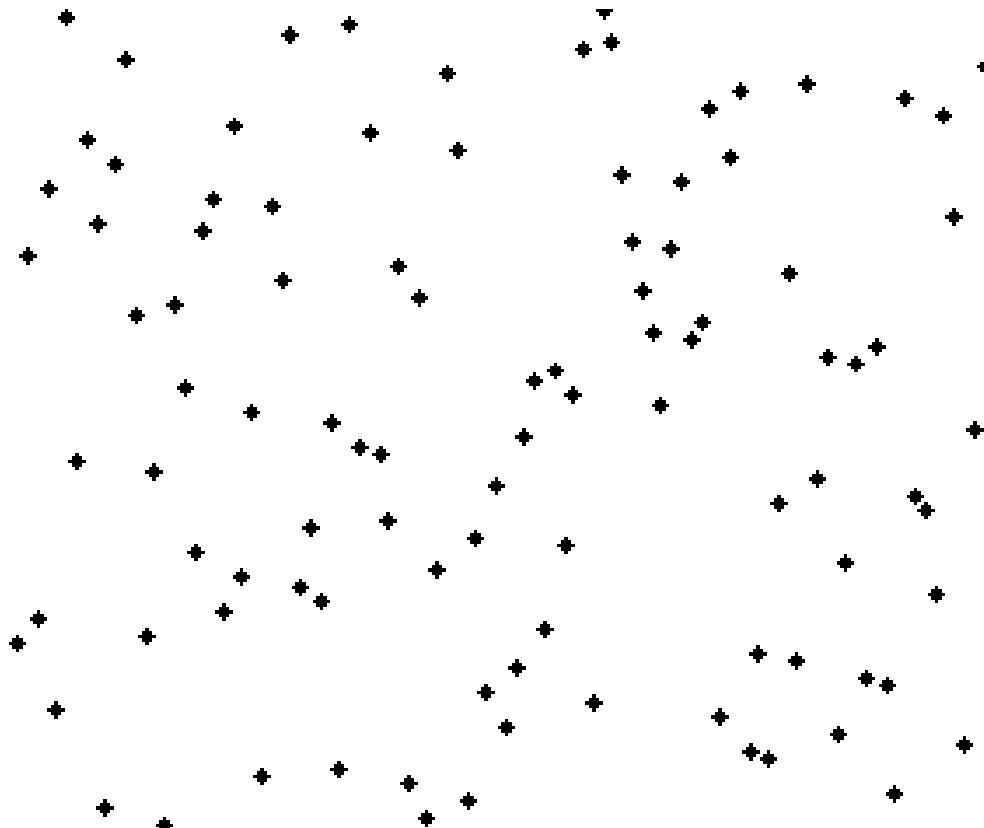
# Analysis of Merge Sort

1. if $lo < hi$	.....	1
2. then $mid \leftarrow \lfloor (lo+hi)/2 \rfloor$	.....	1
3.           MERGE-SORT(A, $lo,mid$ )	.....	$n/2$
4.           MERGE-SORT(A, $mid+1,hi$ )	.....	$n/2$
5.           MERGE(A, $lo,mid,hi$ )	.....	$n$

- Described by recursive equation
- Let  $T(n)$  be the running time on a problem of size  $n$ .
- $T(n) = c$  if  $n=1$   
 $2T(n/2)+cn$  if  $n>1$

$O(n \log n)$

# Merge sort animation



# Merge sort animation



# Quick sort

- The quicksort algorithm was developed in 1960 by [Tony Hoare](#) while in the [Soviet Union](#), as a visiting student at [Moscow State University](#).
- At that time, Hoare worked in a project on [machine translation](#) for the [National Physical Laboratory](#).
- He developed the algorithm in order to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape

# Algorithm

- Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.
- The steps are:
- Pick an element, called a pivot, from the list.
- Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.
- The base case of the recursion are lists of size zero or one, which never need to be sorted.

# Psuedo code

- function quicksort('array')
- if length('array') ≤ 1
- return 'array'    // an array of zero or one elements is already sorted
- select and remove a pivot value 'pivot' from 'array'
- create empty lists 'less' and 'greater'
- for each 'x' in 'array'
- if 'x' ≤ 'pivot' then append 'x' to 'less'
- else append 'x' to 'greater'
- return concatenate(quicksort('less'), 'pivot', quicksort('greater')) // two recursive calls

6 5 3 1 8 7 2 4

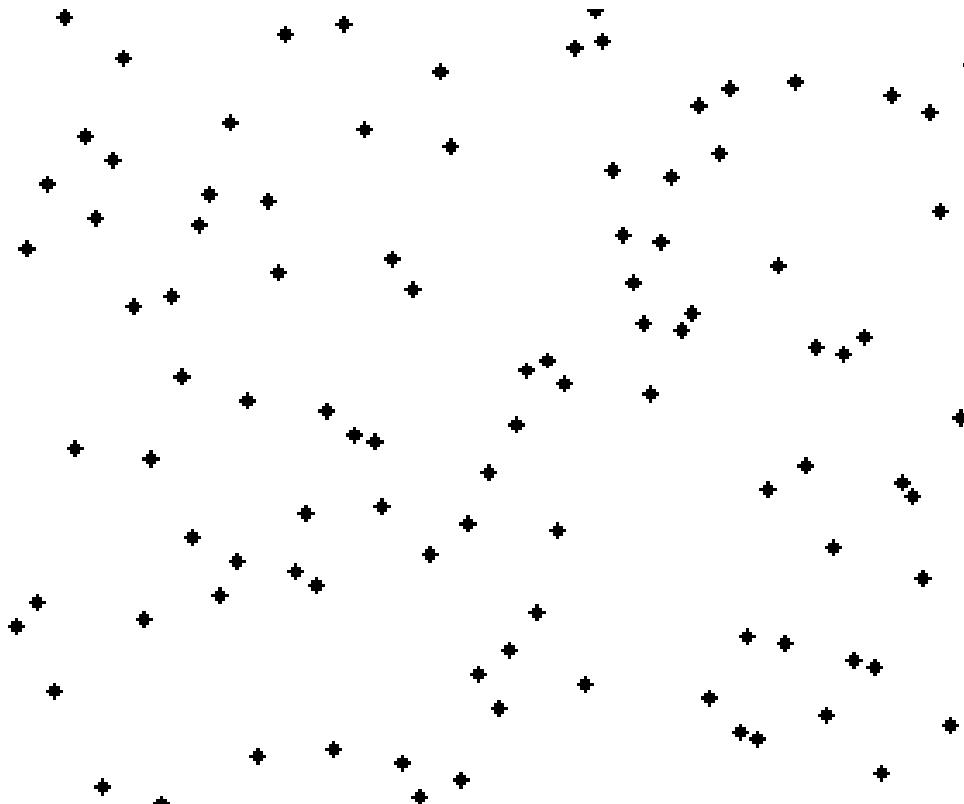
# Properties

- Not stable
- $O(\lg(n))$  extra space
- $O(n^2)$  time, but typically  $O(n \cdot \lg(n))$  time
- With both sub Sorts performed recursively, quick **sort** requires  $O(n)$  extra space for the recursion stack in the worst case when recursion is not balanced.
- To make sure at most  $O(\log N)$  space is used, recurse first into the smaller half of the array, and use a tail call to recurse into the other
- **Quicksort with 3-way partitioning.**
- Use insertion sort, which has a smaller constant factor and is thus faster on small arrays.

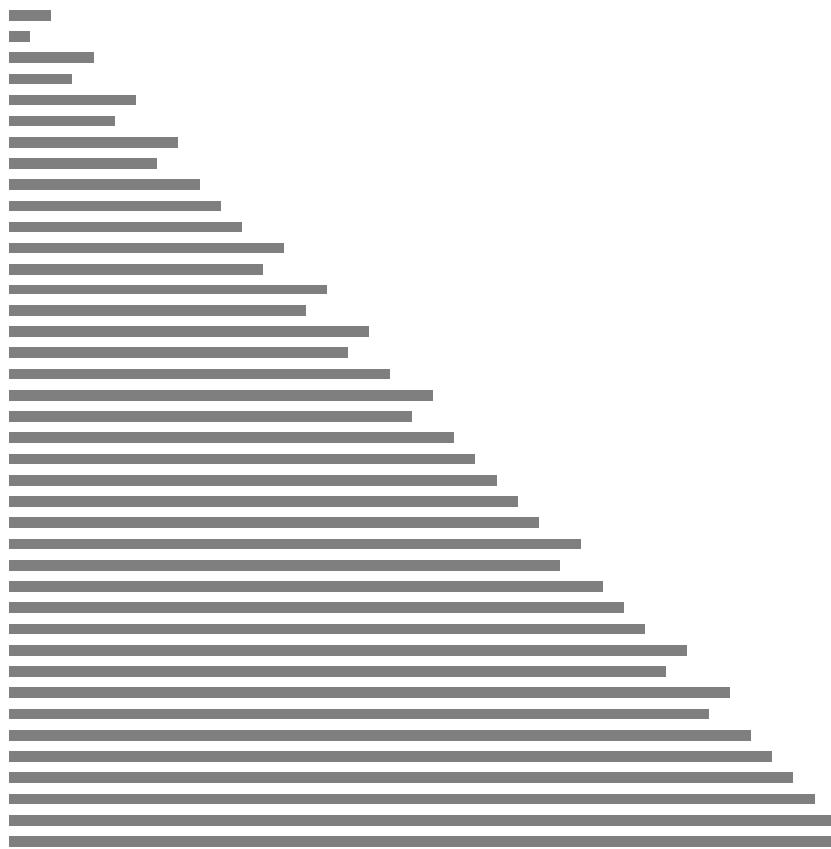
# Pivot selection

- In very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element.
- Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case.
- The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot [[Sedgewick](#)].

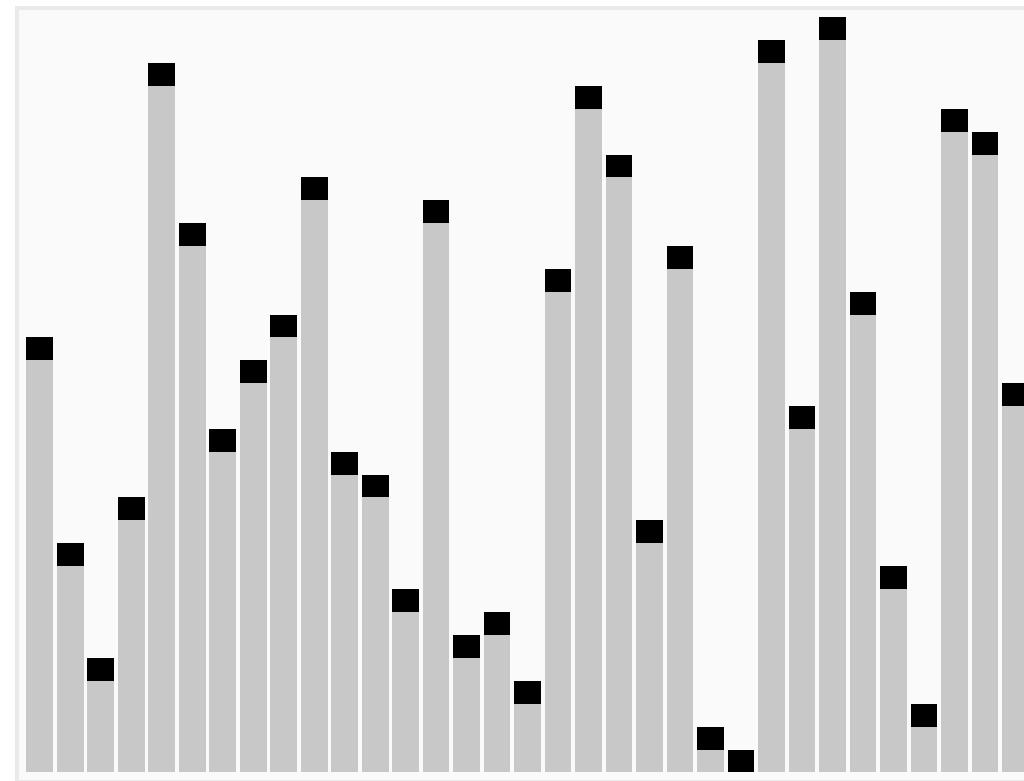
# Quick sort animation



# Quick sort animation



# Quick sorting example



# Divide and Conquer

An algorithm design technique

1. **Divide:** the instance (problem) into a number of subinstances (in most cases 2).
2. **Conquer:** the subinstance by solving them separately.
3. **Combine:** the solutions to the subinstances to obtain the solution to the original problem instance.

# Binary Search

**Input:** An array  $A[1..n]$  of  $n$  elements sorted in nondecreasing order and an element  $x$ .

**Output:**  $j$  if  $x = A[j]$ ,  $1 \leq j \leq n$ , and 0 otherwise.

1. *binarysearch(1, n)*

Procedure *binarysearch(low, high)*

1. if *low > high* then return 0

2. else

3.   *mid*  $\leftarrow (\text{low} + \text{high})/2$

4.   if  $x = A[\text{mid}]$  then return *mid*

5.   else if  $x < A[\text{mid}]$  then return *binarysearch(low, mid-1)*

6.   else return *binarysearch(mid + 1, high)*

7. end if

**C(n)** is the number of comparisons

performed by Algorithm

**BINARYSEARCHREC** in the worst case

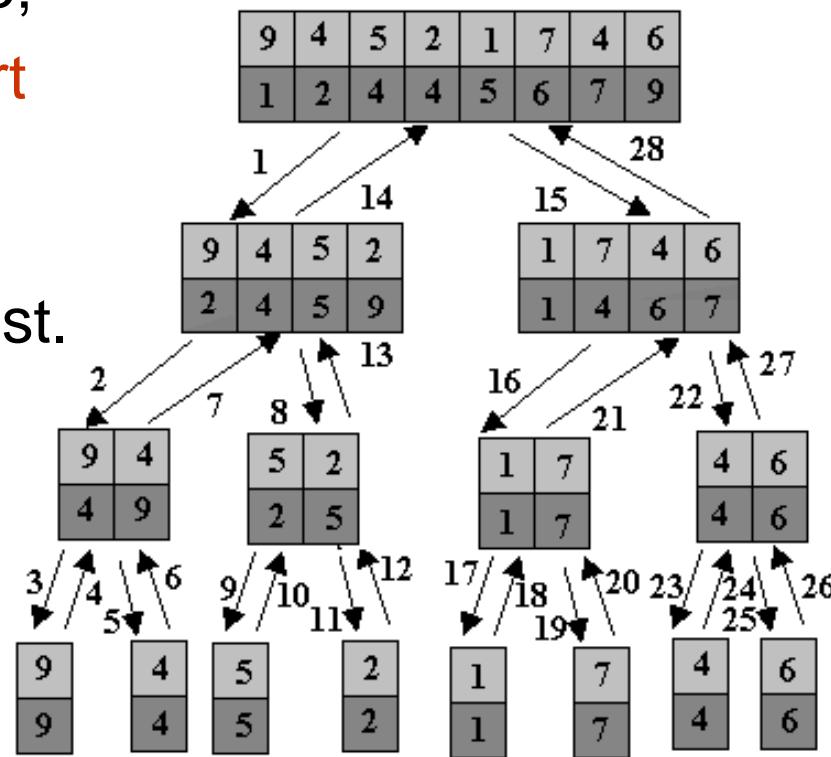
on an array of size **n**.

$$C(n) \leq \begin{cases} 1 & \text{if } n=1 \\ 1+C(\lfloor n/2 \rfloor) & \text{if } n \geq 2. \end{cases}$$

*Binary search is an example of an  $O(\log n)$  algorithm. Thus, twenty comparisons suffice to find any name in the million-name list*

# MERGESORT

1. Divide the whole list into 2 sublists of equal size;
2. Recursively merge sort the 2 sublists;
3. Combine the 2 sorted sublists into a sorted list.



## Algorithm MERGESORT

**Input:** An array  $A[1..n]$  of  $n$  elements.

**Output:**  $A[1..n]$  sorted in nondecreasing order.

*mergesort(A, 1, n)*

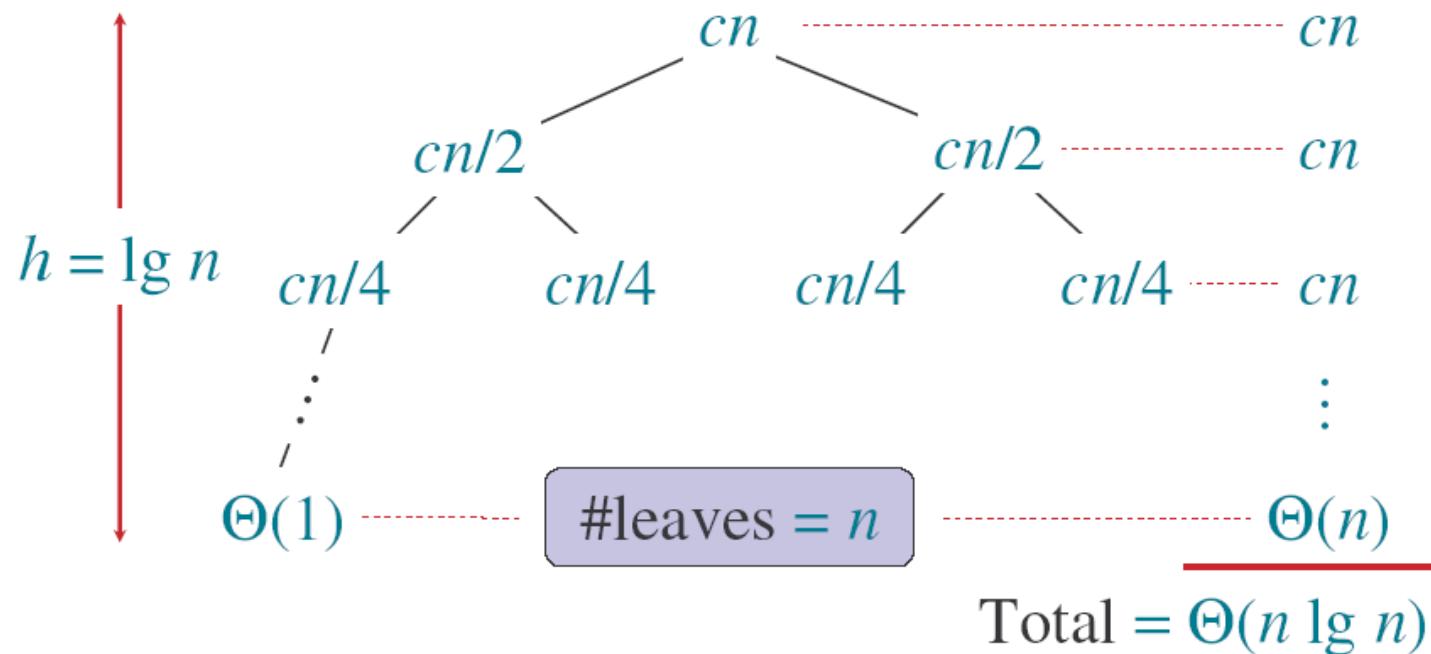
Procedure *mergesort(A, low, high)*

1. if  $low < high$  then
2.      $mid \leftarrow (low + high) / 2$                                $T(1)$
3.     *mergesort(A, low, mid)*                               $T(n/2)$
4.     *mergesort(A, mid + 1, high)*                       $T(n/2)$
5.     MERGE ( $A, low, mid, high$ )                       $T(n)$
6. end if

**Use:** Linear-time *merge* subroutine.

# Analysis for Merge Sort

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



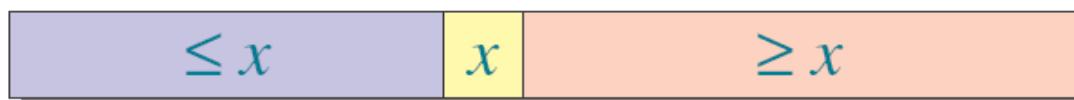
# Quick sort

- Divide-and-conquer algorithm.
- Sorts “in place” (*like insertion sort, but not like merge sort*).

Quicksort an  $n$ -element array:

1. **Divide:** Partition the array into two subarrays around a **pivot**  $x$  such that elements in lower subarray  $\leq x \leq$  elements in upper subarray.
- 2. **Conquer:** Recursively sort the two subarrays.
- 3. **Combine:** Nothing

**Use:** *Linear-time partitioning subroutine.*



Partitioning places all the elements less than the pivot in the *left* part of the array, and all elements greater than the pivot in the *right* part of the array. The pivot fits in the slot between them.

# Partitioning

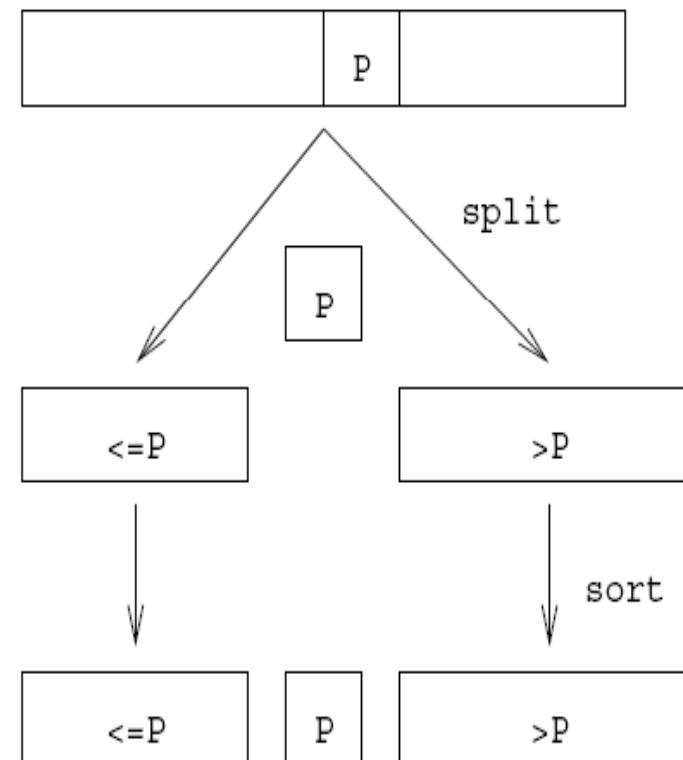
Partition a list  $A[]$  into two non-empty parts.

Pick any value in the array, **pivot**.

$\text{left} = \text{the elements in } A[] \leq \text{pivot}$

$\text{right} = \text{the elements in } A[] > \text{pivot}$

Place The **pivot** in the slot  
**between** them.



An illustration of one step partitioning

**procedure Quicksort(A, p, r)**

**if** p < r **then**

    q  $\leftarrow$  Partition(A, p, r)

    Quicksort(A, p, q - 1)

    Quicksort(A, q + 1, r)

**Partition( A[ ], first, last)**

*//Define the pivot value as the contents of A[First]*

P = A[first] ; Q=first;

*//Initialize Up to First and Down to Last*

Up = first ; down = last;

Repeat

    Repeat up++    until (A[up] > P)

    repeat down- - until (A[down]  $\leq$  P)

    if (up < Down)

        exchange (A[up] , A[down] )

until (up  $\geq$  Down )

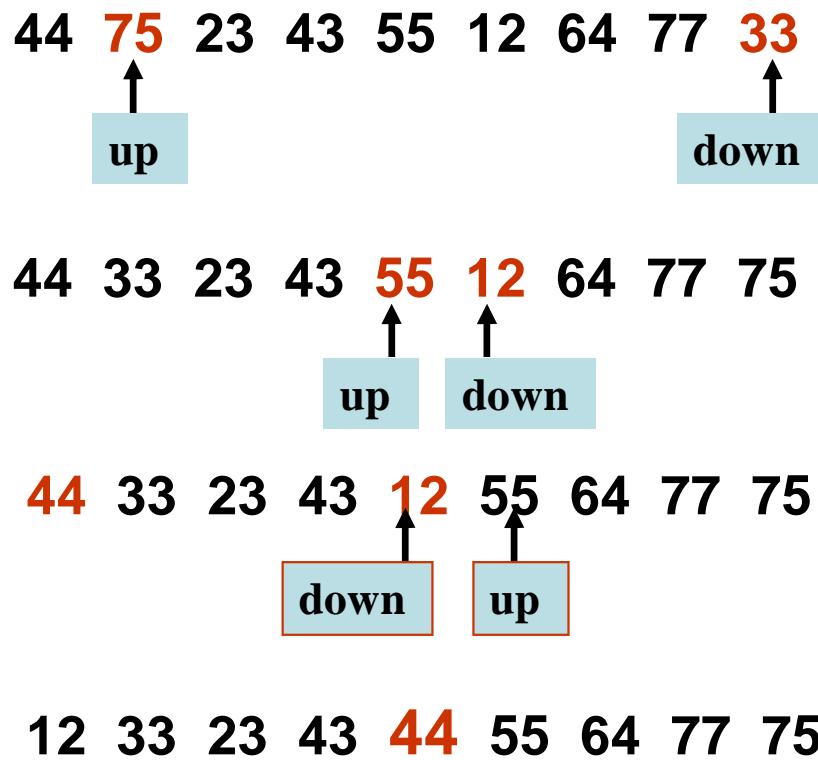
Exchange (A[First] , A[Down])

Q= Down

**Return Q**

## The effects of the linear time partitioning step:

1. The pivot element ends up in the position it retains in the final sorted order.
2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.



***Thus we can sort the elements to the left of the pivot and the right of the pivot independently!***

# Quicksort– best case

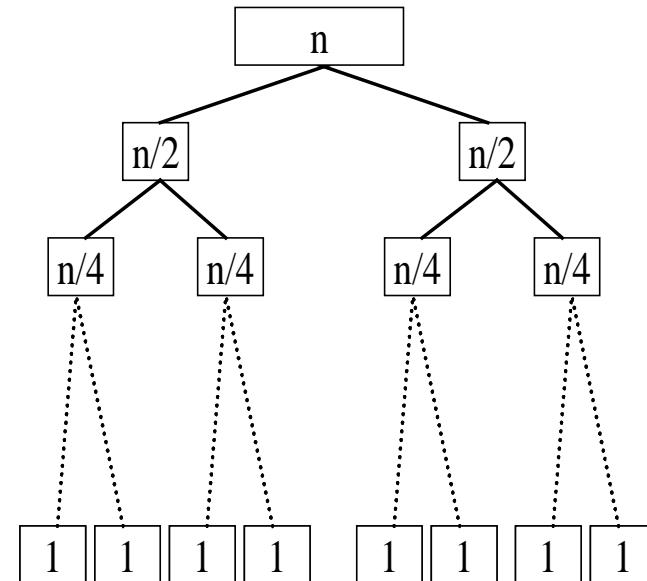
The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size  $n/2$ .

The partition step on each subproblem is linear in its size. Thus the total effort in partitioning the problems of size is  $O(n)$ .

The total partitioning on each level is  $O(n)$ , and it takes  $\log n$  levels of partitions to get to single element subproblems.

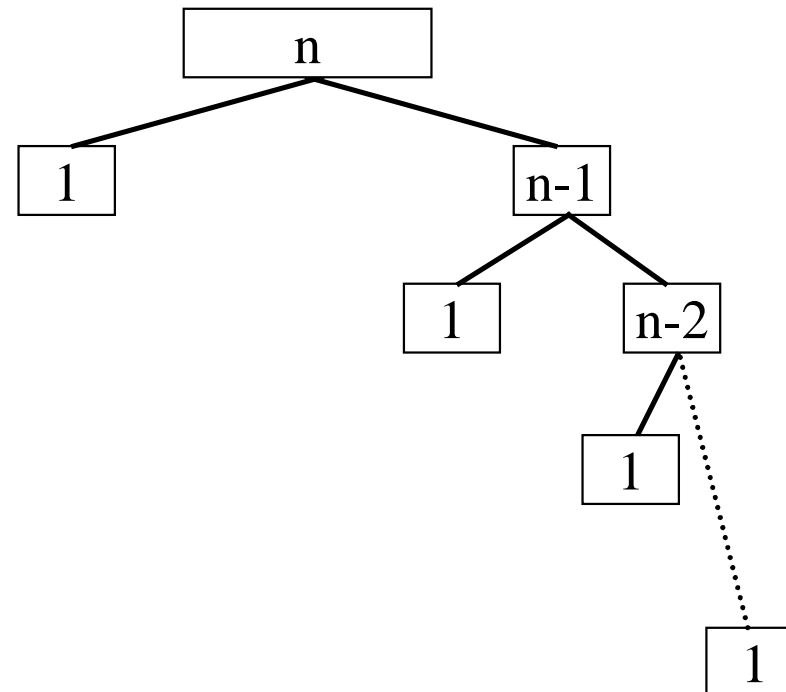
When we are down to single elements, the problems are sorted. Thus the total time in the best case is  $O(n \log n)$ .

**The recursion tree for the best case looks like this**



# Quicksort - Worst Case

- Worst case: the pivot chosen is the largest or smallest value in the array. Partition creates one part of size 1 (containing only the pivot), the other of size  $n-1$ .
- Now we have  $n-1$  levels, instead of  $\log n$ , for a worst case time of  $O(n^2)$



# Heap sort

Heaps and Priority Queues

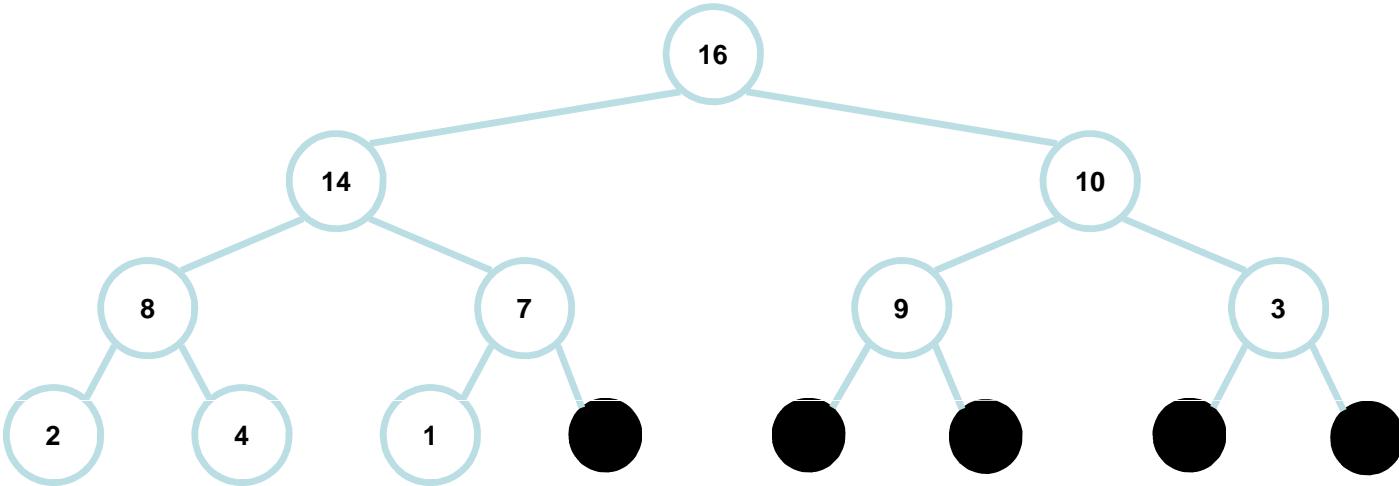
# What is Heap?

An array Representing an *almost complete binary tree*:

- All levels filled except possibly the last
- Last level filled from the left to a point

Let A be an array that implements the heap:

- Root of the tree is A[1].
- Node  $i$  is A[i]
- Parent(i) { return  $\lfloor i/2 \rfloor$ ; }
- Left(i) { return  $2*i$ ; }
- right(i) { return  $2*i + 1$ ; }



Heap can be seen as a complete binary tree think of unfilled slots as null pointers

A= 

16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---

- *What is the height of an n-element heap? Why?*

# Heap Property

$$A[\text{PARENT}(i)] \geq A[i] \quad \text{for all nodes } i > 1$$

- the value of a node is at most the value of its parent
- the largest element in the heap is stored in the root

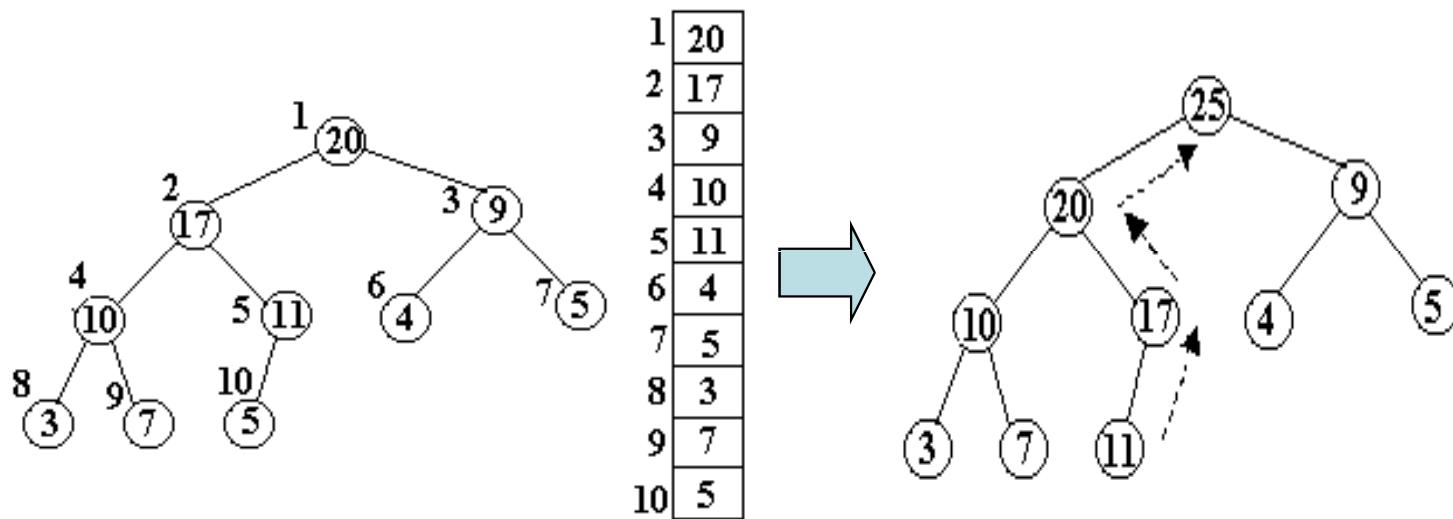
- **Operations on Heaps**

- ***delete-max[H]***: Delete and return an item of maximum key from a nonempty heap **H**.
- ***insert[H,x]***: Insert item **x** into heap **H**.
- ***delete[H,i]***: Delete the **i**th item from heap **H**.
- ***makeheap[A]***: Transform array **A** into a heap.
- ***HEAPSORT(A)***: sorts an array in place.

- Before describing the main heap operations, let us first present two secondary heap operations:
  - **Shift-up**
  - **Shift-down**
- *These are used subroutines in the algorithms that implement heap operations.*

# Shift-up

➤ Suppose the key stored in the 10th position of the heap shown in is changed from 5 to 25.

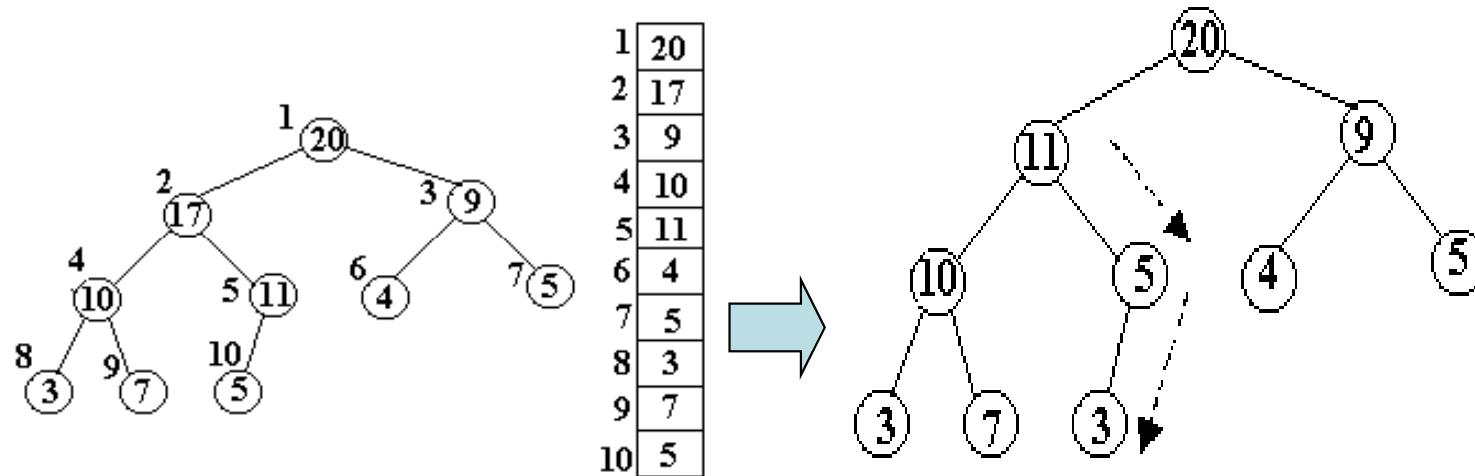


move  $H[i]$  up along the *unique* path from  $H[i]$  to the root until its proper location along this path is found.

At each step along the path, the key of  $H[i]$  is compared with the key of its parent  $H[\lceil i/2 \rceil]$ .

# Shift-down

➤ Suppose we change the key 17 stored in the second position of the heap shown in Fig. 1 to 3.



➤ At each step along the path, its key is compared with the maximum of the two keys stored in its children nodes (if any).

## Algorithm MAKEHEAP

**Input:** An array  $A[1..n]$  of  $n$  elements.

**Output:**  $A[1..n]$  is transformed into a heap

1. **for**  $i \leftarrow n/2$  **downto** 1
2.   **SHIFT-DOWN( $A$  ,  $i$ )**
3. **end for**

# Make heap animation

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort from the smallest to the largest. (NOTE, for 'Building the Heap' step: Larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap.

6 5 3 1 8 7 2 4

# Heap sort Algorithm

- Heapsort is a two step algorithm.
- The first step is to build a heap out of the data.
- The second step begins with removing the largest element from the heap.
- We insert the removed element into the sorted array.
- For the first element, this would be position 0 of the array.
- Next we reconstruct the heap and remove the next largest item, and insert it into the array.
- After we have removed all the objects from the heap, we have a sorted array.

# Heap sort properties

- Heapsort is not a [stable sort](#); merge **sort** is stable.
- The heapsort algorithm itself has  $O(n \log n)$  time complexity

# Algorithm

- function **heapify**(a,count) is
- // (end is assigned the index of the first (left) child of the root)
- end := 1
- 
- while end < count
- // (sift up the node at index end to the proper place such that all nodes above
- // the end index are in heap order)
- siftUp(a, 0, end)
- end := end + 1
- // (after sifting up the last node all nodes are in heap order)
- 
- function **siftUp**(a, start, end) is
- input: start represents the limit of how far up the heap to sift.
- end is the node to sift up.
- child := end
- while child > start
- parent := floor((child - 1) / 2)
- if a[parent] < a[child] then (out of max-heap order)
- swap(a[parent], a[child])
- child := parent (repeat to continue sifting up the parent now)
- else
- return

# Heap creating example animation

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort ascending.

NOTE, for 'Building the Heap' step:

Larger nodes don't stay below smaller node parents.

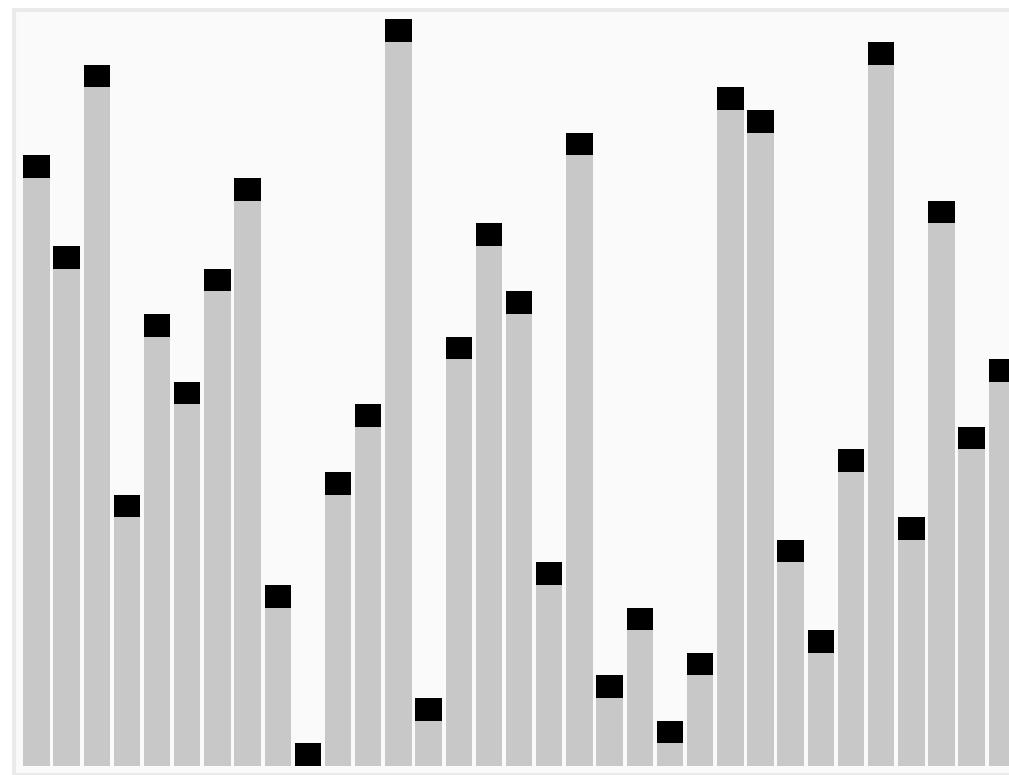
They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap.

6 5 3 1 8 7 2 4

# heap sort animation



# heap sort animation



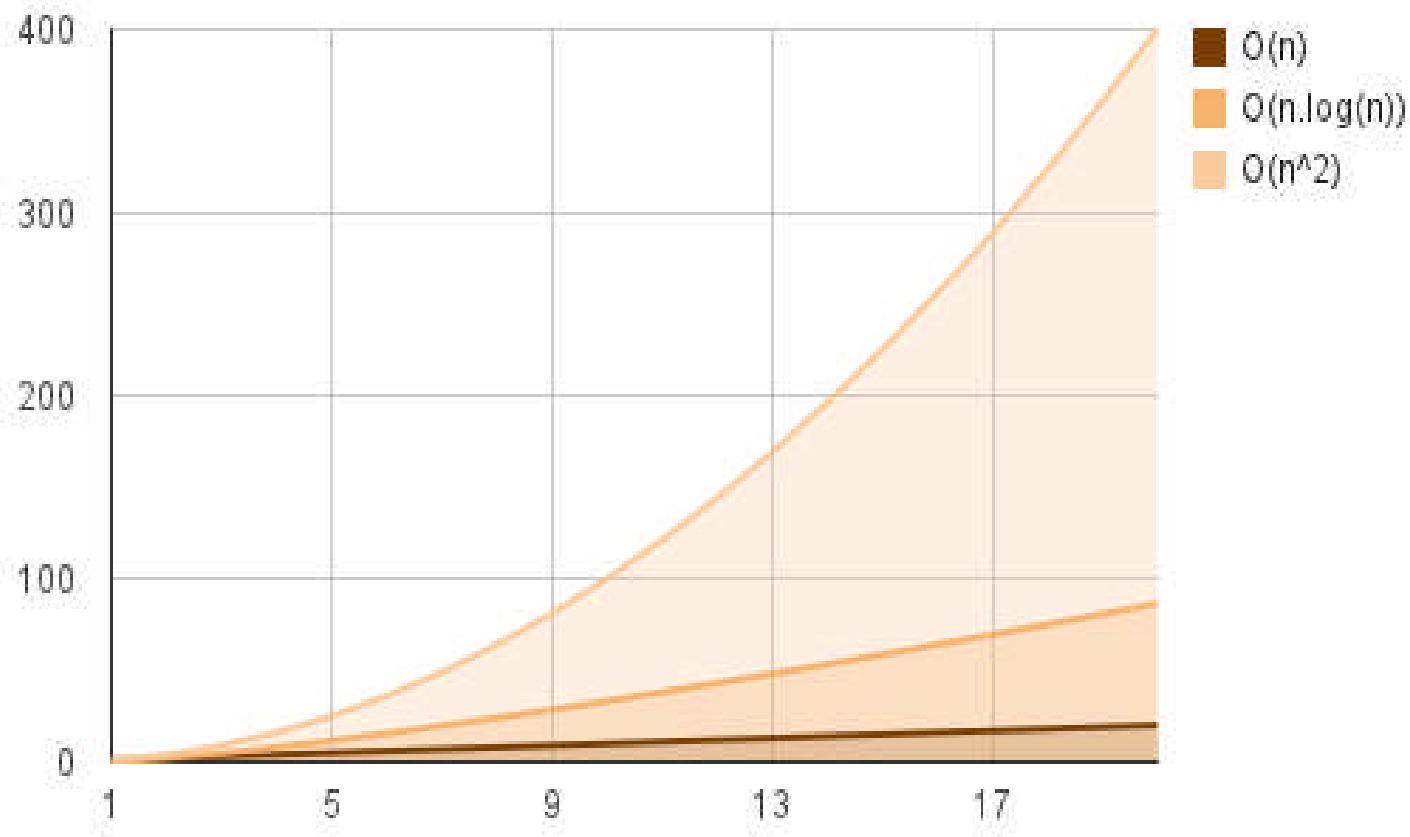
# Radix sort

- Radix **sort** is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- A positional notation is required, but because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers.
- Radix **sort** is not limited to integers.
- Radix **sort** dates back as far as 1887 to the work of Herman Hollerith on tabulating machines

# Radix sort is fast and stable

- Radix **sort** is a fast stable sorting algorithm which can be used to **sort** keys in integer representation order.
- The complexity of radix **sort** is linear, which in terms of omega means  $O(n)$ .
- That is a great benefit in performance compared to  $O(n \cdot \log(n))$  or even worse with  $O(n^2)$

# Radix sort is linear



# Example of radix sorting

1. Unsorted list:

- 523
- 153
- 088
- 554
- 235

2. On Radix 0 (1<sup>st</sup> digit)

- 523
- 153
- 554
- 235
- 088
- ^

Sort on Radix 1 (2nd. digit)

- 523
- 235
- 153
- 554
- 088

Sort on Radix 2 (3rd digit)

- 088
- 153
- 235
- 523
- 554
- ^

# Example of radix sort

1. Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

2. Sorting by least significant digit (1s place) gives: [we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

continued

# Example continued

3 Sorting by next digit (10s place)

gives: [Note: 802 again comes before 2 as 802 comes before 2 in the previous list.]

: 802, 2, 24, 45, 66, 170, 75, 90

4. Sorting by most significant digit (100s place) gives:

: 2, 24, 45, 66, 75, 90, 170, 802

# Radix sort is single pass

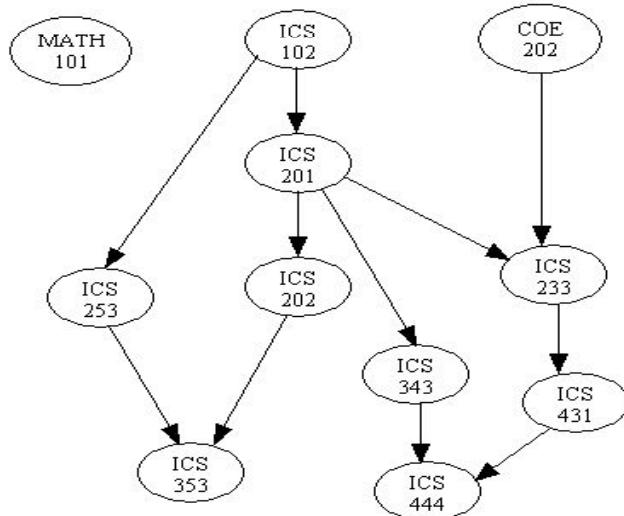
- It is important to realize that each of the above steps requires just a single pass over the data,
- since each item can be placed in its correct bucket without comparing with other items.

## Topological Sort

- Sorting vertices in a graph
- Not unique
- Not usual meaning of sort
- Graph must be directed acyclic graph (DAG) (no cycles).
- Used in scheduling a sequence of jobs or tasks based on their dependencies

# Introduction

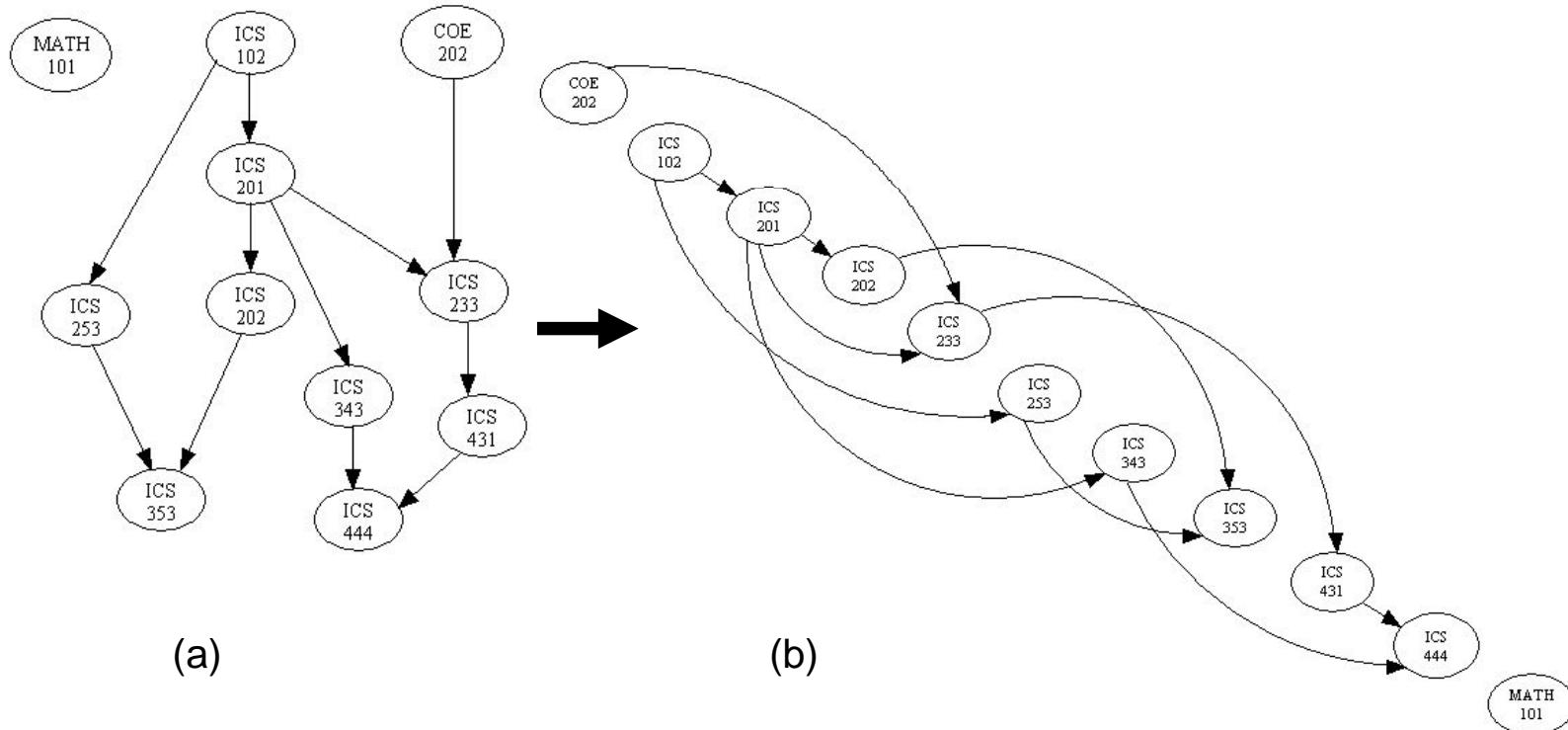
- There are many problems involving a set of tasks in which some of the tasks must be done before others.
- For example, consider the problem of taking a course only after taking its prerequisites.
- Is there any systematic way of linearly arranging the courses in the order that they should be taken?



Yes! - Topological sort.

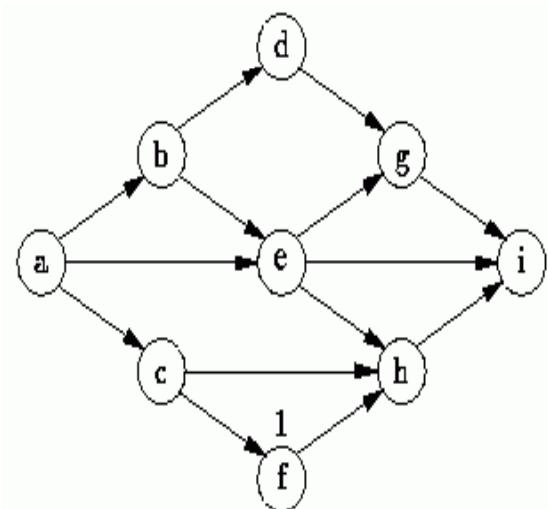
# Definition of Topological Sort

- Topological sort is a method of arranging the vertices in a directed acyclic graph (DAG), as a sequence, such that no vertex appear in the sequence before its predecessor.
- The graph in (a) can be topologically sorted as in (b)



## Topological Sort is not unique

- Topological sort is not unique.
- The following are all topological sort of the graph below:



$s1 = \{a, b, c, d, e, f, g, h, i\}$

$s2 = \{a, c, b, f, e, d, h, g, i\}$

$s3 = \{a, b, d, c, e, g, f, h, i\}$

$s4 = \{a, c, f, b, e, h, d, g, i\}$   
etc.

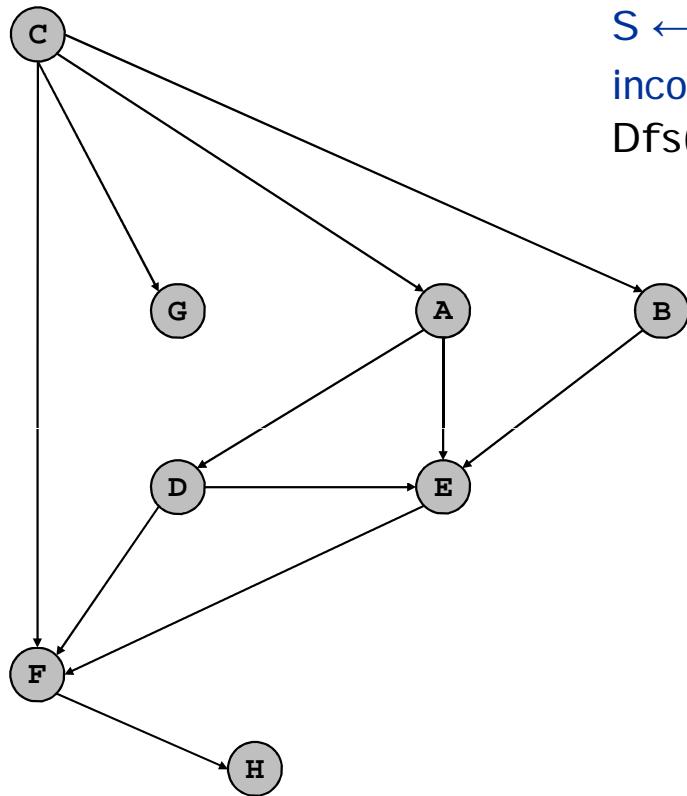
# Topological sort algorithm

1. Input graph G
2. Result  $\leftarrow \{ \}$
3. S  $\leftarrow \{ \text{ Set of all nodes with no incoming edges } \}$
4. while S is non-empty do
5.   Move **n** from S to Result
6.   for each child m of n do:
7.     delete edge **e** (n to m)
8.     if m is root (no more incoming edges) then
9.       add m to S
10. Finally
11. if G still has edges then
12.   return error (G was cyclic)
13. else
14.   return Result (a topologically sorted order)

# DFS based topological sort

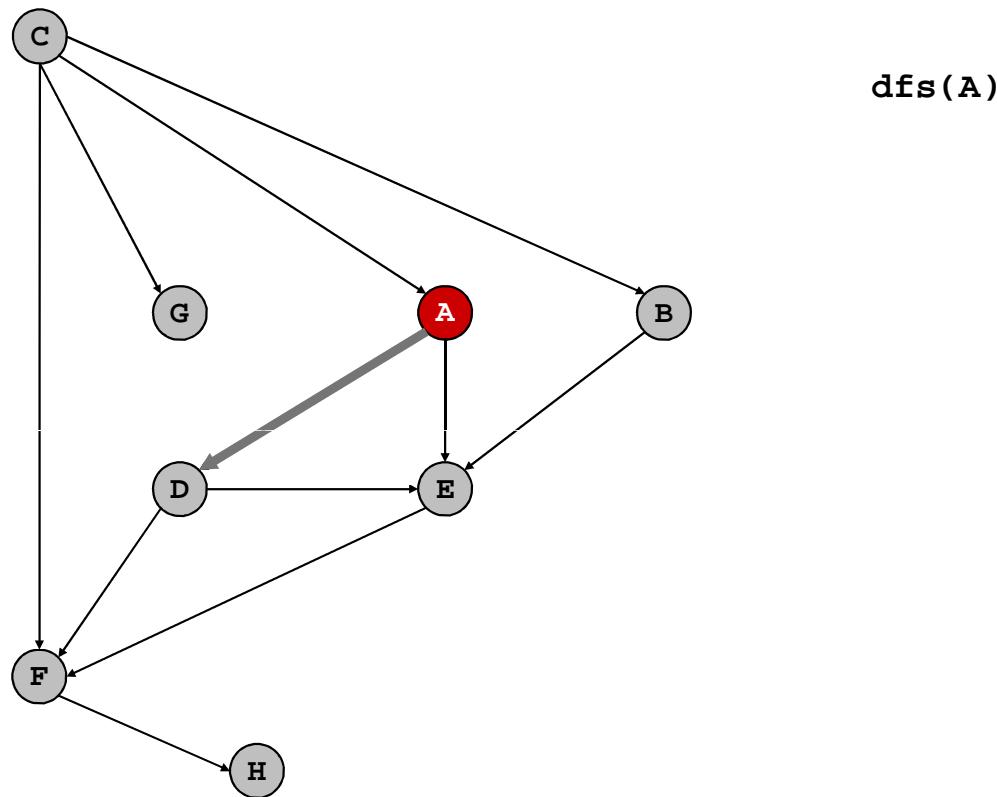
- `result = { };`
- `roots = {nodes with no parent}`
- for `n` in roots:
  - `dfs(n)`
  - function `visit(n)`
  - if `n` is not-visited then
    - mark `n` as visited
    - for child `m` of `n` do:
      - `visit(m)`
    - add `n` to `result`;

## Topological Sort: DFS

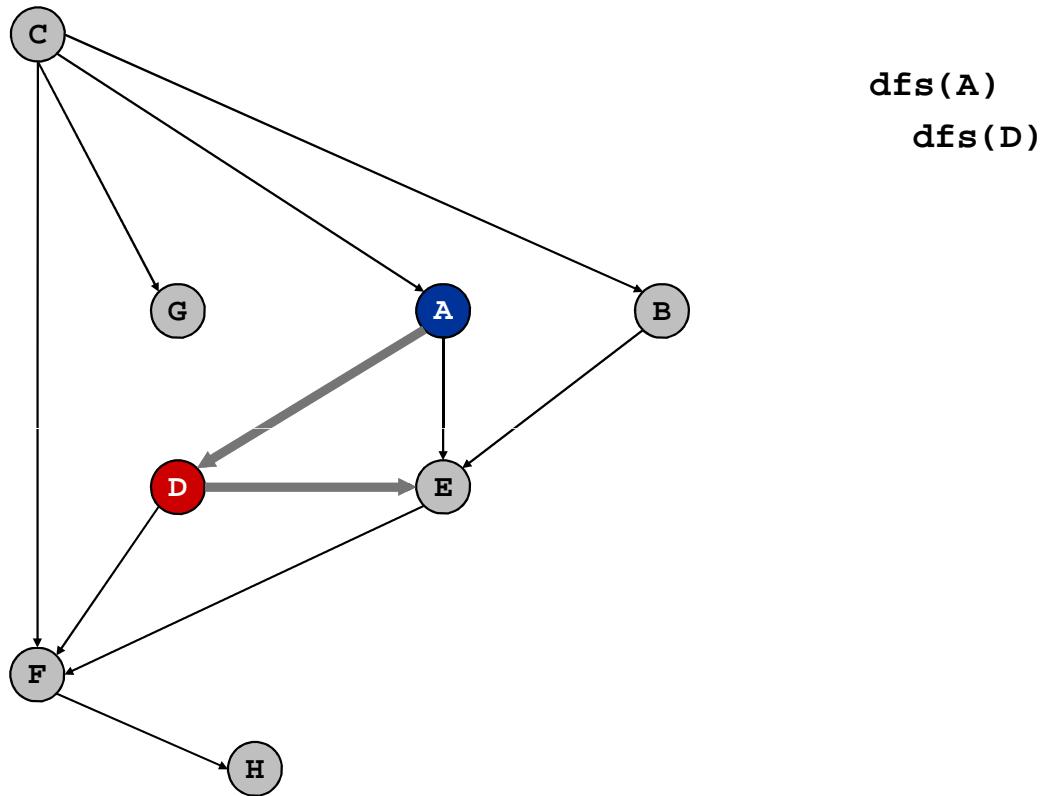


$S \leftarrow$  Set of all nodes with no incoming edges = { c }  
Dfs( c )

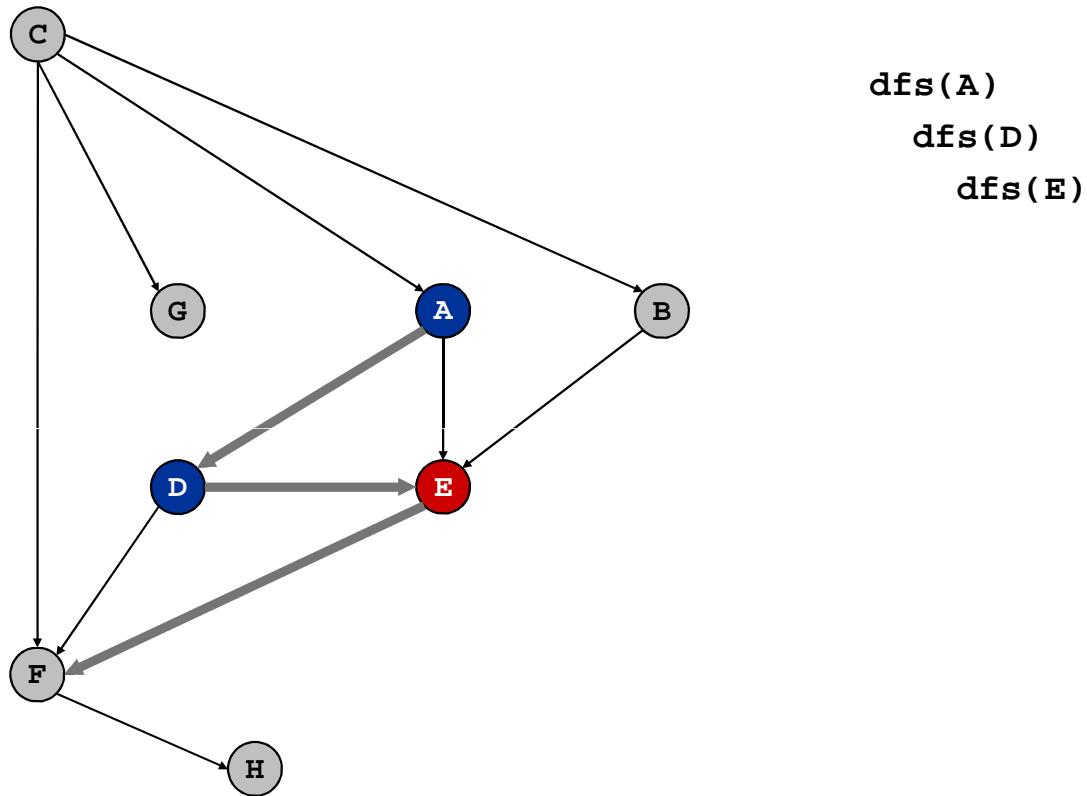
## Topological Sort: DFS



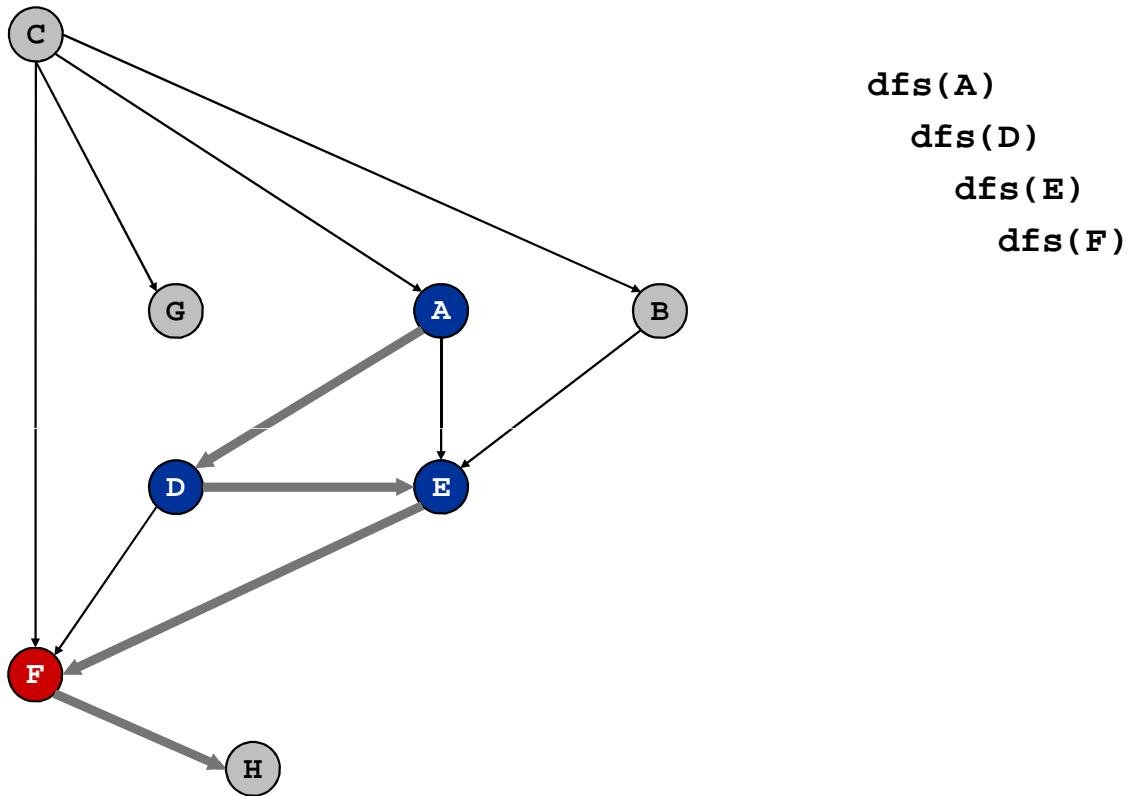
## Topological Sort: DFS



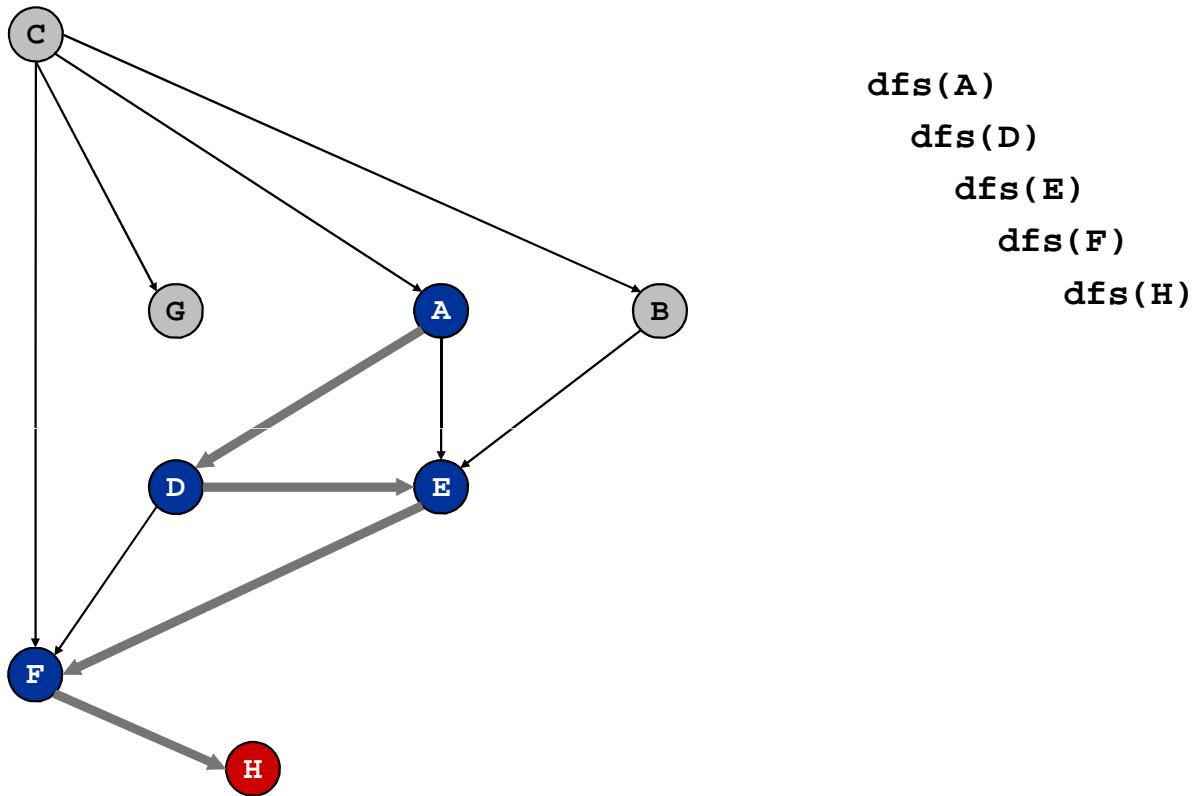
## Topological Sort: DFS



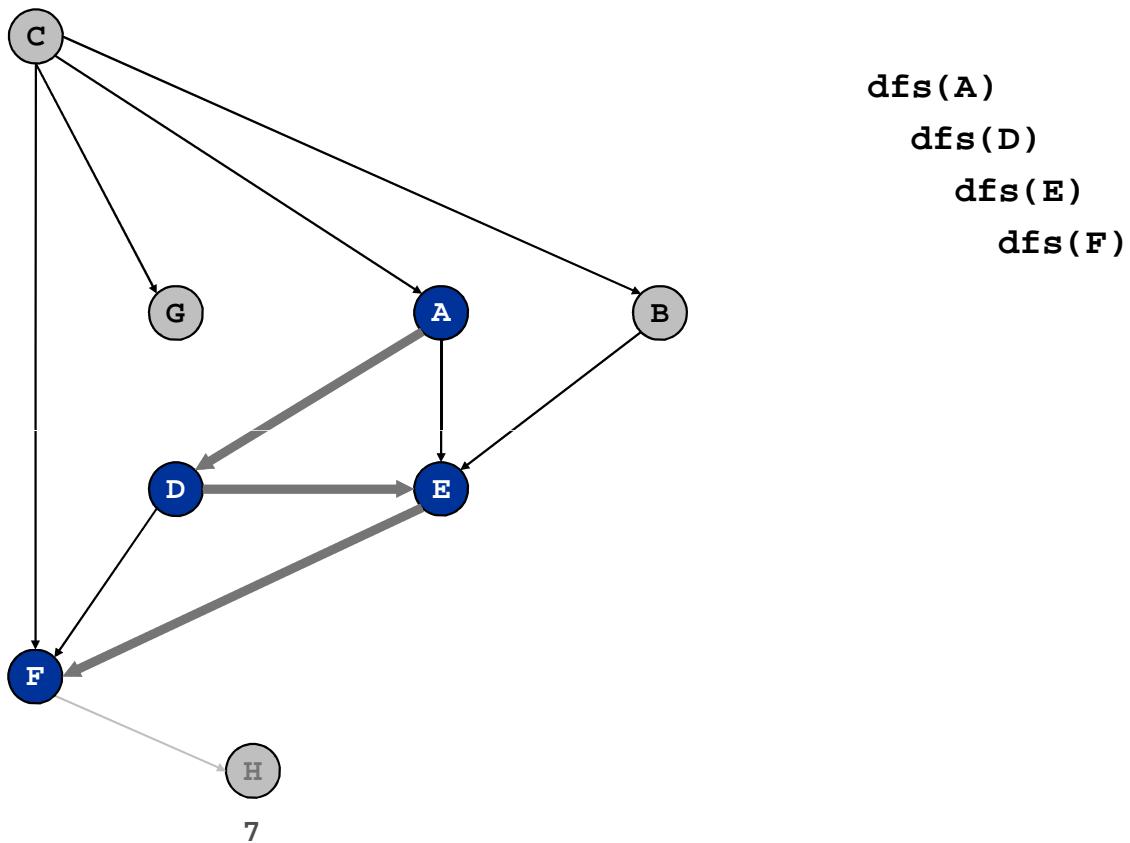
## Topological Sort: DFS



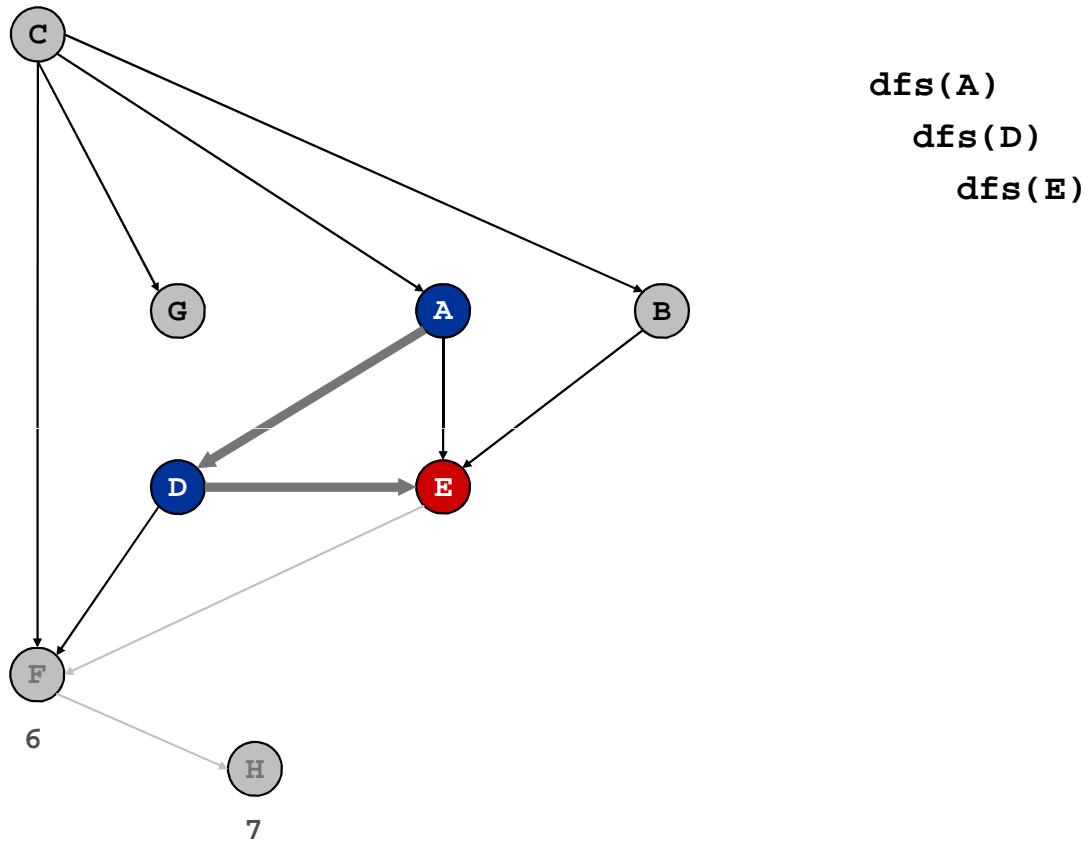
## Topological Sort: DFS



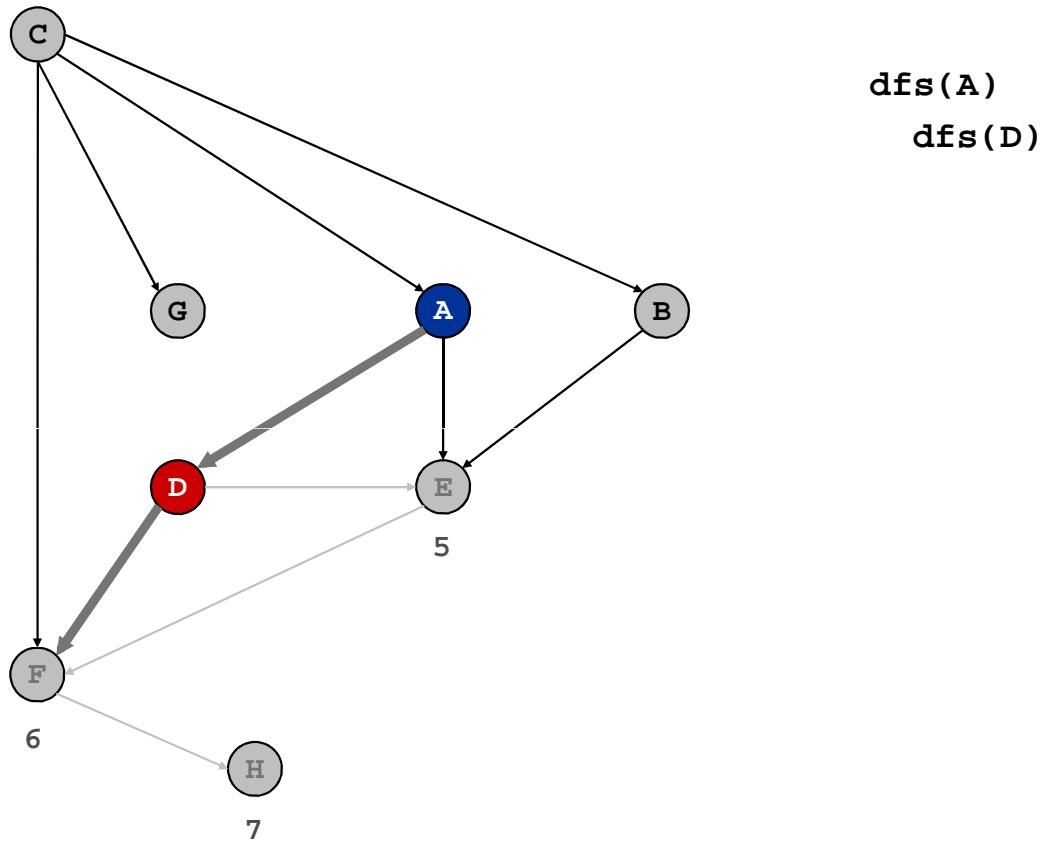
## Topological Sort: DFS



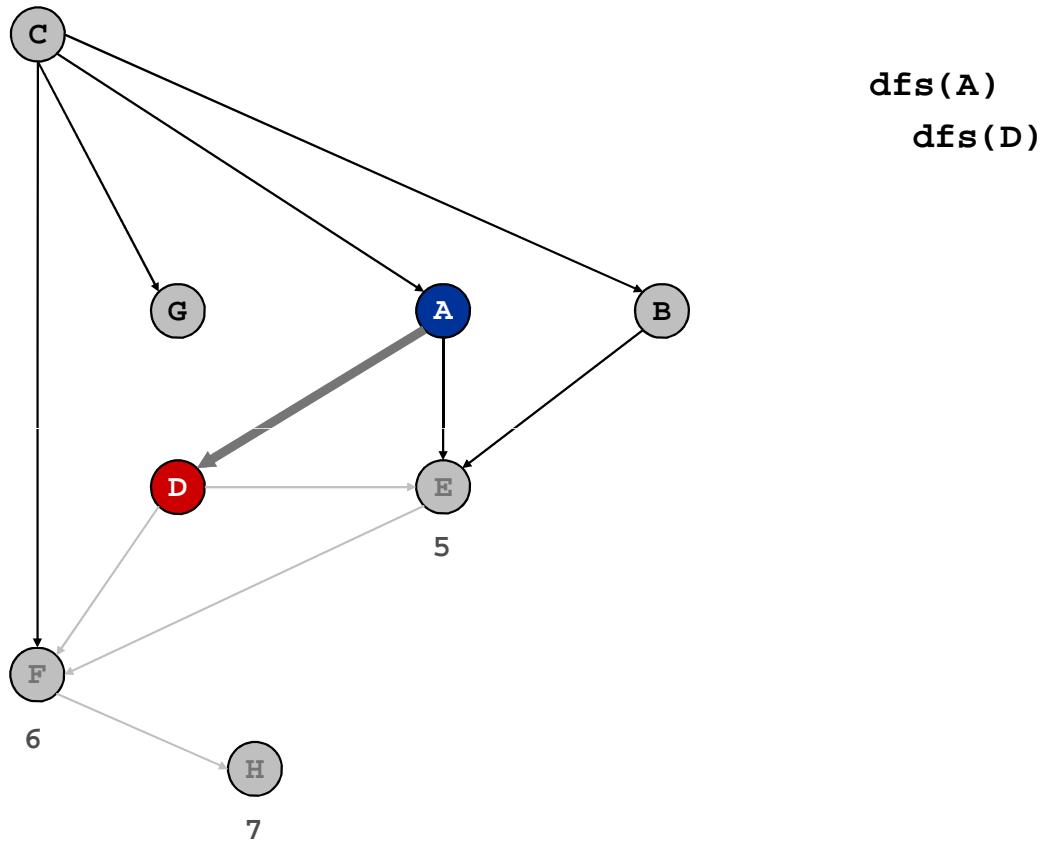
## Topological Sort: DFS



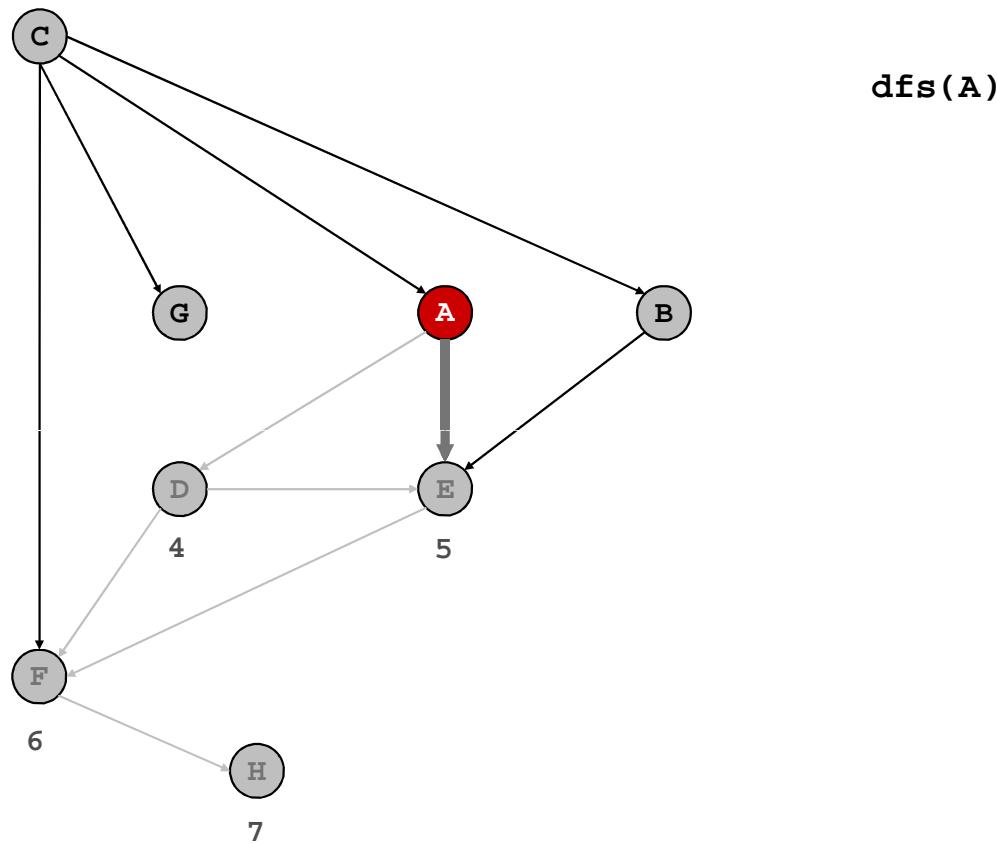
## Topological Sort: DFS



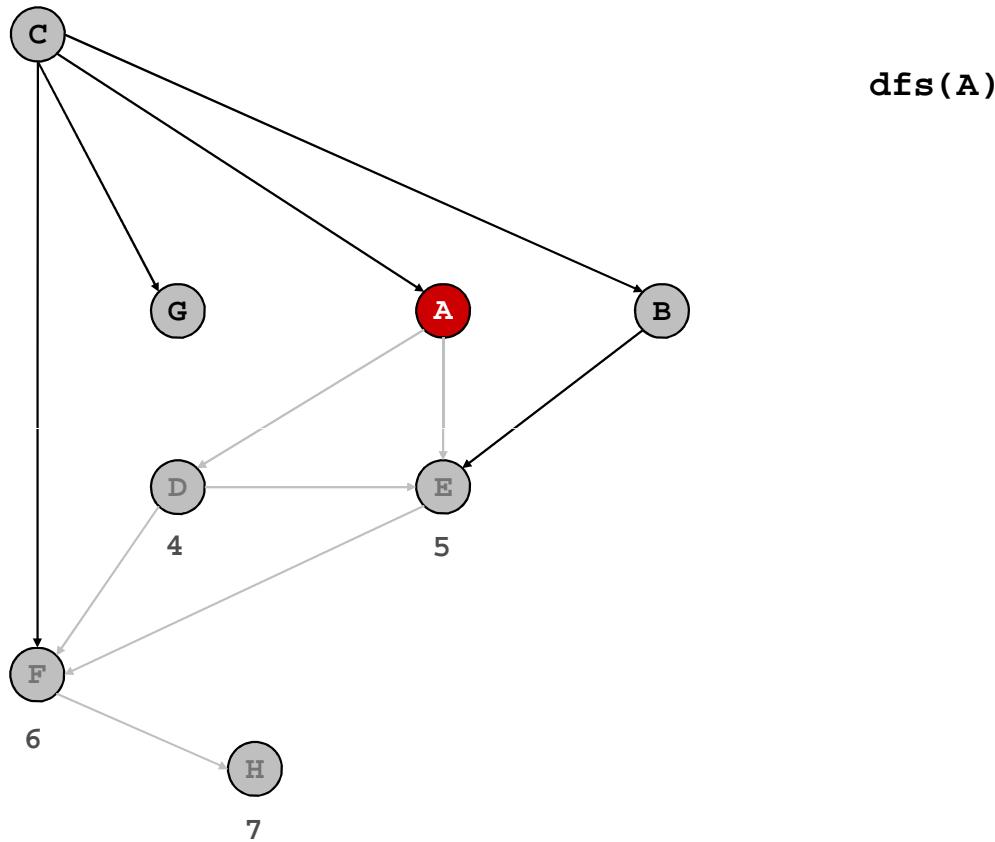
## Topological Sort: DFS



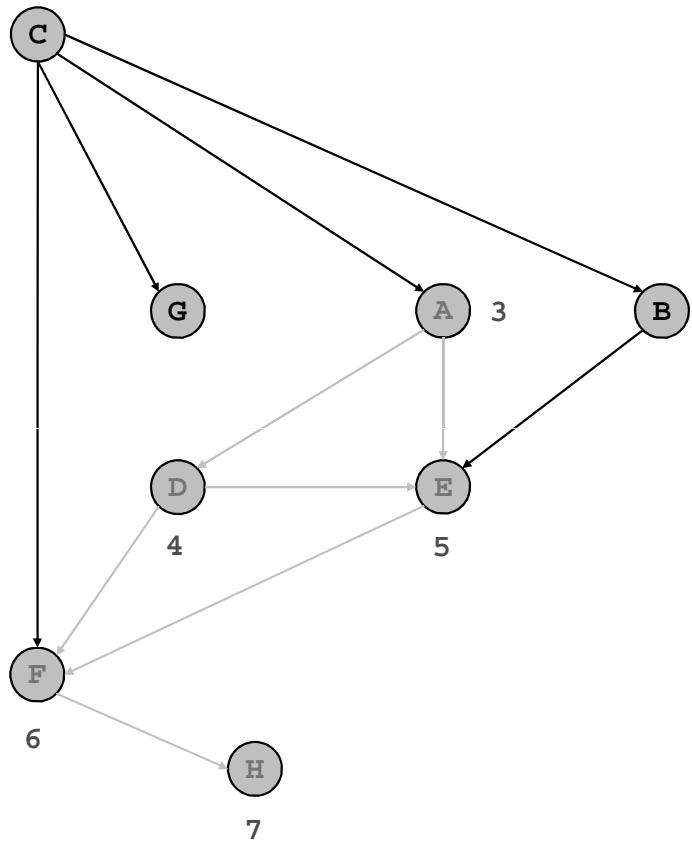
## Topological Sort: DFS



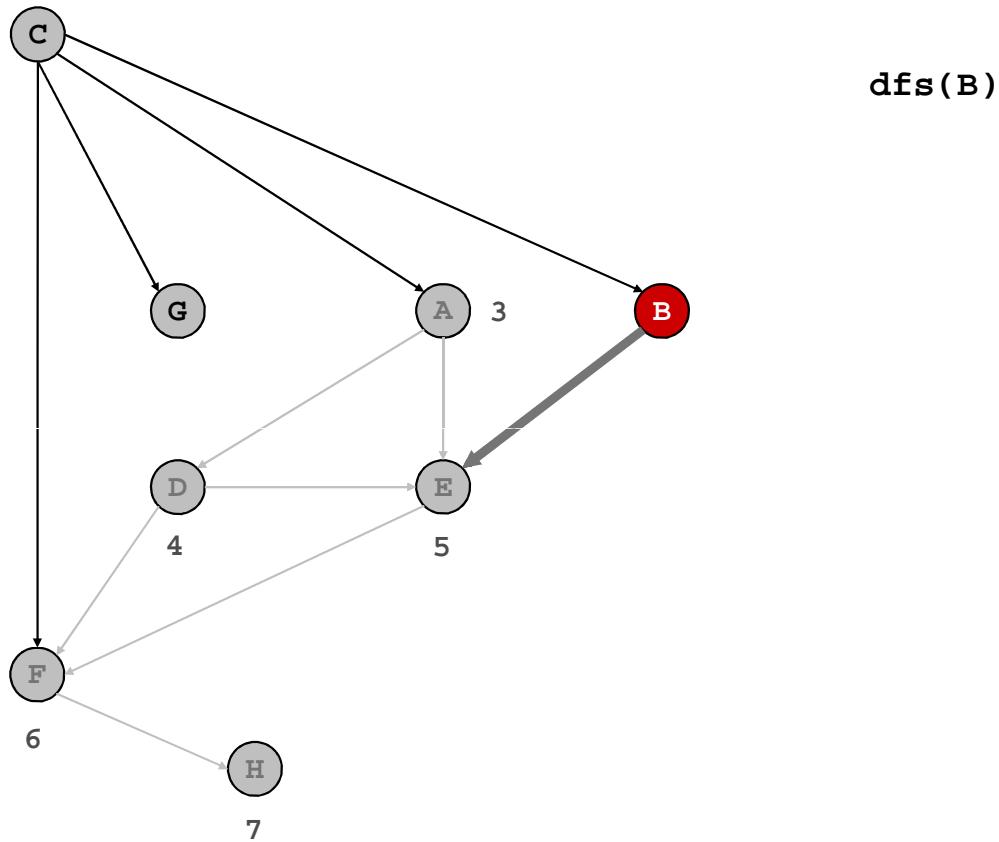
## Topological Sort: DFS



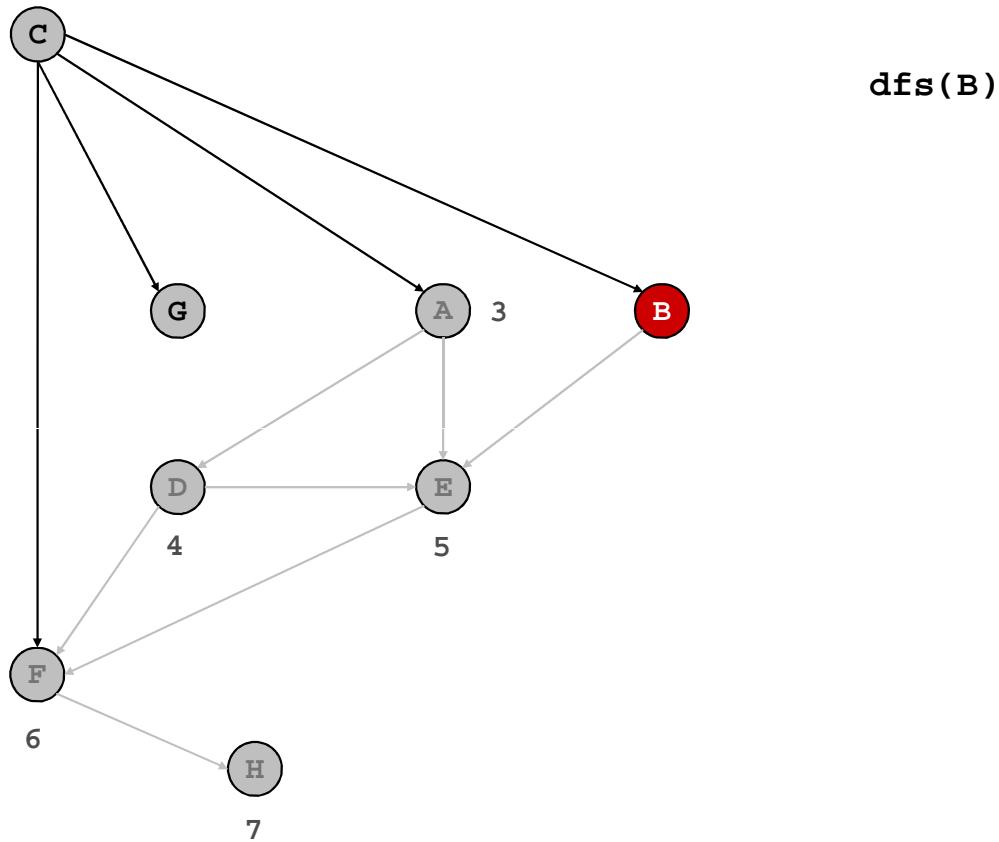
## Topological Sort: DFS



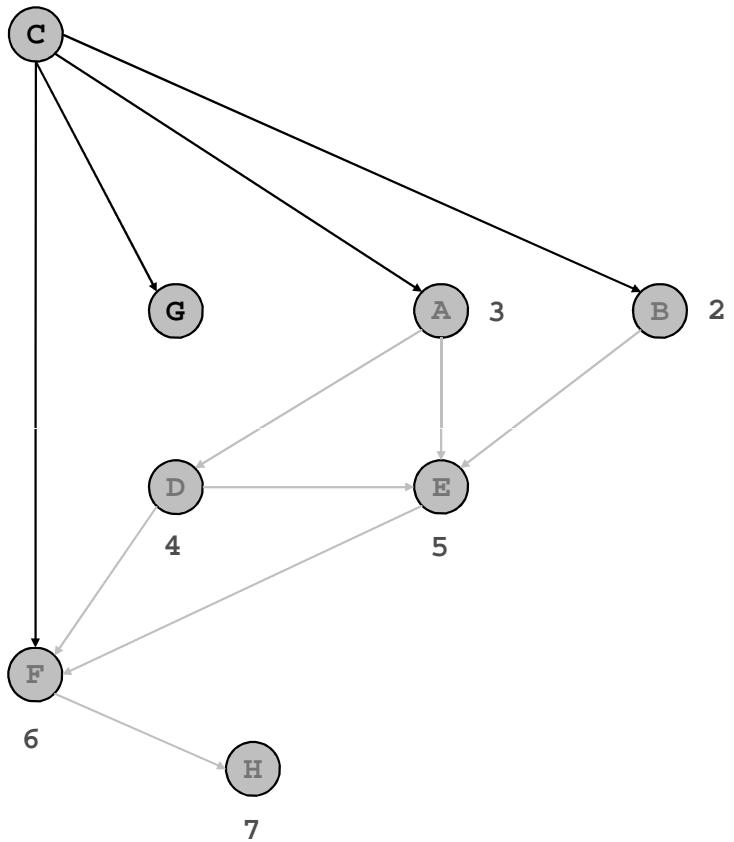
## Topological Sort: DFS



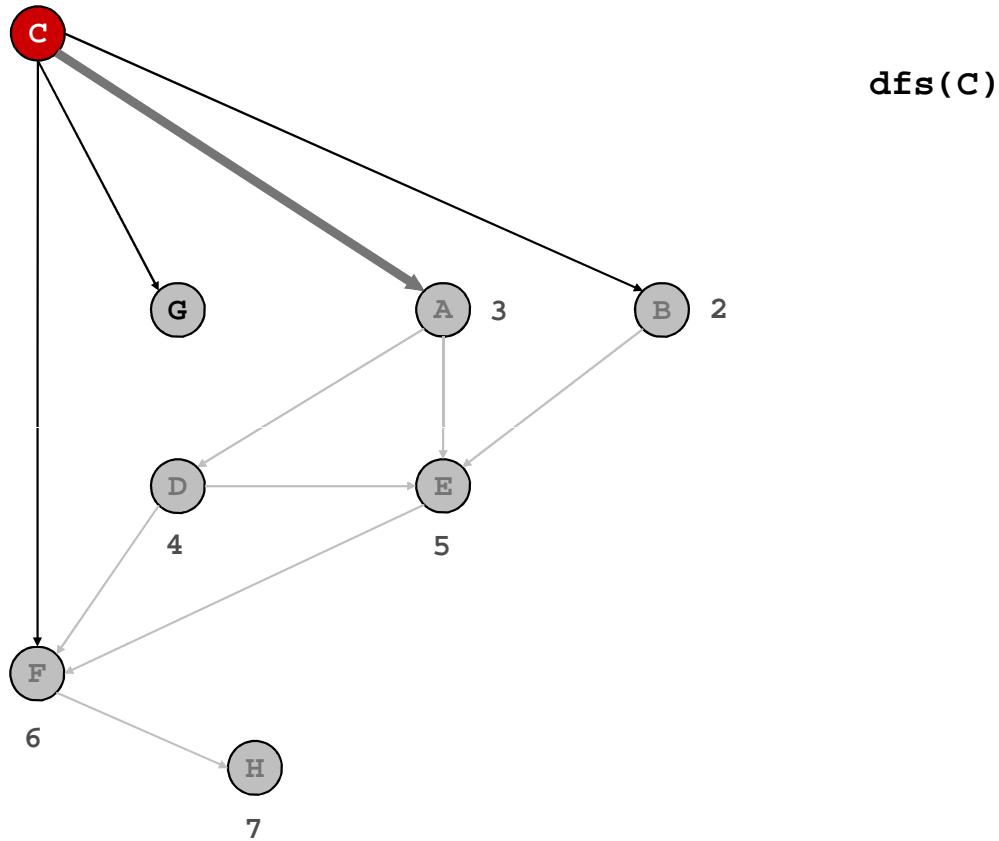
## Topological Sort: DFS



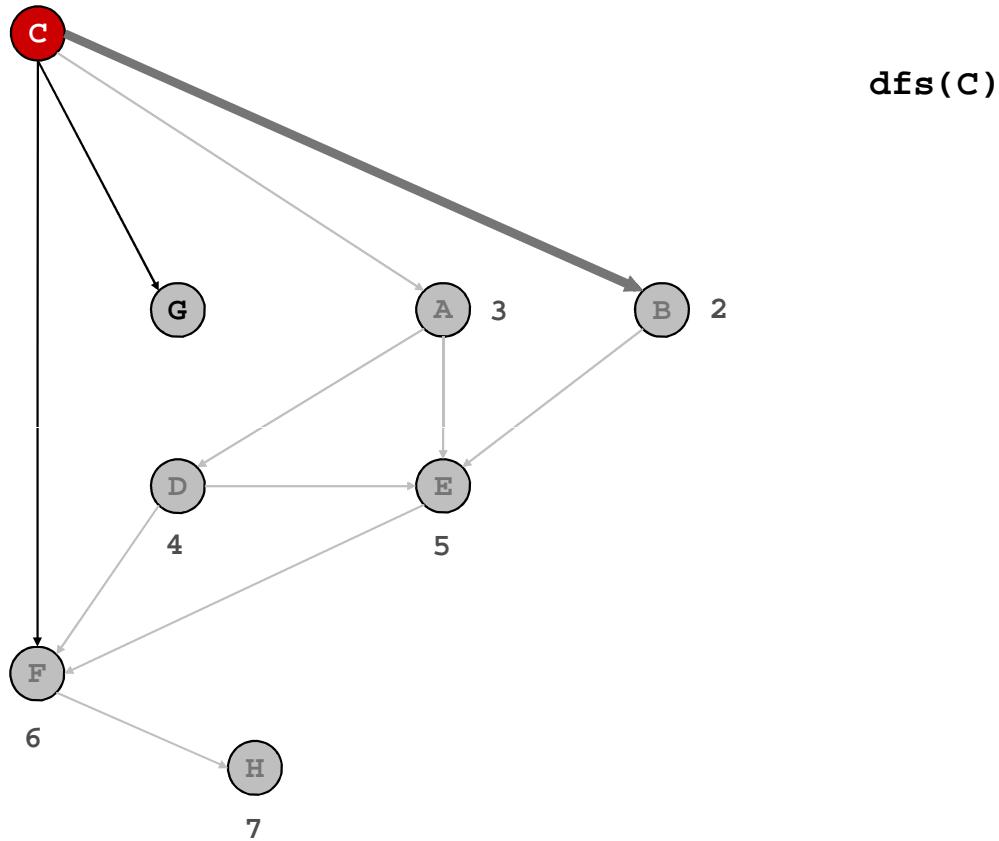
## Topological Sort: DFS



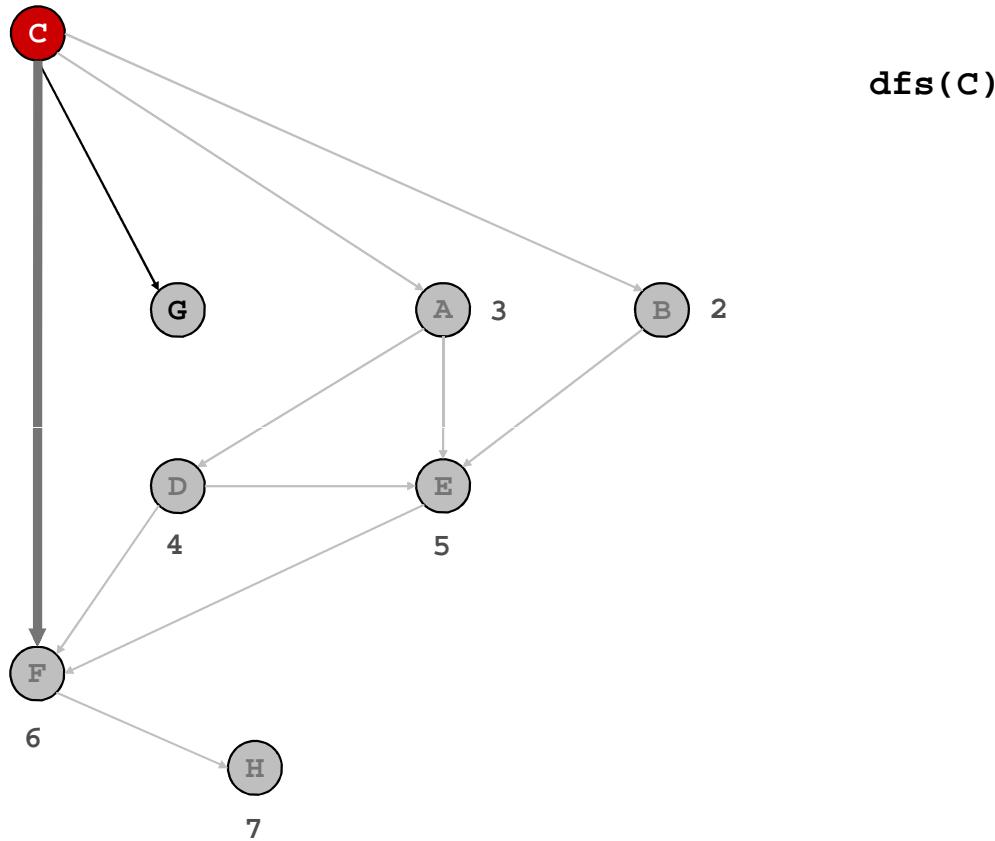
## Topological Sort: DFS



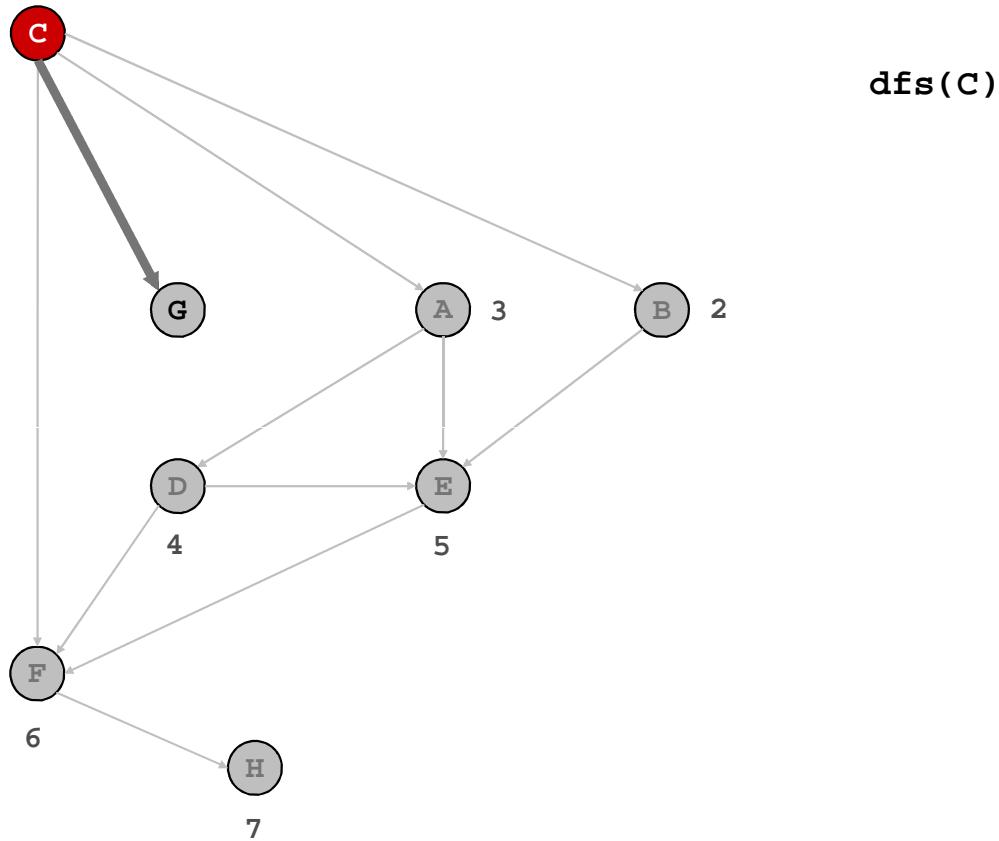
## Topological Sort: DFS



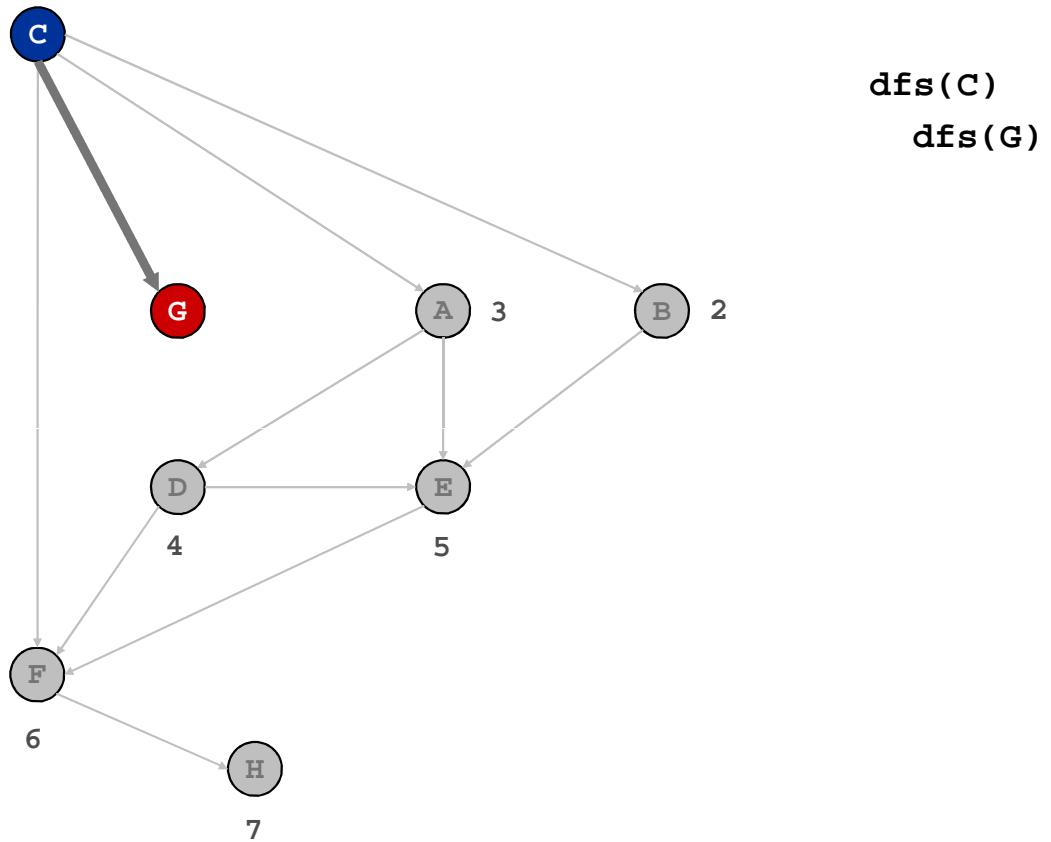
## Topological Sort: DFS



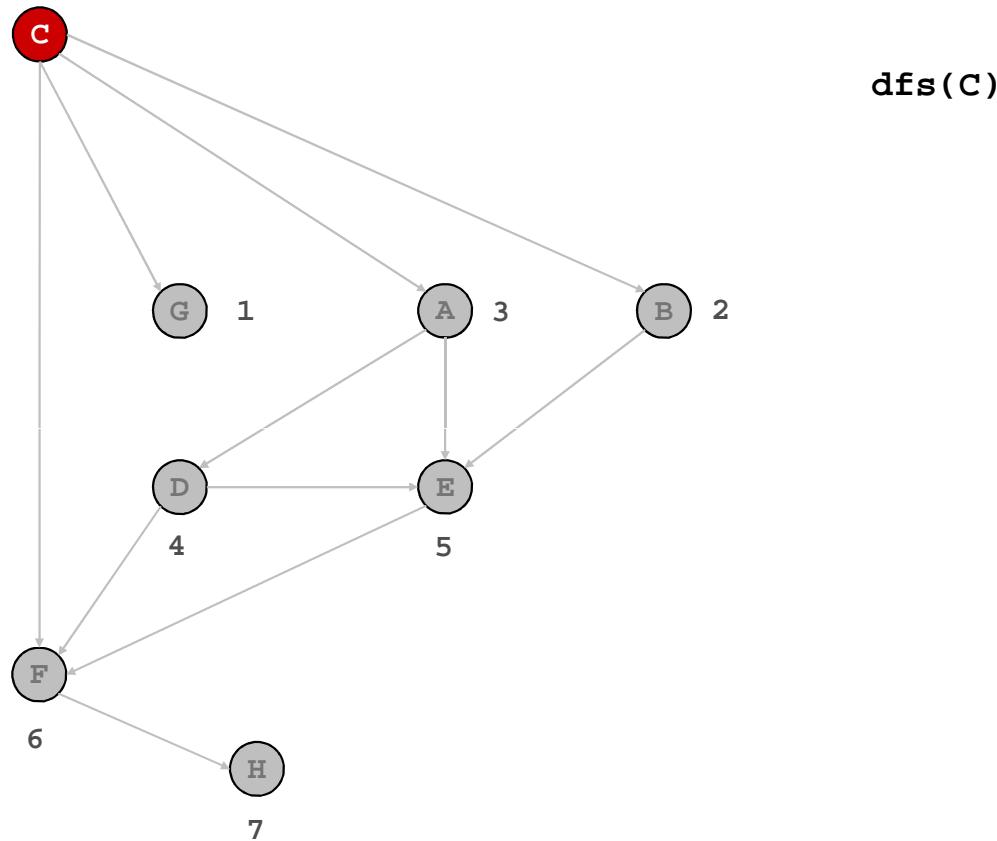
## Topological Sort: DFS



## Topological Sort: DFS

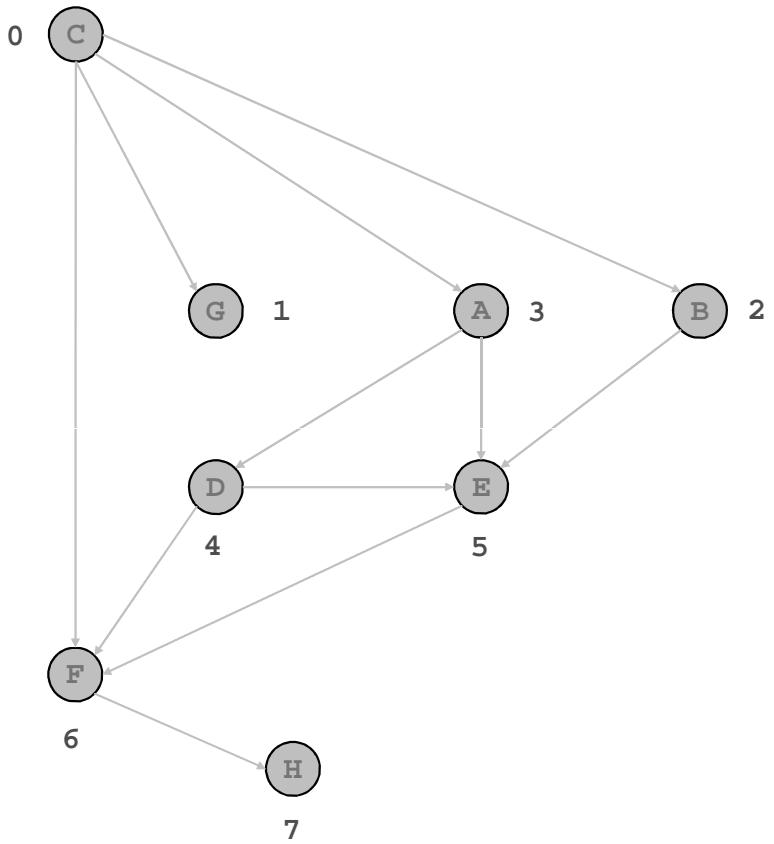


## Topological Sort: DFS

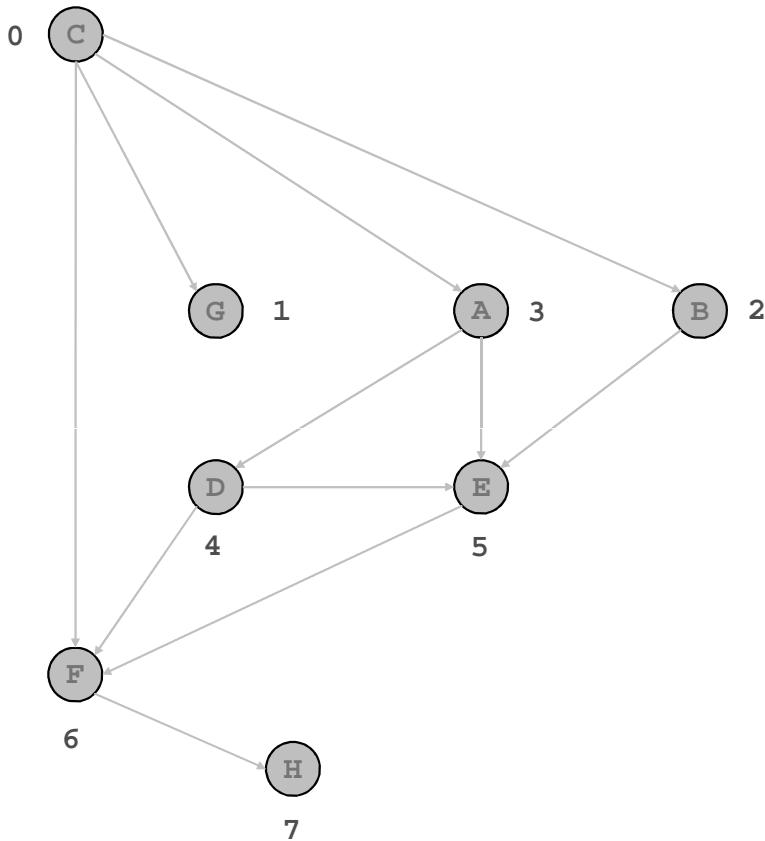


`dfs(C)`

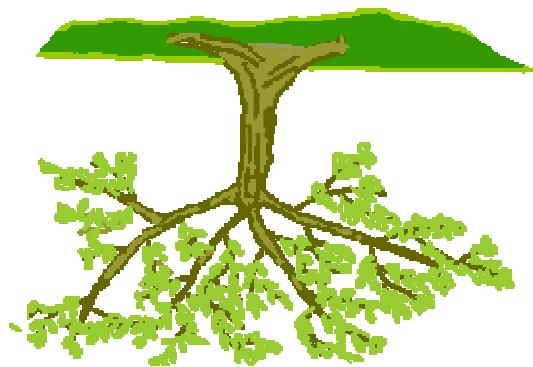
## Topological Sort: DFS



## Topological Sort: DFS

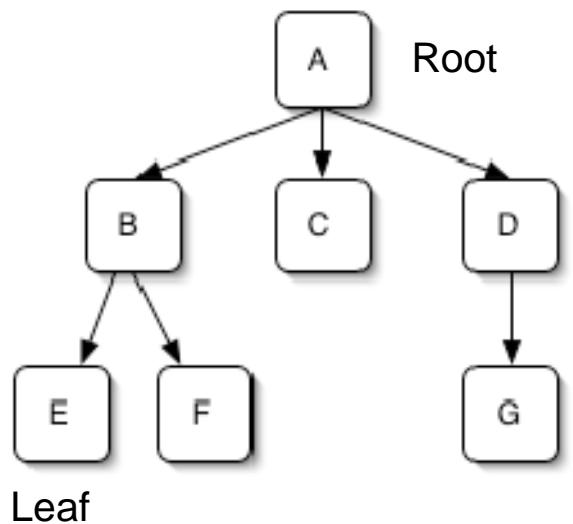


Topological order: **C G B A D E F H**



# Tree

## Data Structure

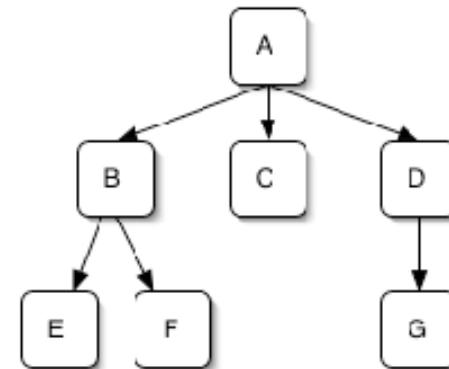


# Tree

1. Is a connected acyclic graph
2. Has a Root
3. Each node may have subtrees

# Tree vocabulary

1. **Root:** A (only node with no parent)
2. **Children(B) = {E, F}**
3. **Siblings(X) = {Nodes with the same parent as X, excluding X}**
4. **Siblings(B) = {C, D}, Siblings(A) = {}**
5. **Descendants(X) = {Nodes below X}**
6. **Descendants(A) = {B,C,D,E,F,G}**
7. **Ancestors(X) = {Nodes between X and the root}**
8. **Ancestors(E) = {B, A}**
9. Nodes with no children are called
10. **leaves or external or *terminal* nodes:** {C, E, F, G}
11. **Internal nodes:** Nodes with children (vertices other than leaves {A, B, D})
12. The **subtree rooted at X** is the tree of all descendants of X, including X.
13. **Branching factor:** number of children



# Tree

- Depth or Level of a vertex is length of the path from the root to the vertex.
  - Root depth = 0
  - $\text{Depth}(x)$  = number of ancestors of  $x$ .
  - $\text{Depth}(x) = 1 + \text{Depth}(\text{parent}(x))$
- Height of a node  $x$  is the length of the longest path from the vertex to a leaf.
- Height of a tree is Height of root

# Path

Path to a node **x** is a sequence of nodes from root to x. How many paths are there to a node?

Height of a node is the length of the LONGEST path from the node to its leaves.

Height of a leaf is 0

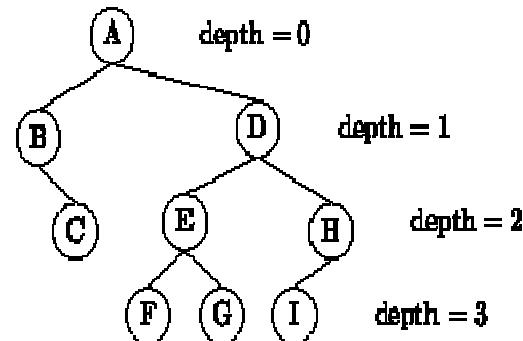
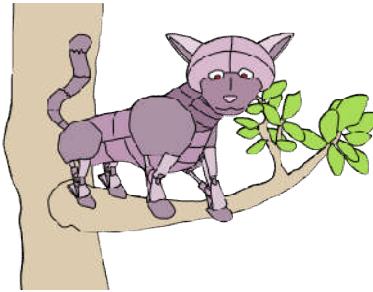
# Tree Traversals

- visit all nodes of a tree, starting from the root. *Use recursion*
  - each element of the tree is **visited** exactly once.
  - **visit** of an element, indicates **action** (*print value, evaluate the operator, etc.*)
- Pre-order traversal: (root → left → right)
- In-order traversal: (left → root → right)
- Post-order traversal: (left → right → root)

**Example of recursive function  
for preorder tree traversal.**

```
preOrder(T) {  
    if(! T) return;  
    visit(T);  
    preOrder(T.left);  
    preOrder(T.right);  
}
```

Pre-order traversal: (root → left → right)  
In-order traversal: (left → root → right)  
Post-order traversal: (left → right → root)

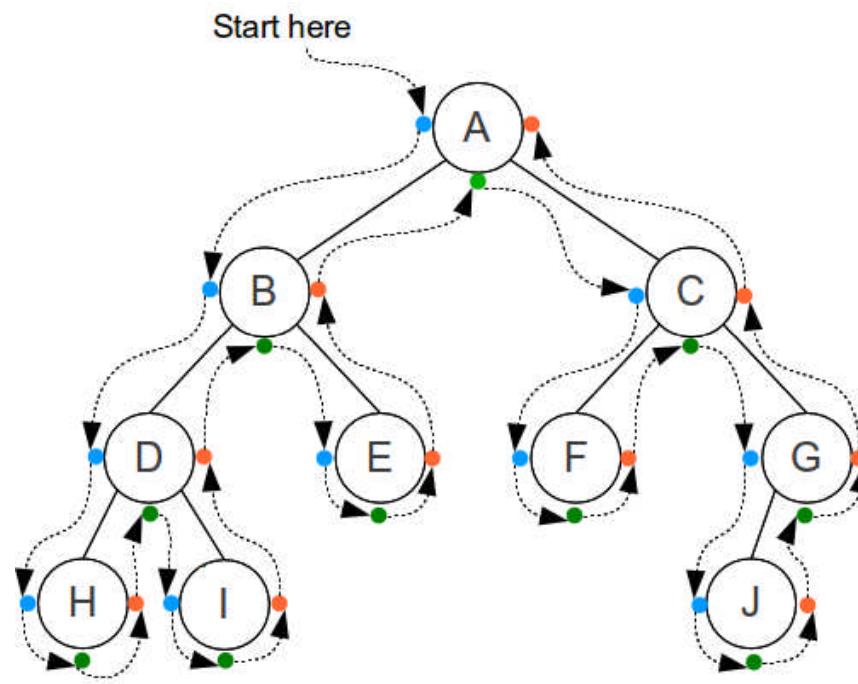


**Preorder:** ABCDEFGHI

**Postorder:** CBFGEIHDA

**Inorder:** BCAFEGDIH

# Traversal example

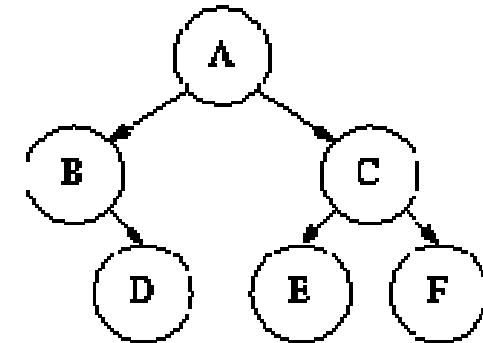


Pre-Order      ABDHIECFGJ

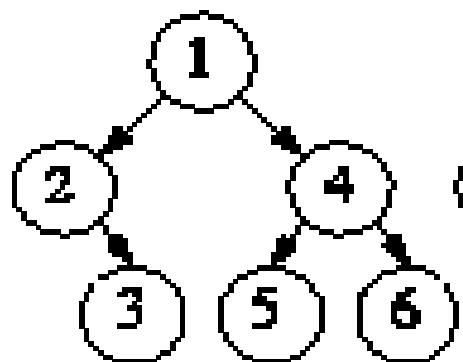
In-Order        HDIBEAFCJG

Post-Order      HIDEBFJGCA

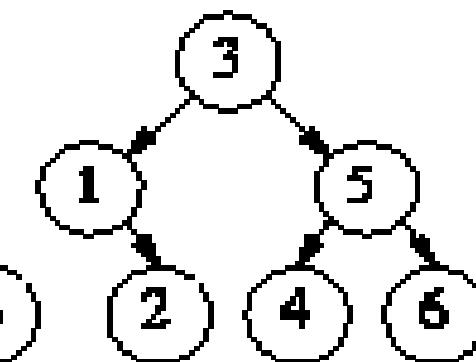
# Tree Traversals



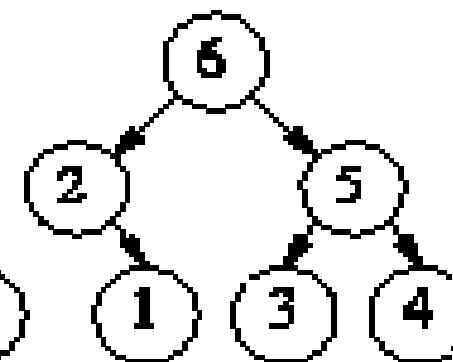
Pre-Order



In-Order



Post-Order

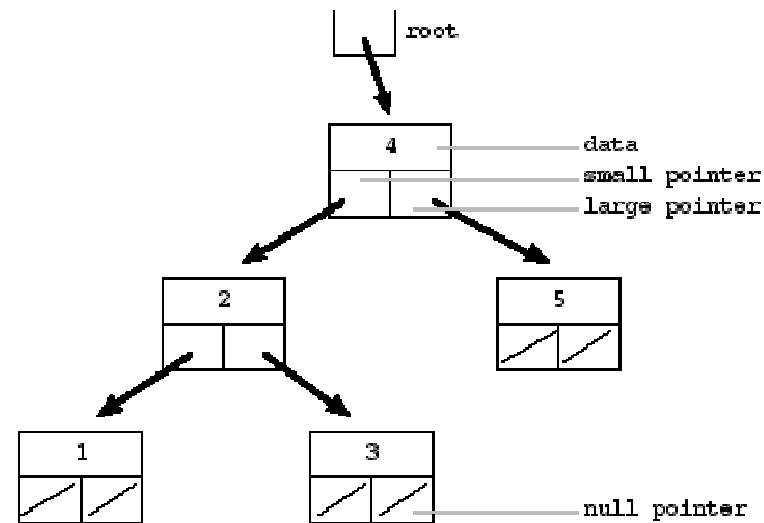


Pre-Order: A-B-D-C-E-F

In-Order: B-D-A-E-C-F

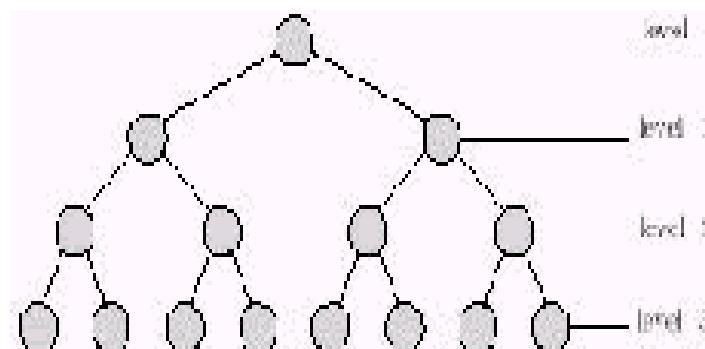
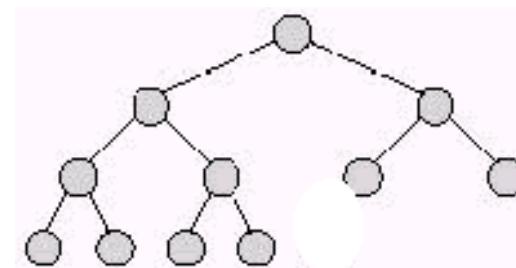
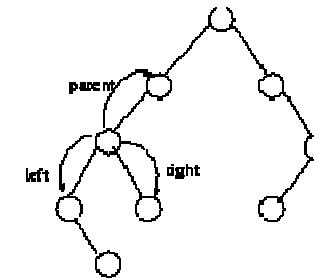
Post-Order: D-B-E-F-C-A

# Binary tree data structure



# Binary Tree

- **Binary Tree** is a tree data structure in which each node has **at most two children**. *the child nodes are called left and right*
- **Full Binary Tree:** If each internal node has exactly two children.
- **Complete Binary Tree:** If it is full and all its leaves are on the same level (i.e. have the same depth)

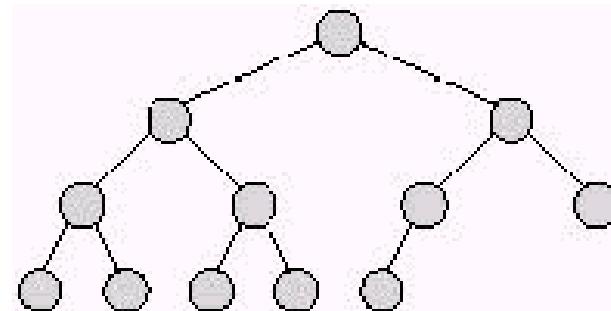


# Almost Complete Binary Tree (ACBT)

- complete binary tree.

Or

- complete binary tree and conditions:
  - All levels are complete except the lowest one.
  - In the last level empty spaces are towards the right.
- If Binary tree has height **k**, then it is between  **$O(\log n)$  and  $n-1$**
- **The height of a complete or almost-complete binary tree with  $n$  vertices is  $\lceil \log n \rceil$ .**
- **In a full binary tree, the number of leaves is equal to the number of internal vertices plus one.**



# Height of trees

Let  $n$  be the number of vertices in a binary tree  $T$  of height  $k$ .

If  $T$  is complete then:

$$n \leq \sum_{j=0..k} 2^j = 2^{k+1} - 1.$$

If  $T$  is almost-complete then:

$$2^k \leq n \leq 2^{k+1} - 1$$

# Storage of Binary Trees in arrays

- Resulting from sequentially numbering the nodes
- Start with the root at 1
  - `Parent(i)` at `int(i/2)`
  - `Left(i)` at `2*i`
  - `right(i)` at `2*i + 1`

# Binary search tree

A binary search tree is a binary tree such that: If  $x$  is a node with key value  $\text{key}[x]$ , If each node of a tree has the following *Binary Search Tree properties*:

- for all nodes  $y$  in left subtree of  $x$ ,  $\text{key}[y] \leq \text{key}[x]$
- for all nodes  $y$  in right subtree of  $x$ ,  $\text{key}[y] \geq \text{key}[x]$

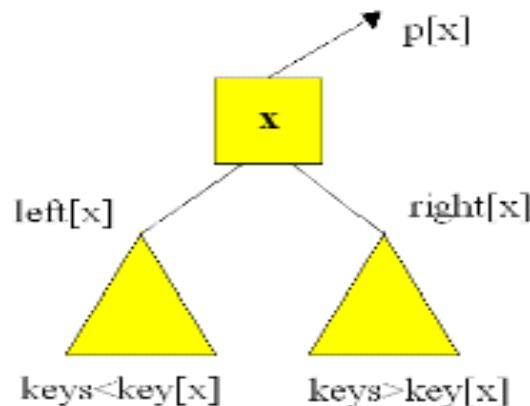
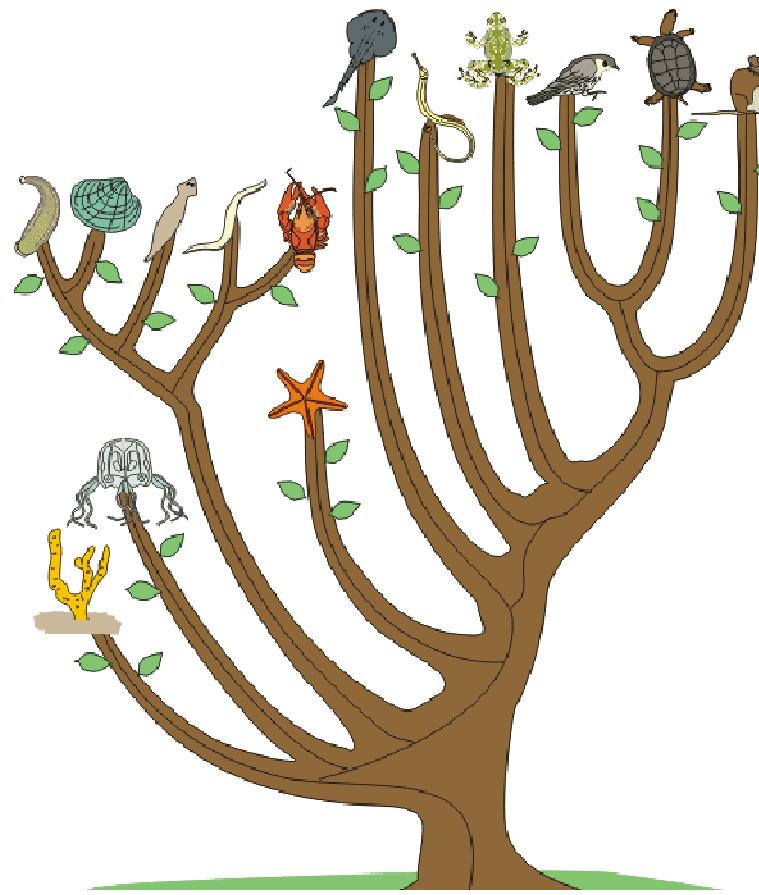


Fig. 1

# Special Trees

- Red black
- Kd trees
- B-trees
- 2-3 trees
- AVL trees
- Fibonacci





# Type of Trees

# Binary search tree (BST)

- BST also called an **ordered** or **sorted binary tree**, is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - Both the left and right subtrees must also be binary search trees.
  - There must be no duplicate nodes.

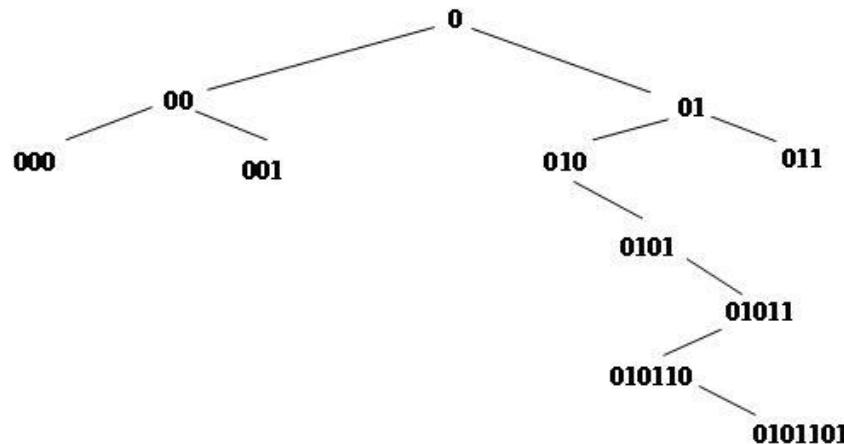
# BST

- The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.
- Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

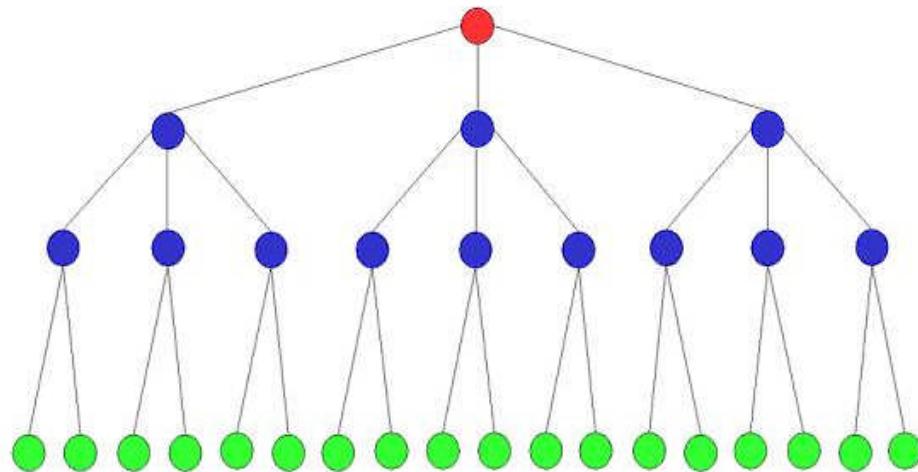
# *Optimal binary search tree*

- If we do not plan on modifying a search **tree**, and
- we know exactly how often each item will be accessed,
- we can construct an *optimal binary search tree*, which is a search **tree**
- where the average cost of looking up an item (the *expected search cost*) is minimized.

# Binary tree get unbalanced ( $O(n)$ to access nodes)



# Balanced tree for faster ( $O(\log n)$ ) to access nodes)



# Balancing trees

- Recall that, for binary-search trees, although the average-case times for the lookup, insert, and delete methods are all  $O(\log N)$ , where  $N$  is the number of nodes in the **tree**, the worst-case time is  $O(N)$ .
- We can **guarantee**  $O(\log N)$  time for all three methods by using a **balanced tree** -- a **tree** that always has height  $O(\log N)$ -- instead of a binary-search **tree**.
  - A number of different balanced trees have been defined, including **AVL trees**, **red-black trees**, and **B trees**.

# Balanced trees

- Balanced search trees are found in many flavors and have been the major data structure used for structures called dictionaries, where insertion, deletion, and searching must take place.
- In 1970, John Hopcroft introduced 2-3 search trees as an improvement on existing balanced binary trees.
- Later they were generalized to B-trees by Bayer and McCreight.
- B-trees were then simplified by Bayer to form red-black trees.

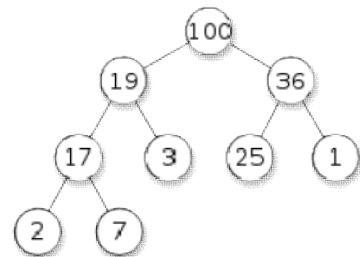
# Different types of trees

V · T · E	Trees in computer science
<b>Binary trees</b>	Binary search tree (BST) · Cartesian tree · MVP Tree · Top tree · T-tree
<b>Self-balancing binary search trees</b>	AA tree · AVL tree · LLRB tree · Red–black tree · Scapegoat tree · Splay tree · Treap
<b>B-trees</b>	B+ tree · B*-tree · B <sup>X</sup> -tree · UB-tree · 2-3 tree · 2-3-4 tree · (a,b)-tree · Dancing tree · Htree
<b>Tries</b>	Suffix tree · Radix tree · Ternary search tree · X-fast trie · Y-fast trie
<b>Binary space partitioning (BSP) trees</b>	Quadtree · Octree · k-d tree · Implicit k-d tree · VP tree
<b>Non-binary trees</b>	Exponential tree · Fusion tree · Interval tree · PQ tree · Range tree · SPQR tree · Van Emde Boas tree
<b>Spatial data partitioning trees</b>	R-tree · R+ tree · R* tree · X-tree · M-tree · Segment tree · Hilbert R-tree · Priority R-tree
<b>Other trees</b>	Heap · Hash tree · Finger tree · Order statistic tree · Metric tree · Cover tree · BK-tree · Doubly chained tree · iDistance · Link-cut tree · Fenwick tree

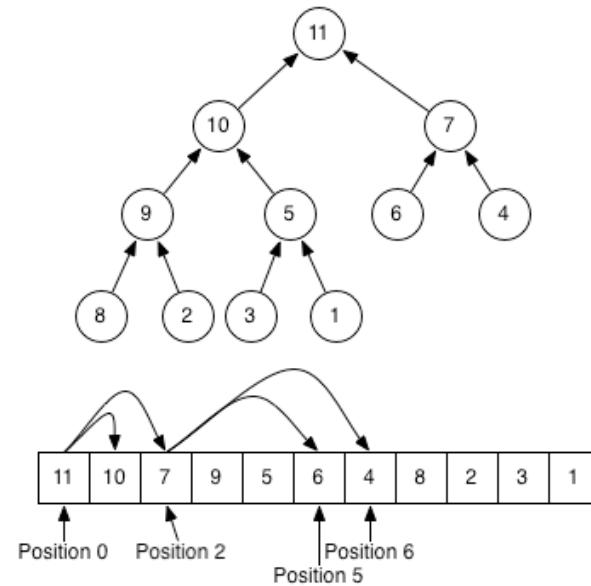
# Heap

A **heap** is a [tree-based data structure](#) that satisfies the *heap property*: If  $A$  is a parent [node](#) of  $B$  then  $\text{key}(A)$  is ordered with respect to  $\text{key}(B)$  with the same ordering applying recursively.

A *heap* data structure should not be confused with the *heap* which is a common name for [dynamically allocated memory](#).

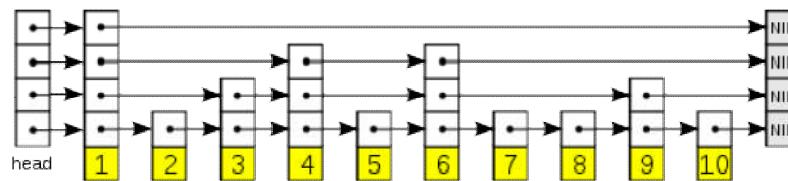


Max heap



# Skip lists (a probabilistic alternative to balanced trees )

- A **skip list** is a [data structure](#) for storing a sorted [list](#) of items using a hierarchy of [linked lists](#) that connect increasingly sparse subsequences of the items. These auxiliary lists allow item lookup with [efficiency](#) comparable to [balanced binary search trees](#) (that is, with number of probes proportional to  $\log n$  instead of  $n$ )

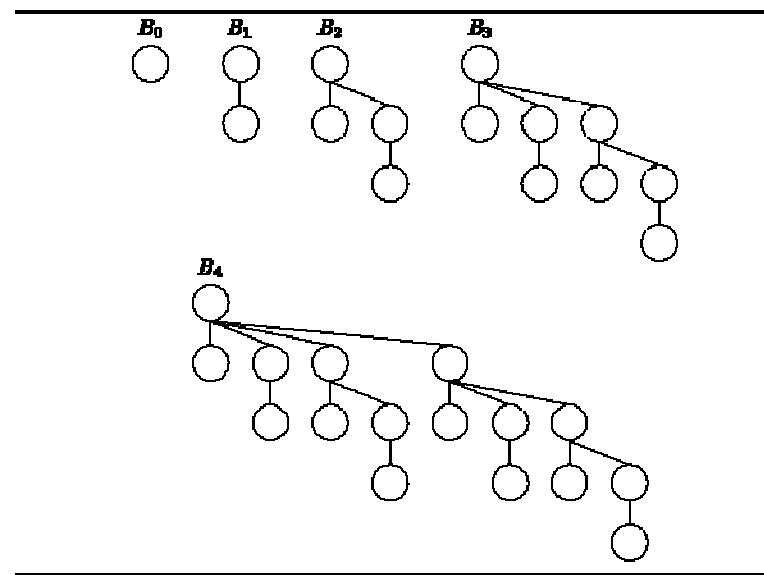


# Skip list

- *Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications.*
- *Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.*

# Binomial tree

- A binomial heap is implemented as a collection of [binomial trees](#). A **binomial tree** is defined recursively: A binomial tree of order 0 is a single node
  - A binomial tree of order  $k$  has a root node whose children are roots of binomial trees of orders  $k-1, k-2, \dots, 2, 1, 0$  (in this order).

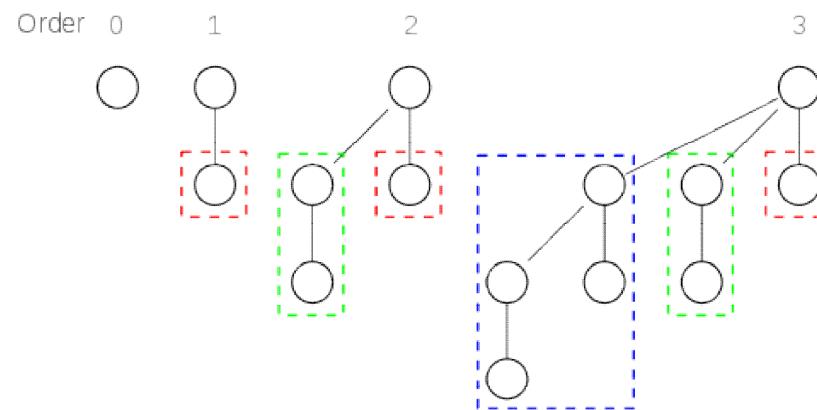


# Binomial heap

- Supports quick merging of two heaps.
- A binomial heap is implemented as a collection of [binomial trees](#) (compare with a [binary heap](#) single [binary tree](#)).
- A **binomial tree** is defined recursively:
  - A binomial tree of order 0 is a single node
  - A binomial tree of order  $k$  has a root node whose children are roots of binomial trees of orders  $k-1, k-2, \dots, 2, 1, 0$  (in this order).
- Properties
  - A binomial tree of order  $k$  has  $2^k$  nodes, height  $k$ .
  - binomial heap with  $n$  nodes consists of at most [log](#)  $n + 1$  binomial trees

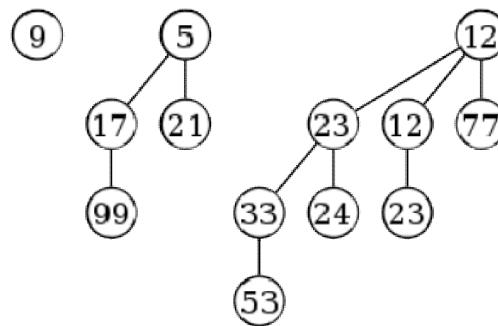
# Example of binomial trees

- Binomial trees of order 0 to 3:
- Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted.
- For example, the order 3 binomial tree is connected to an order 2, 1, and 0 (highlighted as blue, green and red respectively) binomial tree.



# Example of binomial heap

- *Example of a binomial heap containing 13 nodes with distinct keys.*
- *The heap consists of three binomial trees with orders 0, 2, and 3.*



# Binomial heap

- A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:
- Each binomial tree in a heap obeys the [minimum-heap property](#): the key of a node is greater than or equal to the key of its parent.
- There can only be either one or zero binomial trees for each order, including zero order.
- Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a [linked list](#), ordered by increasing order of the tree.

# Fibonacci heaps (FH)

- Invented by [Fredman](#) and [Robert E. Tarjan](#) in 1984 for use in graph algorithms.
- Using Fibonacci heaps for [priority queues](#) improves the asymptotic running time of important algorithms, such as [Dijkstra's algorithm](#) for computing the [shortest path](#) between two nodes in a graph.

# Fibonacci heap

- FH is a collection of **trees** satisfying the **minimum-heap property** (the key of a child is always greater than or equal to the key of the parent).
- The minimum key is always at the root of one of the **trees**.

# Lazy balancing of FH

- FH can be balanced lazily (postponing the work for later operations).
- As a result of a relaxed structure, some operations can take a long time while others are done very quickly.

# Fibonacci heap properties

- every node has degree at most  $O(\log n)$
- Size of subtree rooted in a node of degree  $k$  is at least  $F[k + 2]$ , where  $F[k]$  is the  $k$ th Fibonacci number.

# Uses of Fibobacci heaps

## Fibonacci Heaps

Fibonacci heap history. **Fredman and Tarjan (1986)**

- Ingenious data structure and analysis.
- Original motivation:  $O(m + n \log n)$  shortest path algorithm.
  - also led to faster algorithms for MST, weighted bipartite matching
- Still ahead of its time.

Fibonacci heap intuition.

- Similar to binomial heaps, but less structured.
- Decrease-key and union run in  $O(1)$  time.
- "Lazy" unions.

# Cost of operations on heaps

## Priority Queues

Operation	Heaps			
	Linked List	Binary	Binomial	Fibonacci †
make-heap	1	1	1	1
insert	1	$\log N$	$\log N$	1
find-min	N	1	$\log N$	1
delete-min	N	$\log N$	$\log N$	$\log N$
union	1	N	$\log N$	1
decrease-key	1	$\log N$	$\log N$	1
delete	N	$\log N$	$\log N$	$\log N$
is-empty	1	1	1	1

† amortized



## Tarjan says

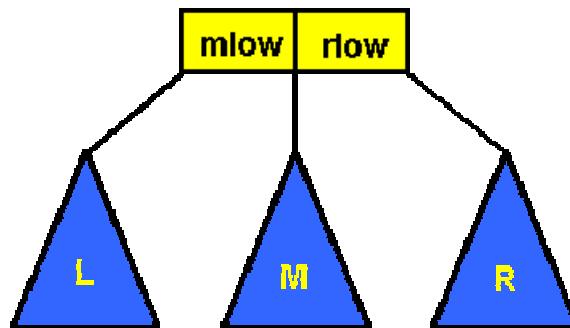
- Once you successfully write a complicated program, it always runs as is. The computer doesn't have to understand the algorithm, it just follows the steps.
- E.g. Scientists are still using fast fortran programs from 1950s.



R. E. Tarjan

# 2-3 Tree

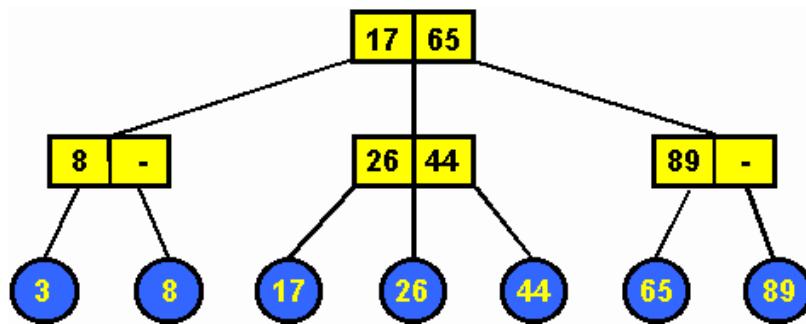
- **2–3 tree** is a type of data structure, a tree where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements.



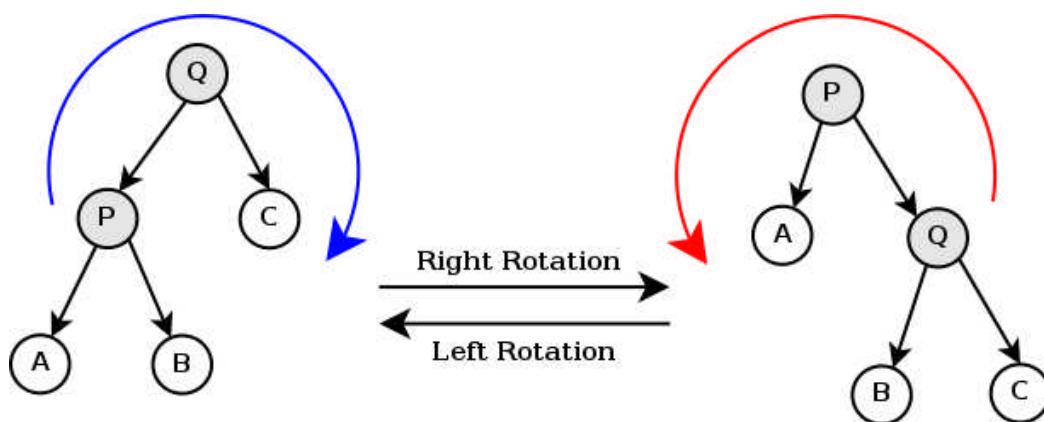
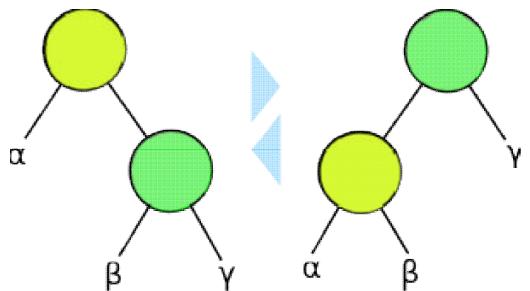
# Rules for 2-3 search trees

- *All data appears at the leaves.*
- *Data elements are ordered from left (minimum) to right (maximum).*
- *Every path through the tree is the same length.*
- *Interior nodes have two or three subtrees.*

# Example of 2-3 tree



# Tree rotation



# Rotate to balance

- A tree can be rebalanced using rotations.
- After a rotation, the side of the rotation increases its height by 1 whilst the side opposite the rotation decreases its height similarly.
- Therefore, one can strategically apply rotations to nodes whose left child and right child differ in height by more than 1.
- Self-balancing binary search trees apply this operation automatically.
- A type of tree which uses this rebalancing technique is the [AVL tree](#), Red-Black, Splay trees.

# Splay Trees

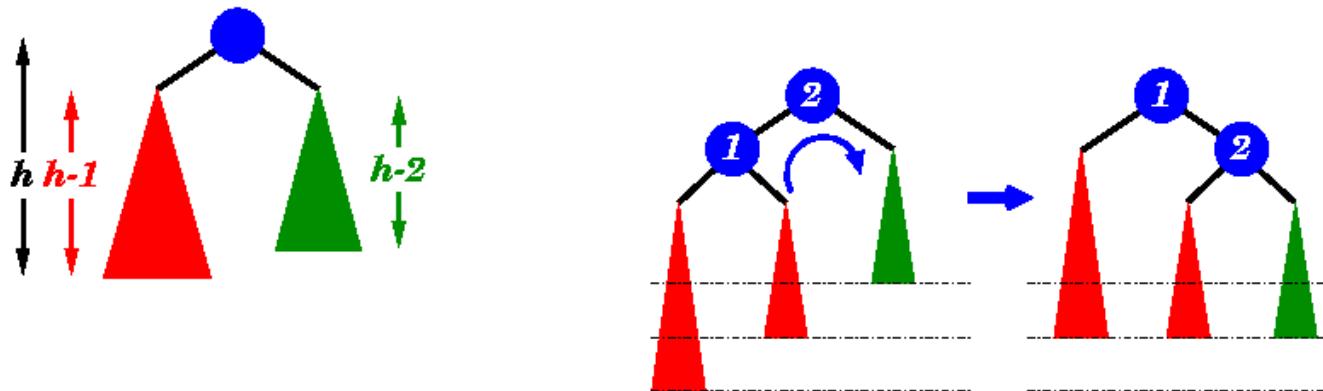
- A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.
- It performs basic operations such as insertion, look-up and removal in  $\mathcal{O}(\log n)$  amortized time.
- For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
- Invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985

# Splay trees

- First perform a standard binary tree search for the element in question, and then use [tree rotations](#) in a specific fashion to bring the element to the top.
- Simple implementation—simpler than other self-balancing binary search trees, such as [red-black trees](#) or [AVL trees](#).
- Perhaps the most significant disadvantage of splay trees is that the height of a splay tree can be linear.

# AVL Tree

- Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

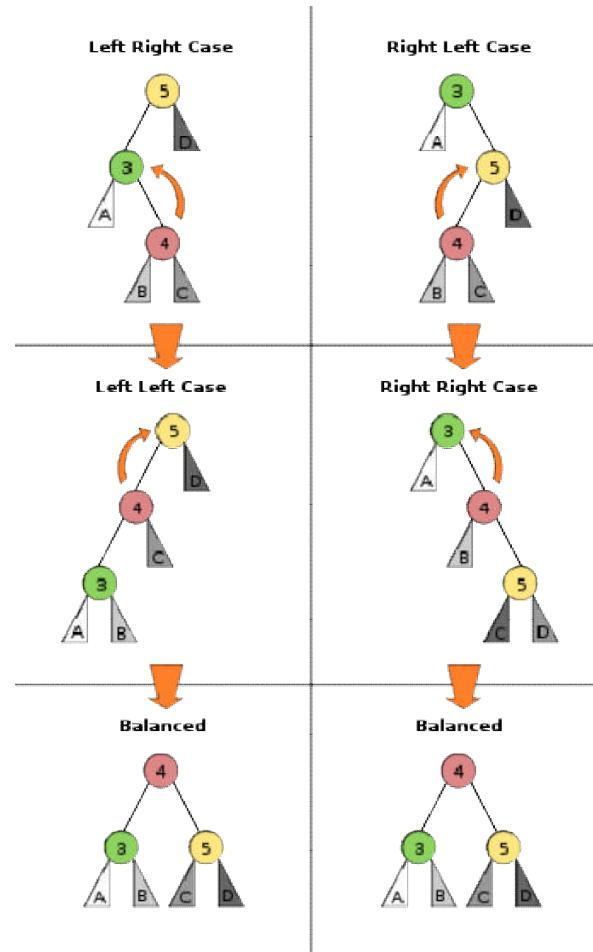


Rotation rebalance the trees

# AVL tree

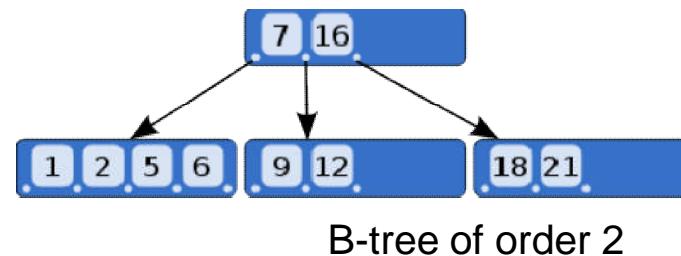
- AVL tree is a self-balancing binary search tree
- In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.
- Lookup, insertion, and deletion all take  $\mathcal{O}(\log n)$  time in both the average and worst cases, where  $n$  is the number of nodes in the **tree** prior to the operation.
- Insertions and deletions may require the **tree** to be rebalanced by one or more tree rotations.

# Rebalancing AVL by rotations



# B-Tree

- **B-tree** is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- The B-tree is a generalization of a binary search tree in that a node can have more than two children



B-tree of order 2

# B-Tree

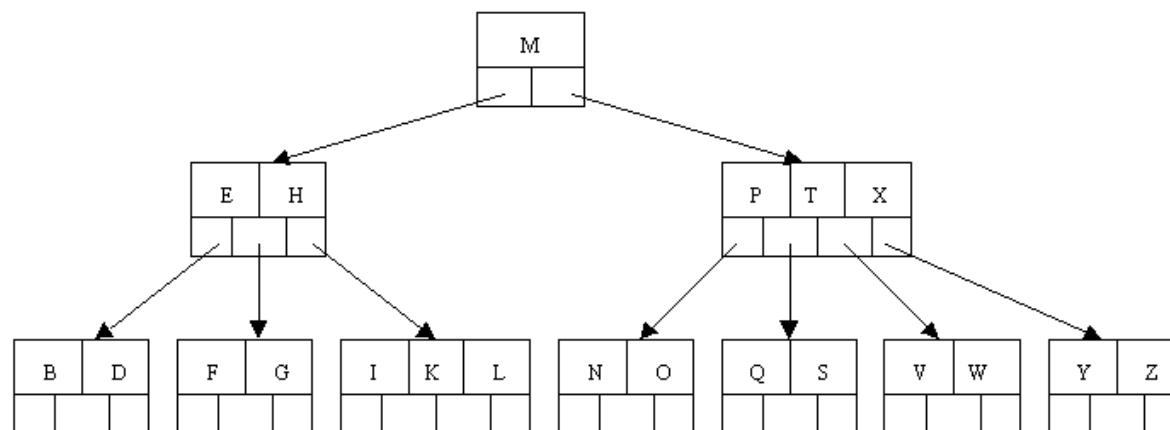
- In order to maintain the pre-defined range, internal nodes may be joined or split.
- Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees,
- but may waste some space, since nodes are not entirely full.

# B-Tree

- Each internal node of a B-tree will contain a number of keys.
- The keys act as separation values which divide its subtrees

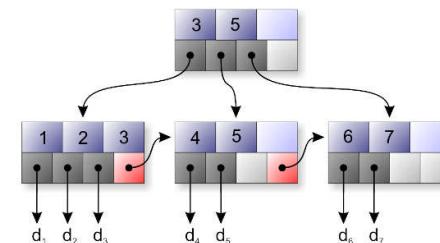
# Example of a B-tree of order 5

- This means that (other than the root node) all internal nodes have at least  $\text{ceil}(5 / 2) = \text{ceil}(2.5) = 3$  children (and hence at least 2 keys).
- The maximum number of children that a node can have is 5 (so that 4 is the maximum number of keys).
- Each leaf node must contain at least 2 keys.
- In practice B-trees usually have orders a lot bigger than 5.

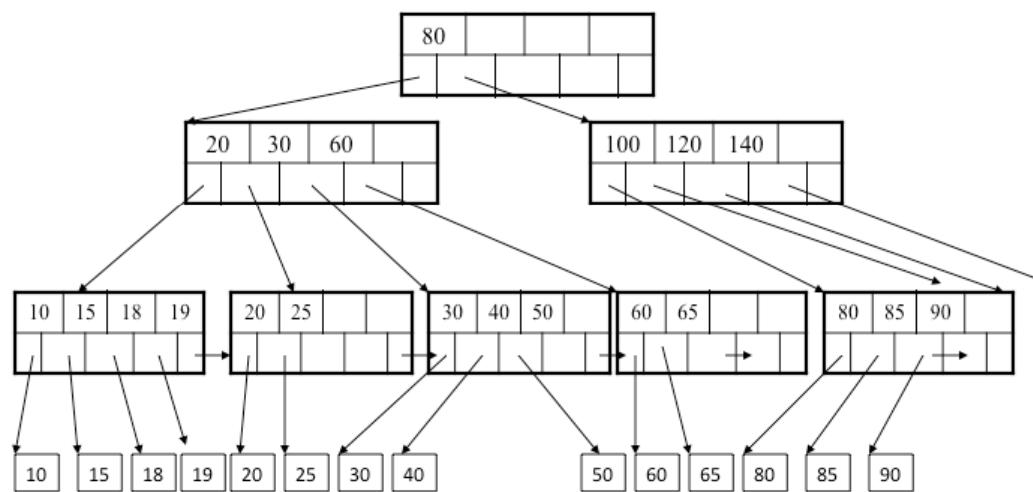


# B+ Trees

- The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context—in particular, filesystems.
- The B+-tree is used as a (dynamic) indexing method in relational database management systems.
- This is primarily because unlike binary search trees, B+ trees have very high fanout (typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the **tree**.

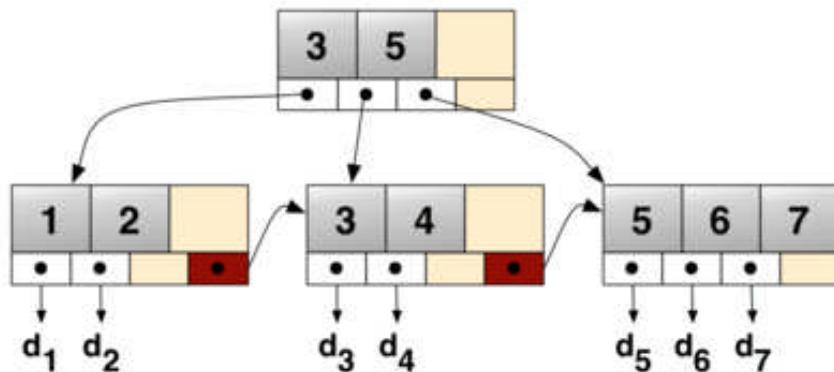


# B+ Tree example



# B+ Tree

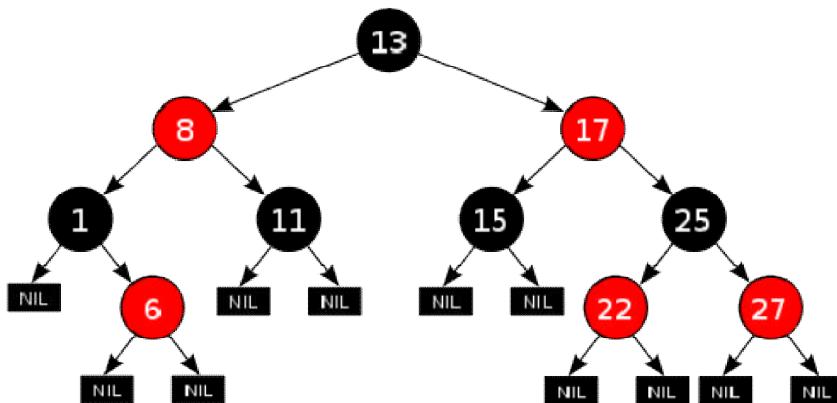
- variant of the original **B-tree** in which all records are stored in the leaves and all leaves are linked sequentially.



# Red-black tree

- A **red–black tree** is a type of self-balancing binary search tree.
- The self-balancing is provided by painting each node with one of two colors (these are typically called 'red' and '**black**', hence the name of the trees) in such a way that the resulting painted **tree** satisfies certain properties that don't allow it to become significantly unbalanced.
- When the **tree** is modified, the new **tree** is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be made efficiently.

# Example of RB tree



# Rbtree

- Tracking the color of each node requires only 1 bit of information per node because there are only two colors.
- The **tree** does not contain any other data specific to it being the red–**black tree** so its memory footprint is almost identical to classic (uncolored) binary search **tree**.
- In many cases the additional bit of information can be stored at no additional memory cost.

# Properties

- A node is either red or **black**.
- The root is **black**. (This rule is sometimes omitted. Since the root can always be changed from red to **black**, but not necessarily vice-versa, this rule has little effect on analysis.)
- All leaves (NIL) are **black**. (All leaves are same color as the root.)
- Both children of every red node are **black**.
- Every simple path from a given node to any of its descendant leaves contains the same number of **black** nodes.
- These constraints enforce a critical property of red–**black** trees: that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf.

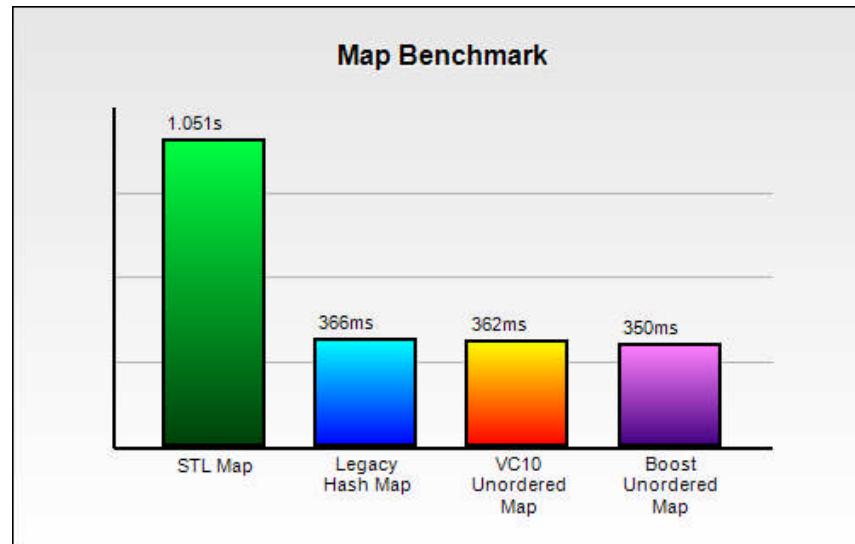
# Uses

- C++ STL Library: Map, Multimap, Set, Multiset
- Java's TreeMap class
- IBM's ISAM (Indexed Sequential Access Method)
- Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time:
- Real-time applications
- Computational geometry
- Completely Fair Scheduler in Linux kernel

# Hash is 2x faster than STL map.

Hash is unordered set.

Map is ordered set.

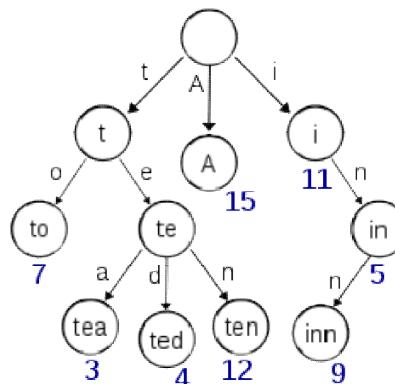


# Trie (prefix tree)

- A **trie**, (retrieval), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.
- Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.
- All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.
- Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.

# Trie (prefix tree)

- A **trie**, (retrieval), is an ordered tree data structure for associative array with strings keys.
- E.g. a trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

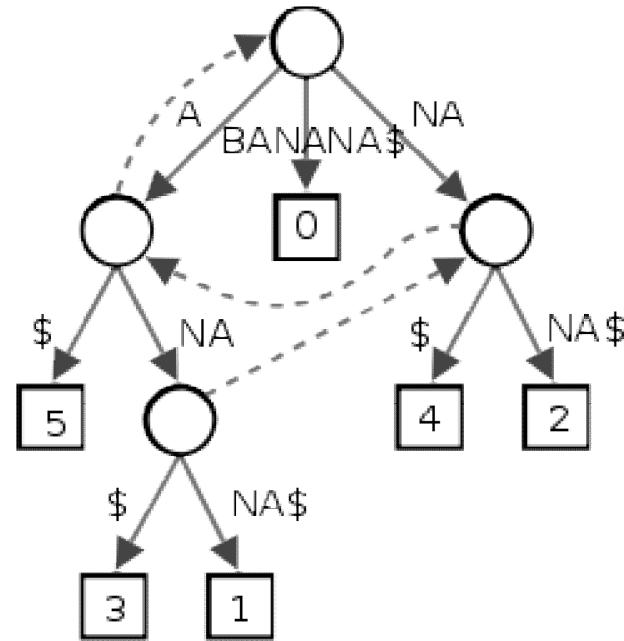


# Suffix tree

- Used for fast string matching using [Ukkonen's algorithm](#)
- Constructing Suffix Tree( $S$ ) takes linear time and linear space in  $\text{length}(S)$ .
- Once constructed, searching is fast.

## Example: Suffix tree for “BANANA”

- The six paths from the root to a leaf (boxes) are the six suffixes A\$, NA\$, ANA\$, NANA\$, ANANA\$ and BANANA\$ of
- The numbers in the leaves give the start position of the corresponding suffix.
- Suffix links drawn dashed.



# Uses of Suffix Tree

- String search, in  $O(m)$  complexity, where  $m$  is the length of the sub-string (but with initial  $O(n)$  time required to build the suffix tree for the string)
- Finding the longest repeated substring
- Finding the longest common substring
- Finding the longest palindrome in a string.
- Bio-Informatics – DNA matching.
- search efficiently with mismatches.

# Suffix tree

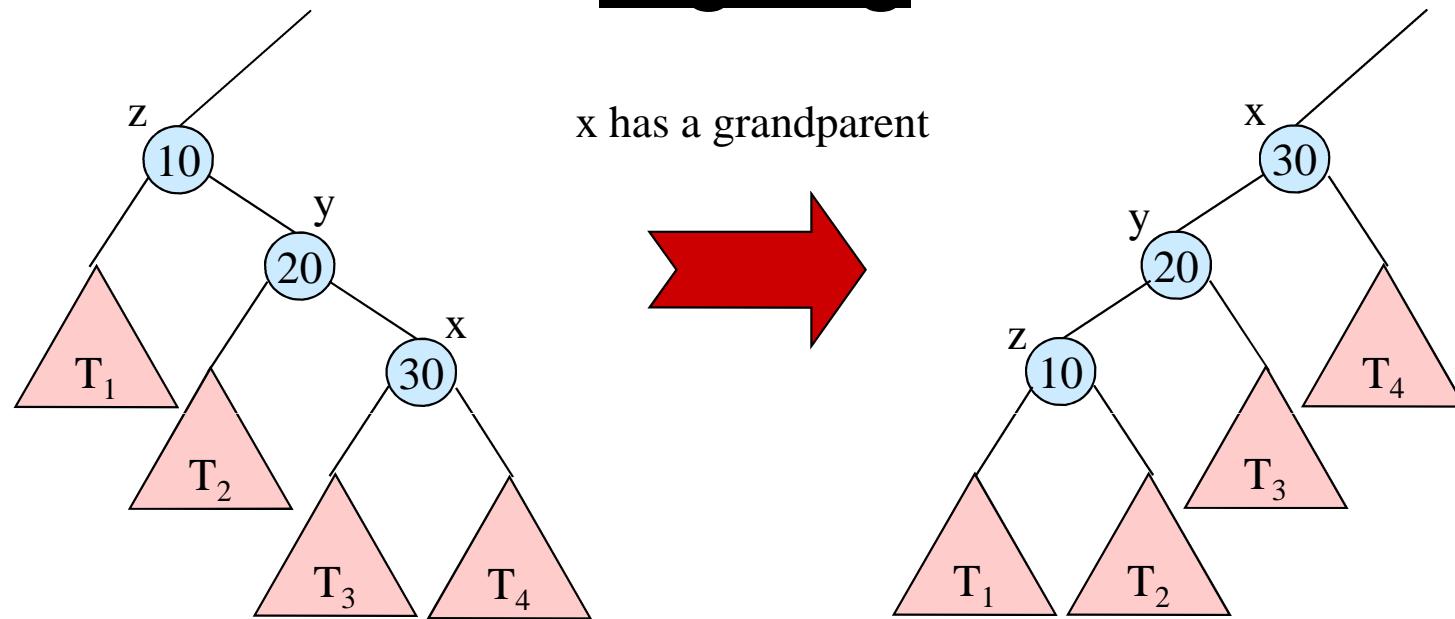
- The paths from the root to the leaves have a one-to-one relationship with the suffixes of S.
- Edges spell non-empty strings,
- Internal nodes (except perhaps the root) have at least two children.
- S padded with '\$' to mark end of string. This ensures that no suffix is a prefix of another, and that there will be n leaf nodes, one for each of the n suffixes of S.
- Generalized suffix tree is a suffix tree made for a set of words instead of only for a single word:
  - It represents all suffixes from this set of words.
  - Each word must be terminated by a different termination symbol or word.

# Splay Tree Demo

# Splay Trees

- At the end of each operation a special step called **splaying** is done.
- The splay operation moves x to the root of the tree.
- The splay operation consists of sub-operations called **zig-zig**, **zig-zag**, and **zig**.
- Splaying ensures that all operations take  $O(\lg n)$  amortized time.

## Zig-Zig

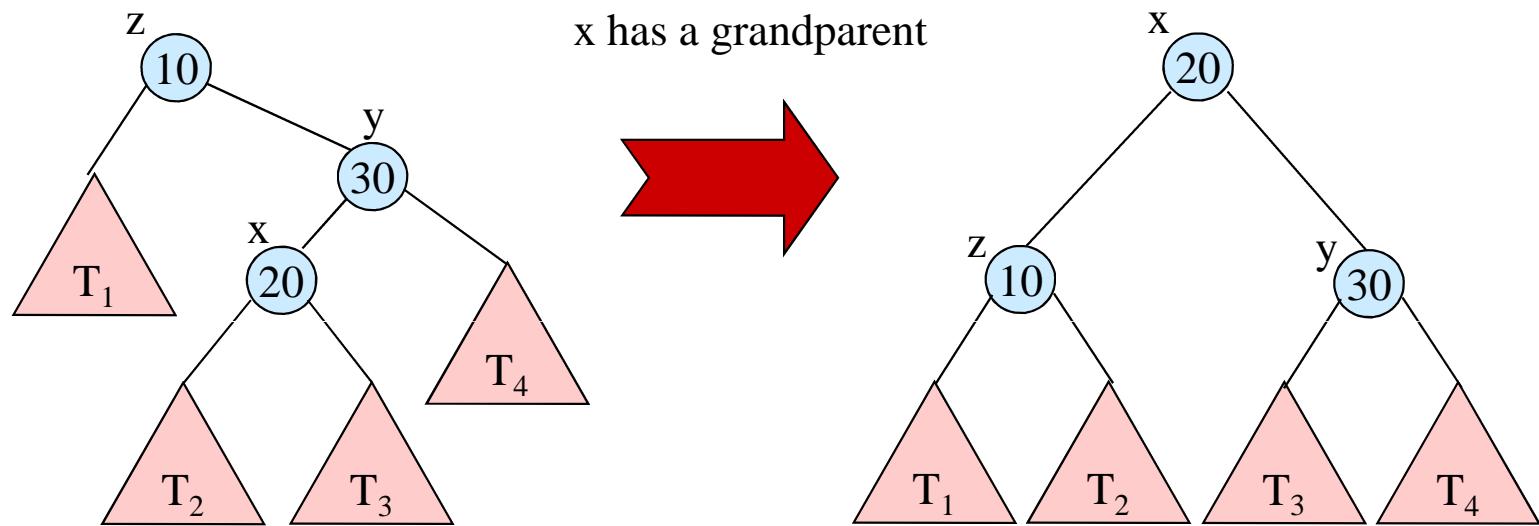


(Symmetric case too)

**Note:**  $x$ 's depth decreases by two.

Splay Trees - 3

## Zig-Zag



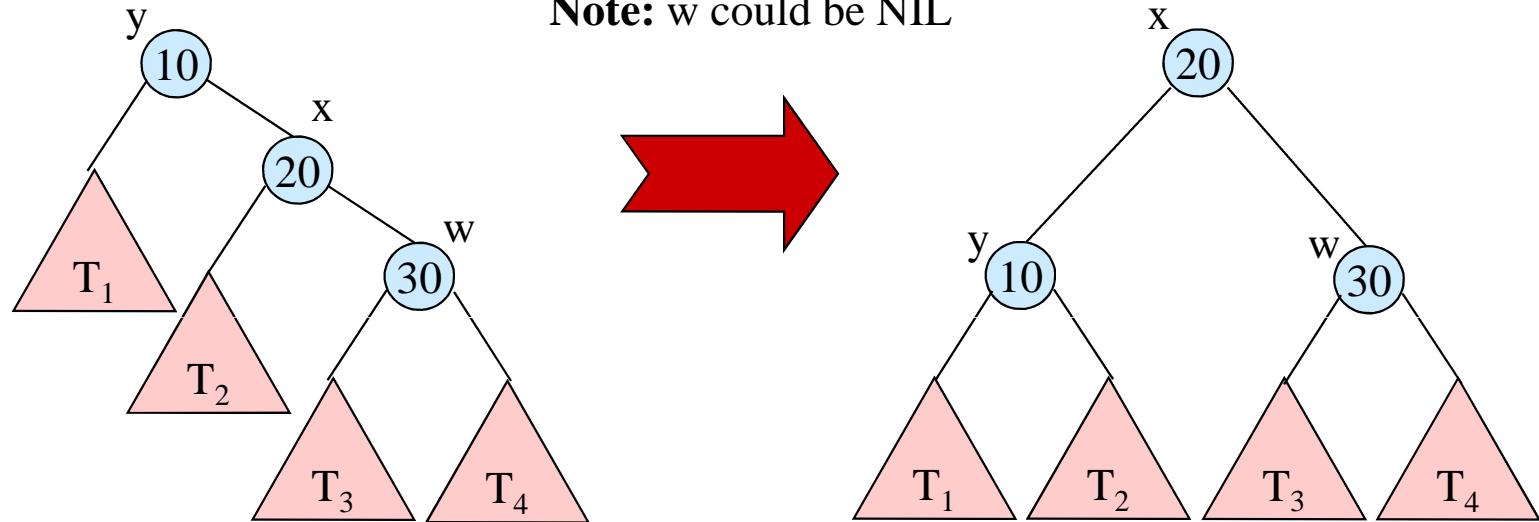
**Note:** x's depth decreases by two.

Splay Trees - 4

# Zig

x has no grandparent (so, y is the root)

**Note:** w could be NIL

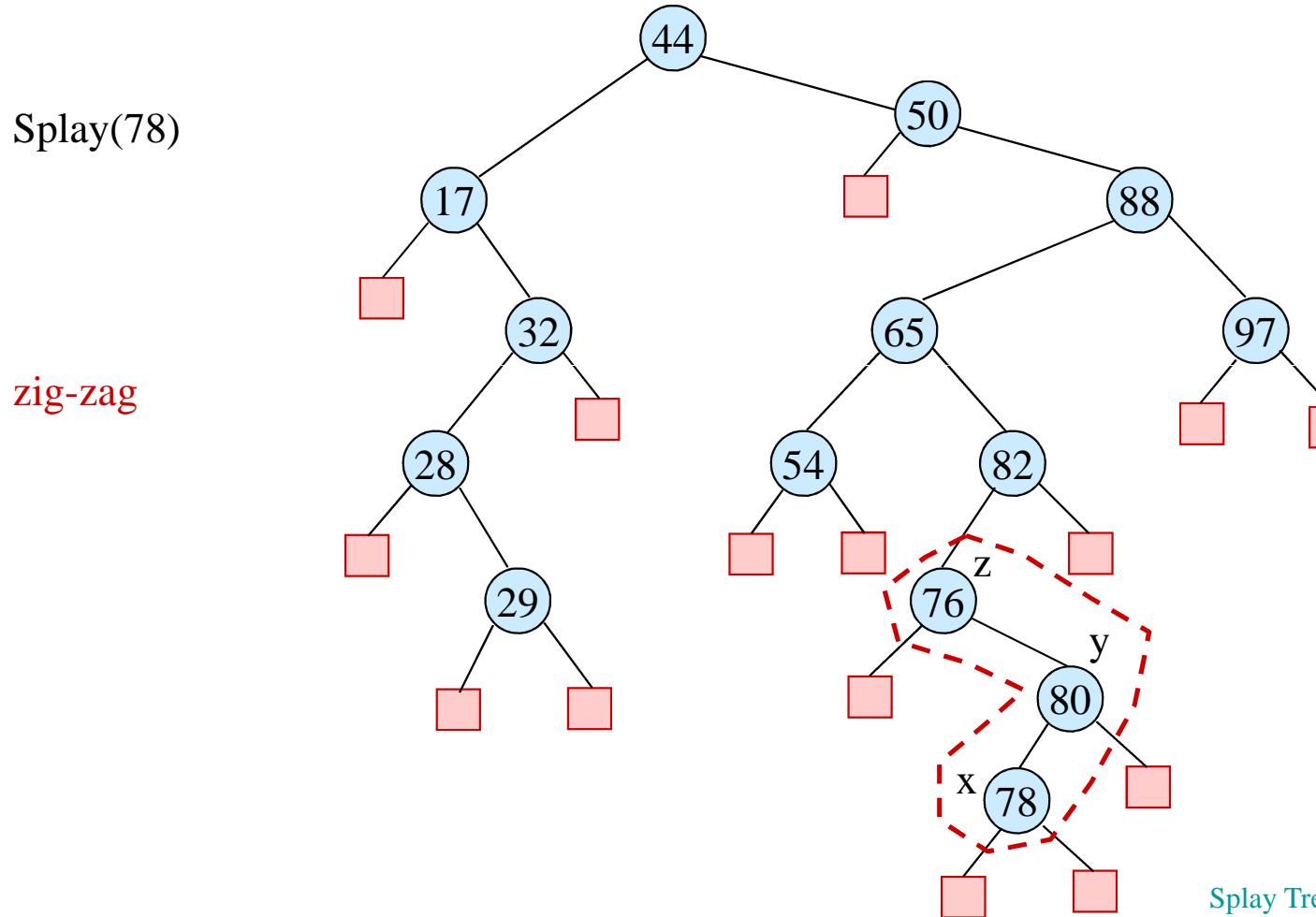


(Symmetric case too)

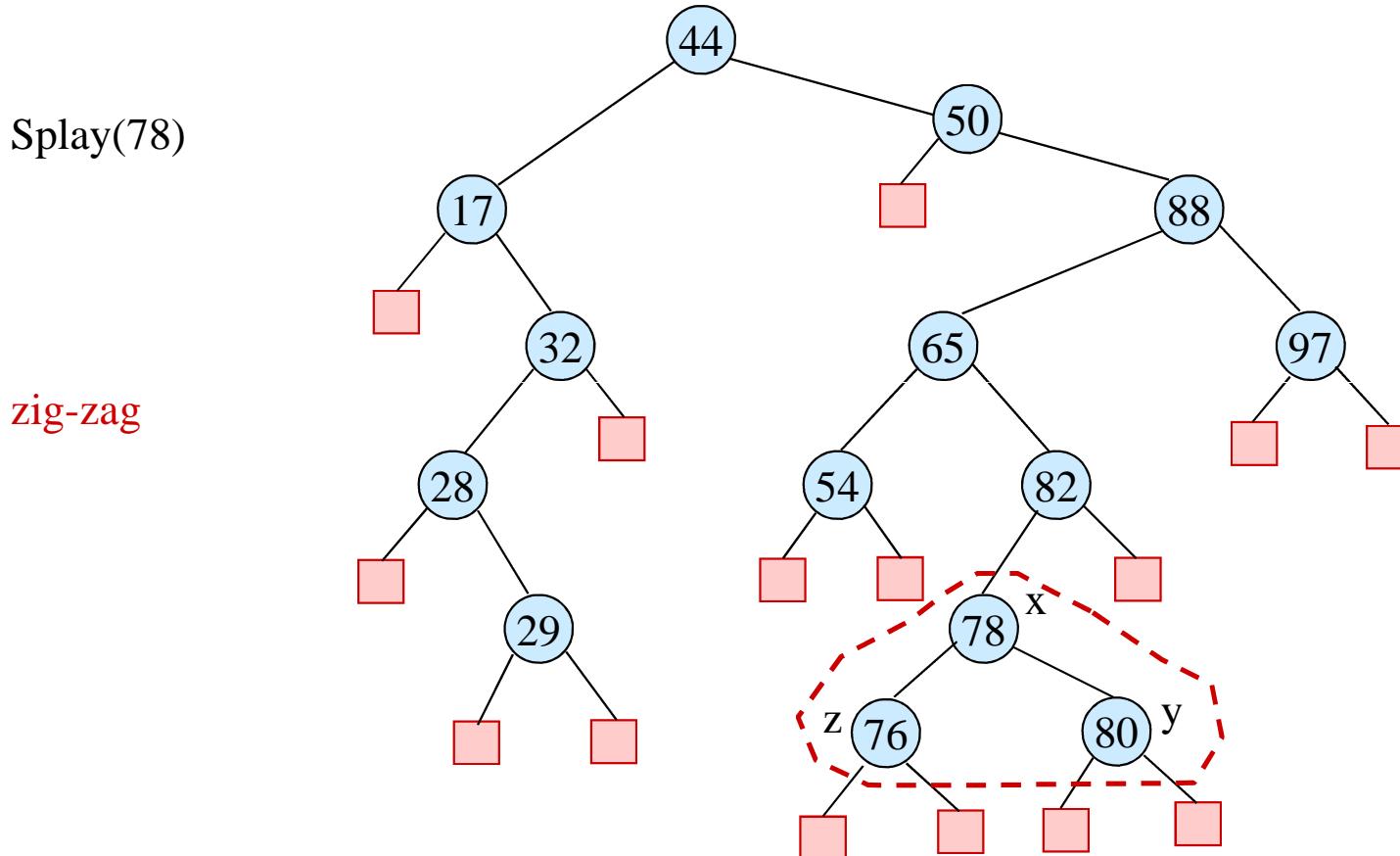
**Note:** x's depth decreases by one.

Splay Trees - 5

# Complete Example

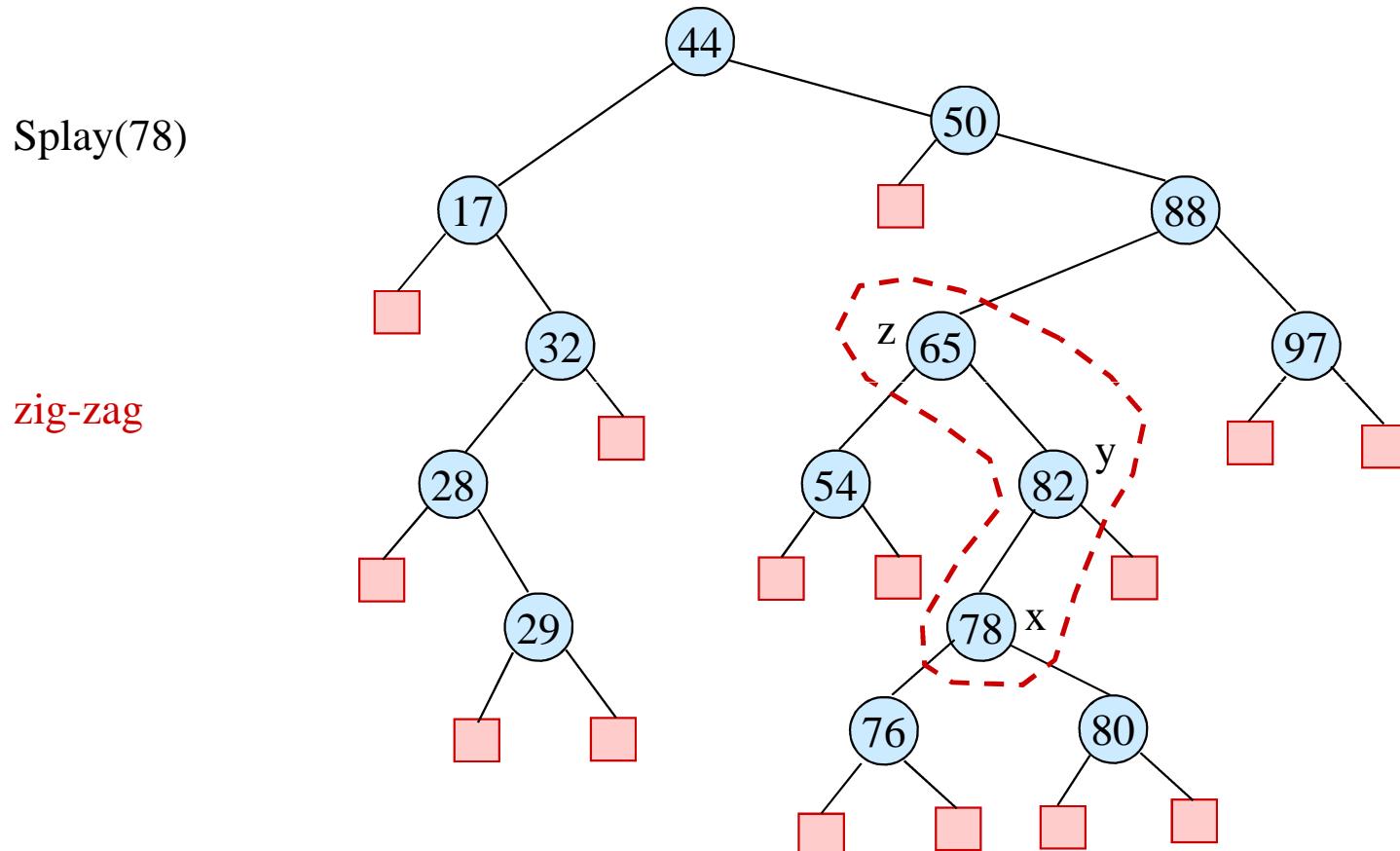


# Complete Example



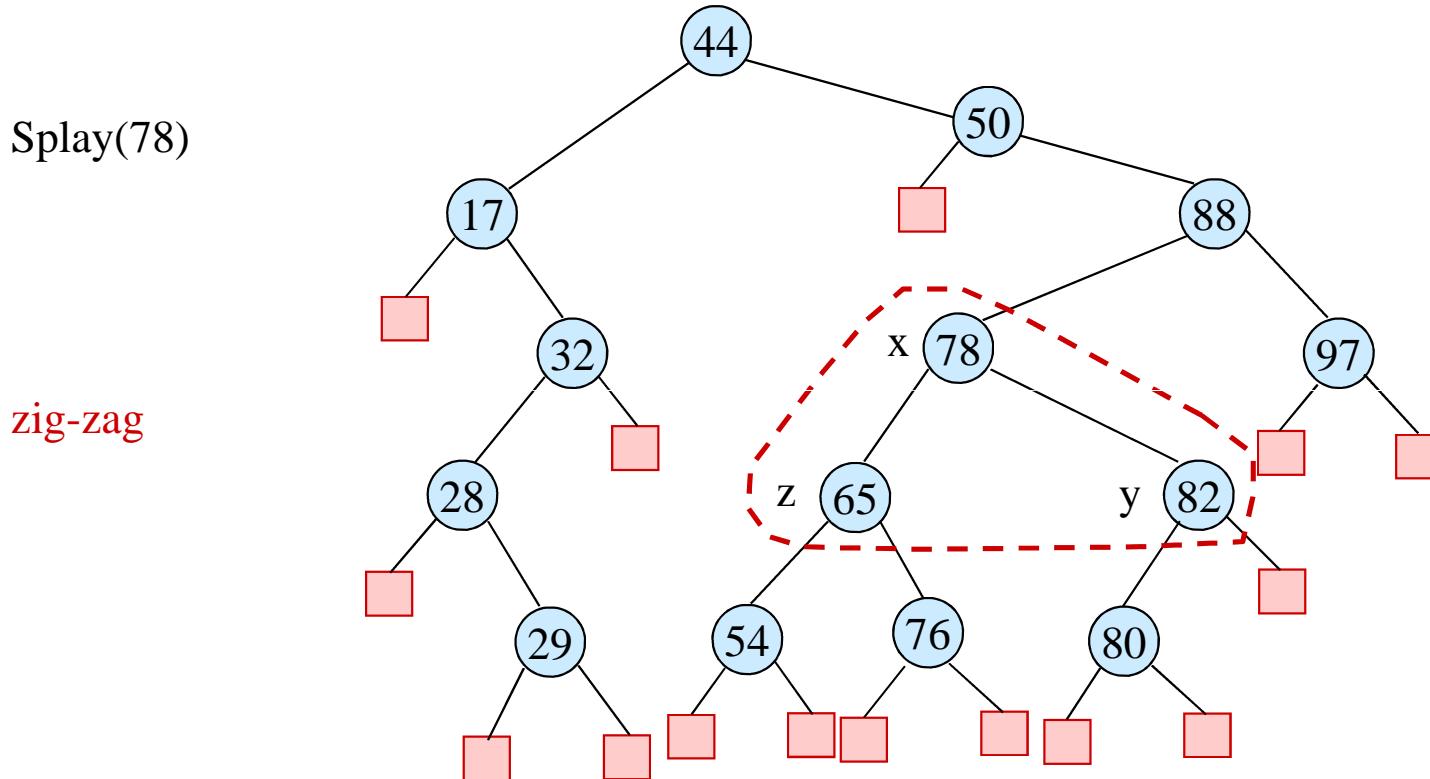
Splay Trees - 7

# Complete Example



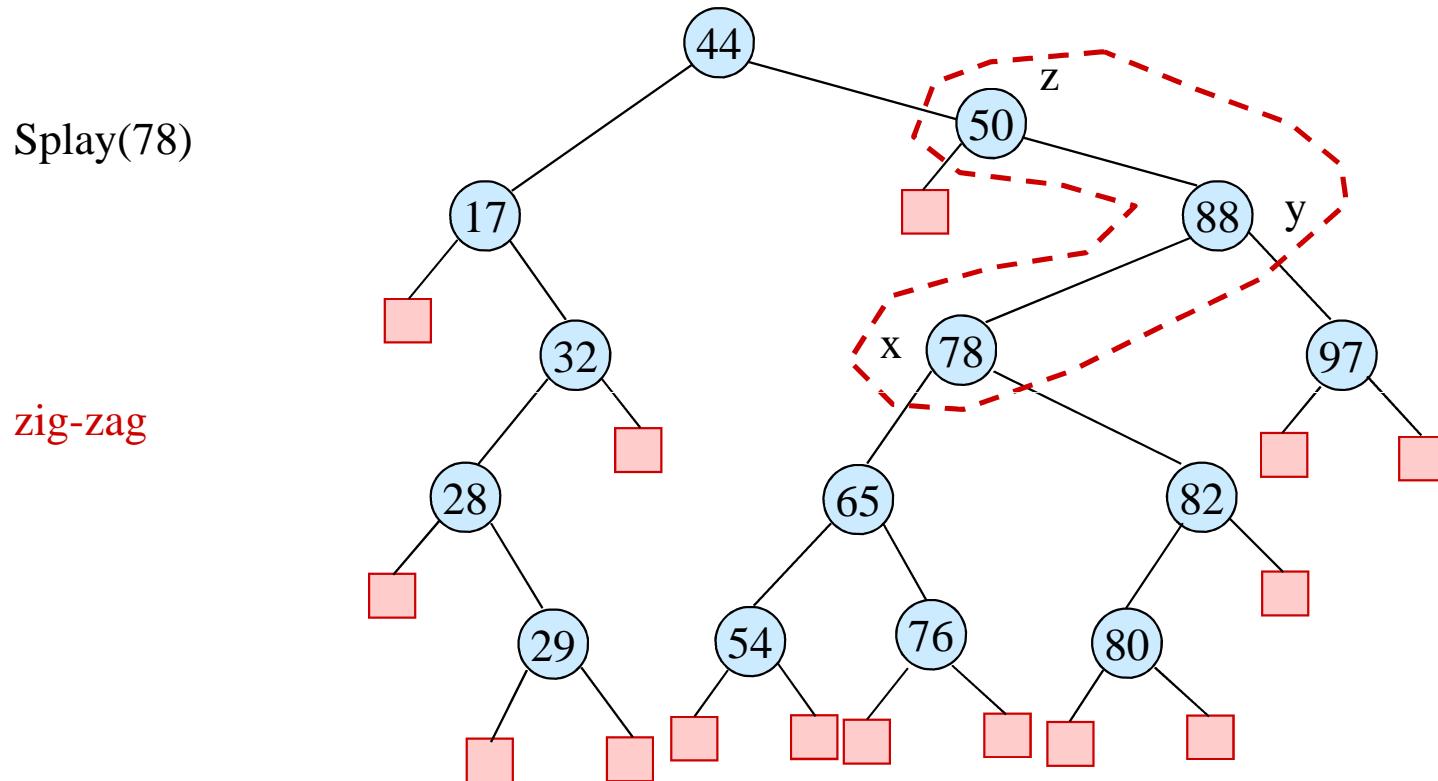
Splay Trees - 8

# Complete Example



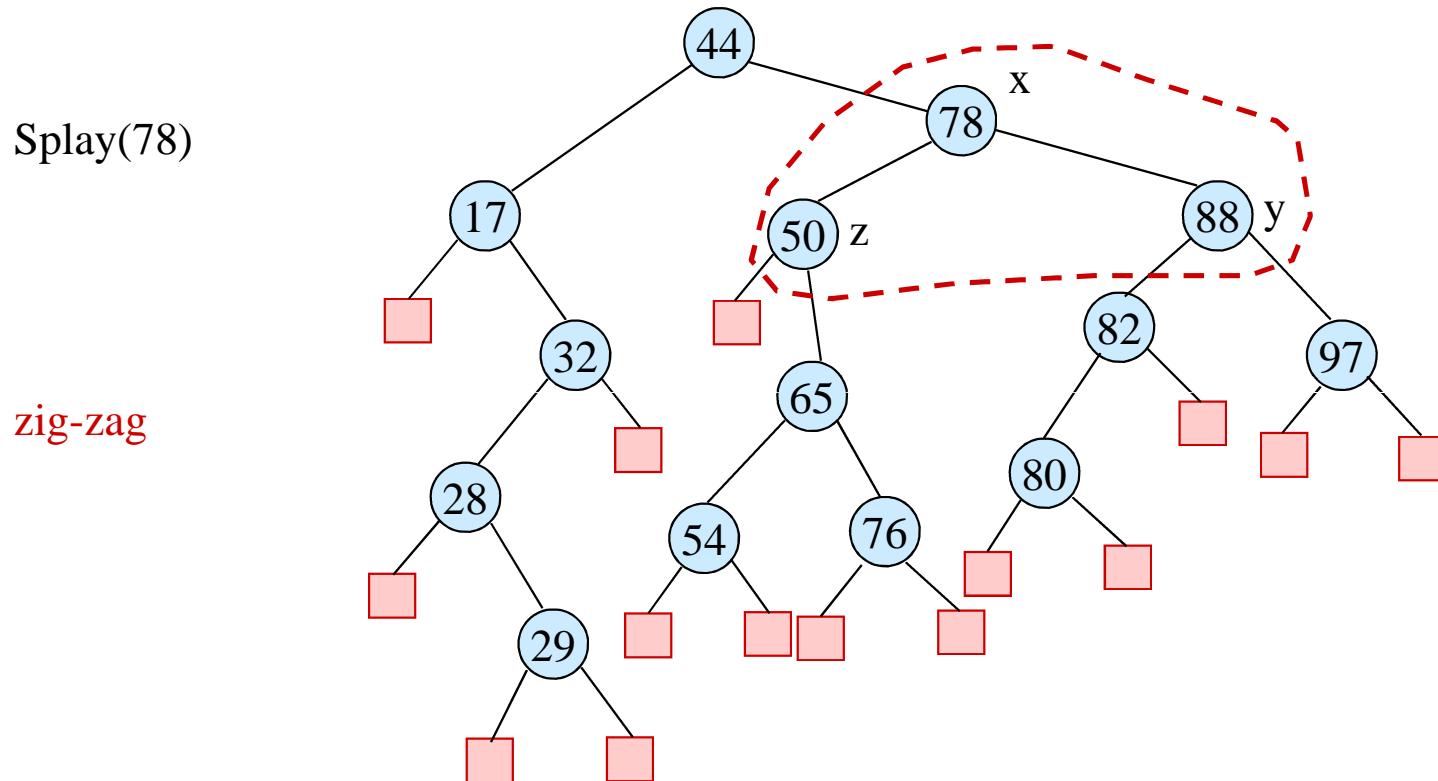
Splay Trees - 9

# Complete Example



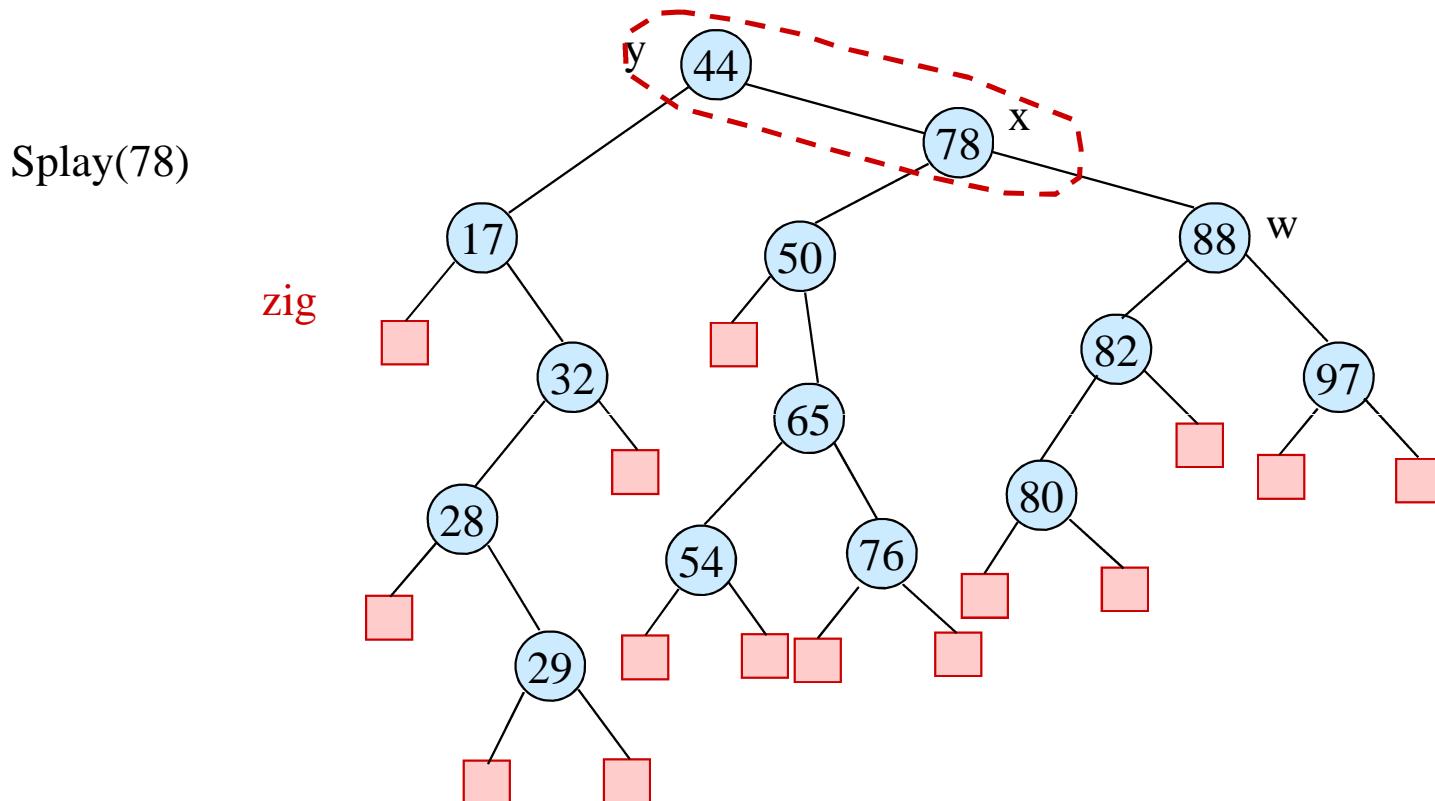
Splay Trees - 10

# Complete Example



Splay Trees - 11

# Complete Example



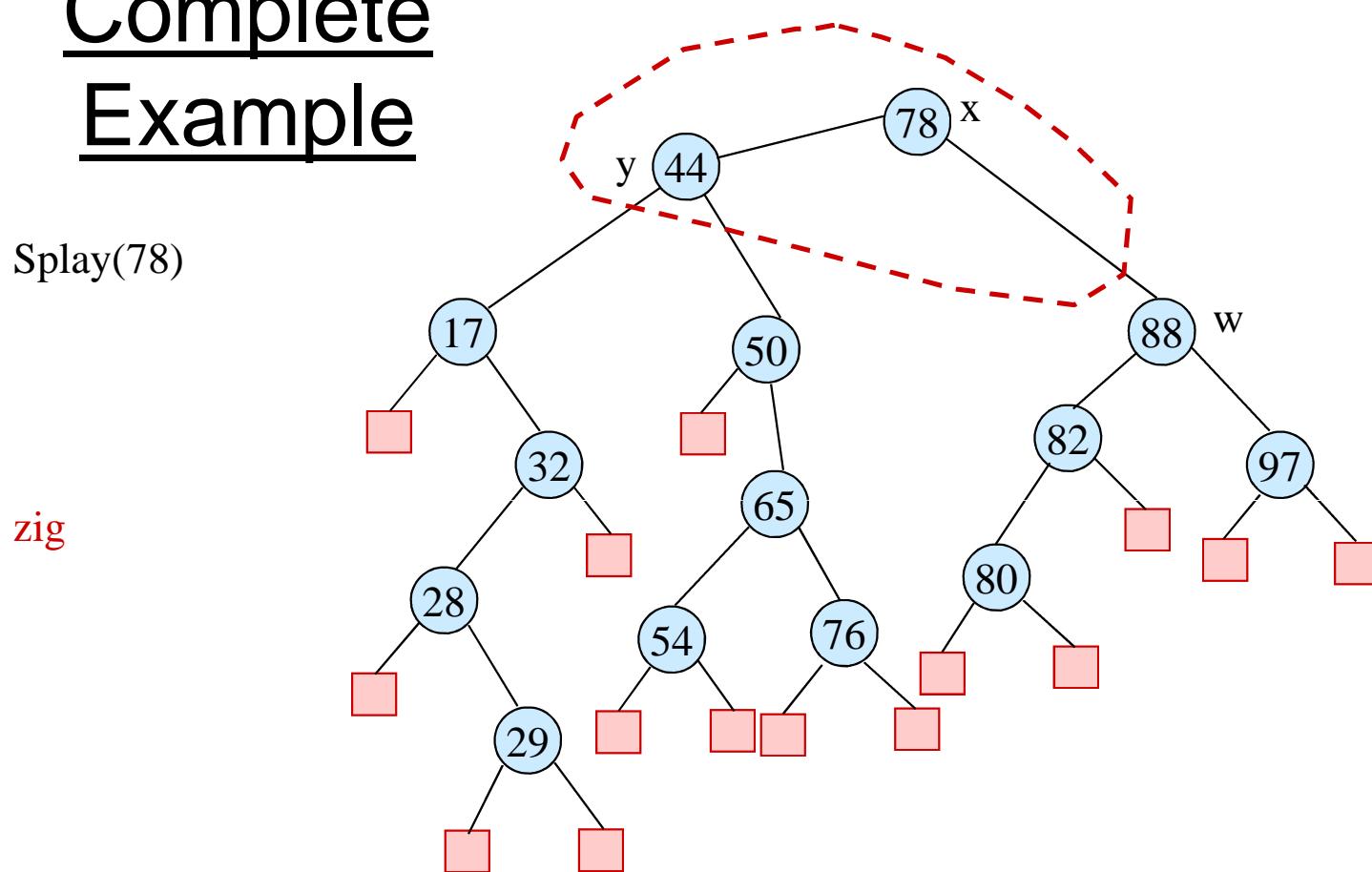
Case 1.  $x$  has no grandparent (*zig*)

If  $x$  is left child of root  $y$ , then rotate  $(xy)R$ .

Else if  $x$  is right child of root  $y$ , then rotate  $(yx)L$ .

Splay Trees - 12

# Complete Example



Splay Trees - 13

# Case 1 of 3

1.  $x$  has no grandparent (*zig*)
  - If  $x$  is left child of root  $y$ , then rotate  $(xy)R$ .
  - Else if  $x$  is right child of root  $y$ , then rotate  $(yx)L$ .

## Case 2 of 3

### 2. $x$ is LL or RR grandchild (*zig-zig*)

- If  $x$  is left child of  $y$ , and  $y$  is left child of  $z$ ,  
then rotate at grandfather  $(yz)R$  and then  
rotate at father  $(xy)R$ .
- Else if  $x$  is right child of  $y$ , and  $y$  is right child  
of  $z$ ,  
then rotate at grandfather  $(yz)L$  and then  
rotate at father  $(xy)L$ .
- If  $x$  has not become the root, then  
continue splaying at  $x$ .

## Case 3 of 3

### 3. $x$ is LR or RL grandchild (*zig-zag*)

- If  $x$  is right child of  $y$ , and  $y$  is left child of  $z$ ,  
then rotate at father  $(yx)L$  and then rotate at  
grandfather  $(xz)R$ .
- Else if  $x$  is left child of  $y$ , and  $y$  is right child  
of  $z$ ,  
then rotate at father  $(yx)R$  and then rotate at  
grandfather  $(xz)L$ .
- If  $x$  has not become the root, then  
continue splaying at  $x$ .

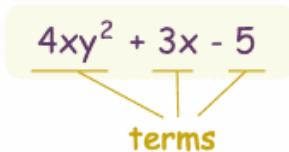
# Faster Multiplication



# Karatsuba's method for Faster Multiplication

# Examples

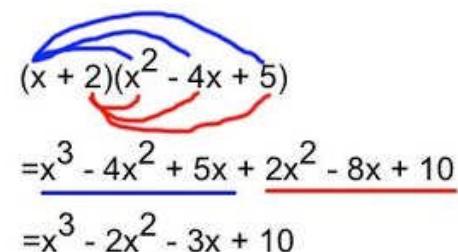
E.g. Polynomial in 2 variables with 3 terms

$$4xy^2 + 3x - 5$$


terms

E.g. Polynomial multiplication

$$\begin{aligned}(2x+3)(5x+4) &= ? \\ &= (2x)(5x) + (2x)(4) + (3)(5x) + (3)(4) \\ &= 10x^2 + 8x + 15x + 12 \\ &= 10x^2 + 23x + 12\end{aligned}$$

$$\begin{aligned}(x+2)(x^2 - 4x + 5) &= x^3 - 4x^2 + 5x + 2x^2 - 8x + 10 \\ &= x^3 - 2x^2 - 3x + 10\end{aligned}$$


# Polynomials

**Polynomial:**  $A(x) = \sum_{j=0}^{n-1} a_j x^j$

**coefficient:** complex, e.g.,  
 $3.25 + 6.999i$

real part                      imaginary part                       $\sqrt{-1}$

**Degree-bound** of  $A(x)$  is  $n$ .

**Degree** is  $k$  if  $a_k$  is the highest non-zero coefficient.

# Polynomial Addition

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

$$C(x) = A(x) + B(x) = \sum_{j=0}^{n-1} c_j x^j, \quad c_j = a_j + b_j$$

$\Theta(n)$  time to compute.

## Example of Polynomial Multiplication

$$\begin{array}{r} 6x^3 + 7x^2 - 10x + 9 \\ -2x^3 \quad \quad \quad + 4x - 5 \\ \hline -30x^3 - 35x^2 + 50x - 45 \\ 24x^4 + 28x^3 - 40x^2 + 36x \\ \hline -12x^6 - 14x^5 + 20x^4 - 18x^3 \\ \hline -12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45 \end{array}$$

Q. What is the complexity of this?

# Usual multiplication

- Normal multiplication would take  $(2n)^2$  cross multiplies and some data manipulation,  $O(n^2)$ .
- Karatsuba multiplication is  $O(n^{1.58})$ .

# Polynomial Multiplication

$$C(x) = A(x) \cdot B(x)$$

$$= \sum_{j=0}^{2n-2} c_j x^j$$

$$\text{where } c_j = \sum_{k=0}^j a_k b_{j-k}$$

↑  
called **convolution**

**Note:** Degree bound of  $C(x)$  is  $2n - 1$ .

Straightforward computation takes  **$\Theta(n^2)$**  time.

# Maple

C:\MAPLEV4\BIN.WIN\WMAPLE32.EXE

```
> P:= x^3 + 5*x^2 + 11*x + 15;
> Q:= 2*x + 3;
> degree(P,x);                  = 3
> coeff(P,x,1);                = 11
> coeffs(P,x);                 = {15, 11, 1, 5}
> subs(x=3,P);                 = 120
```

# Polynomials Maple

```
> P := x^3+5*x^2+11*x+15;
> Q := 2*x + 3;
> m := expand(P * Q);
m = 2 x^4 + 13 x^3 + 37 x^2 + 63 x + 45.

> factor(m);
(2*x+3)*(x+3)*(x^2+2*x+5)

> roots(m); ... {-3,-3/2} ... Two Real roots.
> roots(m, I) ... {-1±2i,-3,-3/2}... Complex roots
```

# Pari

- Free Computer Algebra Software for Linux and windows.
- C:\tools\pari> gp.exe
- gp > m =  $2x^4 + 13x^3 + 37x^2 + 63x + 45;$
- gp > factor(m)
  - [x + 3, 1]
  - [2\*x + 3, 1]
  - [x^2 + 2\*x + 5, 1]
- gp > log(x+1)
  - x - 1/2\*x^2 + 1/3\*x^3 - 1/4\*x^4 + 1/5\*x^5 - 1/6\*x^6...

# Multiplication of polynomials

- We assume P and Q are two polynomials in x of equal size (terms) and of odd degree.
- If they are not, pad them with 0 coefficients.

# Divide and Conquer multiplication

Divide P and Q of size  $2n$  each into smaller polynomials of size  $n$  each:

$P = a_{2n-1}x^{2n-1} + \dots + a_0$  is a poly of size  $2n$ , ( $2n$  terms).  
factor common  $x^n$  from first  $n$  terms:

$$P = x^n(a_{2n-1}x^{n-1} + \dots + a_{n+1}x + a_n) + (a_{n-1}x^{n-1} + \dots + a_0)$$
$$P = A x^n + B$$

Similarly for Q:

$$Q = C x^n + D$$

# Usual multiplication of P\*Q

Divide P and Q of size  $2n$  each into smaller polynomials of size  $n$  each:

- $P = A x^n + B$
- $Q = C x^n + D$
- $P * Q = (A * C)(x^n)^2 + (A * D + B * C)(x^n) + (B * D)$
- There are 4 multiplies of size  $n$  polynomials, plus a size  $2n$  polynomial addition:
- $O(4 * (n^2))$ . What we expected, no better.

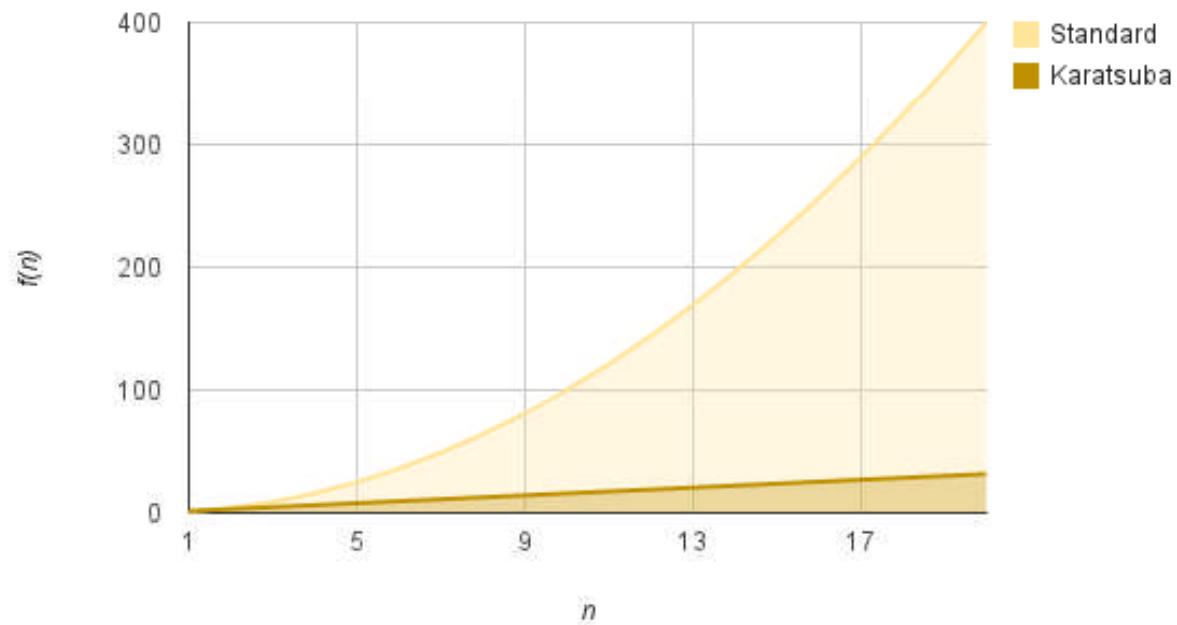
# Karatsuba's multiplication

- $P*Q = (A*C)(x^n)^2 + (A*D+B*C)(x^n) + (B*D)$
- **The new idea:** Instead, compute  $\{R,S,T\}$  in  $3^*$
- $R = (A+B)*(C+D) = \underline{A*C} + \underline{A*D} + \underline{B*C} + \underline{B*D}$  .. mult1
- $S = A*C$  .. mult2
- $T = B*D$  .. mult3
- $U = R-S-T = A*D+B*C$
- $P*Q = S*(x^n)^2 + U*x^n + T.$
- $= A*C*(x^n)^2 + (R-S-T)*x^n + B*D.$
- $= A*C*(x^n)^2 + (A*D+B*C)*x^n + B*D.$
- Cost: 3 multiplies (instead of 4) of size n polynomials and a few additions of polynomials of size  $2n$ .

# Cost of karatsuba's recursive multiplications

- $M(2n) = 3*M(n) + \text{cheaper adds } O(n)....$   
(problem halves, and 3 multiples per stage).
- $M(k) = 3^{\lg(n)} * M(1)$
- $3^{\lg(n)} = 2^{\lg 3 \lg n} = n^{\lg 3} = n^{1.58}$  ( $\lg$  is log base 2).
- Generally if P and Q are about the same size and dense, and “large but not too large” this is good.
- Larger sizes: better methods.
- How large are the coefficients?

$O(n^2)$  grows much faster than  
 $O(n^{\lg 3})$



# Strassen's Faster Matrix Multiplication

# Basic Matrix Multiplication

Suppose we want to multiply two matrices of size  $N \times N$ : for example  $A \times B = C$ .

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$C_{11} = a_{11}b_{11} + a_{12}b_{21}$$

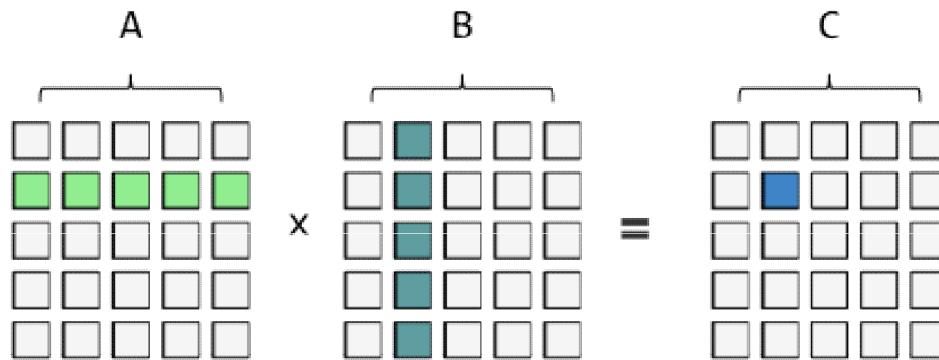
$$C_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$C_{21} = a_{21}b_{11} + a_{22}b_{21}$$

$$C_{22} = a_{21}b_{12} + a_{22}b_{22}$$

2x2 matrix multiplication can be  
accomplished in 8 multiplication. ( $2^{\log_2 8} = 2^3$ )

# Matrix multiplication



$$C[i][j] = \sum(A[i][k] * B[k][j]) \text{ for } k = 0 \dots n$$

In our case:

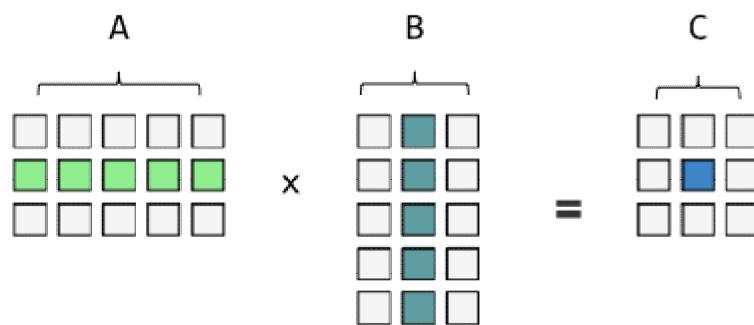
$C[1][1] \Rightarrow$

$$A[1][0]*B[0][1] + A[1][1]*B[1][1] + A[1][2]*B[2][1] + A[1][3]*B[3][1] + A[1][4]*B[4][1]$$

# Rectangular matrices

- $A[m \times n] \times B [n \times p] = C [m \times p]$

- Rows of A == Columns of B
- Rows of C == Rows A
- Columns of C == Columns of B



Rectangular matrix multiplication is only possible when the second dimension of A equals the first dimension of B!

# Matrix multiplication is not-commutative

$$\begin{array}{ccc} \text{A} & \quad \text{B} & \quad \text{C} \\ \left[ \begin{array}{ccccc} \square & \square & \square & \square & \square \\ \color{green}\square & \color{green}\square & \color{green}\square & \color{green}\square & \color{green}\square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \end{array} \right] & \times & \left[ \begin{array}{ccccc} \square & \color{teal}\square & \square & \square & \square \\ \square & \color{teal}\square & \square & \square & \square \\ \square & \color{teal}\square & \square & \square & \square \\ \square & \color{teal}\square & \square & \square & \square \\ \square & \color{teal}\square & \square & \square & \square \end{array} \right] & = & \left[ \begin{array}{ccccc} \square & \square & \square & \square & \square \\ \square & \color{blue}\square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \end{array} \right] \\ \text{B} & \quad \text{A} & \quad \text{C}' \\ \left[ \begin{array}{ccccc} \square & \square & \square & \square & \square \\ \color{teal}\square & \color{teal}\square & \color{teal}\square & \color{teal}\square & \color{teal}\square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \end{array} \right] & \times & \left[ \begin{array}{ccccc} \square & \color{green}\square & \square & \square & \square \\ \square & \color{green}\square & \square & \square & \square \\ \square & \color{green}\square & \square & \square & \square \\ \square & \color{green}\square & \square & \square & \square \\ \square & \color{green}\square & \square & \square & \square \end{array} \right] & = & \left[ \begin{array}{ccccc} \square & \square & \square & \square & \square \\ \square & \color{red}\square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square \end{array} \right] \end{array}$$

# Basic Matrix Multiplication

```
matrix_mult  
for (i = 1...N)  
    for (j = 1..N)  
        compute Ci,j;
```

algorithm

Time analysis

$$C_{i,j} = \sum_{k=1}^N a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^N \sum_{j=1}^N \sum_{k=1}^N c = cN^3 = O(N^3)$$

# Strassens's Matrix Multiplication

- Strassen showed that  $2 \times 2$  matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions, ( $7 = 2^{\lg 7} = 2^{2.807} < 2^3 = 2^{\lg 8} = 8$ )
- This savings can be applied recursively by Divide and Conquer Approach.

# Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data  $S$  in two or more disjoint subsets  $S_1, S_2, \dots$
  - Recur: solve the sub-problems recursively
  - Conquer: combine the solutions for  $S_1, S_2, \dots$ , into a solution for  $S$
- The base case for the recursion are sub-problems of constant size
- Analysis can be done using recurrence equations

# Divide and Conquer Matrix Multiply

$$A \times B = R$$

$$\begin{array}{|c|c|} \hline A_0 & A_1 \\ \hline A_2 & A_3 \\ \hline \end{array} \times \begin{array}{|c|c|} \hline B_0 & B_1 \\ \hline B_2 & B_3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ \hline A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \\ \hline \end{array}$$

- Divide matrices into sub-matrices.
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices

# Divide and Conquer Matrix Multiply

$$\begin{array}{ccc} A & \times & B \\ \boxed{a_0} & \times & \boxed{b_0} \end{array} = \boxed{R} = \boxed{a_0 \times b_0}$$

- Terminate recursion on small base case.

# Strassens's Matrix Multiplication

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22}) * B_{11}$$

$$P_3 = A_{11} * (B_{12} - B_{22})$$

$$P_4 = A_{22} * (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12}) * B_{22}$$

$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

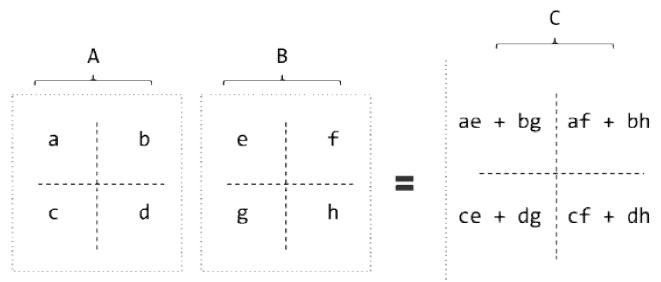
$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

# Strassen's method



$$\begin{aligned}P_1 &= A(F - H) \\P_2 &= (A + B)H \\P_3 &= (C + D)E \\P_4 &= D(G - E) \\P_5 &= (A + D)(E + H) \\P_6 &= (B - D)(G + H) \\P_7 &= (A - C)(E + F)\end{aligned}$$

$$AB = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

After defining the sum P1 ... P7, the product AB can be done in O( $n^{\lg(7)}$ )!

# Comparison

$$\begin{aligned} C_{11} &= P_1 + P_4 - P_5 + P_7 \\ &= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} * (B_{21} - B_{11}) - (A_{11} + A_{12}) * B_{22} + \\ &\quad (A_{12} - A_{22}) * (B_{21} + B_{22}) \\ &= A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{22}B_{21} - A_{22}B_{11} - \\ &\quad A_{11}B_{22} - A_{12}B_{22} + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22} \\ &= A_{11}B_{11} + A_{12}B_{21} \end{aligned}$$

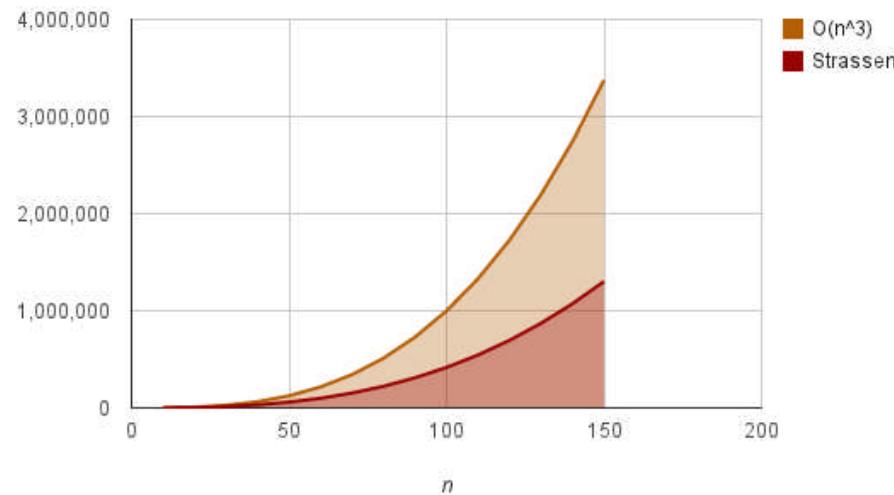
# Strassen Algorithm

```
void matmul(int *A, int *B, int *R, int n) {  
    if (n == 1) {  
        (*R) += (*A) * (*B);  
    } else {  
        matmul(A, B, R, n/4);  
        matmul(A, B+(n/4), R+(n/4), n/4);  
        matmul(A+2*(n/4), B, R+2*(n/4), n/4);  
        matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);  
        matmul(A+(n/4), B+2*(n/4), R, n/4);  
        matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);  
        matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);  
        matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);  
    }  
}
```

Divide matrices in sub-matrices and recursively multiply sub-matrices using matmul(..).

# Strassen vs $O(n^3)$

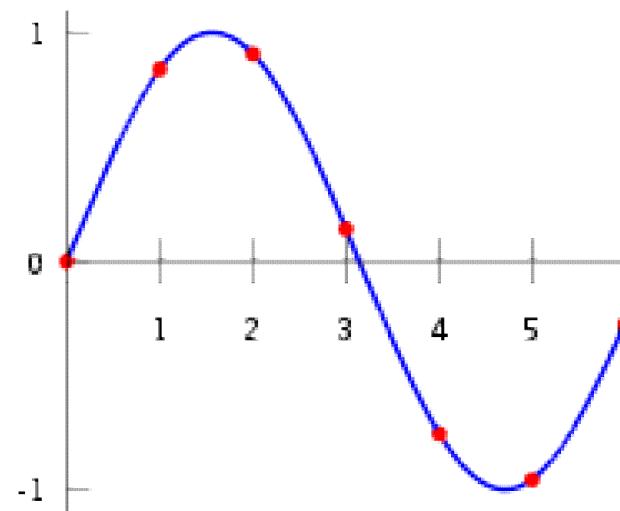
- for small n (usually  $n < 45$ ) the general algorithm is practically a better choice.



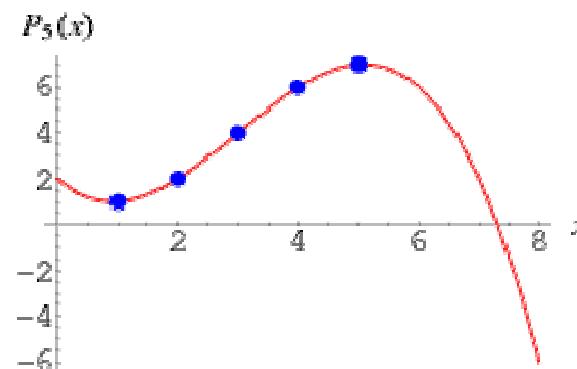
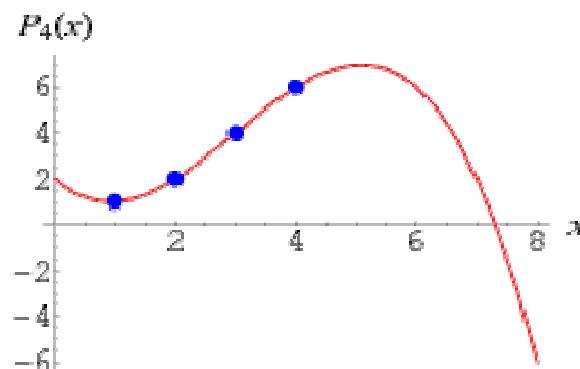
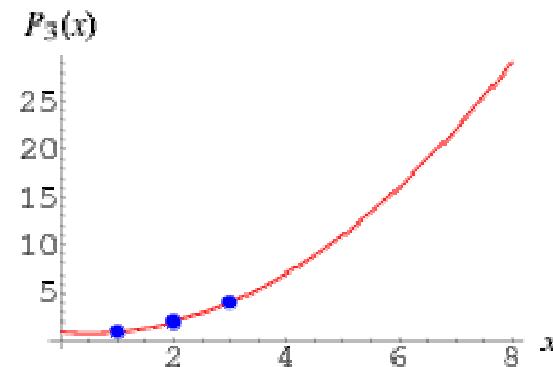
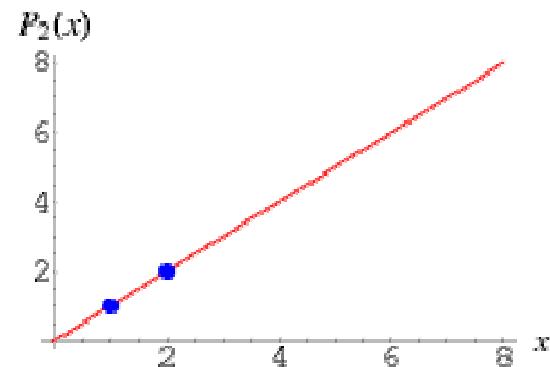
# Faster Multiplication, using Polynomials

# Polynomial interpolation

The red dots denote the data points  $(x_i, y_i)$ ,  
while the blue curve shows the interpolation polynomial.

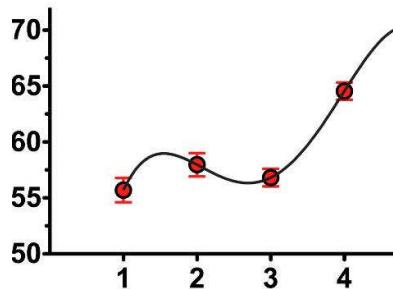


# Polynomial interpolation



# Polynomial interpolation

- Suppose we have 4 data points:  
 $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ , and  $(x_4, y_4)$ .
- There is exactly one polynomial of deg 3,  
 $p(x) = ax^3 + bx^2 + cx + d$
- which passes through the four points.

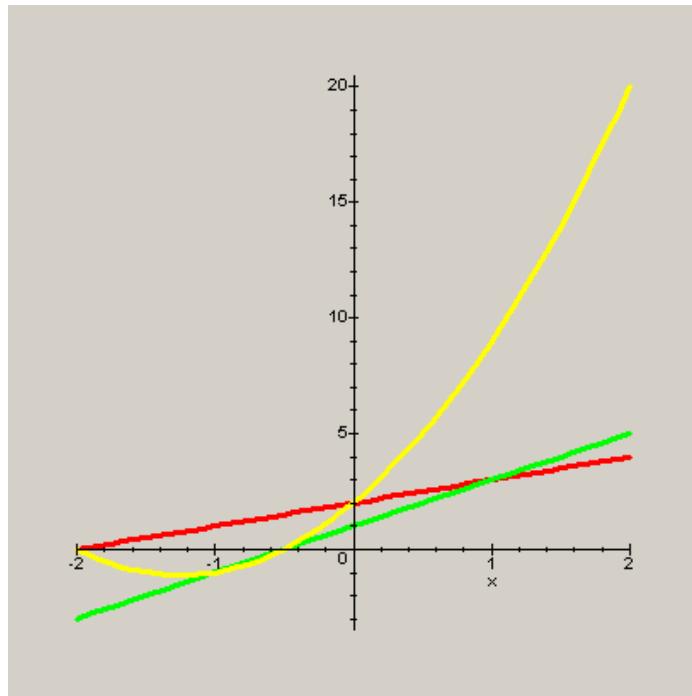


# Interpolate in Maple

```
# Evaluate P(x) and Q(x) at 3 points x=[-1,0,1,-1]
# And Interpolate at 3 points ([x1,x2,x3], [y1,y2,y3])
> p := x -> 1 + 2 * x;
                  p := x -> 1 + 2 x
> q := x -> 2 + x;
                  q := x -> 2 + x
> seq(p(x)*q(x), x=-1..1);
                  -1, 2, 9
> interp([-1,0,1],[-1,2,9], x);
                  2 x^2 + 5 x + 2
```

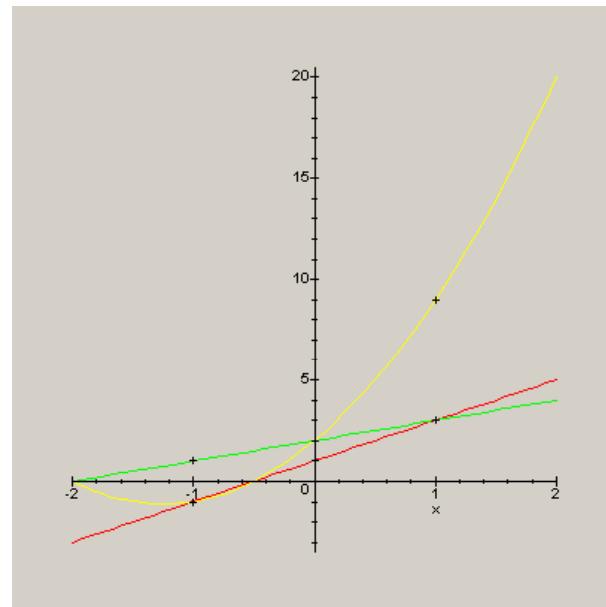
# Plot the graphs

```
> plot({p(x), q(x), p(x)*q(x)},x=-2..2}
```



# All the maple commands together

```
1. p := x -> 1 + 2 * x;  
2. q := x -> 2 + x;  
3. seq(p(x)*q(x), x=-1..1);  
4. interp([-1,0,1],[-1,2,9], x);  
5. py:= {seq([x,p(x)], x=-1..1)};  
6. qy:= {seq([x,q(x)], x=-1..1)};  
7. pqy:= {seq([x,p(x)*q(x)], x=-1..1)};  
8. pl1:=plot({p(x), q(x), p(x)*q(x)},x=-2..2);  
9. pl2:=pointplot(py);  
10. pl3:=pointplot(qy);  
11. pl4:=pointplot(pqy);  
12. with(plots);  
13. display({pl1,pl2,pl3,pl4});
```



## Vandermonde matrix

- Given the points  $\{p(x_i) : i=0..n\}$
- we can compute the polynomial  
 $p(x) = a_n x^n + \dots + a_0$
- using the matrix inverse:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

# Multiply polynomials $h(x) = f(x) * g(x)$ by Interpolation at n points

- A. for  $i=0 \dots n$ ;  
evaluate  $f(i)$  and  $g(i)$ ;  
save the value of  $h(i) = f(i) * g(i)$ .
- B. Interpolate  $\{ (h(i), i) : i=0..n \}$  to the unique polynomial  $h(x)$  that passes through these points:  $(0, h(0)), (1, h(1)), \dots, (n, h(n))$ .

# Horner's rule to evaluate $P(X)$

- Evaluating  $P(x)$  at a point  $x_0$ .
- Brute force takes  $O(n^2)$

$$P(x) = a_0 + x a_1 + x^2 a_2 + x^3 a_3 \dots$$

- ***Horner's rule*** does it in  $O(n)$  by factoring on x:  
$$P(x) = a_0 + x (a_1 + x (a_2 + x (\dots)))$$
- Doing it at n points takes  $O(n^2)$ .

# Horner's Polynomial Evaluation

- Given the coefficients  $(a_0, a_1, a_2, \dots, a_{n-1})$  of  $x^i$  in  $p(x)$ .
- Evaluate  $p(x)$  at  $z$  in  $O(n)$  time using:
- **Function Horner( $A=(a_0, a_1, a_2, \dots, a_{n-1})$ , $z$ ):**  
    If  $n==1$  then return  $a_0$   
    else return  $(a_0 + z * \text{Horner}(A'=(a_1, a_2, \dots, a_{n-1}), z))$

# Using Maple

```
> convert( 5 * x^5 + 3 * x^3 + 22 * x^2 + 55,  
horner);
```

$$55 + (22 + (3 + 5x^2)x)x^2$$

# Homework

Write C program to compute  $p(x)$  using horner rule, given polynomial coeffs as  $p[ ]$  and points  $x[ ]$ .

e.g.  $p(x) = 5 * x^5 + 3 * x^3 + 22 * x^2 + 55$

Horner of  $p(x)$  is  $55+(22+(3+5*x^2)*x)*x^2$

```
double p [ ] = { 55,0,22,3,0,5};  
double x [ ] = { 1, 10, 20};  
for k in 0..2  
    print horner(p, x[k])
```

# Homework solution: horner in C

```
•     double horner( double *coeffs, int n, double x ) {  
•         double y = 0.0;  
•         while ( n-- > 0)  
•             y = y * x + coeffs[n];  
•         return y;  
•     }  
•     #define LEN(A) (sizeof(A)/sizeof(A[0]))  
•     int main( ) {  
•         double k, p[] = { 1, 0, 3 }; // p(x):=1+0x+3x^2 = 1+x(0+3*x));  
•         for(k=-2;k<=2;k++)  
•             printf( "p(%g)= %g\n", k, horner( p, LEN(p), k ) );  
•         return 0;  
•     }
```

# A peculiar way to multiply polynomials $f(x)$ by $g(x)$

How much does this cost?

- Evaluations:  $2*n$  evaluations of size( $n$ ) using Horner's rule is  $O(n^2)$ .
- Interpolation: using “Lagrange” or “Newton Divided Difference” is also  $O(n^2)$ .
- So  $O(n^2)$ , it is slower than Karatsuba.
- BUT....

# Evaluate $P(x)$ faster?

- Our choice of points  $0, 1, 2, \dots, n$  was arbitrary.
- What if we choose “better” points?
- Evaluating  $P(0)$  is almost free.
- Evaluating at symmetric points:  
 $P(c)$  and  $P(-c)$  can be done “faster” by sharing the calculations
- e.g. take coefficients of odd and even powers separately.

# Evaluate $p(-x)$ faster from $p(x)$

- $p1 = P_{\text{odd}} := a_{2n+1}x^{2n+1} + a_{2n-1}x^{2n-1} + \dots + a_1$
- $p2 = P_{\text{even}} := a_{2n}x^{2n} + \dots + a_0$
- $P(x) = P_{\text{odd}}(x^2)x + P_{\text{even}}(x^2)$
- $P(x) = +p1*x+p2$
- $P(-x) = -p1*x+p2.$
- So  $P(c)$  can be computed by evaluating
  - $p1 = P_{\text{even}}(c^2)$
  - $p2 = P_{\text{odd}}(c^2)$
  - and returning  $c*p1+p2$
- Finally,  $P(-c)$  is just  $-c*p1+p2$ , nearly free.

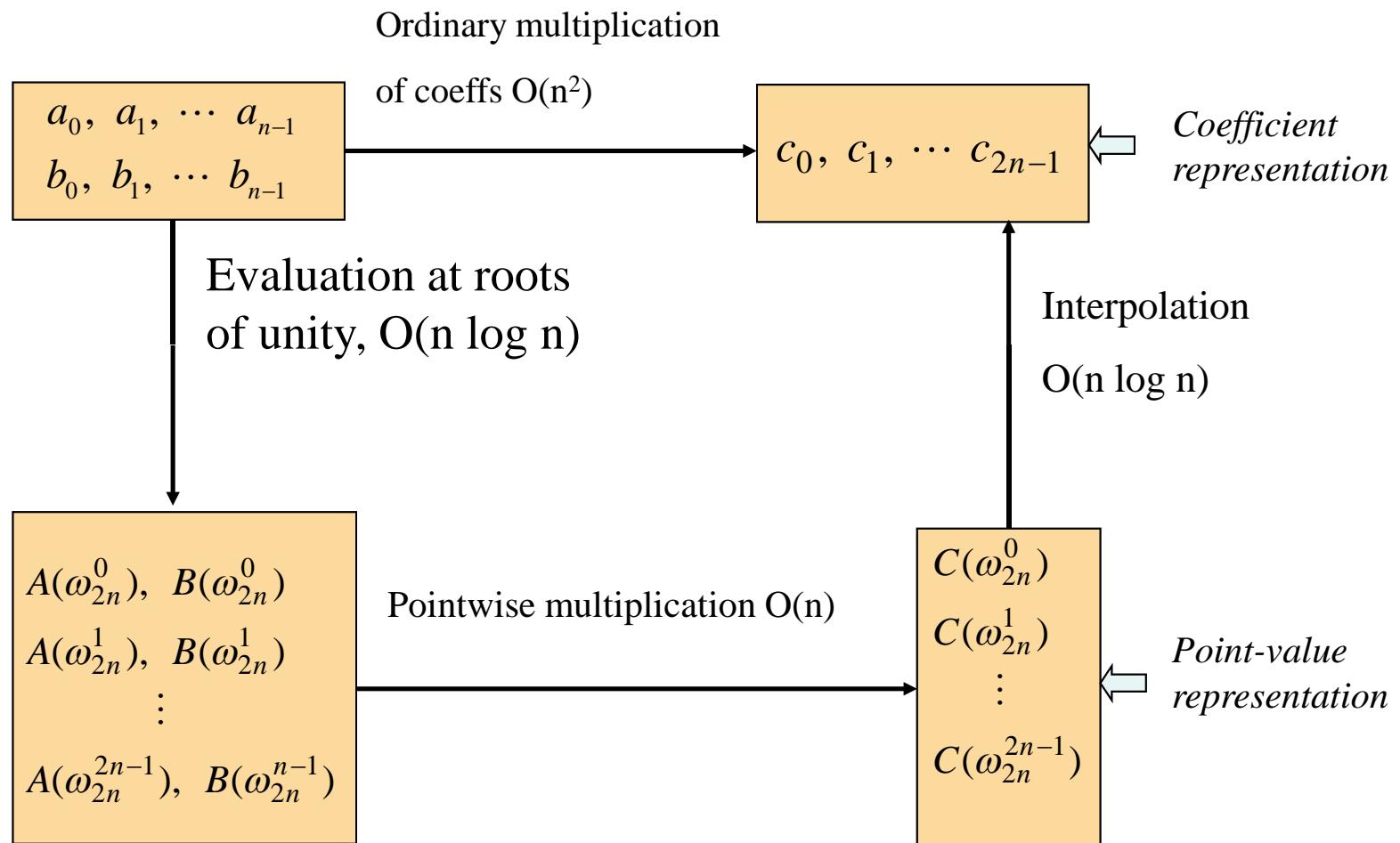
# Are complex numbers better?

- Take a number  $r$  (real, complex, finite field,...)
- Compute the  $2^m$  roots of  $1 = \{1, w, w^2, w^3, \dots w^{2^m}\}$ , these roots have lot of symmetry.
- Evaluate  $P$  at these roots, exploiting symmetry to save on calculations.
- Evaluating  $P(-w^k)$  and  $P(w^k)$  can be done “faster”..
  - take even and odd coefficients separately,
  - by the same trick as before, with  $c^2 = w^{2k}$

# Enter: Numerical FFT

- Evaluate  $P$  at “complex roots of unity.”
- This reduces cost from  $n^2$  to  $O(n \log n)$ ;
- and same complexity for interpolation.

# Fast multiplication of polynomials



# Complex roots of Unity

( $n \times 1$  for evaluating the FFT)

# Complex roots of unity

- *The n-th root of unity* is the complex number  $\omega$  such that  $\omega^n=1$ .
- The value  $w_n=\exp(i \frac{2\pi}{n})$  is called *the principal n-th root of unity*,
- where:  $i=\sqrt{-1}$
- $\exp(i*u)=\cos(u)+i*\sin(u)$ .

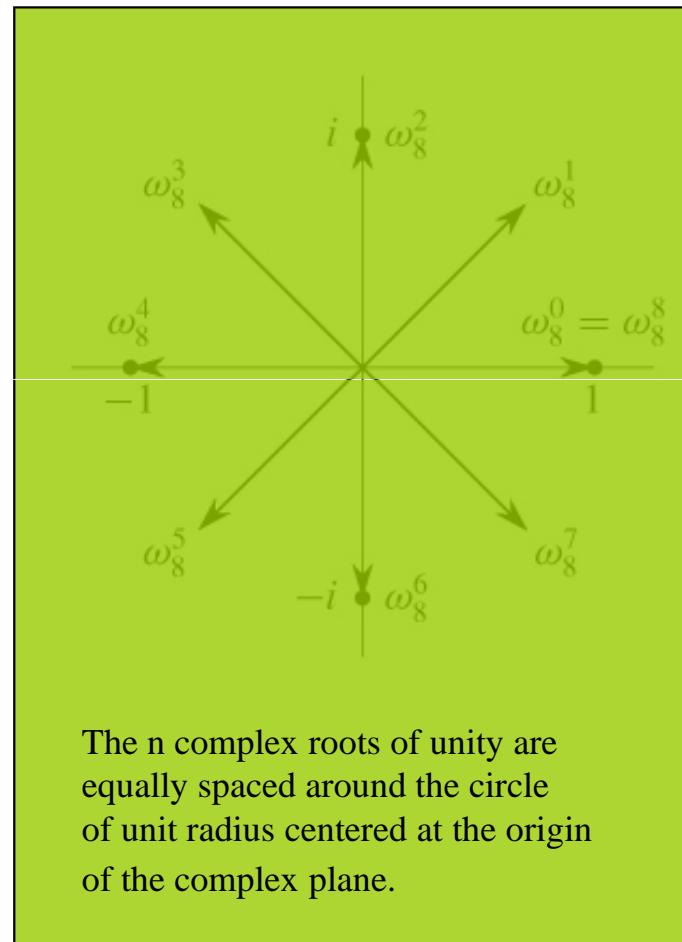
# All the n roots of unity

The n distinct roots of 1 are

$$= \{1, \omega, \omega^2, \dots, \omega^{n-1}\}$$

$$= \{ \exp(k * i 2 \pi / n), k=1..n \}$$

$$\text{with } \omega = e^{2\pi i / n}$$



The n complex roots of unity are  
equally spaced around the circle  
of unit radius centered at the origin  
of the complex plane.

# Properties of the roots of unity

*Lemma-1:*

For any integers  $n \geq 0$  and  $d > 0$ ,

$$W_d n^d k = w_n n^k$$

*Proof:*  $w_d n^d k$

$$= \exp(2\pi i / dn)^d (dk)$$

$$= \exp(2\pi i / n)^k = w_n n^k$$

# Complex roots of unity

*Corollary:* For any even integer  $n > 0$ ,

$$w_n^{n/2} = w_2 = -1$$

Proof:

$$\exp(n/2 * i \pi / n) = \exp(i \pi) = -1$$

# Complex roots of unity

**Lemma-2:** If  $n > 0$  is even, then the squares of the  $n$ -th roots of unity are the  $(n/2)$  roots of unity:

$$(w_n^k)^2 = w_{\{n/2\}^k}$$

**Proof:**

$$\begin{aligned} & (w_n^{k+n/2})^2 \\ &= w_n^{k+2k+n} \\ &= w_n^{k+2k} * w_n^k \\ &= w_n^{k+2k} = (w_n^k)^2 \end{aligned}$$

# Complex roots of unity

***Lemma-3:***

For any integer  $n \geq 1$  and  $k \geq 0$  not divisible by  $n$ ,

$$S = \sum_{j=0..n-1} \{ (w_k^n)^j \} = 0$$

***Proof:***

$S = ((w_n^k)^n - 1) / (w_n^k - 1)$  Geometric series sum.

$$= ((w_n^n)^k - 1) / \dots$$

$$= (1 - 1) / \dots$$

$$= 0$$

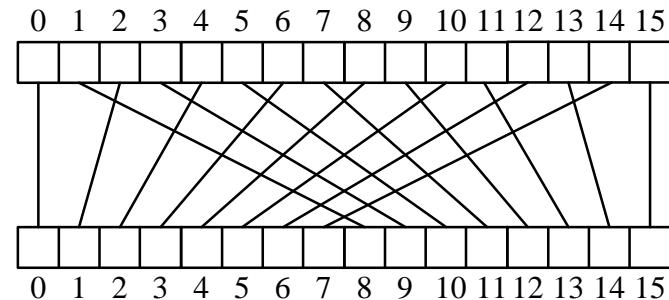
# Properties of Primitive Roots of Unity

- **Inverse Property:** If  $\omega$  is a primitive root of unity, then  $\omega^{-1} = \omega^{n-1}$ 
  - Proof:  $\omega\omega^{n-1} = \omega^n = 1$
- **Cancellation Property:** if  $-n < k < n$ ,  $\sum_{j=0}^{n-1} \omega^{kj} = 0$ 
  - Proof:  
$$\sum_{j=0}^{n-1} \omega^{kj} = \frac{(\omega^k)^n - 1}{\omega^k - 1} = \frac{(\omega^n)^k - 1}{\omega^k - 1} = \frac{(1)^k - 1}{\omega^k - 1} = \frac{1 - 1}{\omega^k - 1} = 0$$
- **Reduction Property:** If  $w$  is a primitive  $(2n)$ -th root of unity, then  $\omega^2$  is a primitive  $n$ -th root of unity:
- Proof: If  $1, \omega, \omega^2, \dots, \omega^{2n-1}$  are all distinct, so is the subset  $1, \omega^2, (\omega^2)^2, \dots, (\omega^2)^{n-1}$
- **Reflective Property:** If  $n$  is even, then  $\omega^{n/2} = -1$ .
  - Proof: By the cancellation property, for  $k=n/2$ :  
$$0 = \sum_{j=0}^{n-1} \omega^{(n/2)j} = \omega^0 + \omega^{n/2} + \omega^0 + \omega^{n/2} + \dots + \omega^0 + \omega^{n/2} = (n/2)(1 + \omega^{n/2})$$
  - Corollary:  $\omega^{k+n/2} = -\omega^k$ .

# FFT

# Fast Fourier Transform

# The Fast Fourier Transform in $O(n \log n)$ .



## The DFT

We wish to evaluate a polynomial  $A(x)$  at each  $n$ -th roots of unity.

Assume  $n$  is a power of 2  
(pad with 0 if required).

# The DFT *Discrete Fourier Transform*

The vector  $y = (y_0..y_{n-1})$  is called the **DFT** of the coefficient vector  $a = (a_0..a_{n-1})$  of a polynomial  $A(x)$ , and written as vector:  $y = \text{DFT}_n(a)$ .

Each  $y_k$  is evaluated at the  $k$ -th root of 1:

$$y_k = A(w_n^k) = \sum_{j=0..n-1} (a_j * w_n^{kj})$$

for  $k=0,1, \dots, n-1$

## Divide the Polynomial into Odd/Even

Write:  $A(x) = A_0(x^2) + x A_1(x^2)$ .

Where

$$A_0(x) = a_0 + a_2 x + a_4 x^2 + ..$$

$$A_1(x) = a_1 + a_3 x + a_5 x^2 + ..$$

# Break the polynomial in odd/even

- If n is even, we can divide a polynomial

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$$

into two polynomials

$$p^{\text{even}}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}$$

$$p^{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}$$

$$p(x) = p^{\text{even}}(x^2) + xp^{\text{odd}}(x^2).$$

## FFT divide and conquer

The problem of evaluating  $A(x)$  at  $\{w_n^0..w_n^{n-1}\}$  reduces to

1. Evaluating the two  $n/2$  degree polynomials  $A_1(x), A_0(x)$  at square of  $\{w_n^0..w_n^{n-1}\}$
2. Combining the results, we get  $y_k = A(w_n^k)$ .

# Recursive FFT

RECURSIVE - FFT( $a$ )

```
1   $n \leftarrow \text{length}[a]$ 
2  if  $n = 1$ 
3    then return  $a$ 
4   $\omega_n \leftarrow e^{2\pi i / n}$ 
5   $\omega \leftarrow 1$ 
6   $a^{[0]} \leftarrow (a_0, a_2, \dots, a_{n-2})$ 
7   $a^{[1]} \leftarrow (a_1, a_3, \dots, a_{n-1})$ 
8   $y^{[0]} \leftarrow \text{RECURSIVE - FFT}(a^{[0]})$ 
9   $y^{[1]} \leftarrow \text{RECURSIVE - FFT}(a^{[1]})$ 
10 for  $k \leftarrow 0$  to  $n/2 - 1$ 
11   do  $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$ 
12      $y_{k+n/2} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$ 
13    $\omega \leftarrow \omega \omega_n$ 
14 return  $y$                                  $\triangleright$   $y$  is assumed to be a vector.
```

# Running time of RECURSIVE-FFT

- Each invocation takes time  $O(n)$ ,  
beside the recursive calls.
- $T(n) = 2 T(n/2) + O(n) = O(n \log n)$ .

# Interpolation

We can write the DFT as the matrix product  $\mathbf{y} = \mathbf{V}\mathbf{a}$  that is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

# Interpolation, inverse matrix

**Theorem:** for  $j$  and  $k$  in  $\{0, 1, \dots, n-1\}$ ,  
the  $(j, k)$  entry of the inverse of matrix is  $w_n^{-k_j}/n$ .  
And  $V^* V^{-1} = I$   
Proof: Next slide, from [Cormen 3e book, pg 913].

iDFT<sub>n</sub>(y) = (a<sub>0..n-1</sub>) can be computed using V<sup>-1</sup>  
 $a_j = 1/n * \sum_{k=0..n-1} \{ y_k * w_n^{-k_j} \}$

# Inverse FFT

## Theorem 30.7

For  $j, k = 0, 1, \dots, n - 1$ , the  $(j, k)$  entry of  $V_n^{-1}$  is  $\omega_n^{-kj}/n$ .

**Proof** We show that  $V_n^{-1}V_n = I_n$ , the  $n \times n$  identity matrix. Consider the  $(j, j')$  entry of  $V_n^{-1}V_n$ :

$$\begin{aligned}[V_n^{-1}V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n.\end{aligned}$$

This summation equals 1 if  $j' = j$ , and it is 0 otherwise by the summation lemma (Lemma 30.6). Note that we rely on  $-(n-1) \leq j' - j \leq n-1$ , so that  $j' - j$  is not divisible by  $n$ , in order for the summation lemma to apply. ■

Given the inverse matrix  $V_n^{-1}$ , we have that DFT $_n^{-1}(y)$  is given by

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

for  $j = 0, 1, \dots, n - 1$ . By comparing equations (30.8) and (30.11), we see that by modifying the FFT algorithm to switch the roles of  $a$  and  $y$ , replace  $\omega_n$  by  $\omega_n^{-1}$ , and divide each element of the result by  $n$ , we compute the inverse DFT (see Ex-

How to find inverse:  $F_n^{-1}$ ?

**Proposition.** Let  $\omega$  be a primitive  $l$ -th root of unity over a field  $L$ . Then

$$\sum_{k=0}^{l-1} \omega^k = \begin{cases} 0 & \text{if } l > 1 \\ 1 & \text{otherwise} \end{cases}$$

**Proof.** The  $l=1$  case is immediate since  $\omega=1$ .

Since  $\omega$  is a primitive  $l$ -th root, each  $\omega^k$ ,  $k \neq 0$  is a distinct  $l$ -th root of unity.

$$\begin{aligned} Z^l - 1 &= (Z - \omega_l^0)(Z - \omega_l^1)(Z - \omega_l^2) \dots (Z - \omega_l^{l-1}) = \\ &= Z^l - (\sum_{k=0}^{l-1} \omega_l^k) Z^{l-1} + \dots + (-1)^l \prod_{k=0}^{l-1} \omega_l^k \end{aligned}$$

Comparing the coefficients of  $Z^{l-1}$  on the left and right hand sides of this equation proves the proposition.

## Inverse matrix to $F_n$

**Proposition.** Let  $\omega$  be an  $n$ -th root of unity. Then,

$$F_n(\omega) \cdot F_n(\omega^{-1}) = nE_n$$

**Proof.** The  $ij^{\text{th}}$  element of  $F_n(\omega)F_n(\omega^{-1})$  is

$$\sum_{k=0}^{n-1} \omega^{ik} \omega^{-ik} = \sum_{k=0}^{n-1} \omega^{k(i-j)} = \begin{cases} 0, & \text{if } i \neq j \\ n, & \text{otherwise} \end{cases}$$

The  $i=j$  case is obvious. If  $i \neq j$  then  $\omega^{i-j}$  will be a primitive root of unity of order  $l$ , where  $l|n$ . Applying the previous proposition completes the proof.

$$F_n^{-1}(\omega) = \frac{1}{n} F_n(\omega^{-1})$$

So,

Evaluating	$y = F_n(\omega) \mathbf{a}$
Interpolation	$\mathbf{a} = \frac{1}{n} F_n(\omega^{-1}) y$

## FFT and iFFT are $O(n \log n)$

By using the FFT and the iFFT,  
we can transform a polynomial of degree  $n$   
back and forth between its coefficient  
representation and a point-value representation  
in time  $O(n \log n)$ .

# The convolution

For any two vectors  $\mathbf{a}$  and  $\mathbf{b}$  of length  $n$  is a power of 2,  
we can do:

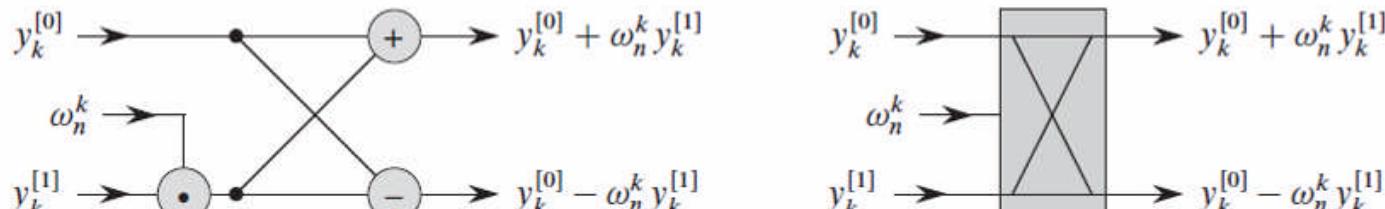
$$\mathbf{a} \otimes \mathbf{b} = DFT_{2n}^{-1}(DFT_{2n}(\mathbf{a}) \cdot DFT_{2n}(\mathbf{b}))$$

# The Butterfly operation

The **for** loop involves computing the value  $\omega_n^k y_k^{[1]}$  twice.

We can change the loop(the butterfly operation):

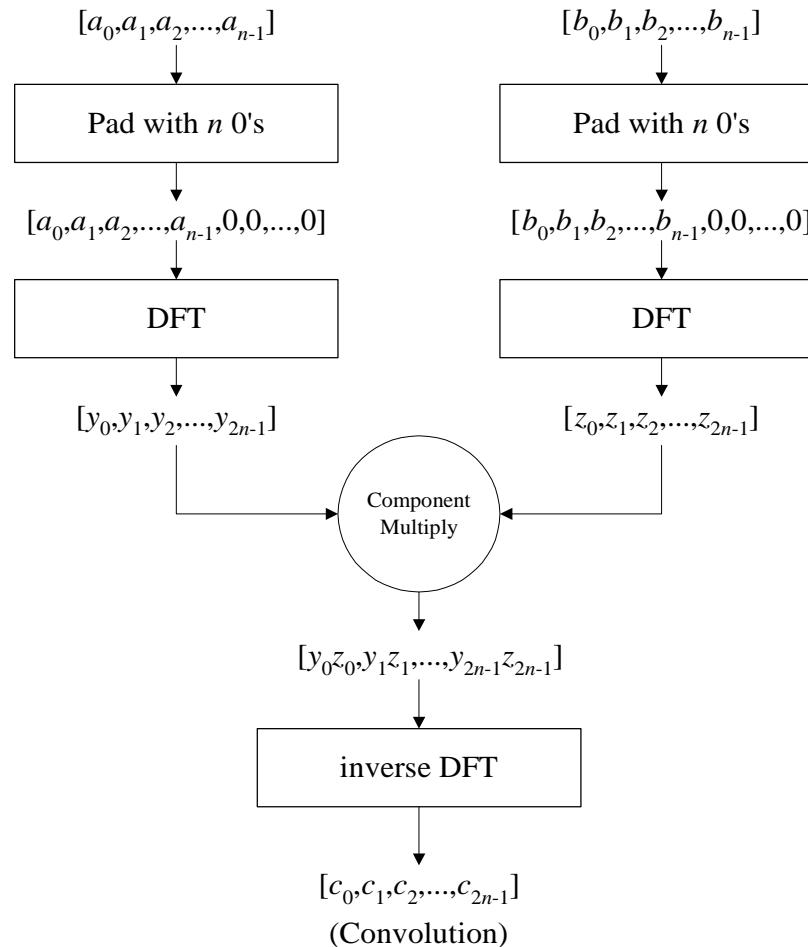
```
for k ← 0 to n/2-1  
    do t ← ωyk[1]  
        yk ← yk[0] + t  
        yk+(n/2) ← yk[0] - t  
        ω ← ωωn
```



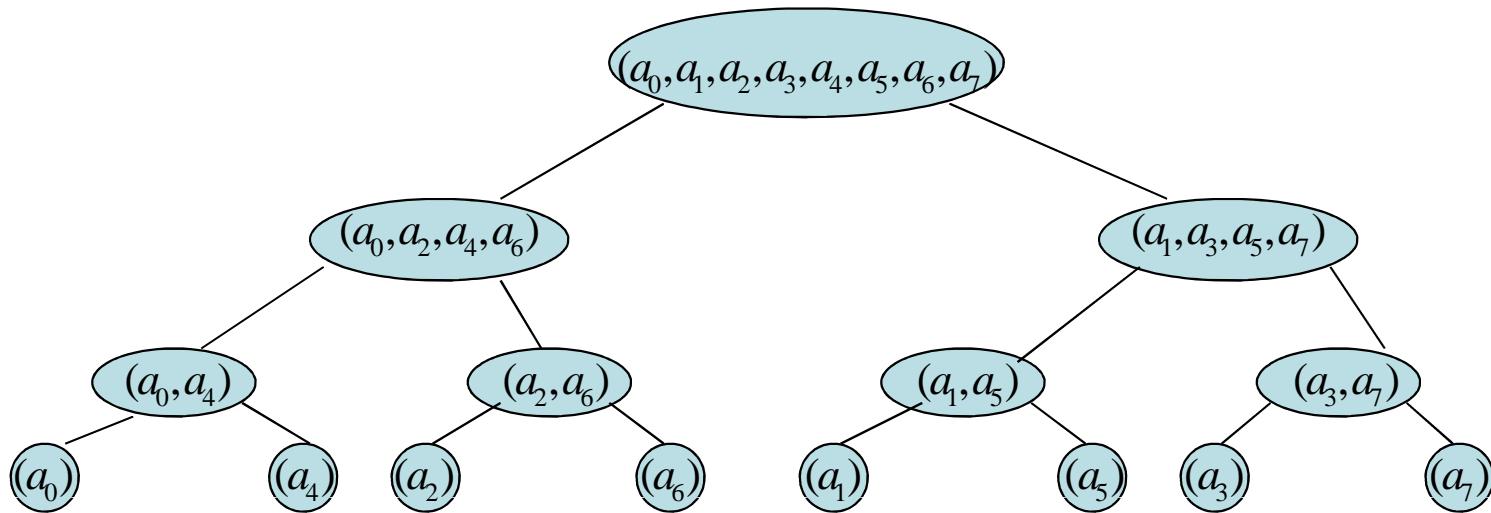
A butterfly operation. (a) The two input values enter from the left, the twiddle factor  $w_n^k$  is multiplied by  $y_k^{[1]}$ , and the sum and difference are output on the right.  
(b) A simplified drawing of a butterfly operation.

# Convolution

- The DFT and the iDFT can be used to multiply two polynomials
- So we can get the coefficients of the product polynomial quickly if we can compute the DFT (and iDFT) quickly



# Iterative FFT



We take the elements in pairs, compute the DFT of each pair, using one butterfly operation, and replace the pair with its DFT

We take these  $n/2$  DFT's in pairs and compute the DFT of the four vector elements.

We take 2 ( $n/2$ )-element DFT's and combine them using  $n/2$  butterfly operations into the final  $n$ -element DFT

## Iterative-FFT Code with bit reversal.

0,4,2,6,1,5,3,7 → 000,100,010,110,001,101,011,111 → 000,001,010,011,100,101,110,111

BIT-REVERSE-COPY(a,A)

*n* ← *length* [a]

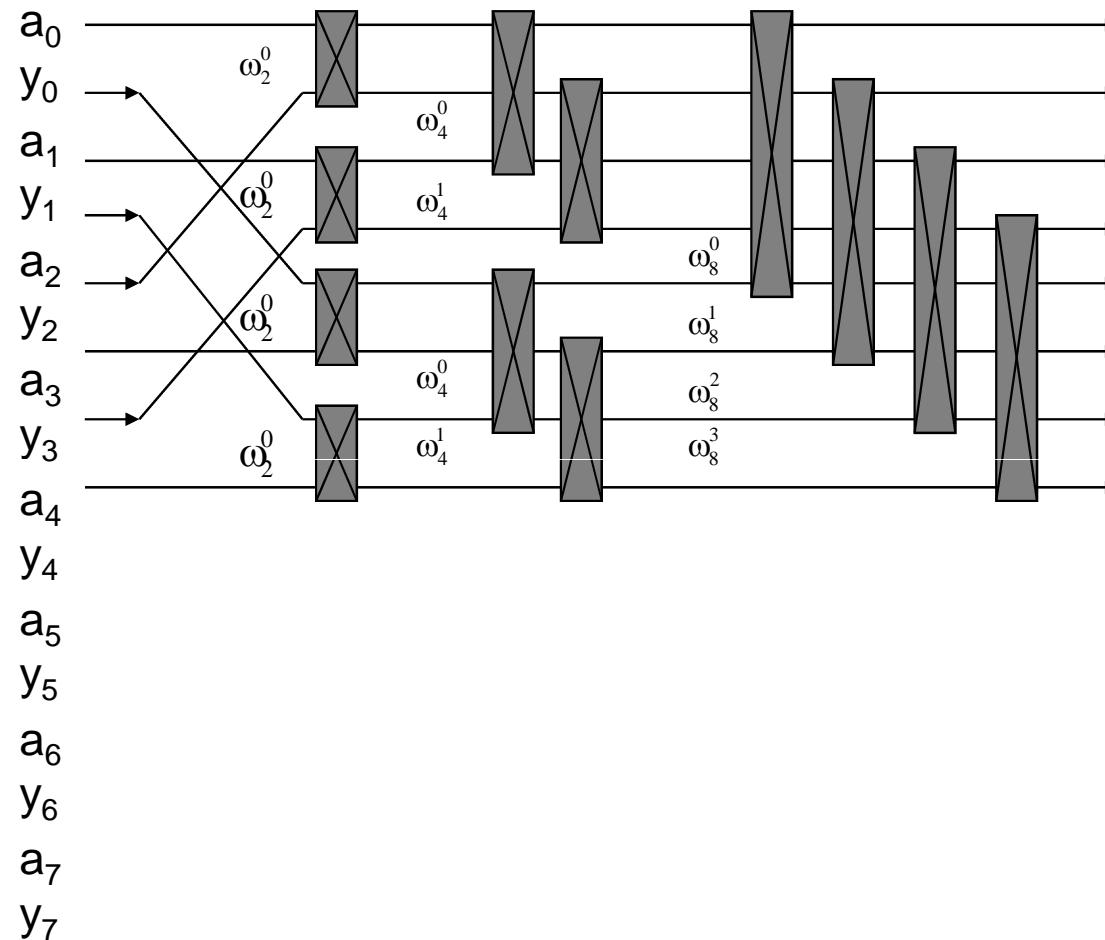
**for** *k* ← 0 **to** *n*-1

**do** *A*[rev(*k*)] ← *a<sub>k</sub>*

ITERATIVE-FFT

1.     BIT-REVERSE-COPY(a,A)
2.     *n* ← *length* [a]
3.     **for** *s* ← 1 **to** log *n*
4.         **do** *m* ← 2<sup>*s*</sup>
5.              $\omega_m \leftarrow e^{2\pi i/m}$
6.         **for** *j* ← 0 **to** *n*-1 **by** *m* ← 1
7.             **for** *j* ← 0 **to** *m*/2-1
8.                 **do for** *k* ← *j* **to** *n*-1 **by** *m*
9.                     **do** *t* ←  $\omega A[k+m/2]$
10.                     *u* ← *A*[*k*]
11.                     *A*[*k*] ← *u*+*t*
12.                     *A*[*k*+*m*/2] ← *u*-*t*
13.      $\omega \leftarrow \omega \omega_m$
14.     **return** *A*

## A parallel FFT circuit



# FFT Computation example in Maple

## Example: $Q=A^*B$

Usual multiplication the following polynomials in  $O(n^2)$ :

$$A(x) := 1 + x + 2x^2;$$

$$B(x) := 1 + 2x + 3x^2;$$

$$Q(x) := A(x)^* B(x);$$

$$= 1 + 3x + 7x^2 + 7x^3 + 6x^4$$

# Fast Polynomial Multiplication

Multiply the polynomials in  $O(n \log(n))$   
using DFT of the coefficient vectors:

$$A = (1, 1, 2, 0, 0)$$

$$B = (2, 1, 3, 0, 0)$$

$$DFT(A) =$$

$$[4.000, (-0.309 - 2.126i), (0.809 + 1.314i),\\ (0.809 - 1.314i), (-0.309 + 2.126i)]$$

$$DFT(B) =$$

$$[6.000, (-0.809 - 3.665i), (0.309 + 1.677i),\\ (0.309 - 1.677i), (-0.809 + 3.665i)]$$

$$A \times B = iDFT(DFT(A) * DFT(B))$$

$$DFT(A) \cdot DFT(B) =$$

$$\begin{aligned} & [24.00, (-7.545 + 2.853i), \\ & (-1.954 + 1.763i), (-1.954 - 1.763i), \\ & (-7.545 - 2.853i)] \end{aligned}$$

and

$$A \times B = iDFT( DFT(A) \cdot DFT(B) ) = (1, 3, 7, 7, 6).$$

$$\text{ie. } (1x^0 + 3x^1 + 7x^2 + 7x^3 + 6x^4)$$

# FFT speedup

If one complex multiplication takes 500ns:

$N$	$T_{DFT}$	$T_{FFT}$
$2^{12}$	8 sec.	0.013 sec.
$2^{16}$	0.6 hours	0.26 sec.
$2^{20}$	6 days	5 sec.

# FFT in Maple

```
# Multiply polynomials A and B using FFT
A(x):=1+x+2*x^2;
B(x):=1+2*x+3*x^2;
Q(x):=expand(A(x)*B(x));
readlib(FFT);
ar := array([1,1,2,0,0,0,0,0]); ai := array([0,0,0,0,0,0,0,0]);
br := array([1,2,3,0,0,0,0,0]); bi := array([0,0,0,0,0,0,0,0]);
FFT(3,ar,ai); af := evalm(ar+I*ai);
FFT(3,br,bi); bf := evalm(br+I*bi);
abf := zip( (x,y)->x * y, af, bf ); # multiply component wise
abfr := evalm(map(x->Re(x), abf)); # real(array)
abfi := evalm(map(x->Im(x), abf)); # imag(array)
iFFT(3,abfr, abfi); # compute the inverse fft
evalm(abfr+I*abfi); # combine the real and imag
# coeffs of Q := [1,3,7,7,6]
```

# FFT Programs in C and Python

# FFT with numpy in python

- `#!/python2.5`
- `# What: fft in python using numpy.`
- `from numpy.fft import fft`
- `from numpy.fft import ifft`
- `from numpy import array`
- `from numpy import set_printoptions, get_printoptions`
- `# print get_printoptions()`
- `set_printoptions(precision = 4)`
- `set_printoptions(suppress=True)`
- `a = array((0, 1, 7, 2, -1, 3, 7, 8, 0, -23, -7, 31, 1, 31, -7, -31))`
- `print "data =", a`
- `y = fft(a)`
- `print "fft(data) =", y`
- `z = ifft(y)`
- `print "ifft(fft(data)) =", z`

# Running fft-numpy.py

- > fft-numpy.py
- data = [ 0 1 7 2 -1 3 7 8  
• 0 -23 -7 31 1 31 -7 -31]
- fft(data) = [  
• 22 +0j -14.2 +10.7j -79.1 +0j -4.8-101.8j  
• 0 -2.j 4.8 -58.2j 79.1 -0j 14.2 +46.3j  
• -22 +0j 14.2 -46.3j 79.1 -0j 4.8 +58.2j  
• 0 +2.j -4.8+101.8j -79.1 +0j -14.2 -10.7j]
- ifft(fft(data)) = [  
• 0+0j 1-0j 7-0j 2-0j -1+0j 3+0j 7-0j 8+0j  
• 0+0j -23-0j -7+0j 31+0j 1+0j 31+0j -7+0j -31+0j]

# fft.c 1 of 2.

```
• double PI = 3.14159265358979;
• const double eps = 1.e-7;
• typedef double complex cplx;

• void fft_rec(cplx buf[], cplx out[], int n, int step, int inv) {
•     if (step < n) {
•         int i;
•         fft_rec(out      , buf      , n, step * 2, inv);
•         fft_rec(out + step, buf + step, n, step * 2, inv);
•         for (i = 0; i < n; i += 2 * step) {
•             int sign = inv? -1 : 1;
•             cplx t = cexp(-I * PI * i * sign / n) * out[i + step];
•             buf[ i / 2]      = out[i] + t;
•             buf[(i + n)/2] = out[i] - t;
•         }
•     }
• }
```

## fft.c continued 2 of 2

- void fft(cplx buf[], int n, int inv) {
- cplx out[n];
- int i;
- assert(is\_power\_of\_two(n));
- for (i = 0; i < n; i++)
- out[i] = buf[i];
- fft\_rec(buf, out, n, 1, inv);
- if (inv) for (i = 0; i < n; i++) buf[i] /= n;
- }
- int main() {
- cplx buf[] = {1, 2, 5, 1, 0, 0, 0, 99};
- fft(buf, 8, 0);
- fft(buf, 8, 1); // invfft
- }

# Running fft.c

- ```
> gcc -std=gnu99 -g -o fft.exe fft.c
> ./fft.exe
• Data: x[0]=(1, 0), x[1]=(2, 0), x[2]=(5, 0), x[3]=(1, 0),
• x[4]=(0, 0), x[5]=(0, 0), x[6]=(0, 0), x[7]=(99, 0),
• FFT: x[0]=(108, 0), x[1]=(71.71, 62.88),
• x[2]=(-4, 98), x[3]=(-69.71, 72.88),
• x[4]=(-96, 0), x[5]=(-69.71, -72.88),
• x[6]=(-4, -98), x[7]=(71.71, -62.88),
• iFFT: x[0]=(1, 0), x[1]=(2, 0), x[2]=(5, 0), x[3]=(1, 0),
• x[4]=(0, 0), x[5]=(0, 0), x[6]=(0, 0), x[7]=(99, 0),
```

# Integers are also polynomials

- Cryptography involves multiplying large (100s of digits) integers. So it must be done efficiently.
- E.g.  $2345 = 2x^3 + 3x^2 + 4x + 5$  ( $x=10$ )
- E.g.  $0x45f = 4x^2 + 5x + f$  ( $x=16$ , hex).

Now we can use fast polynomial multiplication algorithm for integers also.

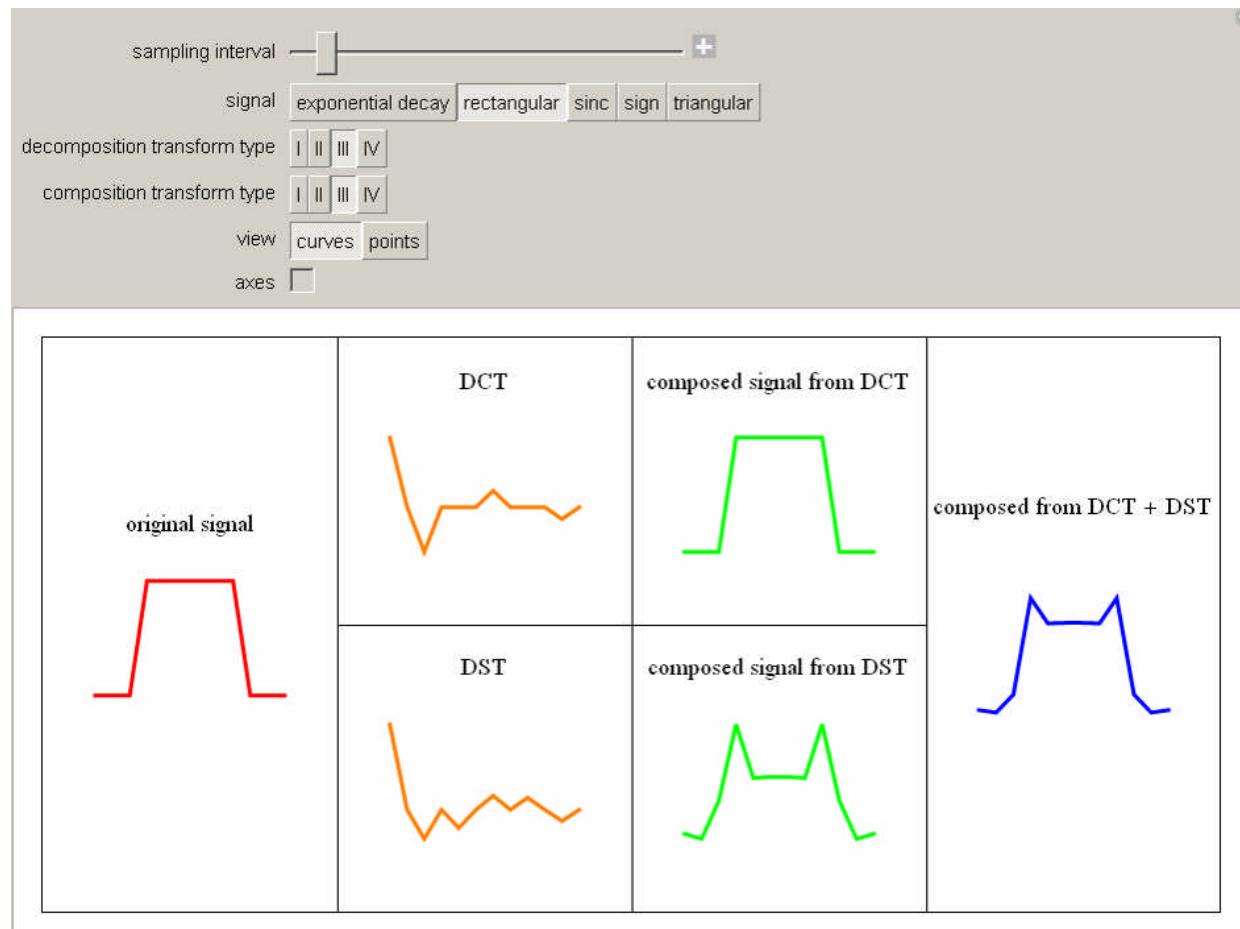
## Example: Compute E to 100,000 decimals using fft-mul.c in 1 second

- > e2718.exe 100000 | head
- Total Allocated memory = 4608 K
- Starting series computation
- Starting final division
- Total time : 1.01 seconds
- Worst error in FFT (should be less than 0.25):  
0.0003662109
- $E = 2.7182818284\ 5904523536\ 0287471352\ \dots$

from <http://xavier.gourdon.free.fr/Constants/constants.html>

# FFT Applications (Mathematica Demonstrations)

# Mathematica demonstration of DCT/DST



# 2D FT

1. 2 dimensional Fourier transforms simply involve a number of 1 dimensional fourier transforms.
2. More precisely, a 2 dimensional transform is achieved by first transforming each row, replacing each row with its transform and then transforming each column, replacing each column with its transform.

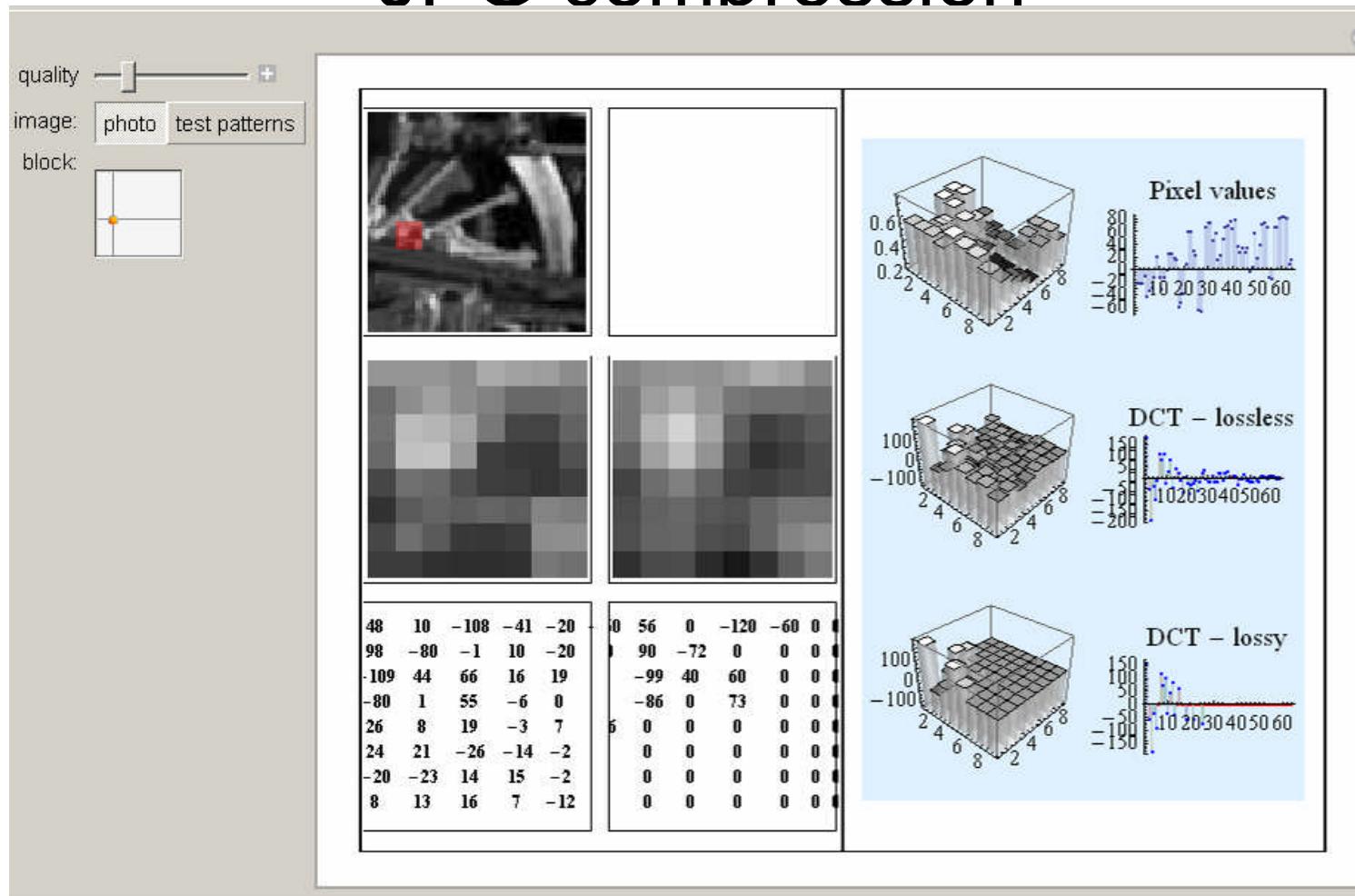
From <http://paulbourke.net/miscellaneous/dft/>

# Images are also integers

1. The JPEG compression algorithm (which is also used in MPEG compression) is based on the two-dimensional discrete cosine transform (DCT) applied to image blocks that are  $8 \times 8$  pixels in size.
2. DCT concentrates information about the pixels in the top-left corner of the  $8 \times 8$  matrix so that the importance of information in the direction of the bottom-right corner decreases.
3. It is then possible to degrade the low information value coefficients by dividing and multiplying them by the so-called quantization matrix.
4. These operations are rounded to whole numbers, so the smallest values become zero and can be deleted to reduce the size of the JPEG.

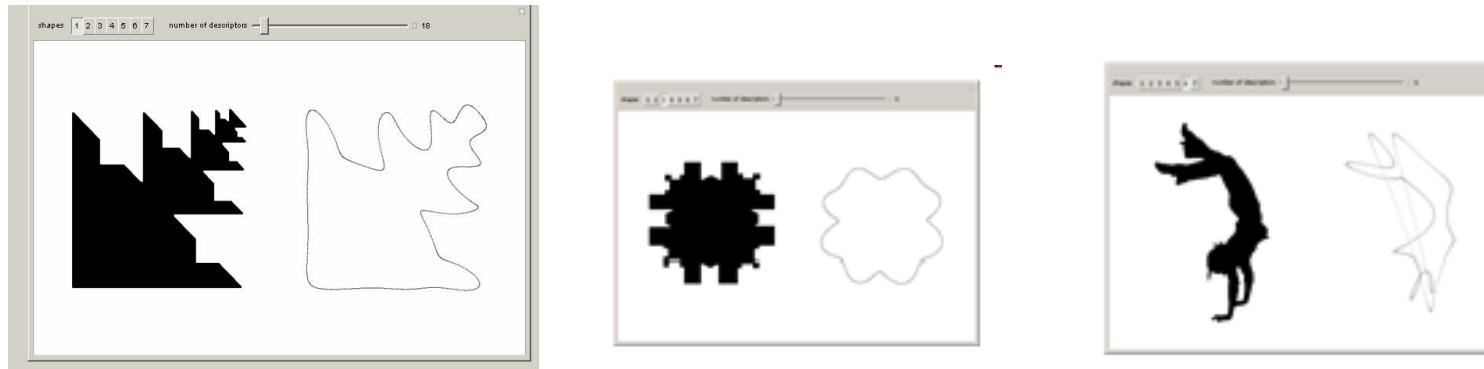
From <http://demonstrations.wolfram.com/JPEGCompressionAlgorithm/>

# Mathematica Demonstration of JPG compression



# Fourier descriptors for shape approximation

1. Fourier descriptors are a way of encoding the shape of a two-dimensional object by taking the Fourier transform of the boundary, where every (x,y) point on the boundary is mapped to a complex number  $x+i y$ .
2. The original shape can be recovered from the inverse Fourier transform.
3. However, if only a few terms of the inverse are used, the boundary becomes simplified, providing a way to smooth or filter the boundary.
4. From <http://demonstrations.wolfram.com/FourierDescriptors/>



# Fourier descriptors details

The Fourier descriptors of a shape are calculated as follows.

1. Find the coordinates of the edge pixels of a shape and put them in a list in order, going clockwise around the shape.
2. Define a complex-valued vector using the coordinates obtained.

For example:

3. Take the discrete Fourier transform of the complex-valued vector.

Fourier descriptors inherit several properties from the Fourier transform.

4. Translation invariance: no matter where the shape is located in the image, the Fourier descriptors remain the same.
5. Scaling: if the shape is scaled by a factor, the Fourier descriptors are scaled by that same factor.
6. Rotation and starting point: rotating the shape or selecting a different starting point only affects the phase of the descriptors.

# Fourier descriptors details

Because the discrete Fourier transform is invertible, all the information about the shape is contained in the Fourier descriptors.

A common thing to do with Fourier descriptors is to set the descriptors corresponding to values above a certain frequency to zero and then reconstruct the shape.

The effect of this is a low-pass filtering of the shape, smoothing the boundary. Since many shapes can be approximated with a small number of parameters, Fourier descriptors are commonly used to classify shapes.

The slider lets you choose how many terms to use in the reconstruction. With more terms, the shape looks more like the original. With fewer terms, the shape becomes smoother and rounder.

The basic method of Fourier descriptors is discussed in R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Englewood Cliffs, NJ: Prentice Hall, 2007.

# Dynamic programming

- Fibonacci Numbers
- Longest Common Subsequence (LCS)
- Shortest Path

# Dynamic programming

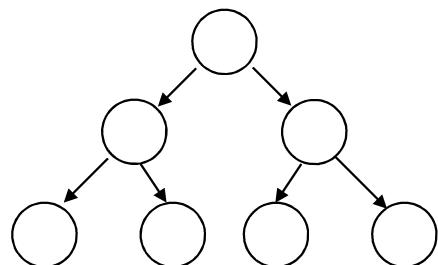
- One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the **same computations being performed repeatedly** since *identical* sub-instances may arise.
- The idea behind ***dynamic programming*** is to avoid this pathology by obviating the requirement to calculate the same quantity twice.
- The method usually accomplishes this by maintaining a *table of sub-instance results*.

# Bottom up

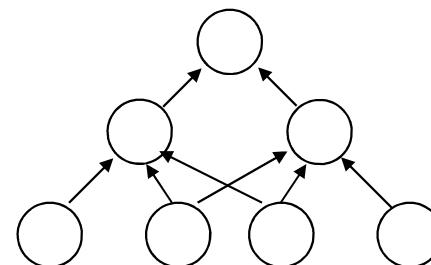
- Dynamic Programming is a **Bottom-Up Technique** in which the smallest sub-instances are *explicitly* solved first and the results of these used to construct solutions to progressively larger sub-instances.
- In contrast, Divide-and-Conquer is a **Top-Down Technique** which *logically* progresses from the initial instance down to the smallest sub-instance via intermediate sub-instances.
- **dynamic programming**. There are a couple of standard ways to progress:
  - memoization
  - converting from top-down to bottom-up

# DYNAMIC PROGRAMMING

- *Dynamic programming* solves a problem by partitioning the problem into sub-problems.
  - If the subproblems are *independent*: use divide-and-conquer method.
  - If the subproblems are *dependent*: use dynamic programming.
- A dynamic programming algorithm solves every subproblem once and **saves its answer to sub-problems in a table** for reuse it.



divide-and-conquer

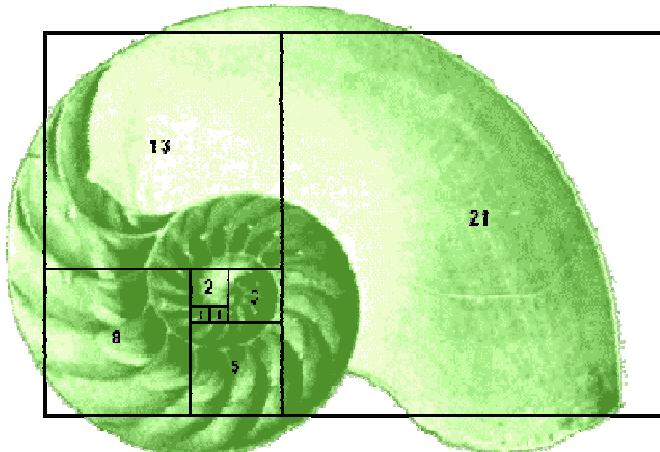


Dynamic Programming

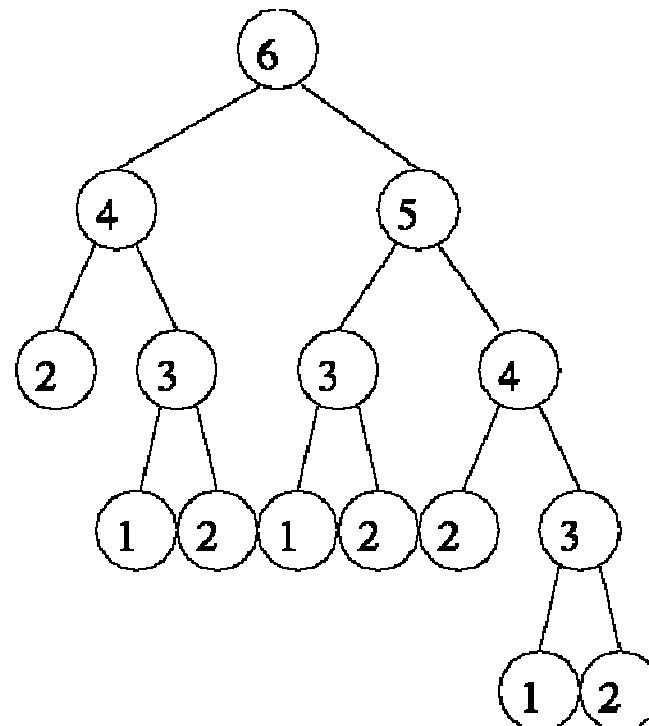
# Computing the Fibonacci sequence the slow way

```
int fibo_very_slow(n)
    if n < 2 then return 1
    else return f(n-1) + f(n-2);
```

$\Theta(2^n)$  time and  $\Theta(n)$  space.



fibo(6) call tree with duplicate calls



# Memoization

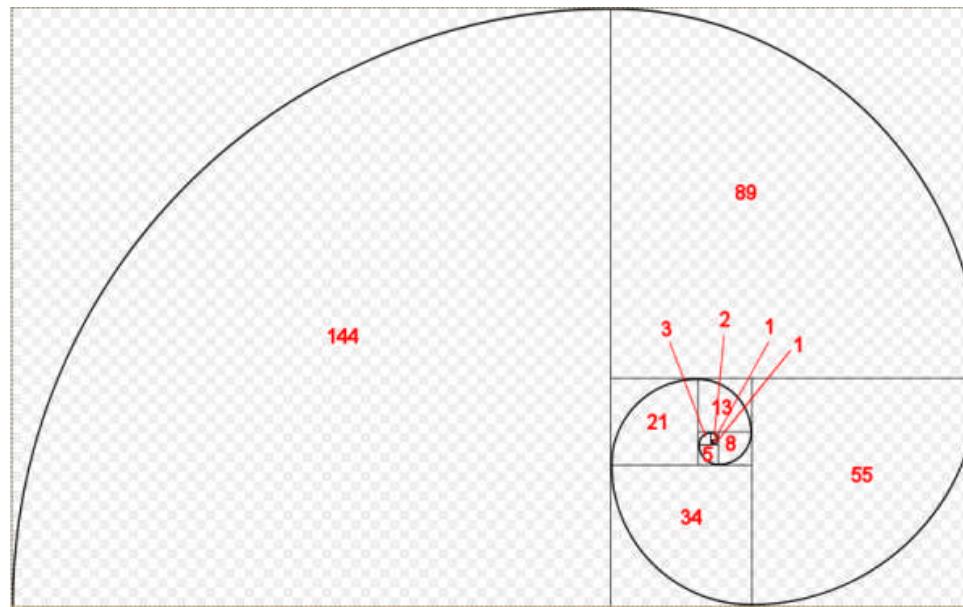
Remember earlier solutions (memoization)

```
int fibo_dynamic(int n)
    static saved[N]
    if n < 2 then return 1
    if saved[n] then return saved[n];
    saved[N] = f(n-1) + f(n-2)
    return saved[N]
```

$\Theta(n)$  time and  $\Theta(n)$  space

# Fibonacci numbers

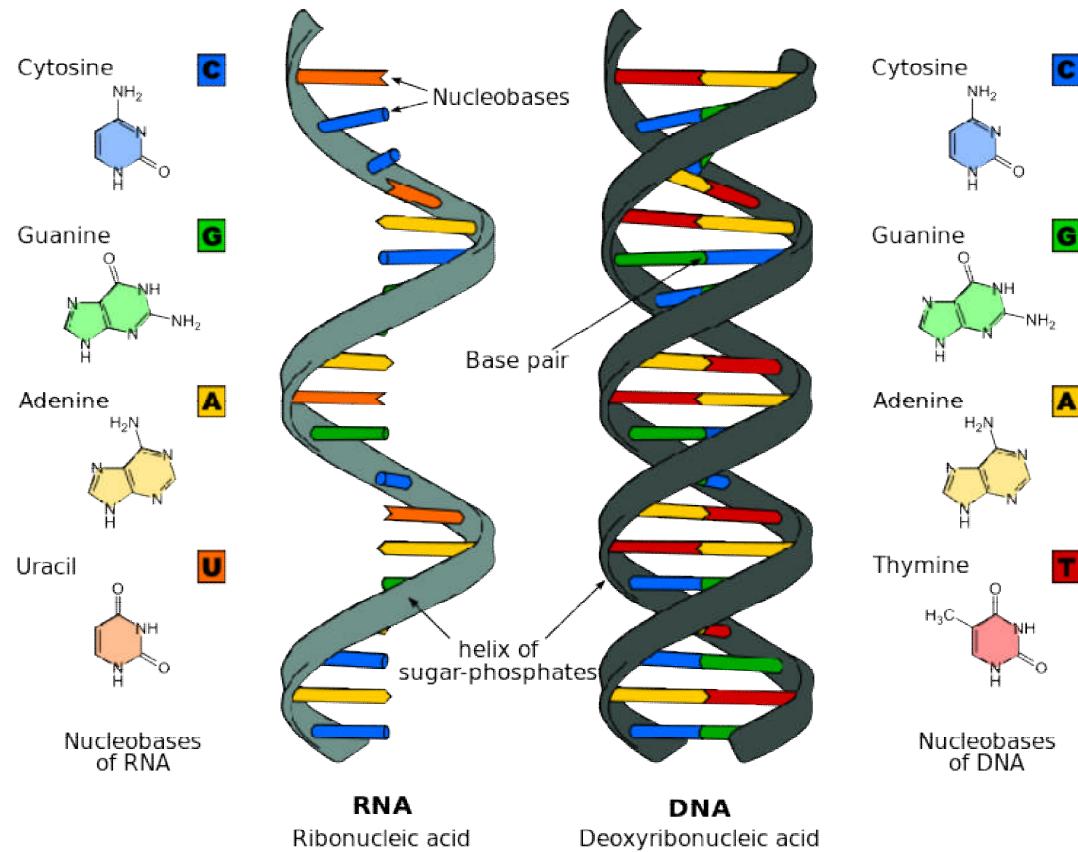
```
int fibo_fast(int n)
If n < 2 then return 1
f1 = f2 = 1;
for k =1 to n
    f1 = f1 + f2
    f2 = f1
return f1
Θ(n) time and
Θ(1) space
```



# LCS: Longest Common Subsequence

- Example of Dynamic programming
- Matching two strings with missing characters

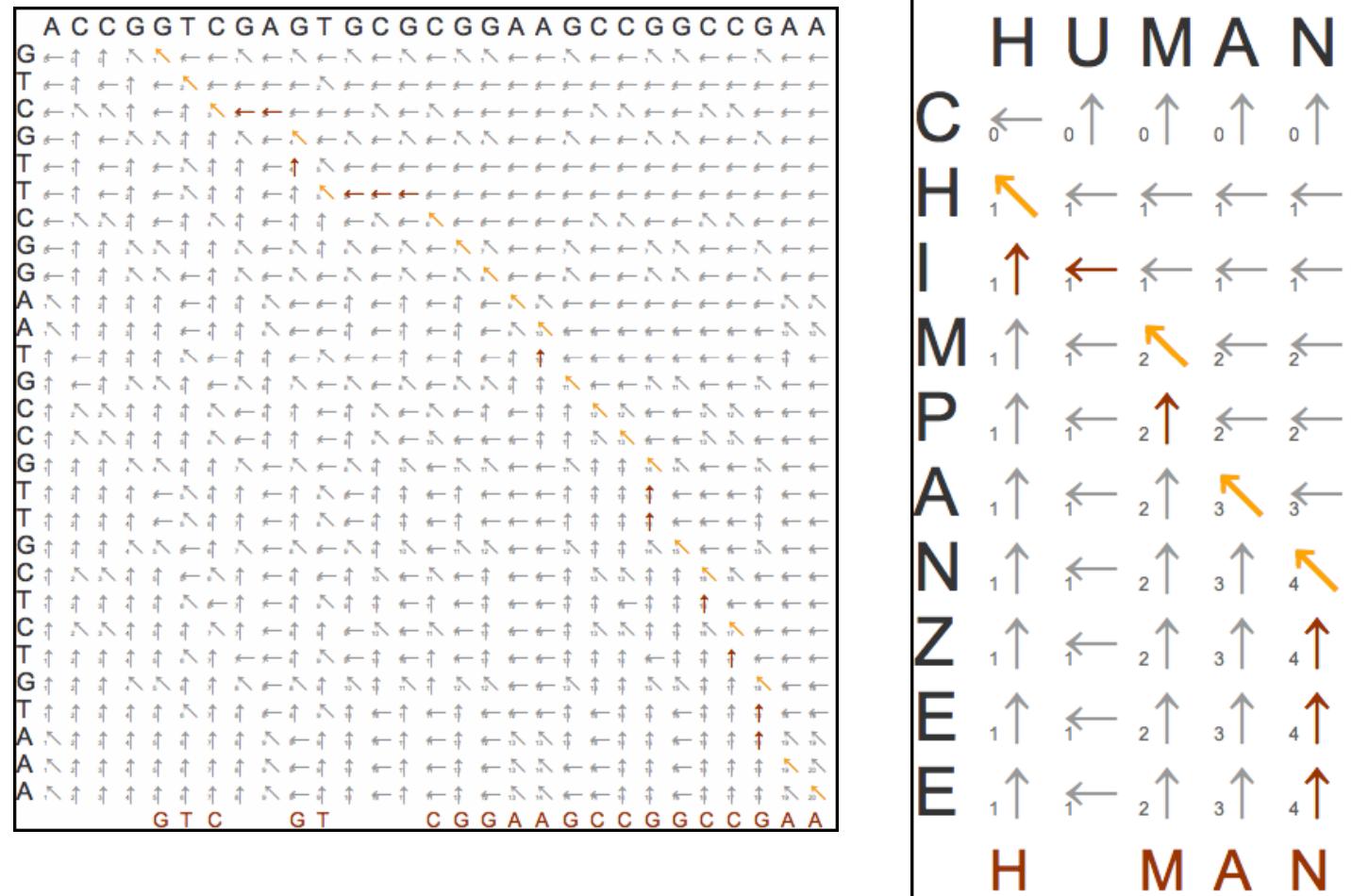
DNA are strings over {C,G,A,T}  
RNA are strings over {C,G,A,U}



# DNA Matching

- The main objective of DNA analysis is to get a visual representation of DNA left at the scene of a crime.
- A DNA "picture" features columns of dark-colored parallel bands and is equivalent to a fingerprint lifted from a smooth surface.
- To identify the owner of a DNA sample, the DNA "fingerprint," or profile, must be matched, either to DNA from a suspect or to a DNA profile stored in a database.

# LCS example in DNA matching



# The longest common subsequence (LCS) problem

**string** : A = b a c a d

**subsequence** of A is formed by deleting 0 or more symbols from A (not necessarily consecutive).

e.g. ad, ac, bac, acad, bacad, bcd.

**Common subsequences** of

A = b a c a d

B = a c c b a d c b

Are: ad, ac, bac, acad.

The **longest common subsequence (LCS)** of A and B:

a c a d.

- *Substring* LONGEST COMMON SUBSEQUENCE
  - CBD is a substring of AB**CBD**A B

### *Subsequence*

- *BCDB* is a subsequence of A**BCBD**A B
- *Common subsequence*
  - *BCA* is a common subsequence of  
 $X = A\textcolor{blue}{B}C\textcolor{blue}{B}D\textcolor{blue}{A}B$  and  $Y = \textcolor{blue}{B}DCABA$
  - *BCBA* is the longest common subsequence of  $X$  and  $Y$   
 $X = A \textcolor{blue}{B} \textcolor{blue}{C} \textcolor{blue}{B} D \textcolor{blue}{A} B$   
 $Y = \textcolor{blue}{B} D \textcolor{blue}{C} A \textcolor{blue}{B} A$



## **LCS problem**

Given two sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$   
to find an LCS of  $X$  and  $Y$ .

- **Brute force approach**
  - Enumerate all subsequences of  $X$  and  $Y$
  - Find the common to both.
  - Find the longest one.
- Is Infeasible: space  $\Theta(2^m + 2^n)$ , time  $\Theta(2^{m+n})$ ,  
(There are  $2^m$  subsequences of  $X$  is)

- LCS data structure is a table c
- $X = X_1 X_2 X_3 \dots \dots \dots X_n$
  - $Y = Y_1 Y_2 Y_3 \dots Y_m$
  - $c[i, j]$  is table of length(LCS( $X[1..i], Y[1..j]$ ))

Where:

$c[0,i] = c[i,0] = 0$  // first row and column are zero.

$$\begin{aligned}c[i,j] &= 1+c[i-1,j-1] && \text{if } X_i == Y_j \\&= \max(c[i,j-1],c[i-1,j]) && \text{otherwise}\end{aligned}$$

- **Time complexity:**  $O(mn)$
- **Space complexity:**  $O(mn)$

# LCS sub-problem table computation

| $j$ | 0     | 1 | 2 | 3  | 4  | 5  | 6  |
|-----|-------|---|---|----|----|----|----|
| $i$ | $y_j$ | B | D | C  | A  | B  | A  |
| 0   | $x_i$ | 0 | 0 | 0  | 0  | 0  | 0  |
| 1   | A     | 0 | 0 | 0  | 0  | 1  | -1 |
| 2   | B     | 0 | 1 | -1 | -1 | 1  | -2 |
| 3   | C     | 0 | 1 | 1  | 2  | -2 | 2  |
| 4   | B     | 0 | 1 | 1  | 2  | 2  | 3  |
| 5   | D     | 0 | 1 | 2  | 2  | 2  | 3  |
| 6   | A     | 0 | 1 | 2  | 2  | 3  | 4  |
| 7   | B     | 0 | 1 | 2  | 2  | 3  | 4  |

// first row and column are zero.

$$c[0,i] = c[i,0] = 0$$

$$\begin{aligned} c[i,j] &= 1 + c[i-1,j-1] \text{ if } X_i == Y_j \\ &= \max(c[i,j-1], c[i-1,j]) \text{ otherwise} \end{aligned}$$

PRINT-LCS( $i, j$ )

1 if  $i = 0$  or  $j = 0$

2 then return

3 if  $c[i, j] == 1 + c[i-1,j-1]$  then

4 PRINT-LCS( $i - 1, j - 1$ )

5 print  $x[i]$

6 else if  $c[i, j] == c[i-1,j]$  then

7 PRINT-LCS( $b, X, i - 1, j$ )

8 else PRINT-LCS( $b, X, i, j - 1$ )

# LCS example

|     | -1  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | C   | A   | G   | A   | T   | A   | G   | A   | G   |     |
| -1  | (0) | (0) | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 0 A | 0   | 0   | (1) | 1   | 1   | 1   | 1   | 1   | 1   | 1   |
| 1 G | 0   | 0   | 1   | (2) | (2) | (2) | (2) | 2   | 2   | 2   |
| 2 C | 0   | 1   | 1   | (2) | (2) | (2) | (2) | 2   | 2   | 2   |
| 3 G | 0   | 1   | 1   | 2   | 2   | 2   | 2   | (3) | 3   | 3   |
| 4 A | 0   | 1   | 2   | 2   | 3   | 3   | 3   | 3   | (4) | (4) |

This corresponds to the four following alignments:

$$\left( \begin{array}{cccccccccc} - & A & G & C & - & - & - & G & A & - \\ C & A & G & - & A & T & A & G & A & G \end{array} \right)$$

$$\left( \begin{array}{cccccccccc} - & A & G & - & C & - & - & G & A & - \\ C & A & G & A & - & T & A & G & A & G \end{array} \right)$$

$$\left( \begin{array}{cccccccccc} - & A & G & - & - & C & - & G & A & - \\ C & A & G & A & T & - & A & G & A & G \end{array} \right)$$

$$\left( \begin{array}{cccccccccc} - & A & G & - & - & - & C & G & A & - \\ C & A & G & A & T & A & - & G & A & G \end{array} \right)$$

# Exercises

Build the LCS table for

1. X=ABC, Y=ABC
2. X=AAA, Y=BBB
3. X=ABA, Y=BAB

# Solutions

Build the LCS table for

1.  $X=ABC, Y=ABC$
2.  $X=AAA, Y=BBB$
3.  $X=ABA, Y=BAB$

| j | 0 | 1             | 2 | 3 |
|---|---|---------------|---|---|
| i | y | A             | B | C |
| 0 | x | 0             | 0 | 0 |
| 1 | A | 0 ↘ 1 ← 1 ← 1 |   |   |
| 2 | B | 0 ↑ 1 ↘ 2 ← 2 |   |   |
| 3 | C | 0 ↑ 1 ↑ 2 ↘ 3 |   |   |

| j | 0 | 1             | 2 | 3 |
|---|---|---------------|---|---|
| i | y | B             | B | B |
| 0 | x | 0             | 0 | 0 |
| 1 | A | 0 ↑ 0 ↑ 0 ↑ 0 |   |   |
| 2 | A | 0 ↑ 0 ↑ 0 ↑ 0 |   |   |
| 3 | A | 0 ↑ 0 ↑ 0 ↑ 0 |   |   |

| j | 0 | 1             | 2 | 3 |
|---|---|---------------|---|---|
| i | y | B             | A | B |
| 0 | x | 0             | 0 | 0 |
| 1 | A | 0 ↑ 0 ↘ 1 ← 1 |   |   |
| 2 | B | 0 ↘ 1 ↑ 1 ↘ 2 |   |   |
| 3 | A | 0 ↑ 1 ↘ 2 ↑ 2 |   |   |

# Homework

Find LCS of

1. X=TAGAGA,

Y=GAGATA

2. X=YOUR\_FIRST\_NAME

Y=YOUR\_LAST\_NAME

# Homework Solution

Find LCS of X=TAGAGA, Y=GAGATA

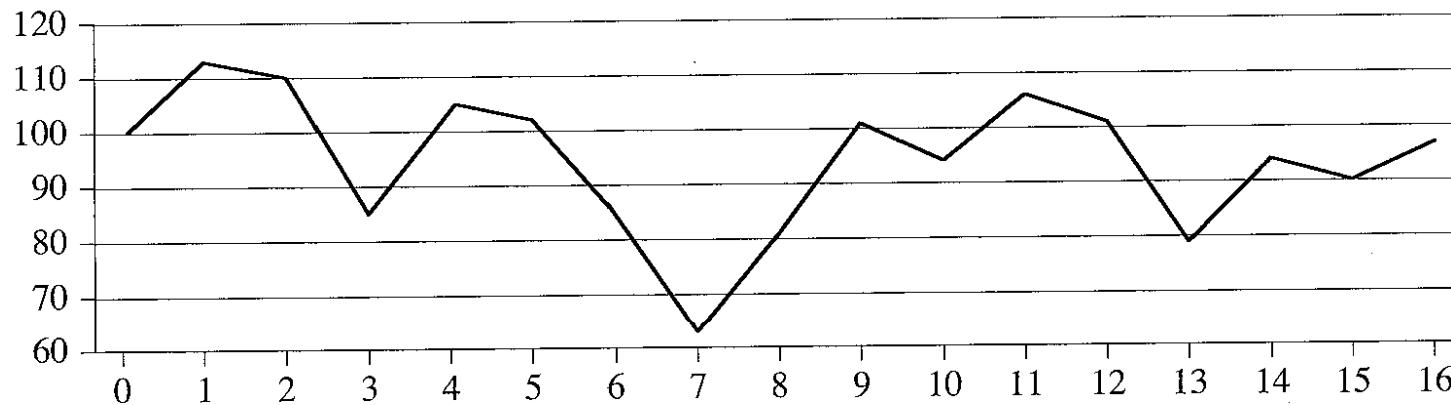
|   |   |   |     |     |     |     |     |     |
|---|---|---|-----|-----|-----|-----|-----|-----|
|   | j | 0 | 1   | 2   | 3   | 4   | 5   | 6   |
| i | y | G | A   | G   | A   | T   | A   |     |
| 0 | x | 0 | 0   | 0   | 0   | 0   | 0   | 0   |
| 1 | T | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↑ 1 | ↑ 1 |
| 2 | A | 0 | ↑ 0 | ↑ 1 | ↑ 1 | ↑ 1 | ↑ 1 | ↑ 2 |
| 3 | G | 0 | ↑ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↑ 2 | ↑ 2 |
| 4 | A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 | ↑ 3 |
| 5 | G | 0 | ↑ 1 | ↑ 2 | ↑ 3 | ↑ 3 | ↑ 3 | ↑ 3 |
| 6 | A | 0 | ↑ 1 | ↑ 2 | ↑ 3 | ↑ 4 | ↑ 4 | ↑ 4 |

# Max subarray sum

(example of dynamic programming)

# Max subarray sum

Example:

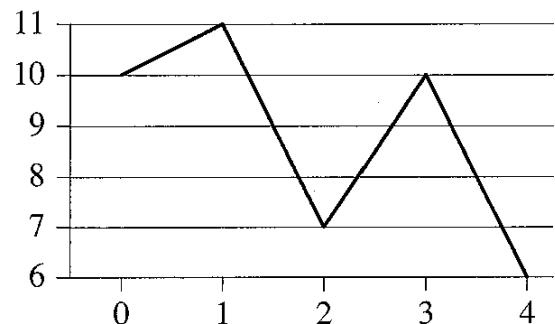


| Day    | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8  | 9   | 10 | 11  | 12  | 13  | 14 | 15 | 16 |
|--------|-----|-----|-----|-----|-----|-----|-----|-----|----|-----|----|-----|-----|-----|----|----|----|
| Price  | 100 | 113 | 110 | 85  | 105 | 102 | 86  | 63  | 81 | 101 | 94 | 106 | 101 | 79  | 94 | 90 | 97 |
| Change |     | 13  | -3  | -25 | 20  | -3  | -16 | -23 | 18 | 20  | -7 | 12  | -5  | -22 | 15 | -4 | 7  |

**Figure 4.1** Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

# Max subarray sum

**Another Example:** buying low and selling high, even with perfect knowledge, is not trivial:



| Day    | 0  | 1  | 2  | 3  | 4  |
|--------|----|----|----|----|----|
| Price  | 10 | 11 | 7  | 10 | 6  |
| Change |    | 1  | -4 | 3  | -4 |

**Figure 4.2** An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

# Max subarray sum

**First Solution:** compute the value change of each subarray corresponding to each pair of dates, and find the maximum.

1. How many pairs of dates:  $\binom{n}{2}$
2. This belongs to the class  $\Theta(n^2)$
3. The rest of the costs, although possibly constant, don't improve the situation:  $\Omega(n^2)$ .

Not a pleasant prospect if we are rummaging through long time-series (Who told you it was easy to get rich???), even if you are allowed to post-date your stock options...

# Max subarray sum

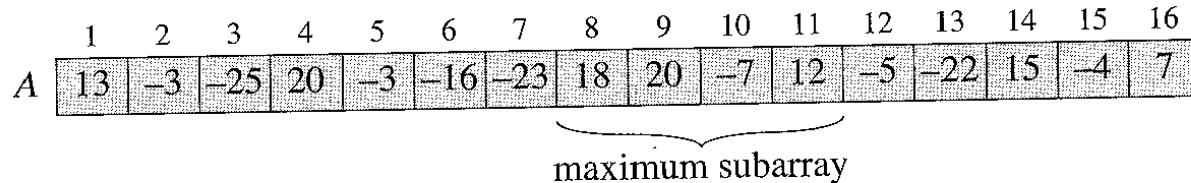
We are going to find an algorithm with an  $O(n^2)$  running time (i.e. **strictly** asymptotically faster than  $n^2$ ), which should allow us to look at longer time-series.

**Transformation:** Instead of the daily price, let us consider the daily change:  $A[i]$  is the difference between the closing value on day  $i$  and that on day  $i-1$ .

The problem becomes that of finding a contiguous subarray the sum of whose values is maximum.

On a first look this seems even worse: roughly the same number of intervals (one fewer, to be precise), and the requirement to add the values in the subarray rather than just computing a difference:  
 $\Omega(n^3)$ ?

## Max subarray sum



**Figure 4.3** The change in stock prices as a maximum-subarray problem. Here, the subarray  $A[8..11]$ , with sum 43, has the greatest sum of any contiguous subarray of array  $A$ .

It is actually possible to perform the computation in  $\Theta(n^2)$  time by

1. Computing all the daily changes;
2. Computing the changes over 2 days (one addition each)
3. Computing the changes over 3 days (one further addition to extend the length-2 arrays... etc... check it out. You'll need a two-dimensional array to store the intermediate computations.

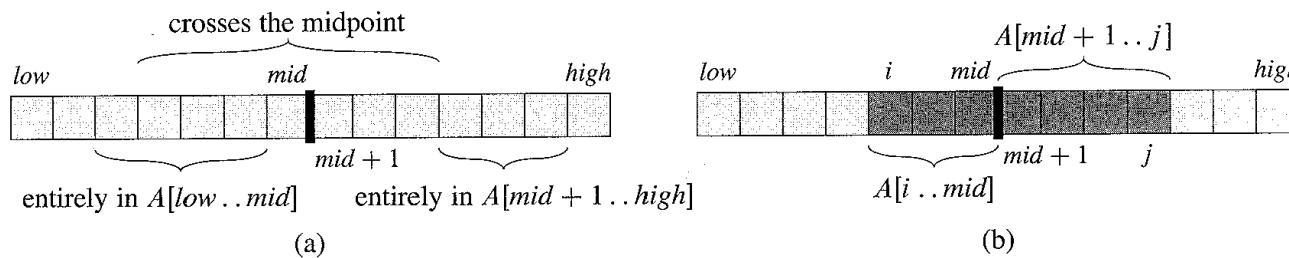
Still bad though. Can we do better??

# Max subarray sum

How do we divide?

We observe that a maximum contiguous subarray  $A[i \dots j]$  must be located as follows:

1. It lies entirely in the left half of the original array:  $[low \dots mid]$ ;
2. It lies entirely in the right half of the original array:  $[mid + 1 \dots high]$ ;
3. It straddles the midpoint of the original array:  $i \leq mid < j$ .



**Figure 4.4** (a) Possible locations of subarrays of  $A[low \dots high]$ : entirely in  $A[low \dots mid]$ , entirely in  $A[mid + 1 \dots high]$ , or crossing the midpoint  $mid$ . (b) Any subarray of  $A[low \dots high]$  crossing the midpoint comprises two subarrays  $A[i \dots mid]$  and  $A[mid + 1 \dots j]$ , where  $low \leq i \leq mid$  and  $mid < j \leq high$ .

# Max subarray sum

The “left” and “right” subproblems are smaller versions of the original problem, so they are part of the standard Divide & Conquer recursion. The “middle” subproblem is not, so we will need to count its cost as part of the “combine” (or “divide”) cost.

The crucial observation (and it may not be entirely obvious) is that **we can find the maximum crossing subarray in time linear in the length of the A[low...high] subarray.**

**How?**  $A[i, \dots, j]$  must be made up of  $A[i \dots mid]$  and  $A[m+1 \dots j]$  – so we find the largest  $A[i \dots mid]$  and the largest  $A[m+1 \dots j]$  and combine them.

# Max subarray sum

The Algorithm:

FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )

```
1  left-sum = -∞
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum = -∞
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

# Max subarray sum

The total numbers of iterations for both loops is exactly  $\text{high-low+1}$ .

The left-recursion will return the **indices and value** for the largest contiguous subarray in the left half of  $A[\text{low...high}]$ , the right recursion will return the **indices and value** for the largest contiguous subarray in the right half of  $A[\text{low...high}]$ , and FIND-MAX-CROSSING-SUBARRAY will return the **indices and value** for the largest contiguous subarray that straddles the midpoint of  $A[\text{low...high}]$ .

It is now easy to choose the contiguous subarray with largest value and return its endpoints and value to the caller.

# Max subarray sum

The recursive algorithm:

FIND-MAXIMUM-SUBARRAY( $A, low, high$ )

```
1  if  $high == low$ 
2      return ( $low, high, A[low]$ )           // base case: only one element
3  else  $mid = \lfloor (low + high)/2 \rfloor$ 
4      ( $left-low, left-high, left-sum$ ) =
            FIND-MAXIMUM-SUBARRAY( $A, low, mid$ )
5      ( $right-low, right-high, right-sum$ ) =
            FIND-MAXIMUM-SUBARRAY( $A, mid + 1, high$ )
6      ( $cross-low, cross-high, cross-sum$ ) =
            FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ )
7      if  $left-sum \geq right-sum$  and  $left-sum \geq cross-sum$ 
8          return ( $left-low, left-high, left-sum$ )
9      elseif  $right-sum \geq left-sum$  and  $right-sum \geq cross-sum$ 
10         return ( $right-low, right-high, right-sum$ )
11     else return ( $cross-low, cross-high, cross-sum$ )
```

# Max subarray sum

The analysis:

1. The base case  $n = 1$ . This is easy:  $T(1) = \Theta(1)$ , as one might expect.
2. The recursion case  $n > 1$ . We make the simplifying assumption that  $n$  is a power of 2, so we can always “split in half”.  
 $T(n) = \text{cost of splitting} + 2T(n/2) + \text{cost of finding largest midpoint-straddling subarray} + \text{cost of comparing the three subarray values and returning the correct triple.}$   
The cost of splitting is constant =  $\Theta(1)$ ; the cost of finding the largest straddling subarray is  $\Theta(n)$ ; the cost of the final comparison and return is constant  $\Theta(1)$ .

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1).$$

# Max subarray sum

We finally have:

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

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution  $T(n) = \Theta(n \lg n)$ .

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.