Divide-Conquer-Glue Algorithms

Mergesort and Counting Inversions

Tyler Moore

CS 2123, The University of Tulsa

Some slides created by or adapted from Dr. Kevin Wayne. For more information see http://www.cs.princeton.edu/~wayne/kleinberg-tardos. Some code reused or adapted from Python Algorithms by Magnus Lie Hetland.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide up problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

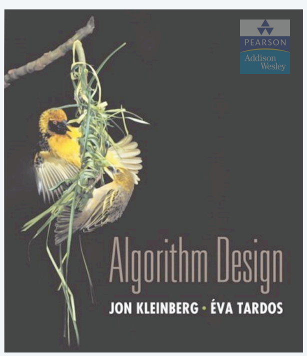
Most common usage.

- Divide problem of size n into two subproblems of size $n/2$ in linear time.
- Solve two subproblems recursively.
- Combine two solutions into overall solution in linear time.

Consequence.

- Brute force: $\Theta(n^2)$.
- Divide-and-conquer: $\Theta(n \log n)$.

attributed to Julius Caesar



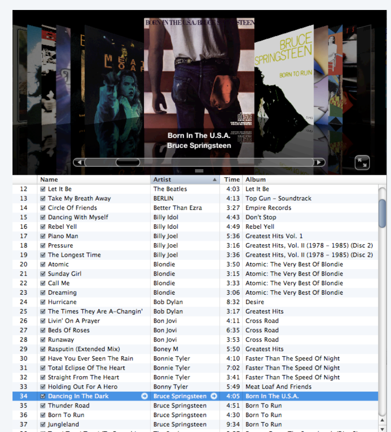
SECTION 5.1

5. DIVIDE AND CONQUER

- *mergesort*
- *counting inversions*
- *closest pair of points*
- *randomized quicksort*
- *median and selection*

Sorting problem

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



Sorting applications

Obvious applications.

- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

Some problems become easier once elements are sorted.

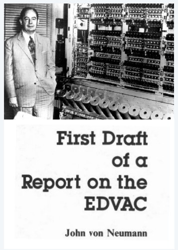
- Identify statistical outliers.
- Binary search in a database.
- Remove duplicates in a mailing list.

Non-obvious applications.

- Convex hull.
- Closest pair of points.
- Interval scheduling / interval partitioning.
- Minimum spanning trees (Kruskal's algorithm).
- Scheduling to minimize maximum lateness or average completion time.
- ...

Mergesort

- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.



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

Merging

Goal. Combine two sorted lists A and B into a sorted whole C.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i \le b_j$, append a_i to C (no larger than any remaining element in B).
- If $a_i > b_j$, append b_j to C (smaller than every remaining element in A).

sorted list A

3	7	10	a_i	18

sorted list B

2	11	b_j	17	23
5	2			

merge to form sorted list C

2	3	7	10	11					

Canonical Divide-Conquer-Glue Algorithm

```
def divide_and_conquer(S, divide, glue):
    if len(S) == 1: return S
    L, R = divide(S)
    A = divide_and_conquer(L, divide, glue)
    B = divide_and_conquer(R, divide, glue)
    return glue(A, B)
```

Mergesort in Python

```python
def mergesort(seq):
    mid = len(seq)/2                    #Midpoint for division
    lft, rgt = seq[:mid], seq[mid:]
    if len(lft) > 1: lft = mergesort(lft)#Sort by halves
    if len(rgt) > 1: rgt = mergesort(rgt)
    res = []                            #Merge sorted halves
    while lft and rgt:                  #Neither half is empty
        if lft[-1] >= rgt[-1]:          #lft has greatest last value
            res.append(lft.pop())       #Append it
        else:                           #rgt has greatest last value
            res.append(rgt.pop())       #Append it
    res.reverse()                       #Result is backward
    return (lft or rgt) + res           #Also add the remainder
```


How can we measure the time complexity of recursive algorithms?

- Measuring the time complexity of iterative algorithms is usually straightforward: count the inputs, check for loops, etc.
- We know that certain operations can take linear time, constant time, logarithmic time, etc.
- Running those operation in a loop n times produces a multiplicative factor
- But how can we do this for recursive algorithms? With recurrence relations

Recurrence Relations

- Recurrence relations specify the cost of executing recursive functions.
- Consider mergesort
  1. Linear-time cost to divide the lists
  2. Two recursive calls are made, each given half the original input
  3. Linear-time cost to merge the resulting lists together
- Recurrence: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- Great, but how does this help us estimate the running time?

A useful recurrence relation

Def. $T(n)$ = max number of compares to mergesort a list of size $\leq n$.
Note. $T(n)$ is monotone nondecreasing.

Mergesort recurrence.

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

Solution. $T(n)$ is $O(n \log_2 n)$.

Assorted proofs. We describe several ways to prove this recurrence.
Initially we assume n is a power of 2 and replace $\leq$ with $=$.

8

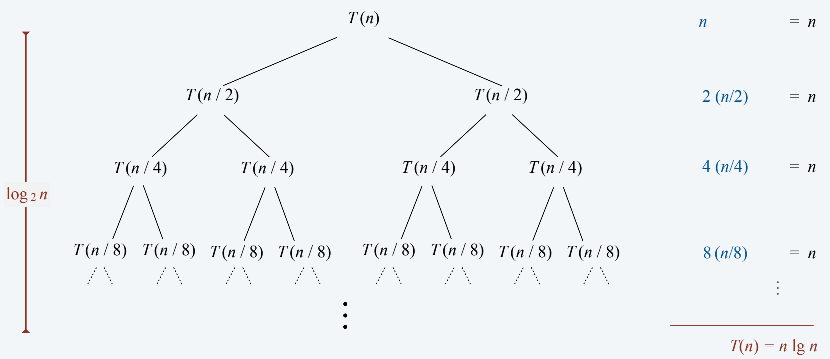
Divide-and-conquer recurrence: proof by recursion tree

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

assuming n is a power of 2

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

Pf 1.



Proof by induction

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

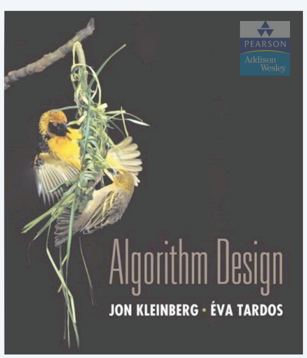
assuming n is a power of 2

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

Pf 2. [by induction on n]

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

$$\begin{aligned} T(2n) &= 2\,T(n) + 2n \\ &= 2\,n \log_2 n + 2n \\ &= 2\,n\,(\log_2 (2n) - 1) + 2n \\ &= 2\,n \log_2 (2n). \quad \blacksquare \end{aligned}$$



SECTION 5.3

5. DIVIDE AND CONQUER

- mergesort
- *counting inversions*
- closest pair of points
- randomized quicksort
- median and selection

Counting inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: $1, 2, \ldots, n$.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j are inverted if $i < j$, but $a_i > a_j$.

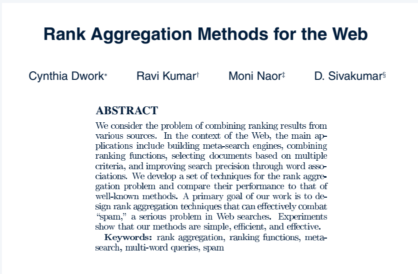
	A	B	C	D	E
me	1	2	3	4	5
you	1	3	4	2	5

2 inversions: 3–2, 4–2

Brute force: check all $\Theta(n^2)$ pairs.

Counting inversions: applications

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's tau distance).



Rank Aggregation Methods for the Web

Cynthia Dwork* Ravi Kumar† Moni Naor‡ D. Sivakumar§

ABSTRACT

We consider the problem of combining ranking results from various sources. In the context of the Web, the main applications include building meta-search engines, combining ranking functions, selecting documents based on multiple criteria, and improving search precision through word associations. We develop a set of techniques for the rank aggregation problem and compare their performance to that of well-known methods. A primary goal of our work is to design rank aggregation techniques that can effectively combat "spam," a serious problem in Web searches. Experiments show that our methods are simple, efficient, and effective.

Keywords: rank aggregation, ranking functions, meta-search, multi-word queries, spam

Counting inversions: divide-and-conquer

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

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 ∈ A and b ∈ 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: how to combine two subproblems?

Q. How to count inversions (a, b) with $a \in A$ and $b \in B$?
A. Easy if A and B are sorted!

Warmup algorithm.

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

list A

7	10	18	3	14

list B

17	23	2	11	16

sort A

3	7	10	14	18

sort B

2	11	16	17	23

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

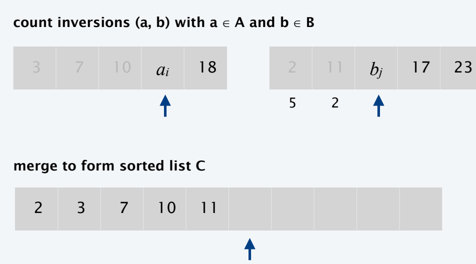
3	7	10	14	18

2	11	16	17	23
5	2	1	1	0

Counting inversions: how to combine two subproblems?

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming 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 left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.



Counting inversions: divide-and-conquer algorithm implementation

Input. List L.
Output. Number of inversions in L and sorted list of elements L'.

```
SORT-AND-COUNT (L)

IF list L has one element
    RETURN (0, L).

DIVIDE the list into two halves A and B.
(r_A , A) ← SORT-AND-COUNT(A).
(r_B , B) ← SORT-AND-COUNT(B).
(r_AB , L') ← MERGE-AND-COUNT(A, B).

RETURN (r_A + r_B + r_AB , L').
```


Counting inversions: divide-and-conquer algorithm analysis

Proposition. The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.

Pf. The worst-case running time $T(n)$ satisfies the recurrence:

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