CMSC 451: Divide and Conquer

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sections 5.1–5.3 of *Algorithm Design* by Kleinberg & Tardos.

Greedy Recap

Greedy algorithms are usually very natural.

Many problems have nice greedy solutions:

1. Topological Sorting (ch. 3)
2. Interval Scheduling (4.1)
3. Interval Partitioning (4.1)
4. Minimizing Lateness (4.2)
5. Optimal Scheduling (4.3)
6. Shortest Paths (Dijkstra's) (4.4)
7. Minimum Spanning Tree (4.5)
8. Maximum Separation Clustering (4.7)
9. Matroids: Max-Weight

Greedy Recap, 2

We've seen some general patterns for algorithms:

1. Sort and then scan (matroid greedy algorithm)
2. TreeGrowing

And for proof techniques:

1. **Greedy "stays ahead"**: any choice the OPT can make, the greedy can make and will
2. **Exchange**: we can transform an OPT solution into greedy with series of small changes.
3. **Matroid**: Hereditary Subset System with Augmentation Property

Divide and Conquer

Divide and Conquer is a different framework.

Related to *induction*:

- Suppose you have a "box" that can solve problems of size $\leq k < n$
- You use this box on some subset of the input items to get partial answers
- You combine these partial answers to get the full answer.

But: you construct the "box" by recursively applying the same idea until the problem is small enough to be solved by brute force.

Merge Sort

```
MergeSort(L):
    if |L| = 2:
        return [min(L), max(L)]
    else:
        L1 = MergeSort(L[0, |L|/2])
        L2 = MergeSort(L[|L|/2+1, |L|-1])
        return Combine(L1, L2)
```

- In practice, you sort in-place rather than making new lists.
- `Combine(L1,L2)` walks down the sorted lists putting the smaller number onto a new list. Takes $O(n)$ time
- Total time: $T(n) \leq 2T(n/2) + cn$.

To Solve a Recurrence

Given a recurrence such as $T(n) \leq 2T(n/2) + cn$, we want a simple upper bound on the total running time.

Two common ways to "solve" such a recurrence:

1. Unroll the recurrence and see what the pattern is.
   Typically, you'll draw the recursion tree.

2. Guess an answer and prove that it's right.

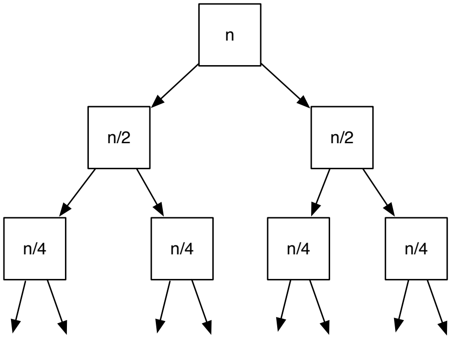
Solving Recurrences

Draw the first few levels of the tree.

Write the amount of work done at each level in terms of the level.

Figure out the height of the tree.

Sum over all levels of the tree.



$T(n) \leq 2T(n/2) + cn$

Each level is cn. There are $\log n$ levels, so $T(n)$ is $O(n \log n)$.

Substitution Method

Substitution method is based on induction. We:

1. Show $T(k) \leq f(k)$ for some small k.
2. Assume $T(k) \leq f(k)$ for all $k < n$.
3. Show $T(n) \leq f(n)$.

$T(n) \leq 2T(n/2) + cn$

Base Case: $2c \log 2 = 2c \geq T(2)$

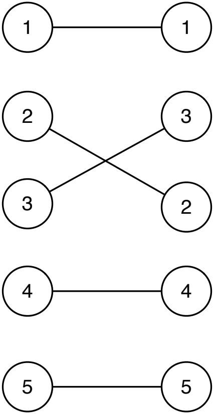
Induction Step:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2c(n/2)\log(n/2) + cn \\ &= cn[(\log n) - 1] + cn \\ &= cn \log n \end{aligned}$$

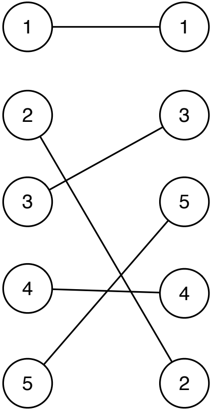
Counting Inversions

Comparing Rankings

Suppose two customers rank a list of movies.



similar



more different

A measure of distance

What's a good measure of how dissimilar two rankings are?

A measure of distance

What's a good measure of how dissimilar two rankings are?

We can count the number of inversions:

- Assume one of the rankings is $1, 2, 3, \ldots, n$.
- Denote the other ranking by $a_1, a_2, \ldots, a_n$.
- An inversion is a pair (i, j) such that $i < j$ but $a_j < a_i$.

Two identical rankings have no inversions.

How many inversions do opposite rankings have?

A measure of distance

What's a good measure of how dissimilar two rankings are?

We can count the number of <u>inversions</u>:

- Assume one of the rankings is $1, 2, 3, \ldots, n$.
- Denote the other ranking by $a_1, a_2, \ldots, a_n$.
- An <u>inversion</u> is a pair (i, j) such that $i < j$ but $a_j < a_i$.

Two identical rankings have no inversions.

How many inversions do opposite rankings have? $\binom{n}{2}$

How can we count inversions quickly?

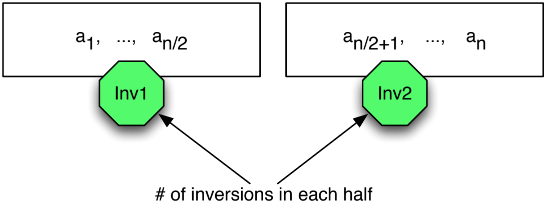
How can we count inversions quickly?

- **Brute Force:** check every pair: $O(n^2)$.
- Some sequences might have $O(n^2)$ inversions, so you might think that it might take as much as $O(n^2)$ time to count them.
- In fact, with divide and conquer, you can count them in $O(n \log n)$ time.

Basic Divide and Conquer

Count the number of inversions in the sequence $a_1, \ldots, a_n$.

Suppose I told you the number of inversions in the first half of the list and in the second half of the list:



What kinds of inversions are not accounted for in `Inv1 + Inv2`?

Half-Crossing Inversions

The inversions we have to count during the merge step:

$\boxed{a_1, \; \ldots, \; a_{n/2}} \quad \boxed{a_{n/2+1}, \; \ldots, \; a_n}$

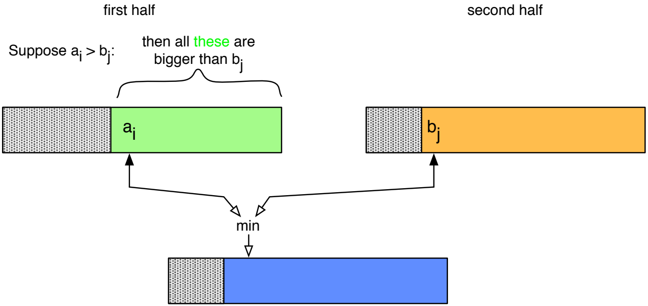
$$a_i > a_j$$

The crux is that we have to count these kinds of inversion in $O(n)$ time.

What if each of the half lists were sorted?

Suppose each of the half lists were sorted.

If we find a pair $a_i > a_j$, then we can infer many other inversions:



Each of the green items is an inversion with b_j.

Merge-and-Count

```
MergeAndCount(SortedList A, SortedList B):
  a = b = CrossInvCount = 0
  OutList = empty list
  While a < |A| and b < |B|:    // not at end of a list
     next = min(A[a], B[b])
     OutList.append(next)

     If B[b] == next:
        b = b + 1
        CrossInvCount += |A| - a //inc by # left in A
     Else
        a = a + 1
   EndWhile
   Append the non-empty list to OutList
   Return CrossInvCount and OutList
```

Sorted!

Note that MergeAndCount will produce a sorted list as well as the number of cross inversions.

SortAndCount

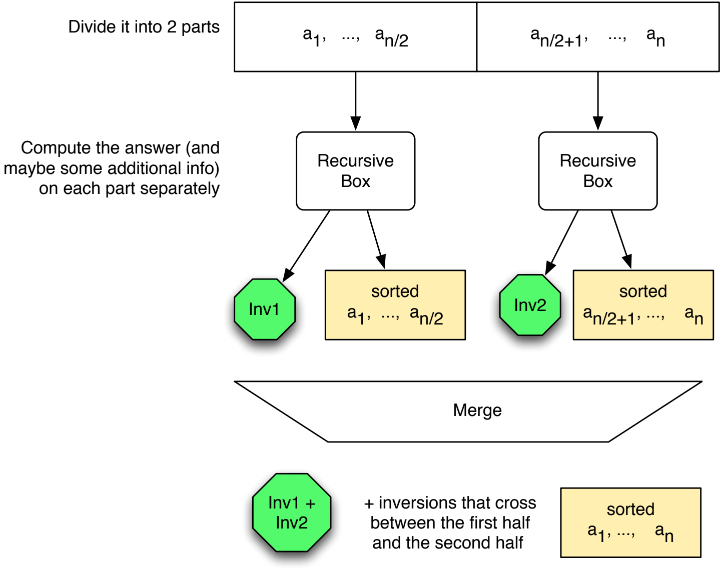
```
SortAndCount(List L):
    If |L| == 1: Return 0

    A, B = first & second halves of L

    invA, SortedA = SortAndCount(A)
    invB, SortedB = SortAndCount(B)

    crossInv, SortedL = MergeAndSort(SortedA, SortedB)
    Return invA + invB + crossInv  and SortedL
```

Algorithm Schematic



Running time?

What's the running time of `SortAndCount`?

Running time?

What's the running time of `SortAndCount`?

Break the problem into two halves.

Merge takes $O(n)$ time.

$$T(n) \leq 2T(n/2) + cn$$

$\implies$ Total running time is $O(n \log n)$.