Decrease and Conquer

Agenda

1. Overview
2. Decrease-by-a-constant: Insertion Sort
3. Decrease-by-a-constant: Topological Sorting
4. Decrease-by-a-constant-factor Algorithms
5. Variable-size Decrease Algorithms

1. Overview

Learning objectives

- Understand the ***Decrease & Conquer*** approach
- Understand and apply:
  - Decrease-by-**a-constant** algorithms (Insertion Sort and Topological Sort)
  - Decrease-by-**a-constant-factor** algorithms (Binary Search, Fake Coin problem, and Fast Exponentiation)
  - **Variable-size** decrease algorithms (Binary Search Tree, Interpolation Search, and Quick Select)

Ferrying Soldiers over a River

- A detachment of 25 soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier.
- How can the soldiers get across the river and leave the boys in joint possession of the boat?
- How many times does the boat pass from shore to shore?

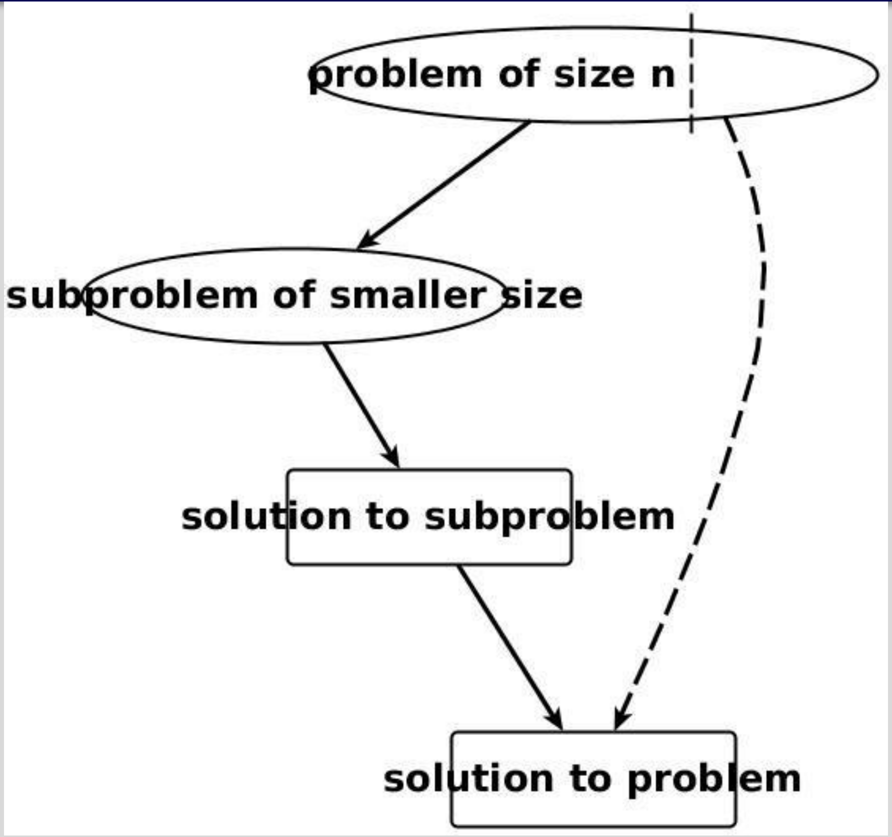
Decrease and Conquer Approach

Process:

1. Reduce a problem instance of size N to a smaller instance of the same problem of size M
2. Solve the smaller instance
3. Extend the solution of the smaller instance to obtain the solution to the original instance

Sometimes referred to as the ***inductive*** or ***incremental*** approach.

Decrease and Conquer approach



Decrease and Conquer approach

Decrease-by-**a-constant**

- `M = N – c for some constant c`
- **Insertion Sorting**
- **Topological Sorting**
- Algorithms for generating permutations and subsets

Decrease-by-**a-constant-factor**

- `M = N / c for some constant c`
- **Binary Search**
- **Fake-coin Problem**
- Josephus Problem

Decrease and Conquer approach

Variable-size-decrease

- `M = N - c for some variable c`
- Search, Insert and Delete in a **Binary Search Tree**
- Euclid’s Algorithm (to find the greatest common divisor)
- **Interpolation Search**
- **Quick Select**

Agenda

1. Overview
2. Decrease-by-a-constant: Insertion Sort
3. Decrease-by-a-constant: Topological Sorting
4. Decrease-by-a-constant-factor Algorithms
5. Variable-size decrease Algorithms
6. Summary

2. Insertion Sort

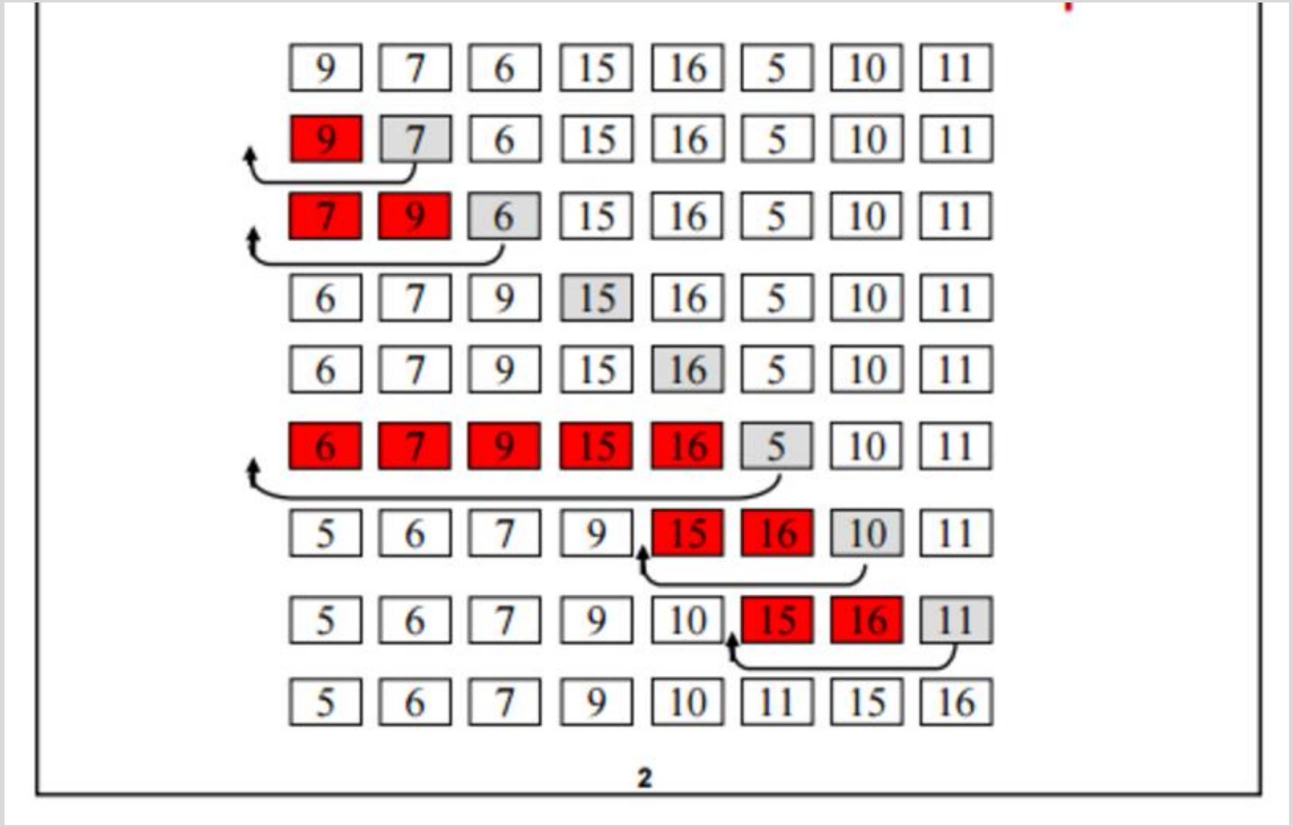
Insertion Sort – Idea

1. Consider each element one at a time (left to right).
2. Insert each element in its proper place among those already considered. (i.e., insert into an already sorted sub-file). This is a right to left scan.
3. It is a decrease-by-**a-constant** algorithm
   - Why? Because every time the array that needs to be sorted the problem gets smaller by one

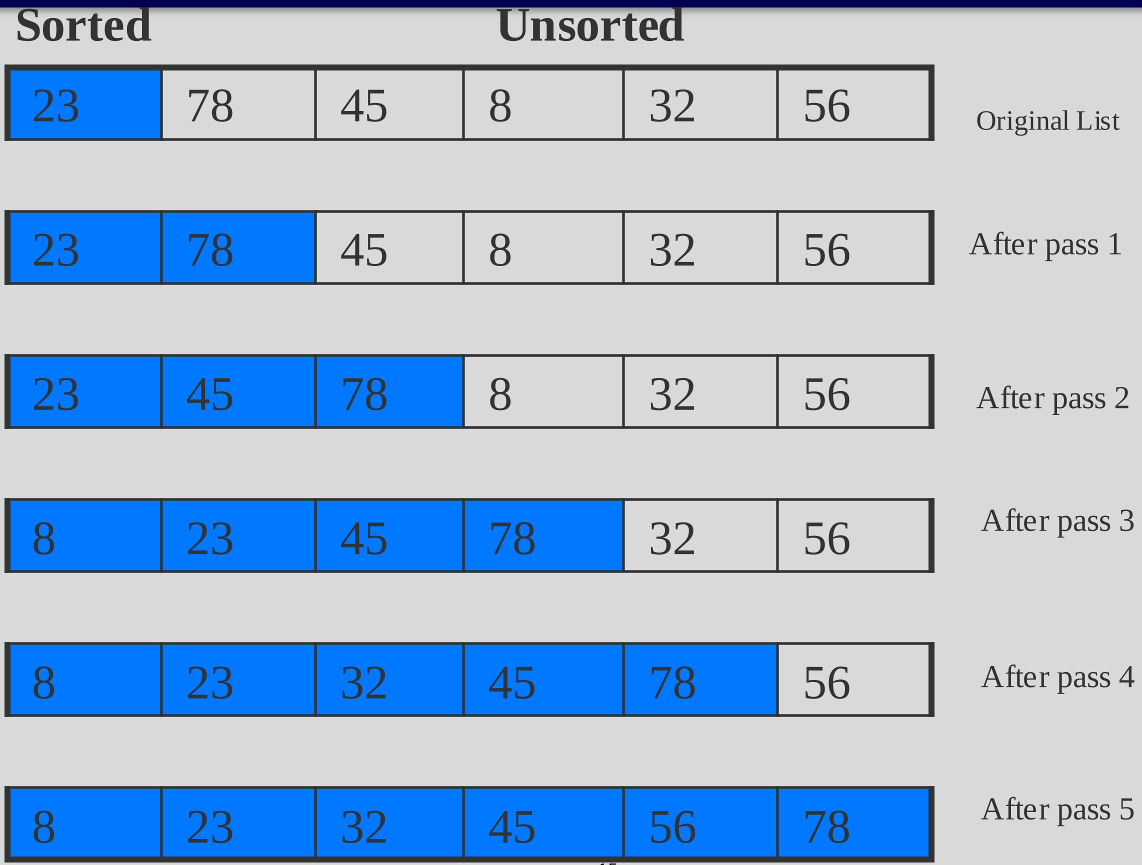
Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs
    - Most common sorting technique used by card players
- The list is divided into two parts: sorted and unsorted
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sub-list, and inserted at the appropriate place
- A list of *n* elements will take at most *n-1* passes to sort the data.

Insertion Sort



Insertion Sort



Sorted			Unsorted			
23	78	45	8	32	56	Original List
23	78	45	8	32	56	After pass 1
23	45	78	8	32	56	After pass 2
8	23	45	78	32	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

Insertion Sort – Pseudocode

ALGORITHM **InsertionSort** ($A[0 \dots n-1]$)
/* Sort an array using an insertion sort. */
/* INPUT : An array $A[0 \dots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \dots n-1]$ sorted in order. */

```
1: for $i = 1$ to $n-1$ do
2:     $v = A[i]$   set the key to be swapped
3:     $j = i-1$
4:     while $j \geq 0$ and $A[j] > v$ do
5:         $A[j+1] = A[j]$
6:         $j = j-1$
7:     end while
8:     $A[j+1] = v$
9: end for
```

A[j] v

$\boxed{5\ 3\ 4\ 1}$

1st pass	3 5 4 1
2nd pass	3 4 5 1
3rd pass	1 3 4 5

Insertion Sort – Time Complexity

- **Worst Case:** the input is in reverse order

$$C_w(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \mathcal{O}(n^2)$$

- **Average Case:** For randomly ordered data, we expect each item to move halfway back.

$$C_a(n) \approx \frac{n^2}{4} \in \mathcal{O}(n^2)$$

Insertion Sort – Time Complexity

- **Best Case:**

  1. In terms of the input, what is the scenario/circumstance where we can achieve the best case?

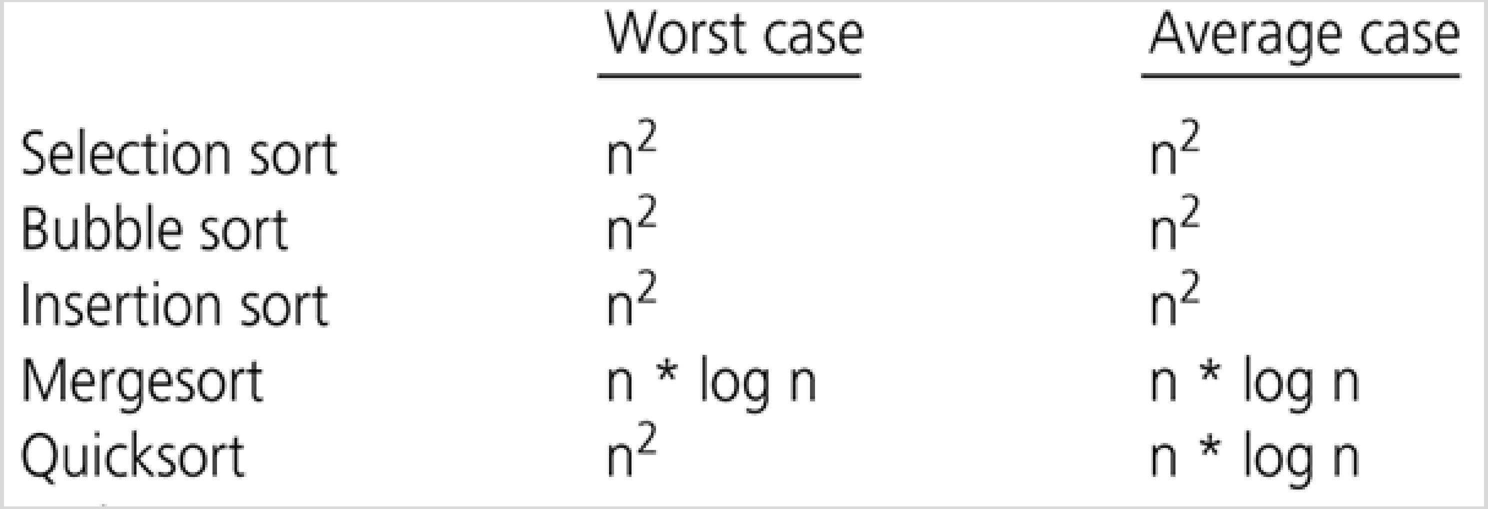
  2. What is the time complexity for the best case?

Is Insertion Sort stable???

Summary of Insertion Sort

- Running time depends on not only the size of the array but also the contents of the array
- **Best-case:➔ O(n)**
    - Array is already sorted in ascending order
    - Inner loop will not be executed
    - The number of moves: 2*(n-1) ➔ O(n)
    - The number of key comparisons: (n-1) ➔ O(n)
- **Worst-case: ➔ O(n^2)**
    - Array is in reverse order
    - Inner loop is executed i-1 times, for i = 2,3, …, n
    - The number of moves: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ O(n^2)
    - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ O(n^2)
- **Average-case: ➔ O(n^2)**
    - We have to look at all possible initial data organizations.
- **So, Insertion Sort is O(n^2)**

Comparison of Sorting Algorithms



	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$

Summary

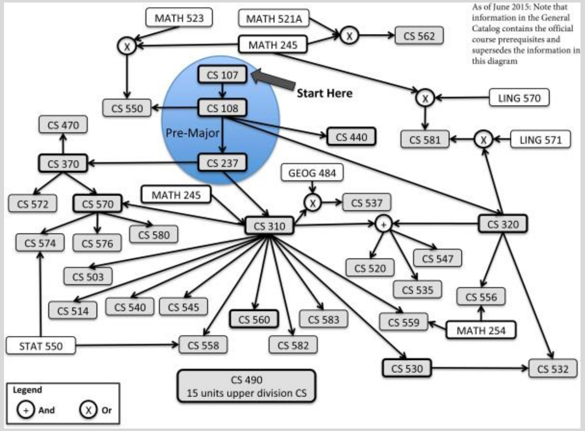
- Insertion sort, bubble sort and selection sort are all O(n^2)
  - In a particular case, one might be better than another. For small problems, insertion sort is a good choice. However, for large problems they all are slow
- Quick sort and merge sort are very efficient
  - The average case for quick sort is **O($n*\log_2 n$) and is** among the fastest known. However, its worst-case behavior of **O(n^2)** is significantly slower than merge sort
  - Merge sort is not quite as fast as quick sort, but its complexity is consistently good - **O($n*\log_2 n$)**. Merge sort also has the disadvantage of requiring an **extra array**

3. Topological Sorting

Imagine the following problems

Job scheduling with order dependencies

What is the order the jobs should be processed to avoid breaking these dependencies?



Subject selection

What are the order the subjects could be taken to ensure we have all the pre-requisites?

Topological Sort – Approaches

There are two different approaches to solve this problem:

- DFS Method
- Source Removal Method (focus of this lecture)
- Problem Modelling
  - A directed graph can be used to represent all jobs
  - If there is an edge from job A to job B: job A must finish before job B

Topological Sort – DFS

- Create a stack and mark all jobs as unvisited
- Iterate through all jobs
  - If the current job J is visited => skip it
  - Process all jobs pointed to by J recursively
  - Mark J as visited
  - Push J to the stack
- Pop all jobs from the stack and print them in popped order

Topological Sort – Source Removal

Source Removal Method:

1. Choose a source vertex
2. A source vertex is the one that has ***no incoming edges***
3. Delete the vertex and all incident edges and append the vertex to topological ordered list
4. Repeat the selection of a source vertex and deletion process for the remaining graph until no vertices are left

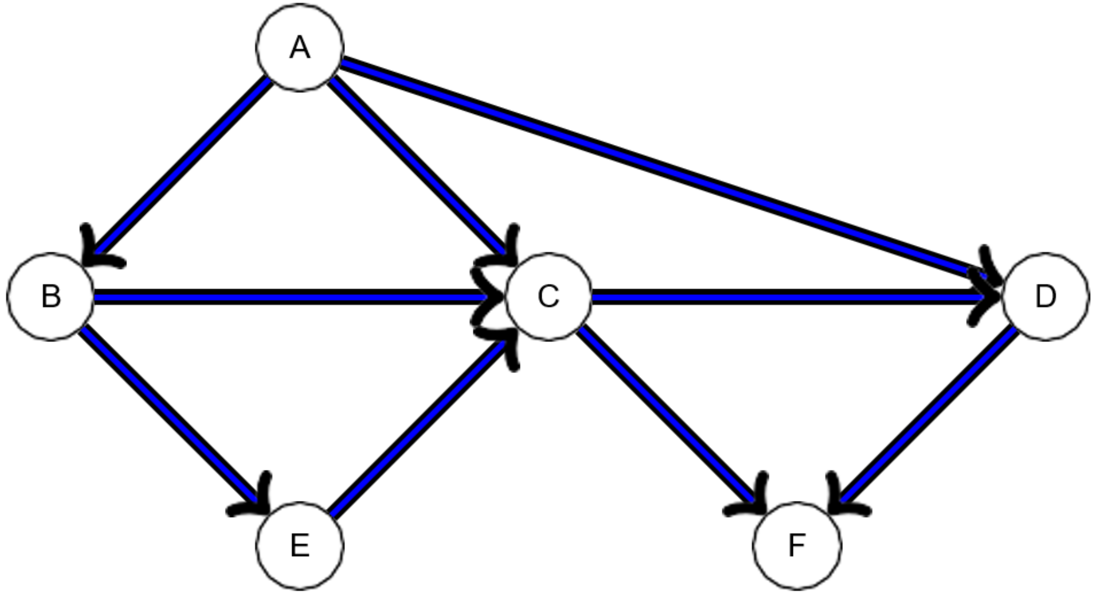
Source Removal - Pseudocode

```
Mark all vertices in V as unvisited
Create an empty queue Q
Calculate the indegree for all vertices
For each vertex v in V
 if v.indegree == 0
    Q.enqueue(v)
    Mark v as visited
While Q is not empty
  add (u = Q.dequeue()) to the result
  for each vertex w pointed to by u
    if w is unvisited
      w.indegree -= 1
      if w.indegree == 0
        Q.enqueue(w)
        Mark w as visited
Return result
```

Topological sort – Demo

Degree(A) = 0, visited
Degree(B) = 1
Degree(C) = 3
Degree(D) = 2
Degree(E) = 1
Degree(F) = 2

Queue: A

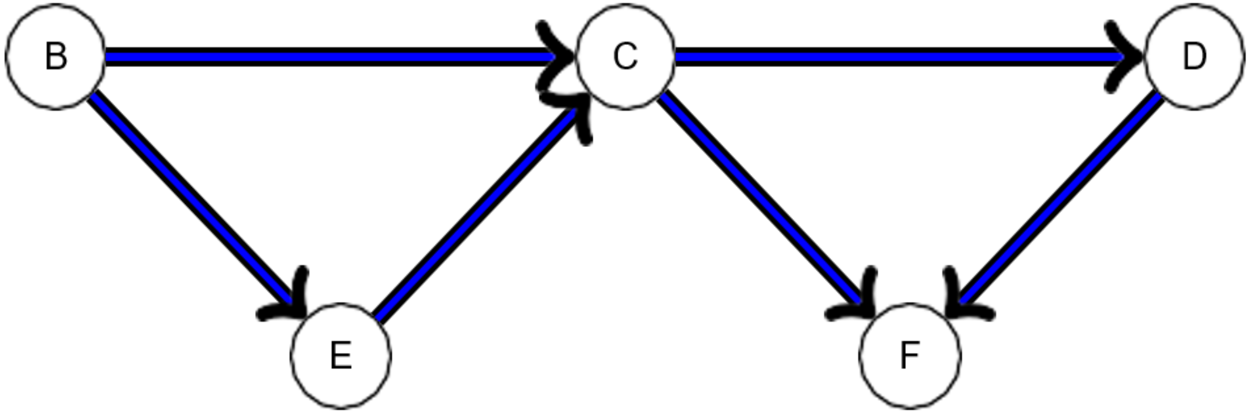


Solution :

Topological sort – Demo

Degree(A) = 0, visited
Degree(B) = 0, visited
Degree(C) = 2
Degree(D) = 1
Degree(E) = 1
Degree(F) = 2

Queue: B

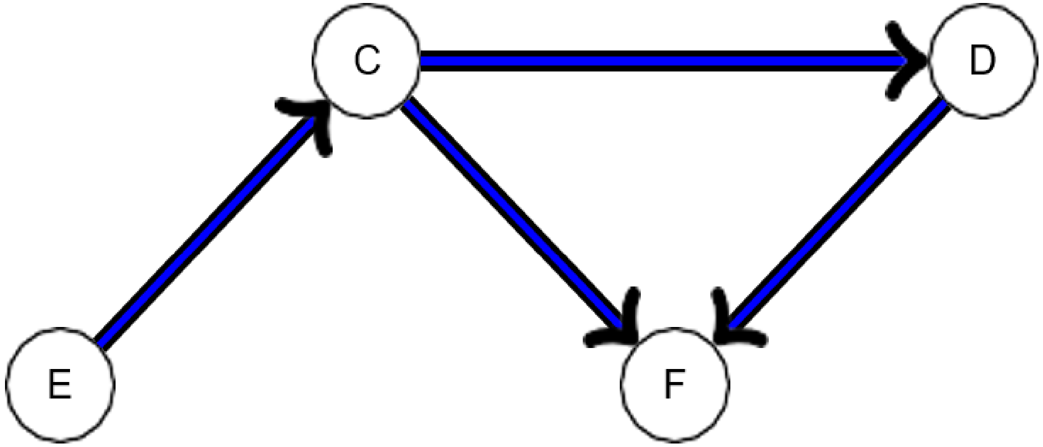


Solution : A

Topological sort – Demo

Degree(A) = 0, visited
Degree(B) = 0, visited
Degree(C) = 1
Degree(D) = 1
Degree(E) = 0, visited
Degree(F) = 2

Queue: E

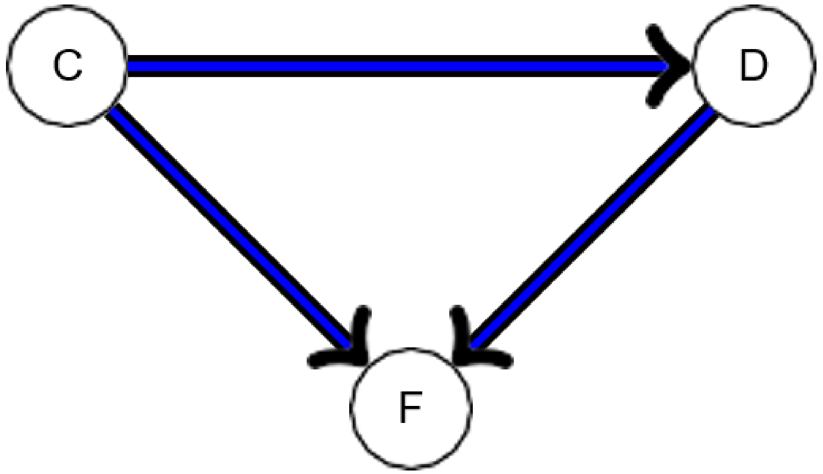


Solution : A B

Topological sort – Demo

Degree(A) = 0, visited
Degree(B) = 0, visited
Degree(C) = 0, visited
Degree(D) = 1
Degree(E) = 0, visited
Degree(F) = 2

Queue: C

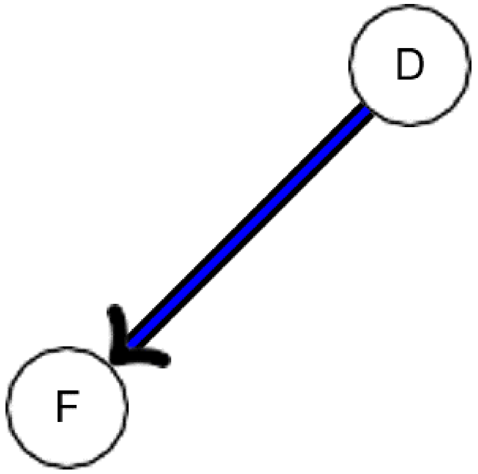


Solution : A B E

Topological sort – Demo

Degree(A) = 0, visited
Degree(B) = 0, visited
Degree(C) = 0, visited
Degree(D) = 0, visited
Degree(E) = 0, visited
Degree(F) = 1

Queue: D

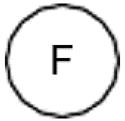


Solution : A B E C

Topological sort – Demo

Degree(A) = 0, visited
Degree(B) = 0, visited
Degree(C) = 0, visited
Degree(D) = 0, visited
Degree(E) = 0, visited
Degree(F) = 0, visited

Queue: F



Solution : A B E C D

Topological sort – Demo

Degree(A) = 0, visited
Degree(B) = 0, visited
Degree(C) = 0, visited
Degree(D) = 0, visited
Degree(E) = 0, visited
Degree(F) = 0, visited

Queue:

Solution : A B E C D F

Topological Sort – Summary

- Topological sorting can have more than one solution, and often does for very large DAGs
- Source removal algorithm:
  - How do you find a source (or determine that such a vertex does not exist) in a digraph represented by an adjacency matrix? What is the time efficiency?
  - How do you find a source (or determine that such a vertex does not exist) in a digraph represented by an adjacency list? What is the time efficiency?

4. Decrease-by-a-constant-factor Algorithms

Decrease-by-a-constant-factor Approaches

- Algorithms that use this approach divide the problem into parts (half, thirds, etc), and then recursively operate on one of the halves, thirds etc.
- Hence, at each iteration, we decrease the problem by a constant factor (a half, a third etc).
- We study three examples:
  - Binary search
  - Fake coin problem
  - Fast exponentiation

Binary Search

- Binary search is a worst-case optimal algorithm for searching in a sorted sequence of elements.
- Given a sorted sequence, compare the value in the array at position *n/2* with a key k .
    - If A[*n/2*] > k , compare k with the midpoint of the lower half.
    - If A[*n/2*] < k , compare k with the midpoint of the upper half.
    - If A[*n/2*] = k , return n/2 (the index of the position of k).

Binary Search – Demo

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92

BINARY SEARCH(5)

Binary Search – Demo

k should be in
this region

A[*n/2*] > 5 (k)

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92

BINARY SEARCH(5)

Binary Search – Demo

A[*n/2]* > 5 (k)

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92

BINARY SEARCH(5)

Binary Search – Demo

A[*n/2]* < 5 (k)

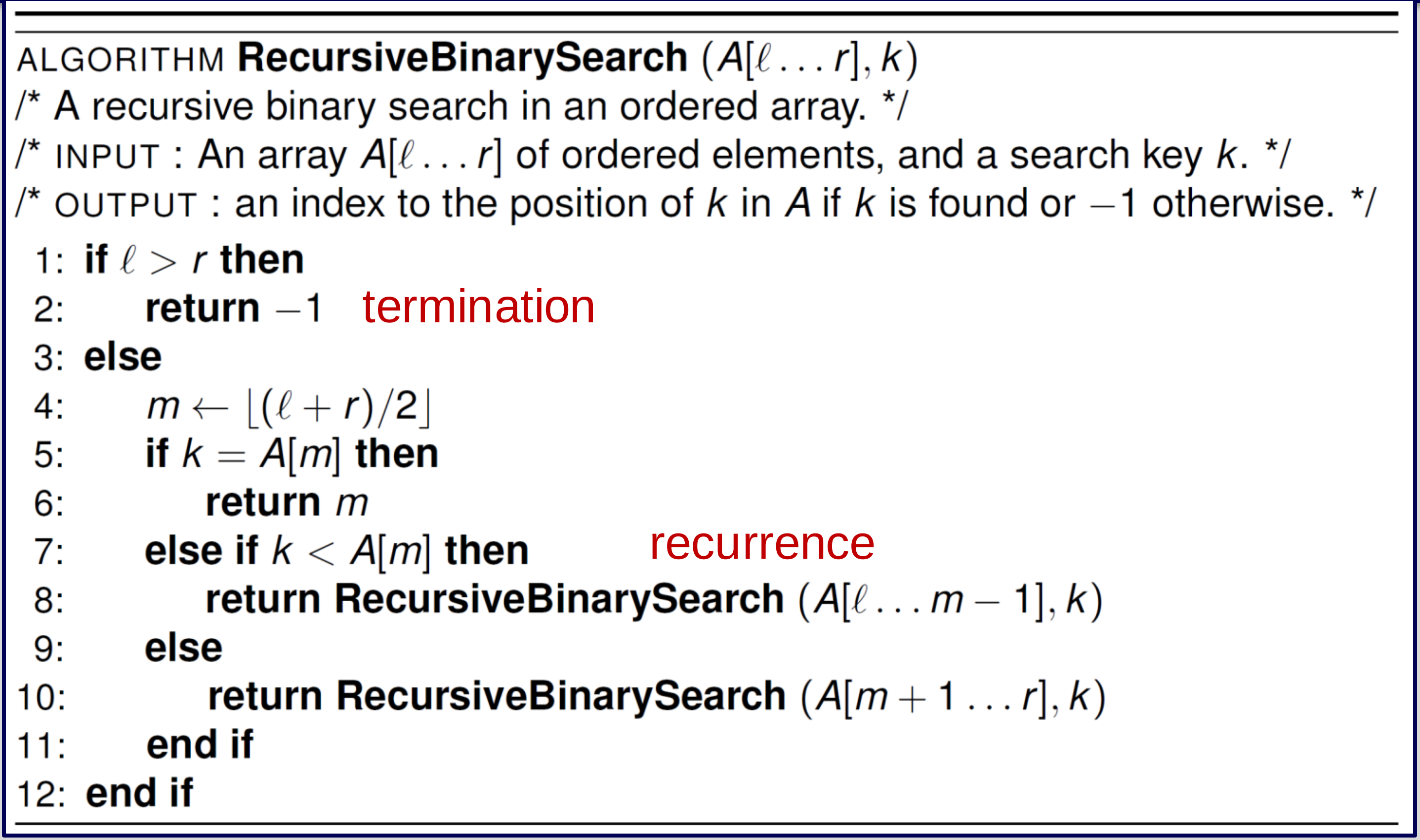
1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92

K should be on the left…

BINARY SEARCH(5)

Oh wait, it's there ☺

Binary Search – Recursive



```
ALGORITHM RecursiveBinarySearch ($A[\ell \dots r], k$)
/* A recursive binary search in an ordered array. */
/* INPUT : An array $A[\ell \dots r]$ of ordered elements, and a search key $k$. */
/* OUTPUT : an index to the position of $k$ in $A$ if $k$ is found or $-1$ otherwise. */
if $\ell > r$ then
    return $-1$    termination
else
    $m \leftarrow \lfloor (\ell + r)/2 \rfloor$
    if $k = A[m]$ then
        return $m$
    else if $k < A[m]$ then        recurrence
        return RecursiveBinarySearch ($A[\ell \dots m-1], k$)
    else
        return RecursiveBinarySearch ($A[m+1 \dots r], k$)
    end if
end if
```

Binary Search – Time complexity

C(n) = C([n/2]) + 1 for n > 1 and C(1) = 1

Apply Master Theorem: `T(n) = aT(n/b) + f(n)`

- `f(n) = O(`n^p`) where p < c`
  - Then, `T(n) = O(`n^c`)`
- `f(n) = O(`$n^c * \log^k n$`) k >= 0`
  - Then, `T(n) = O(`$n^c \log^{k+1} n$`)`
- `f(n) = O(`n^p`) where p > c` **AND** `a*f(n/b) <= k*f(n) for some k < 1`
  - Then, `T(n) = O(f(n))`

Case 2, k = 0 => C(n) = O(lg(N))

Binary Search – Time complexity

- Worst case of O(log(n)) is only achieved if:
  - Array is sorted
  - The array has O(1) access to any position

Fake Coin Problem

- Given a stack of *n* identical-looking coins which contains *exactly one* fake coin (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

Solving Fake Coin Problem

- The solution is to use a ***decrease-by-half*** algorithm:
  - Divide into two sub-stacks of n/2 coins and weigh on scale
  - The lighter sub-stack contains the fake coin
  - Repeat process with the lighter sub-stack until we have two coins remaining. The fake coin must be one of the two.

Solving Fake Coin Problem

- The recurrence relation for number of weighing-s:
- $C(n) = C([n/2]) + 1$ for $n > 1$, $C(1) = 0$
- Gives worst case of O(logn).

Fast Exponentiation

- You want to calculate (X ^ N), N can be very large
- The simple approach: execute multiplication N times
- The simple approach will not work for very large N
- However, because X ^ N = (X ^ (N / 2)) ^ 2
- Reduce the problem of size N to a similar problem of size N/2

Fast Exponentiation

- Cost: $C(N) = C(N/2) + 1$
- Complexity: $O(\lg(N))$
- Application: use in public key cryptography
  - We want to calculate: X ^ N % M
  - (A * B) % M = ((A % M) * (B % M)) % M
  - X^N % M = ((X^(N/2) % M) ^ 2) % M

5. Variable-size Decrease Algorithms

Interpolation Search

- Can be considered an “improved” version of **Binary Search**
  - Imagine that we can go even faster than O(logN)
- We can exploit additional knowledge of our values, i.e., use more than just comparisons

Interpolation Search

- **Interpolation Search:** estimate where the search key is based on the collection's distribution
    - **Simple:** use max and min values and assume equal distribution
    - **More advanced:** approximation of real distribution (e.g., histograms)
- Example: search for "Algorithm" in a 500-page dictionary. Which page do you check first?

Simple Interpolation Search – Idea

- Assume **equal distribution**–values within the ***sorted*** array A are equally distributed in [A[0], A[n - 1]]
- We can estimate the position pos of the search key with value k by using the formula (with l being the start value and r being the end value)

$$\mathrm{pos} = \mathrm{l} + (\mathrm{r} - \mathrm{l}) * \frac{\mathrm{k} - \mathrm{A[l]}}{\mathrm{A[r]} - \mathrm{A[l]}}$$

- If A[pos] == k, search key is found at pos
- If $\mathrm{k} > \mathrm{A[pos]}$, we update $\mathrm{l} = \mathrm{pos} + 1$; else $\mathrm{r} = \mathrm{pos} - 1$

Interpolation Search – Demo

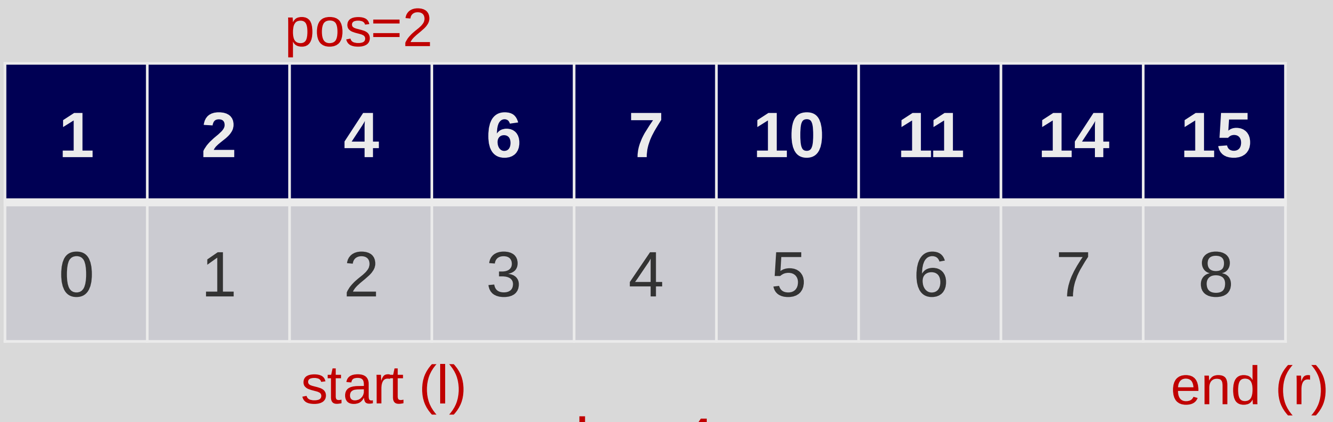
1	2	4	6	7	10	11	14	15
0	1	2	3	4	5	6	7	8

start (l) pos=1 end (r)

k = 4 > A[pos] (2)

start (l)	end (r)	pos
0	8	$0 + (8-0) * \frac{4 - A[0]}{A[8] - A[0]} = 8 * \frac{4-1}{15-1} = 8 * \frac{3}{14} = 1.71 \approx 1$

Interpolation Search – Demo



k = 4

start (l)	end (r)	pos
2	8	$2 + (8 - 2) * \frac{4 - A[2]}{A[8] - A[2]} = 2 + 6 * \frac{4 - 4}{15 - 4} = 2 + 0 = 2$

Interpolation Search – Analysis

- **Average case:** Interpolation Search on equally distributed data requires O(log(log(n)) comparisons
- **Worst case:** O(n), e.g., A is large and contains only values larger than the search key value

Quick Select

- Problem: find the k-th smallest element in an array
- **Quick Select** or **Quick Search**–adopts the idea of **Quick Sort** to search for a value
  - Choose a pivot
  - Partition the array
  - Instead of recursing for both sides, we just recur only for the part that contains the k-th smallest element
    - If k < pivot, recur for the left; else recur for the right

Quick Select – Analysis

- Since we recur only for **one subarray at a time** (instead of two as in Quick Sort), the time complexity reduces from O(N*logN) to O(N)
- Recurrence relation
  - QuickSort: $T(N) = 2*T(N/2) + N$
  - Quick Select: $T(N) = T(N/2) + N$
- **Worst case:** $O(N^2)$

Quick Select – Pseudocode

```
select(arr, k, left, right)
  p = partition(arr, left, right)   // Lomuto partition
  if (p + 1) == k
    return arr[p]
  if (p + 1) < k
    return select(arr, k, p + 1, right)
  return select(arr, k, left, p - 1)
```

Wrapping things up

Learning objectives

- Understand the ***Decrease & Conquer*** approach
- Understand and apply:
    - Decrease-by-**a-constant** algorithms (Insertion Sort and Topological Sort)
    - Decrease-by-**a-constant-factor** algorithms (Binary Search, Fake Coin, and Fast Exponentiation)
    - **Variable-size** decrease algorithms (Interpolation Search and Quick Select)