

# Lecture 2: Recursive Algorithms

## 1 Reductions and Sub-Routines

- Solving a problem by **reducing** it (or a sub-problem of it) to another problem is the most fundamental technique in algorithm design.
- Specifically, algorithm  $A$  may use another algorithm  $B$  as a sub-routine.
- This has numerous advantages:
  - **Code Verification**: the correctness of  $A$  is independent of  $B$ .
  - **Code Reuse**: a great time-saver.
- A simple but very powerful special case of this paradigm is when the algorithm calls itself!
  - This method is called **recursion**.

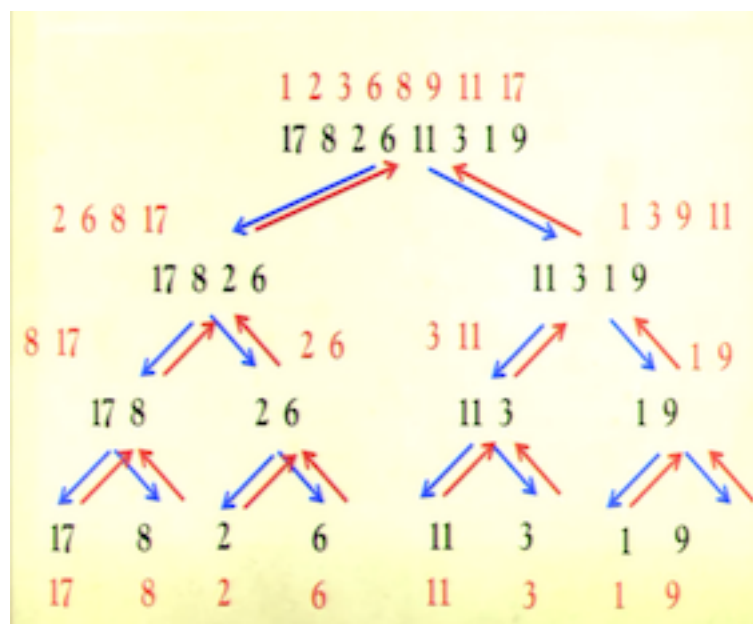
## 2 MergeSort

- We can sort  $n$  numbers into non-decreasing order using the following algorithm:

MergeSort( $x_1, x_2, \dots, x_n$ )

If  $n = 1$  then output  $x_1$

Else output Merge{MergeSort( $x_1, \dots, x_{\frac{n}{2}}$ ), MergeSort( $x_{\frac{n}{2}+1}, \dots, x_n$ )}



- Two Problems:

- Does the algorithm work?

Yes!

→ The algorithm calls itself on smaller instances

- The division process terminates with a set of base cases of size 1.

→ MergeSort trivially works on the base cases.

→ So, given the validity of the Merge Step, the correctness of the algorithm follows by **strong induction**.

- As long as base case is correct and merge step works, everything will be fine.

- If so, is it efficient (polynomial time)?

Yes! Look at the recursive formula.

→ To analyze this we represent the running time  $T(n)$  via a **recurrence**:

Recursive Formula:  $T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n$

- $2 \cdot T(\frac{n}{2})$ : Recurse on two problems with half the size.
- $c \cdot n$ : It takes linear time to merge two sorted lists.

Base Case:  $T(1) = 1$

- Or we can use  $T(c) = O(1)$  for any constant  $c$ .

→ The Running Time of MergeSort

- Theorem: MergeSort runs in time  $O(n \cdot \log n)$
- Proof:

1. By adding dummy numbers, we may assume  $n$  is a power of two:  $n = 2^k$
2. We can unwind the recursive formula as follows:

$$T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n$$

Proof [cont.]

- But this unwinding operation can be repeated:

$$\begin{aligned}T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n \\&= 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}\right) + cn \\&= 2^2 \cdot T\left(\frac{n}{4}\right) + 2 \cdot cn \\&= 2^2 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + c \cdot \frac{n}{4}\right) + 2 \cdot cn \\&= 2^3 \cdot T\left(\frac{n}{8}\right) + 3 \cdot cn \\&= 2^3 \cdot \left(2 \cdot T\left(\frac{n}{16}\right) + c \cdot \frac{n}{8}\right) + 3 \cdot cn \\&= 2^4 \cdot T\left(\frac{n}{16}\right) + 4 \cdot cn \\&\vdots \\&= 2^k \cdot T(1) + k \cdot cn \quad \text{— Since } n = 2^k\end{aligned}$$

Proof [cont.]

- Thus we have:

$$\begin{aligned}T(n) &= 2^k \cdot T(1) + k \cdot cn \\&= n \cdot T(1) + k \cdot cn \\&= n \cdot (1 + k \cdot c)\end{aligned}$$

- But  $c$  is a constant so:

$$T(n) = O(n \cdot k)$$

- Furthermore,  $k = \log n$ , so we get that:

$$T(n) = O(n \cdot \log n)$$



### 3 Binary Search

- We can search for a key  $k$  in a sorted array of cardinality  $n$  using the binary search algorithm:

BinarySearch( $a_1, a_2, \dots, a_n : k$ )

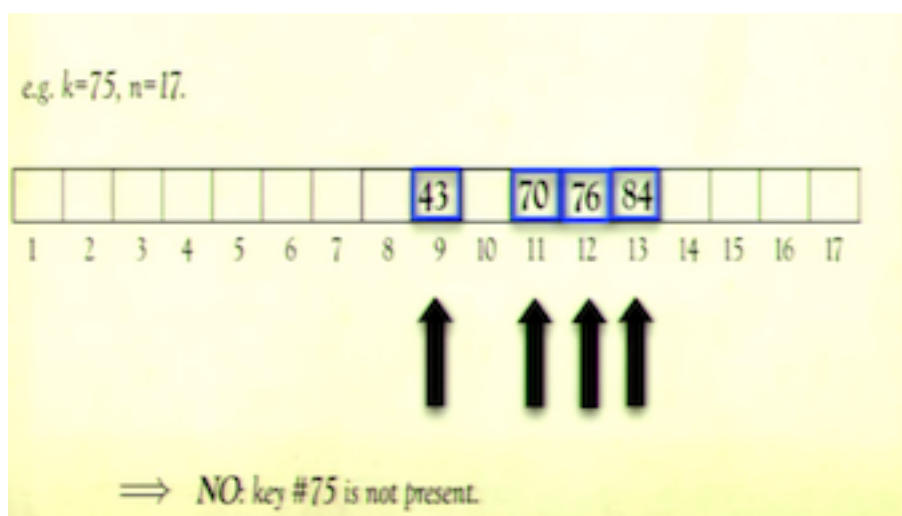
While  $n > 0$  do:

If  $a_{\frac{n}{2}} = k$  output YES

Else if  $a_{\frac{n}{2}} > k$  output BinarySearch( $a_1, a_2, \dots, a_{\frac{n}{2}-1} : k$ )

Else if  $a_{\frac{n}{2}} < k$  output BinarySearch( $a_{\frac{n}{2}+1}, \dots, a_n : k$ )

Output NO



- Does this work?
  - The validity of the binary search follows simply by strong induction. (The base case is trivially true.)
- Running Time?
  - Recurrence:
$$\text{Recursive Formula: } T(n) = T\left(\frac{n}{2}\right) + c$$
$$\text{Base Case: } T(1) = 1$$
  - Theorem: Binary Search runs in time  $O(\log n)$ 
    1. By adding dummy numbers, we may assume  $n$  is a power of two:
$$n = 2^k$$
    2. We can unwind the recursive formula as follows:

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + c \\
 &= \left(T\left(\frac{n}{4}\right) + c\right) + c \\
 &= T\left(\frac{n}{4}\right) + 2 \cdot c
 \end{aligned}$$

- Again this unwinding operation can be repeated:

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{4}\right) + 2 \cdot c \\
 &= \left(T\left(\frac{n}{8}\right) + c\right) + 2 \cdot c \\
 &= T\left(\frac{n}{8}\right) + 3 \cdot c \\
 &\vdots \\
 &= T\left(\frac{n}{2^k}\right) + k \cdot c
 \end{aligned}$$

- Hence:

$$\begin{aligned}
 T(n) &= T(1) + k \cdot c \quad \text{--- Since } n = 2^k \\
 &= 1 + \log n \cdot c
 \end{aligned}$$

- This gives the claimed running time:

$$T(n) = O(\log n)$$

## 4 Divide and Conquer Algorithms

- A **divide and conquer** algorithm recursively breaks up a problem of size  $n$  in smaller sub-problems such that:
  - There are exactly  $a$  sub-problems.
  - Each sub-problem has size at most  $\frac{1}{b} \cdot n$
  - Once solved, the solutions to the sub-problems can be combined to produce a solution to the original problem in time  $O(n^d)$
- So the run-time of a divide and conquer algorithm satisfies the recurrence:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

- MergeSort and Binary Search are indeed **divide and conquer** algorithms.

	Recursion Formula	$a$	$b$	$d$
MergeSort	$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n^1)$	2	2	1
Binary Search	$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + O(n^0)$	1	2	0

## 5 Non-Military Applications of Divide and Conquer

- Divide and Conquer has many other non-military, practical applications:
  - Big Data
  - Distributed Algorithms
  - Clustering and Classification
  - MapReduce

## 6 Dummy Entries

- MergeSort actually has the recurrence:

$$\hat{T}(n) = \hat{T}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \hat{T}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c \cdot n$$

- Recall we got around this by adding dummy entries:
  - We found  $\hat{n}$  the smallest power of 2 greater than  $n$ .

- For this case, MergeSort then does have recurrence:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

- But we also have:

$$\hat{T}(n) \leq T(\bar{n}) = O(\bar{n} \cdot \log \bar{n}) = O(n \cdot \log n)$$

- Here is another way to solve the recurrence:

$$\hat{T}(n) = \hat{T}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \hat{T}\left(\left\lceil \frac{n}{2} \right\rceil\right) + c \cdot n$$

- As we only want to upper bound the running time, we can use:

$$\hat{T}(n) \leq \hat{T}\left(\frac{n}{2} + 1\right) + c \cdot n$$

- This +1 does not seem to fit with our methodology, but we can fix this by applying a **domain transformation**.

- Domain Transformation

- For the domain transformation, simply set:  $T(n) = \hat{T}(n + 2)$
- Thus we have:  $T(n) = T\left(\frac{n}{2}\right) + \hat{c} \cdot n$

$$\begin{aligned} T(n) &= \hat{T}(n+2) \\ &\leq \hat{T}\left(\frac{n+2}{2} + 1\right) + c \cdot (n+2) \\ &\leq \hat{T}\left(\frac{n+2}{2} + 1\right) + \hat{c} \cdot n \\ &= \hat{T}\left(\frac{n}{2} + 2\right) + \hat{c} \cdot n \\ &= T\left(\frac{n}{2}\right) + \hat{c} \cdot n \end{aligned}$$

- Of course, we can solve this recurrence as:  $T(n) = O(n \cdot \log n)$
- Therefore,  $\hat{T}(n) = T(n - 2) = O(n \cdot \log n)$
- As well as ceilings and floors, domain transformations can be used to simplify many other recurrences; e.g. removing lower order terms.