ICCS312 Algorithms and Tractability (Term 1/2021-22)

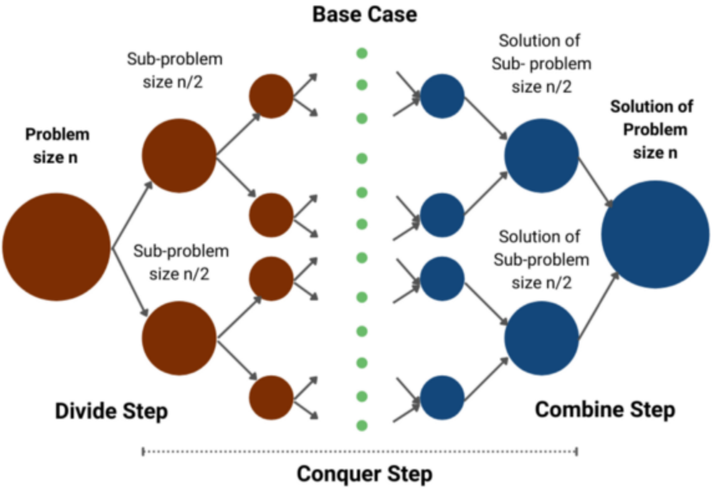
Lecture 3: Divide and Conquer (I)

Sunsern Cheamanunkul

Mahidol University International College

September 21, 2021

Divide-and-Conquer



Divide-and-Conquer

The divide-and-conquer strategy solves a problem by:

1. Divide instance of problem into ~~two or more~~ smaller instances
2. Recursively solving these instances
3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer

The real work of a divide conquer algorithm happens at three different places:

1. when we partition the problem into smaller sub-problems.
2. when the problems are small enough that we can solve them easily.
3. when we combine solutions to sub-problems back to a solution for the original problem.

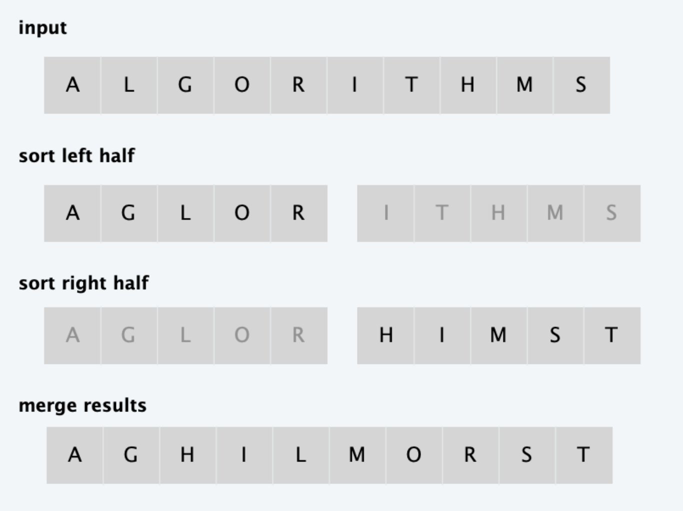
Mergesort

Problem: Given a list L of n elements from a totally ordered universe, rearrange them in ascending order.

Idea:

- Split the list into left and right halves.
- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.

Mergesort


input
A L G O R I T H M S
sort left half
A G L O R
I T H M S
sort right half
A G L O R
H I M S T
merge results
A G H I L M O R S T

Mergesort

T(n) be the running time of MergeSort(L)

Algorithm: MergeSort(L)

```
If len(L) == 1: return L
```
— base case

```
Divide L into two halves A and B.
A = MergeSort(A)
B = MergeSort(B)
L = Merge(A,B)
return L
```

$\leftarrow T(\lceil n/2 \rceil)$

$\leftarrow T(\lceil n/2 \rceil)$

$\leftarrow O(n)$

- What is running time?

Analyzing Mergesort

Let $T(n)$ denote the maximum number of compares to mergesort a list of length n

We will have the following *recurrence relation*.

$$T(n) \leqslant \begin{cases} 0 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } n > 1 \end{cases}$$

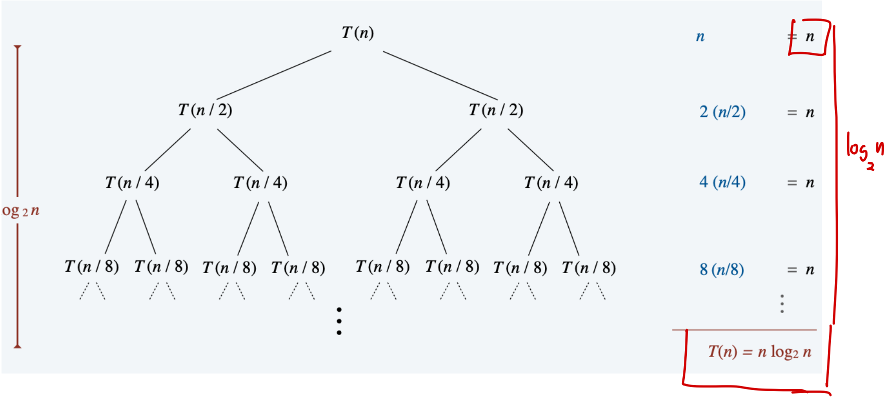
Recursion tree

Proposition

If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$ (assuming n is a power of 2).

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

Recursion tree



Recursion tree

Proposition

If $T(n)$ satisfies the following recurrence, then $T(n) = n\log_2 n$ (assuming n is a power of 2).

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

Proof

- Base case: when $n = 1$, $T(1) = 0 = n\log_2 n$ ✓
- Inductive hypothesis: assume $T(n) = n\log_2 n$
- Goal: show that $T(2n) = 2n\log_2(2n)$

Recursion tree

Proof. (Cont.)

$$\begin{aligned} T(2n) &= 2T(n) + 2n && \text{by our def.} \\ &= 2n\log_2 n + 2n && \text{by our IH} \\ &= 2n(\log_2(2n) - 1) + 2n \\ &= 2n\log_2(2n) \end{aligned}$$

■

$$\log_2 n + 1 - 1 = \log_2 n + \log_2 2 - 1$$
$$= \log_2 2n - 1$$

Back to Mergesort

Proposition

If $T(n)$ satisfies the following recurrence, then $T(n) \leqslant n\lceil \log_2 n \rceil$.

$$T(n) \leqslant \begin{cases} 0 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } n > 1 \end{cases}$$

Proof. [by strong induction]

- Base case: when $n = 1$, $T(1) = 0 = n \log_2 n$
- Let $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$
- Note that $n_1 + n_2 = n$
- Induction step: assume true for $\cancel{n =} 1, 2, \ldots, n-1$

Back to Mergesort

Proof. (Cont.)

$$
\begin{aligned}
T(n) &\leqslant T(n_1) + T(n_2) + n \\
&\leqslant n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n && \text{IH} \\
&\leqslant n_1 \lceil \log_2 n_2 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\
&= n \lceil \log_2 n_2 \rceil + n \\
&\leqslant n(\lceil \log_2 n \rceil - 1) + n \\
&= n \lceil \log_2 n \rceil
\end{aligned}
$$

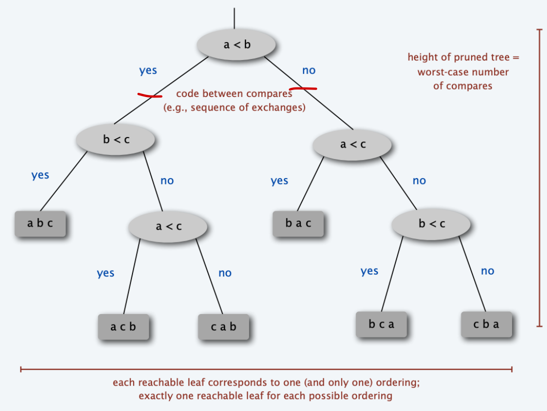
$$\boxed{\lceil \log_2 n_2 \rceil \le \lceil \log_2 n \rceil - 1}$$

■

Detour: Sorting lower bound

Question: Are there a faster algorithm than Mergesort?

Comparison tree



Sorting lower bound

Theorem

Any deterministic compare-based sorting algorithm must make $\Omega(n \log n)$ compares in the worst-case.

Proof.

Assume the input array contain distinct elements $a_1, a_2, \ldots, a_n$. Worst-case number of compares equals to the height h of the pruned comparison tree. Binary tree of height h has $\leqslant 2^h$ leaves. Potentially, $n!$ different ordering $\Rightarrow n!$ reachable leaves.

Sorting lower bound

Theorem

Any deterministic compare-based sorting algorithm must make $\Omega(n \log n)$ compares in the worst-case.

Proof. (Cont.)

$$2^h \geqslant \text{number of leaves} \geqslant n!$$

Thus,

$$h \geqslant \log_2 n! \geqslant \log_2 e\left(\frac{n}{e}\right)^n = \log_2 e + \log_2\left(\frac{n}{e}\right)^n = n \cdot \log_2 \frac{n}{e}$$

$\log_2 e +$

$$= \geqslant n \log_2 n - n/\ln 2 \qquad \text{By Stirling's formula} = \ldots$$

$$\geqslant \overline{\overline{n \log_2 n}}$$

∎

Stirling's formula: $n! \geqslant e\left(\frac{n}{e}\right)^n$

Counting inversions

- Given a list of integers L of size n.
- We say that two elements $L[i]$ and $L[j]$ form an inversion if $L[i] > L[j]$ and $i < j$.
- So if array is already sorted then inversion count is 0 and, on the other hand, if array is sorted in reverse order then the inversion count is at its maximum.
- Our goal here is to design an algorithm to count the number of inversions given an array.

Counting inversions

- Naive solution: Consider all possible pairs and just count.
- What is the running time for this? $\Theta(n^2)$
- Can we do better?

Counting inversions

- Naive solution: Consider all possible pairs and just count.
- What is the running time for this?
- Can we do better?
- Yes, let's do it divide-and-conquer style!

Counting inversions

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversion in each list.
- Combine: count inversions (a, b) with $a \in A$ and $b \in B$
- Return sum of three counts

Counting inversions

input

1	5	4	8	10	2	6	9	3	7

count inversions in left half A

1	5	4	8	10

5-4

count inversions in right half B

2	6	9	3	7

6-3 9-3 9-7

count inversions (a, b) with a $\in$ A and b $\in$ B

1	5	4	8	10

2	6	9	3	7

4-2 4-3 5-2 5-3 8-2 8-3 8-6 8-7 10-2 10-3 10-6 10-7 10-9

output 1 + 3 + 13 = 17

Counting inversions

- But, how to count inversions (a, b) with $a \in A$ and $b \in B$?
- Easy if both A and B are sorted.

Idea: Binary Search

- Assume A and B are already sorted
- For each element $b \in B$
  - binary search in A to find how elements in A are greater than b.

1 3 5 7

A

2 4 6 8

B

$\frac{n}{2}$

$3+2+1=6$

$\Theta(n \log n)$

Counting inversions

list A

7	10	18	3	14

list B

20	23	2	11	16

sort A

3	7	10	14	18

sort B

2	11	16	20	23

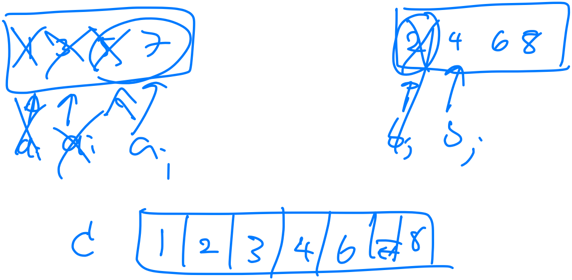
binary search to count inversions (a, b) with a ∈ A and b ∈ B

3	7	10	14	18

2	11	16	20	23
5	2	1	0	0

Counting inversions

- We can do better by performing *Merge-and-Count*
- Assume A and B are sorted.
- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element in B.
- If $a_i > b_j$, then b_j is inverted with any element left in A.
- Append smaller element to sorted list C.



Inv. Count : 0+3+0
+2+1
= 6

Counting inversions

$T(n)$

Algorithm: Sort-And-Count(L)

```
If len(L) == 1: return (0,L)

Divide L into A and B.
(rA, A) = Sort-And-Count(A)
(rA, B) = Sort-And-Count(B)
(rAB, L) = Merge-And-Count(A,B)
return (rA+rB+rAB, L)
```

$O(1)$ — $T(n/2)$ — $T(n/2)$ — $O(n)$

- What is running time?

$$T(n) = 2T(n/2) + O(n)$$

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

Discussion: Nuts & Bolts

Problem: A disorganized carpenter has a mixed pile of n nuts and n bolts.

- The goal is to find the corresponding pairs of nuts and bolts.
- Each nut fits exactly one bolt and each bolt fits exactly one nut.
- By fitting a nut and a bolt together, the carpenter can see which one is bigger (but cannot directly compare either two nuts or two bolts).

Example:

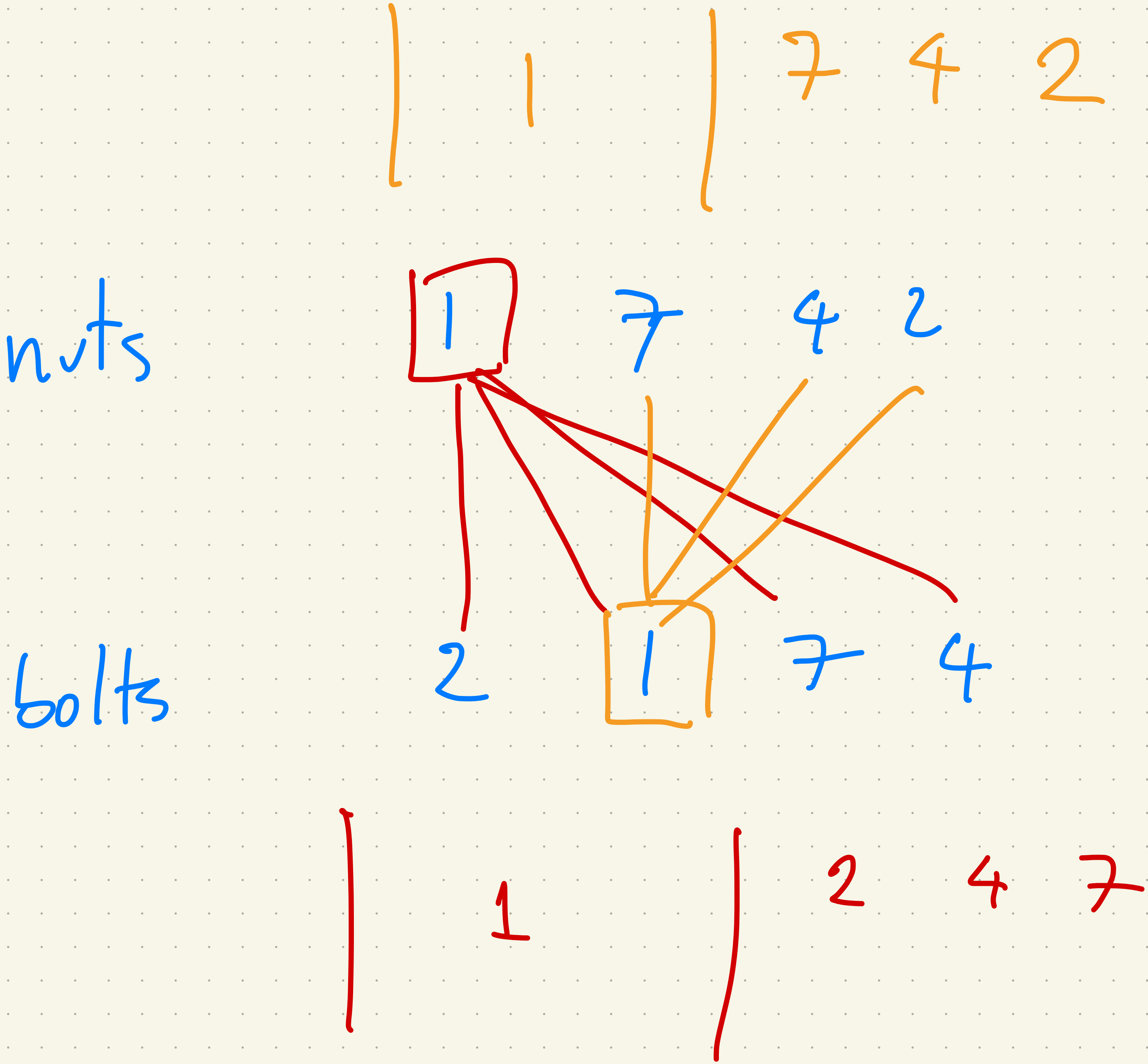
```
nuts = [1,7,4,2]
bolts = [2,1,7,4]
Allowed Operations: Compare(nuts[i], bolts[j])
```

- Challenge: Design an algorithm that makes $O(n \log n)$ compares.

nuts 1 7 $\boxed{4}$ 2

bolts 2 1 7 4

bolt 12 4 7



1
7 4 2
nuts
1 7 4 2
bolts
2 1 7 4
1
2 4 7

```
nut&bolt(nuts, bolts): {
    p = nuts[0]
    bolts_L, bolts_H, b_p = partition(bolts, p)
    nuts_L, nuts_H, _ = partition(nuts, b_p)
    nut&bolts(nut_L, bolts_L)
    nut&bolts(nuts_H, bolts_H)
    return ___
}
```

$\Theta(n)$ (partition steps)

$T(m)$

$T(n-m)$

$$T(n) \le T(m) + T(n-m) + \Theta(n)$$