Divide and Conquer

Foundations of Algorithms
Guven

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Reading Assignment

- Cormen, Chapter 4

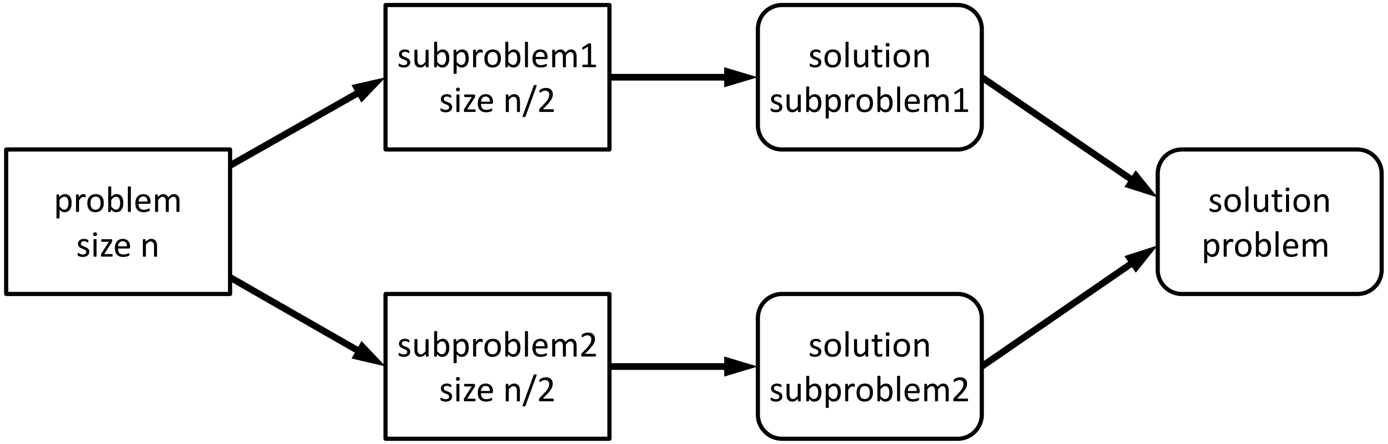
Outline

- Basic Idea
- Recurrence
- Mergesort
- Quicksort
- Binary Search
- Binary Tree Algorithms
- Strassen's Matrix Multiplication
- Closest Pair

Divide and Conquer

- DC Algorithm design paradigm:
  - Divide instance of the problem into two or more smaller instances (disjoint)
  - Solve smaller instances recursively
  - Obtain solution to original (larger) instance by combining these (smaller) solutions
- Multi-branched recursion
  - e.g. Fibonacci sequence: $F_n = F_{n-1} + F_{n-2}$
  - e.g. Fast Fourier Transform
  - Note, in practice recursion is avoided by direct iteration

Divide and Conquer Paradigm



- Divide and conquer ⟷ e.g. Hadoop MapReduce
  - Not overlapping solutions
  - Not dynamic programming where the previous results are used in the next iteration
  - Each subproblem has an independent solution

Mergesort

- Split array A in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A

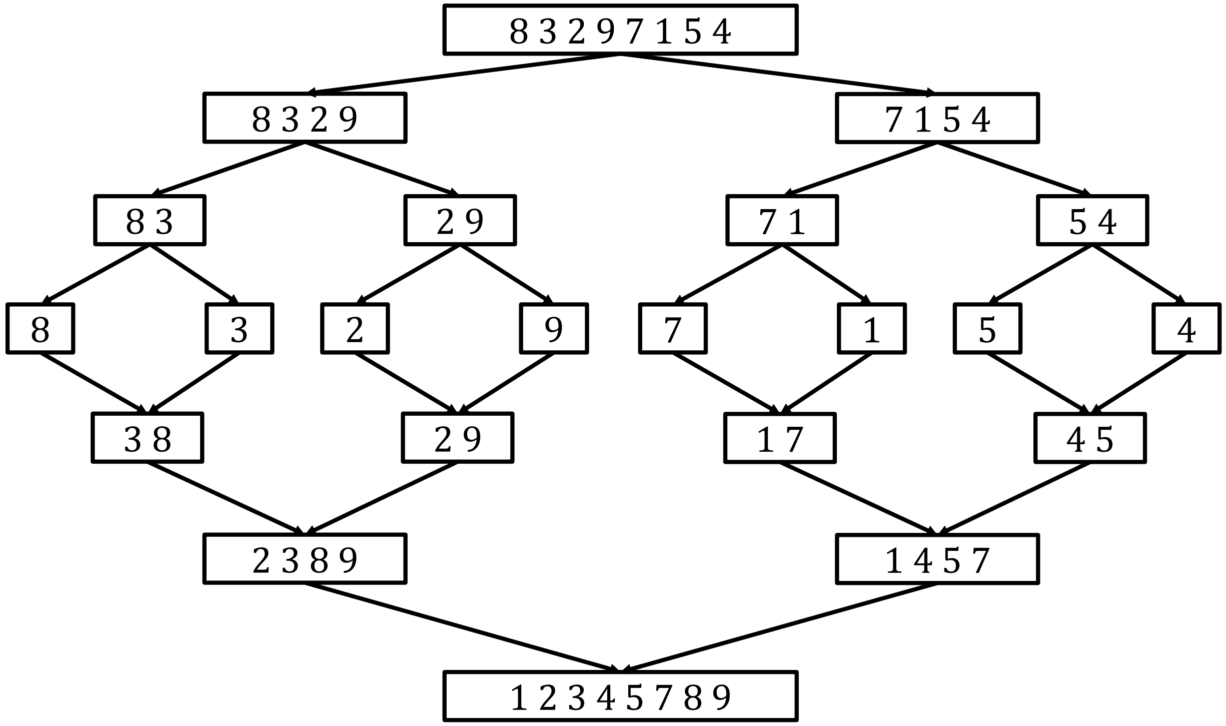
Mergesort

```
def mergesort(A):   # length n
    n=len(A)
    if n>1:
        B = A[:floor(n/2)]
        C = A[ceil(n/2):]
        mergesort(B)
        mergesort(C)
        merge(B,C,A)
```

Mergesort

```
def merge(B,C,A):  # B & C already sorted
  i=j=k=0; p=len(B); q=len(C)
  while i<p and j<q:
    if B[i]<=C[j]:
      A[k]=B[i]; i+=1
    else:
      A[k]=C[j]; j+=1
    k+=1
  if i==p:
    A[k:p+q]=C[j:q]
  else:
    A[k:p+q]=B[i:p]
```

Mergesort Example



Mergesort Complexity

- Best/worst have same efficiency

  $\Theta(n \log n)$

  $T(n) = 2T(n/2) + \Theta(n)$, $T(1)=0$
- Merge function complexity

  $\Theta(p+q)=\Theta(n)$ comparisons
- Space requirement

  not in place

  $\Theta(n)$
- Recursion is not necessary to implement

Recurrence

- An algorithm contains a recursive call
  - calls itself
  - running time by recurrence equation
- Example Mergesort
  - Each divide has subarray of size n/2
  - Two of those divides
  - Merge takes Θ(n)
  - $T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & if\ n > 1 \end{cases}$

Recurrence Master Theorem

- $T(n)=aT(n/b)+f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$
- Master Theorem
  - If $a < b^d$, then $T(n) \in \Theta\left(n^d\right)$
  - If $a = b^d$, then $T(n) \in \Theta\left(n^d \log n\right)$
  - If $a > b^d$, then $T(n) \in \Theta\left(n^{log_b a}\right)$

Recurrence Bound by Substitution

- Mergesort O(n)=n log n
- Use induction to show the O(n) bound holds for Mergesort T(n)
- $T(n)=2T(\lfloor n/2 \rfloor)+n$
- <u>Guess</u> that $T(n) \leq c\, n \log n$
- For m<n this bound holds, set $m=\lfloor n/2 \rfloor$
  - $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) \Rightarrow T(n) \leq c\, n \log(n)$

Quicksort

- Select a pivot - partitioning element
  - e.g. first element in the given array
- Rearrange the list
  - all the elements in the first s positions ≤ pivot
  - all the elements in the remaining n-s positions ≥ pivot
- Exchange the pivot with the last element in the first partition
  - i.e. partition[:-1] ⇆ pivot
- Quicksort the two partitions recursively

Quicksort

```
def quicksort(A):
  n = len(A)
  if n==1 or n==0:
    return A
  else:
    pivot=A[0]; i=0
    for j in range(n-1):  # theta(n)
      if A[j+1] < pivot:
        A[j+1],A[i+1] = A[i+1],A[j+1]  # swap
        i += 1
    A[0],A[i] = A[i],A[0]  # re-place pivot
    par1=quicksort(A[:i]); par2=quicksort(A[i+1:])
    return par1+[A[i]]+par2
```

Quicksort Complexity

- Best case, split in the middle

  $O(n \log n)$

- Worst case, sorted array (!)

  $O(n^2)$

  $T(n) = T(n-1) + \Theta(n)$

- Average case, random arrays

  $O(n \log n)$

Quicksort Improvements

- Better pivot selection
  - median-of-three partitioning
- Switch to insertion sort on small arrays
- Convert to iterative implementation
- Entropy optimal sorting
  - When duplicate sort keys exist frequently

Binary Search

- The most efficient algorithm for searching keys in sorted arrays
  - $O(n \log n)$ is already expended to sort the array
- Algorithm to find the index where A[m]==key
  - For an array A with length n
  - $m = \lfloor n/2 \rfloor$
  - If k == A[m], stop and return m
  - Else search k in A[:m-1] if k<A[m] or search k in A[m+1:]

Binary Search

```
from math import floor
def binarysearch(A,key):  # A is sorted
  n=len(A); l=0; r=n-1
  while l <= r:
    m = int(floor((l+r)/2))
    if key == A[m]:
      return m
    else:
      if key < A[m]:
        r = m-1
      else:  # key > A[m]
        l = m+1
  return -1  # was not found
```

Analysis of Binary Search

- Time complexity

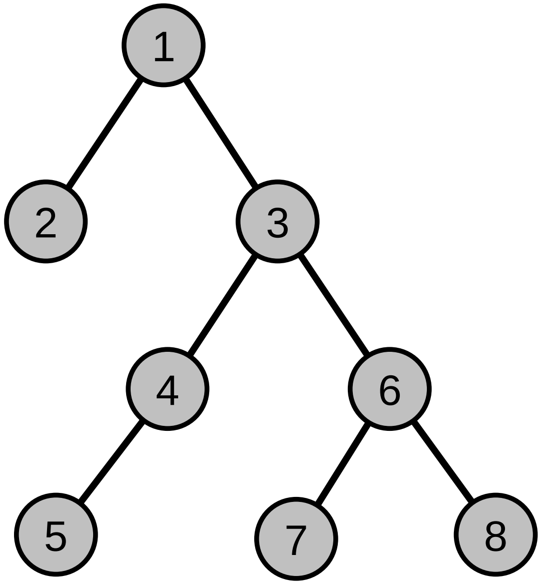
  $T(n) = T(n/2) + O(1)$
- Worst case: $O(\log n)$
  - Note the sort time
- Best case: 1
- Input has to be an array
  - as opposed to linked list, tree, etc.
- Perhaps not a true divide-and-conquer algorithm
  - since only one partition is solved at each level

Binary Tree Algorithms

- Binary Tree (BT) is already divided (in the sense DC)
- Unlike linear data structures, trees can be traversed in a few different ways
- Traversals
  - Preorder
  - Inorder
  - Postorder

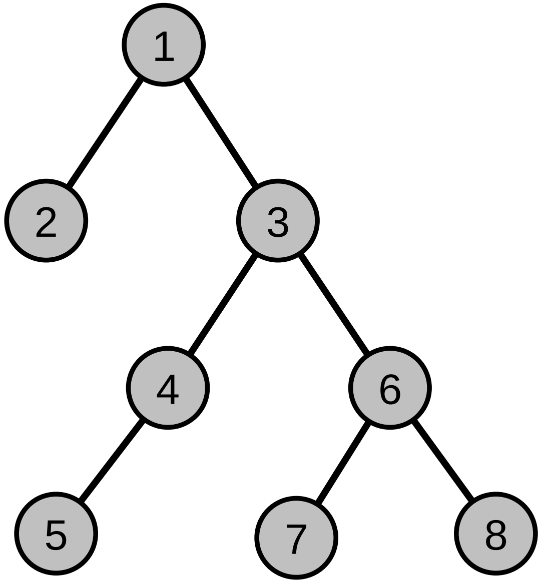
An Ordered Tree

- Ordered Tree is a directed tree with siblings ordered



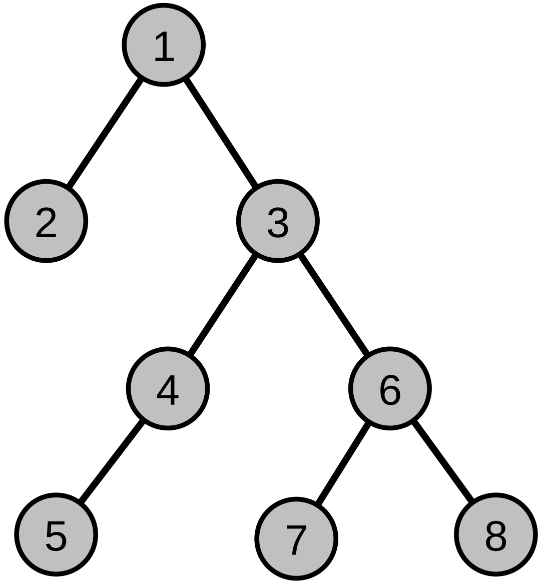
Preorder Binary Tree Traversal

- Preorder : 1, 2, 3, 4, 5, 6, 7, 8
  1. root (order vertices as pushed on a stack)
  2. preorder left subtree
  3. preorder right subtree



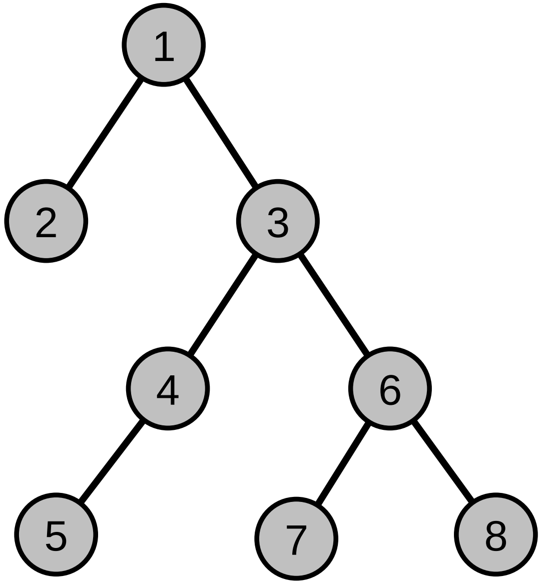
Postorder Binary Tree Traversal

- Postorder: 2, 5, 4, 7, 8, 6, 3, 1
  1. postorder left subtree
  2. postorder right subtree
  3. root (order vertices as popped off stack)



Inorder Binary Tree Traversal

- Inorder: 2, 1, 4, 5, 3, 7, 6, 8
  1. inorder left subtree
  2. root
  3. postorder right subtree



- Is there a mistake above?

Inorder Binary Tree Traversal

```
def inorder(T):  # T is a binary tree
  if len(T) > 0:
    inorder(T.left)
    print(T.root)
    inorder(T.right)
```

- Complexity: Θ(n)

Computing the Height of a BT

- h(T) = max(h(T.left), h(T.right)) + 1

```
def height(Tnode):  # Tree data structure
  if node is None:
    return 0
  else:
    return max(height(Tnode.left),
               height(Tnode.right)) + 1
```

Matrix Multiplication

- C = AB
  - $\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$
  - $= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$
- 8 multiplications, 4 additions
- Time complexity $\Theta(n^3)$

Strassen's Matrix Multiplication

- $C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$
- $m_1=(a_{11}+a_{22})(b_{11}+b_{22})$
- $m_2=(a_{12}+a_{22})b_{11}$
- $m_3=a_{11}(b_{12}-b_{22})$
- $m_4=a_{22}(b_{21}-b_{11})$
- $m_5=(a_{11}+a_{12})b_{11}$
- $m_6=(a_{21}-a_{11})(b_{11}+b_{12})$
- $m_7=(a_{12}-a_{22})(b_{21}+b_{22})$

- 7 multiplications, 18 additions

Strassen's Algorithm

- Strassen (1969) discovered that the product of two matrices can be computed as follows:

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

- $= \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$

Analysis of Strassen's Algorithm

- Pad 0 to make size of the matrix a power of 2
- Number of multiplications
  - M(n) = 7M(n/2), M(1) = 1
  - By master theorem, M(n) = $7^{log2^n} \approx n^{2.807}$
- Note, current GPUs has very fast, 1-clock-cycle multipliers
  - One of the reasons deep learning is realizable now

Closest Pair Problem

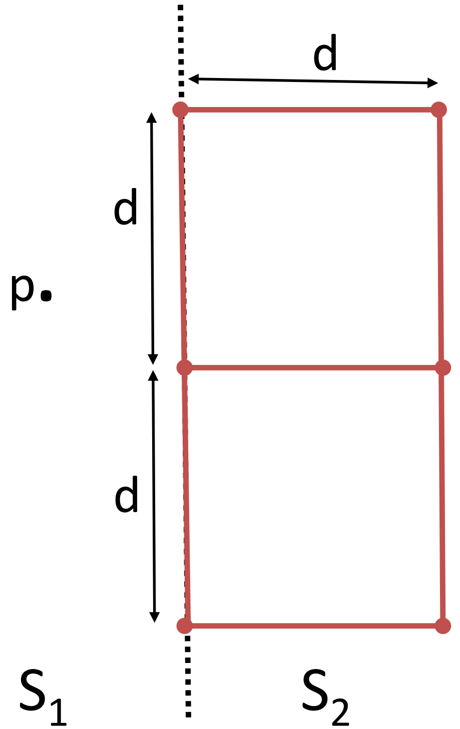
- Find the closest pair
  - Naive approach $O(n^2)$
- Algorithm
  - In 2 dimensions, each point is the tuple (x,y)
  - Fundamental algorithm to other more complex algorithms
  - e.g. Hierarchical clustering
  - d-dimensional algorithm
  - O(n log n) divide-and-conquer algorithms exists

Closest Pair Algorithm

- Sort points by x and by y
- Divide the points into two (equal size) subsets S_1 and S_2
- Find recursively the closest pairs for S_1 and S_2 (i.e. d_1, d_2)
- Set $d=\min(d_1,d_2)$
  Focus on the strip of width 2d
  Let C_1 and C_2 be subsets of points in the strip, in S_1 and S_2
- For every point p in C_1, inspect points in C_2 that may be closer to p than d

Closest Pair Algorithm

- Geometrically, at most 6 points possibly can be in the rectangle d-by-2d
- p will be checked at most 6 points in the rectangle d-by-2d
- Thus, for S_1, at most 6n/2 number of checks needed to find a d' (if exists) which is less than d
- Also for S_2, at most 6n/2 number of checks needed to find a d' which is less than d



Closest Pair Algorithm

- Complexity
  - $T(n) = 2T(n/2) + M(n) + \text{sorting}$
  - $M(n) \in \Theta(n)$
  - $T(n) \in \Theta(n \log n) + \text{sorting}$

  - Complexity = $O(n \log n)$