



# Divide and Conquer

*Algorithm design techniques*



## The plan

---

1. *Problem instance is divided into several smaller instances of the same problem – ideally about same size.*
2. *The smaller instances are solved (maybe recursively, maybe using different algorithm).*
3. *If necessary the solutions for smaller instances are combined to get solution for whole problem.*



## example

- ◆ Compute sum of  $n$  numbers

$A_0, \dots, A_{n-1}$ .

If  $n > 1$ , we can divide problem to two instances.

Compute sum of first  $n/2$  numbers and to compute sum of second  $n/2$  numbers.

For  $n=1$ , the answer is  $A_0$

Once both sums are computed, we can add their values

$$A_0 + \dots + A_{n-1} = (A_0 + \dots + A_{(n/2)-1}) + (A_{n/2} + \dots + A_{n-1})$$

- ◆ Is this effective way how to compute sum of  $n$  numbers??
- ◆ No!



- ◆ **Not every *divide and conquer* algorithm is more effective than *brute force* solution!**
- ◆ *To use divide and conquer, we **divide** problem to smaller parts, and then, sometimes, **combine** solutions of sub problems.*
  - ◆ *This is extra work!*
  - ◆ *It must offer better final result than other methods.*
  - ◆ *(By it's the nature, this technique is most suitable for parallel algorithms, where each instance is solved on different processor).*



# Mergesort

- It sorts given array  $A[0..n-1]$  by dividing it to two parts  $A[0..(n/2)-1]$  and  $A[n/2 .. n-1]$ , sorting each of them recursively and merging two smaller arrays into single sorted one.

**Mergesort( $A[0..n-1]$ )**

**if  $n > 1$  then**

**copy  $A[0..(n/2)-1]$  to  $B[0..(n/2)-1]$ ;**

**copy  $A[n/2..n-1]$  to  $C[0..(n/2)-1]$ ;**

**Mergesort( $B[0..(n/2)-1]$ );**

**Mergesort( $C[0..(n/2)-1]$ );**

**Merge( $B, C, A$ );**



- ◆ **Merging** of two sorted arrays is a trick applied here
  - ◆ *Two pointers are initialized to point to first elements of the arrays to merge*
  - ◆ *Elements pointed to are compared, smaller of them is added to result array. Pointer to that smaller element is incremented to next element.*
  - ◆ *This continues until one of arrays is empty (pointer point to end of it), then the remaining elements of other array are copied.*



8 3 2 9 7 1 5 4

8 3 2 9 | 7 1 5 4

8 3 | 2 9 | 7 1 | 5 4

8 | 3 | 2 | 9 | 7 | 1 | 5 | 4 -- starting to merge

3 8 | 2 9 | 1 7 | 4 5

2 3 8 9 | 1 4 5 7

1 2 3 4 5 7 8 9 -- done





**Merge(B[0,...,p-1],C[0,...,q-1],A[0,...,p+q-1])**

*// Merges two sorted arrays*

*i = 0; j = 0; k = 0;*

**while** *i < p and j < q* **do**

**if** *B[i] <= C[j]* **then**

*A[k] = B[i]; i++;*

**else** *A[k] = C[j]; j++;*

*k++;*

**od**

**if** *i == p*

*copy C[j..q-1] to A[k .. p+q-1]*

**else** *copy B[i..p-1] to A[k .. p+q-1]*



- ◆ *Efficiency?*

- ◆ *Assume that  $n$  is a power of 2*

$$C(n) = 2 * C(n/2) + C_{merge}(n) \text{ for } n > 1, C(1) = 0$$

$C_{merge}$  – one comparison, after that the size of inputs to merge decrease by one. In worst case we need  $n-1$  comparisons.

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \text{ for } n > 1, C_{worst}(1) = 0$$

*As a result*

$$C_{worst}(n) = n \log n - n + 1$$

- ◆ *Problem with mergesort is extra space needed for storing array (can be solved by **inplace merge**).*



# Quicksort

- ◆ *Mergesort divided input by the position in array (middle)*
- ◆ *Quicksort divide input by its **value!***
  - ◆ *It partitions array **A[0..n-1]***
  - ◆ *All elements before particular position **s** are smaller or equal **A[s]**, and all elements after that position are greater or equal **A[s]***
  - ◆ *After such partition is achieved, **A[s]** is in its final position*
  - ◆ *We recursively sort two subarrays*



- ◆  $A[0] \dots A[s-1]$     **$A[s]$**     $A[s+1] \dots A[n-1]$   
     $\leq A[s]$                        $\geq A[s]$





▶ **Quicksort(A[l,..,r])**

**if**  $l < r$

$s = \text{Partition}(\mathbf{A}[l..r]);$  // *s is a split position*

**Quicksort(A[l..s-1]);**

**Quicksort(A[s+1..r]);**





## Partitioning( $A[l..r]$ )

- ♦ *Selected element with which the rest of arrays is compared – **pivot***
  - ♦ *(there are several methods how to select one)*
  - ♦ *For now,  $p = A[l]$*



▶ *Partition*( $A[l..r]$ )

$p = A[l];$

$i=l; j = r+1;$

**repeat**

**repeat**  $i++$  **until**  $A[i] \geq p;$

**repeat**  $j--$  **until**  $A[j] \leq p;$

$swap(A[i], A[j]);$

**until**  $i \geq j$

$swap(A[i], A[j]);$

$swap(A[l], A[j]);$

**return**  $j;$





- ◆  $C_{best}(n) = n \log n$  – each iteration splits array to two parts
  - ◆  $C_{best}(n) = 2 * C_{best}(n/2) + n$ , for  $n > 1$ ;  $C(1) = 0$ ;
- ◆  $C_{worst}(n)$  – after each partitioning, one of the arrays is empty (input already sorted)
  - ◆  $C_{worst}(n) = \Theta(n^2)$
- ◆ Average case  $C_{avg}(n) = 1.38 * n \log n$



## Pivot selection

- ◆ *Selection of pivot influences the performance of algorithm!*
- ◆ *See possible worst case if we start with  $p = A[l]$*
- ◆ *Possibilities:*
  - ◆ *Pick the first element in array*
  - ◆ *Pick the last element in array*
  - ◆ *Pick the middle element in array*
  - ◆ *Pick random one*
  - ◆ **Median-of-three** – *we take middle one, first one and last one and pick one with middle value*



# Binary search

- ◆ *It searches in **sorted** array*
  - ◆ *Compares key **K** with middle element **A[m]***
  - ◆ *If they match, algorithm stops*
  - ◆ *Otherwise the operation is performed recursively on first half of array – if **K** < **A[m]**, or second half otherwise*
- ◆ *NOTE! Sequential search do not need sorted array to work!*
  - ◆ *If we include sorting of array to whole execution time of binary search it is MUCH slower than sequential*



## Nonrecursive BinarySearch

```
BinarySearch(A[0..n-1],K)
l = 0; r = n-1;
while l <= r do
    m = (l+r)/2;
    if K == A[m] then return m;
    else if K < A[m] then r = m-1;
    else l = m+1;
od
return -1;
```





$$K = 6$$

1 3 4 6 11 **18** 23 25 39 42 47

$$6 < 18$$

1 3 **4** 6 11

$$6 > 4$$

**6** 11

$$6 == 6$$

- ◆  $C_{\text{best}}(n) = 1$
- ◆  $C_{\text{worst}}(n) = \log n$
- ◆  $C_{\text{avg}}(n) = \log n$



## Binary tree traversal

- Binary tree  $T$  is defined as a finite set of nodes that is empty or consists of root and two disjoint binary trees  $T_L$  and  $T_R$  – the left and right subtree of root
  - The Binary tree is special case of ordered tree (NOTE: this is not binary search tree yet)
- Definition divides binary tree to left and right subtree – it is simple to apply divide and conquer techniques (the division is already part of data structure).



## Height of a tree

- ◆ *Height of a tree is defined as longest path from root to a leaf*
  - ◆ *It can be computed as maximum of root's left and right subtrees heights plus 1*
  - ◆ *Height of empty tree is -1*

*Height(T)*

**if  $T=0$  then return -1**

**else return  $\max\{\text{Height}(T_L), \text{Height}(T_R)\} + 1$**

- ◆  $A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$ , for  $n(T) > 0$ ,  $A(0)=0$



## Three traversals

---

- ▶ **Preorder traversal**, *the root is visited before the left and right subtrees are visited*
- ▶ **Inorder traversal**, *the root is visited after visiting its left subtree but before visiting the right subtree*
- ▶ **Postorder traversal**, *the root is visited after visiting left and right subtree*



## Closest Pair

- ◆  $P_1=(x_1,y_1), \dots, P_n=(x_n,y_n)$ , ( $n$  is order of two)
- ◆ The points are sorted by their  $x$  coordinate
- ◆ We divide points to subsets  $S_1$  and  $S_2$  ( $n/2$  points in each) by drawing a line  $x=c$ .
- ◆ We find, recursively the closest pairs in subset  $S_1$  and  $S_2$ .  $d = \min(d_1, d_2)$ .
- ◆ Closest pairs can lie on opposite sides of separating line –  $d$  may not be the result yet. We need to test this.



- ◆ *We are interesting about a points that are within  $d$  distance from separating line!*
  - ◆ *Distane between any other points across separating line would be more then  $d$*
- ◆ *Lets call  $C_1$  and  $C_2$  the subsets of  $S_1$  and  $S_2$  that are closser to separating line then  $d$*
- ◆ *For every point  $P(x,y)$  in  $C_1$  we need to inspect points in  $C_2$  that may be closer then  $d$* 
  - ◆ *Such points must have their coordinates within interval  $[y-d,y+d]$*



- ◆ *There can be at MOST 6 such points in  $C_2$* 
  - ◆ *Any point in  $C_2$  is at least distance  $d$  apart from each other*
  - ◆ *We can also maintain points in  $C_1$  and  $C_2$  sorted by  $y$  coordinate*
  - ◆ *Time of this merging  $M(n)$  is  $O(n)$*
- ◆  *$T(n) = 2T(n/2) + M(n) \rightarrow O(n \log n)$*



## The theory

- ◆ *Instance of a problem of size  $n$  can be divided into several instances of size  $n/b$ , with  $a$  of them needing to be solved.*
  - ◆ *Assume the  $n$  is power of  $b$  to simplify computation*
  - ◆  *$T(n) = aT(n/b) + f(n)$  – **general divide and conquer recurrence***
  - ◆  *$f(n)$  – *time spent on dividing problem and combining solution**



$$T(n) = aT(n/b) + f(n)$$

- $T(n) = aT(n/b) + f(n)$

*Master Theorem: If  $f(n)$  is  $\Theta(n^d)$ , where  $d \geq 0$  then*

*$T(n)$  is  $\Theta(n^d)$ , if  $a < b^d$*

*$\Theta(n^d \log n)$ , if  $a = b^d$*

*$\Theta(n^{\log_b a})$ , if  $a > b^d$*

- *Same holds for  $O$  and  $\Omega$*





## For shortest pair

- ◆ *If we apply the Master theorem to our shortest pair solution:  $T(n) = aT(n/b) + f(n)$  – Master Theorem*
- ◆  $T(n) = 2T(n/2) + M(n)$ 
  - ◆  $a = 2, b = 2, d = 1$
  - ◆  $2 == 2^1$
  - ◆  $O(n^1 \log n)$