
Divide &
Conquer

Divide and Conquer

Divide and Conquer is a **runtime improvement technique**, that you can apply on top of existing algorithms of any type (Brute Force, Greedy, Dynamic Programming, etc.)

- MergeSort and QuickSort are the standard examples of Divide and Conquer.
- Take a large problem, and break it up into two or more smaller pieces (divide)
- Recursively solve the smaller pieces, by continuing to break them up until you reach constant-sized base cases.
- Merge your solved pieces together to get the overall solution (conquer)

Recursion

Dynamic Programming recursion is **completely different** than Divide & Conquer recursion.

- DP recursion is sequential (bite-sized decisions)
- D&C recursion is parallelized, with each subproblem independent of the others.
- DP recursion can be turned into a much more efficient iterative algorithm.
- D&C recursion is hard to unroll, and does not net an asymptotic improvement to runtime.

Counting Inversions

You are writing software for Pandora270.

- Each user ranks some number n of songs, from favorite to least favorite.
- You want to find users with similar tastes in music, so that you can recommend songs.
- You will take the songs that two users have in common, and compare their relative rankings.
- User 1 ranks 5 songs, conveniently named 1, 2, 3, 4, 5, ranked in that order.
- User 2 ranks them 1, 3, 4, 2, 5. How similar are these rankings?
- In one sense, they only agree on their favorite and least favorite songs, so they are 40% similar.
- Why is that an unfair metric?

Inversions

1 2 3 4 5

1 3 4 2 5

How many songs do we actually disagree on?

- Just song 2, which is off by two places.

An **inversion** is a pair of songs i and j, such that i < j, but song j ranked better than song i.

How many inversions could there be for n songs?

- C(n,2) = $\frac{n(n-1)}{2}$, or 10 when n=5.

What are our inversions for the above example?

- {2,3} and {2,4}.

We have 2 out of 10 possible inversions, so we should say that our lists are **80%** similar.

Brute-Force

We want to count the number of inversions in a list of n numbers (or n songs).

- There are $\Theta(n^2)$ possible inversions, so we could count each inversion in quadratic time by checking each one manually.

Maybe we can do better, using Divide & Conquer!

Consider the following ranking:

1 5 4 8 10 2 6 9 12 11 3 7

Divide in O(1) time:

1 5 4 8 10 2 | 6 9 12 11 3 7

Recursively solve each piece:

5 inversions | 8 inversions

Then merge our two solutions together.

A First Attempt

Our recursive call needs a base case.

- When you're down to a single song, return 0.

If the left subproblem returns 5 inversions, and the right subproblem returns 8 inversions, do we just return 5+8 for our total inversions?

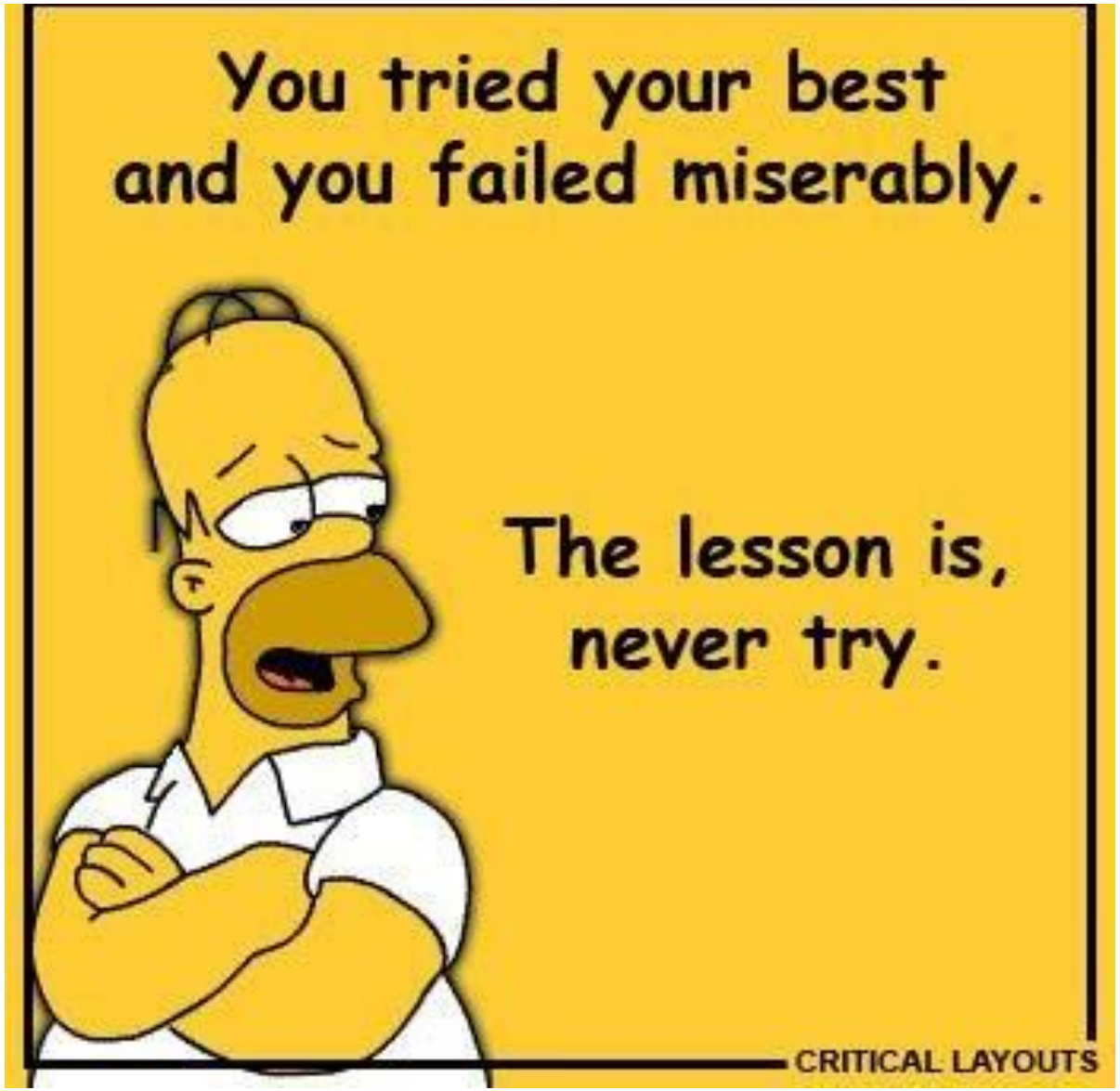
- We also need to count every inversion spanning the division (which turns out to be 9 inversions).

What's the recurrence relation for our divide & conquer algorithm?

- $f(n) = 2f(\frac{n}{2}) + \Theta(n^2)$

$f(n) = \Theta(n^2)$

- Well that was useless…


You tried your best
and you failed miserably.
The lesson is,
never try.
CRITICAL LAYOUTS

A Second Attempt

Often your first attempt at a D&C algorithm will net no improvement. It **will** identify which part of the algorithm needs to be improved.

- The combine phase is the problem.

How would we need to store the songs in each list, so as to count the inversions more easily?

- How does MergeSort merge so easily?

We will both count inversions **and** sort the list.

- We are not losing any information, because we've already counted the inversions we're eliminating when sorting.

The Algorithm

1 5 4 8 10 2 6 9 12 11 3 7

1 5 4 8 10 2 | 6 9 12 11 3 7

5 inversions | 8 inversions

1 2 4 5 8 10 | 3 6 7 9 11 12

2 4 5 8 10 6 7 9 11

1 2 3 4 5 6 7 8 9 10 11 12

+4 +2 +2 +1

When pulling from the right list, we check how many numbers are remaining in the left list (n-i+1, where n is the total number in the left list, and i is our iterator).

- In linear time, we both merged the two lists, and counted up the 9 inversions.

This is the MergeSort variant for counting inversions, and thus achieves n log n: BubbleSort is the traditional n^2 algorithm for this.

All-Pairs Shortest Paths Strikes Back

ASP[i,x,z] stores the length of the shortest path from x to z, using $\leq$ i edges.

- ASP[i,x,x] = 0
- ASP[0,x,z] = ∞
- ASP[i,x,z] = $\min_{\langle x,y\rangle \in E}(c_{\langle x,y\rangle} + ASP[i-1, y, z])$

For i = 0 to n-1

    For all nodes z

        For all nodes x

            Calculate ASP[i,x,z]

- Answers at ASP[n-1]

Runtime = $\Theta(mn^2)$

- In Dynamic Programming, we figured out which node to visit next. What would we do for Divide & Conquer?

All-Pairs Shortest Paths

We will figure out which node to visit in the middle.

It's still, at its core, Dynamic Programming, as we will try all possible middle-nodes and take the best.

- ASP[i,x,z] = $\min_{y \in V}$(ASP[$\lfloor \frac{i}{2} \rfloor$, x, y] + ASP[$\lceil \frac{i}{2} \rceil$, y, z])
- ASP[1,x,z] = $c_{\langle x,z \rangle}$
- ASP[i,x,x] = 0

For i = 1 to n-1

    For all nodes x

        For all nodes z

            Calculate ASP[i,x,z]

Runtime = $\Theta(n^4)$

…oops

A Second Attempt

What values of i do we actually need to calculate?

For i = 0 to log n

    For all nodes x

        For all nodes z

            Calculate ASP[i,x,z]

ASP[i,x,z] stores the length of the shortest path from x to z, using no more than $\mathbf{2}^i$ edges.

- Answers are at ASP[$2^{\lceil \log n \rceil}$]

Runtime = $\Theta(n^3 \log n)$

- This is usually better than running Bellman-Ford n times, but depends on how dense the graph is.

Integer Multiplication

Presumably you recall how to multiply numbers:

```
   12
x 13
   36
 120
 156
```

The same idea works for numbers in binary:

```
      1100
   x  1101
      1100
    110000
   1100000
10011100
```

Elementary Math

What is the runtime to multiply two n-bit integers, using this method?

- $\Theta(n^2)$

We will try to improve this, using Divide & Conquer.

- We want to multiply two n-bit integers, X and Y.
- Let x_F be the first $\frac{n}{2}$ bits of X, and x_L be the last bits.
- Similarly for y_F and y_L.
- So, $x = x_F \cdot 2^{\frac{n}{2}} + x_L$ and $y = y_F \cdot 2^{\frac{n}{2}} + y_L$
- $x \cdot y = x_F \cdot y_F \cdot 2^n + (x_F \cdot y_L + y_F \cdot x_L) \cdot 2^{\frac{n}{2}} + x_L \cdot y_L$

Our base case is multiplying two 1-bit numbers.

What's the recurrence relation?

- $f(n) = 4\, f(\frac{n}{2}) + \Theta(n)$

A Second Attempt

Where's the problem in our recurrence relation?

- The 4. We have too many subproblems.

How can we do this with only 3 subproblems?

- Hint: our first multiplication will be $(x_F + x_L) \cdot (y_F + y_L)$
- $M_2 = x_F \cdot y_F$
- $M_3 = x_L \cdot y_L$
- $x \cdot y = M_2 \cdot 2^n + (M_1 - M_2 - M_3) \cdot 2^{\frac{n}{2}} + M_3$

$f(n) = 3\, f(\frac{n}{2}) + \Theta(n)$

- $f(n) = \Theta(n^{\log 3}) \approx n^{1.58}$

Sequence Alignment

The **edit distance** between two strings $X = x_1 \ldots x_n$ and $Y = y_1 \ldots y_m$ is the minimal number of changes to transform X into Y.

- SA[i,j] is the edit distance of $X = x_i \ldots x_n$ and $Y = y_j \ldots y_m$.
- SA[i,m+1] = n+1-i
- SA[n+1,j] = m+1-j
- SA[i,j] = SA[i+1,j+1], if $x_i = y_j$.
- SA[i,j] = 1 + min(SA[i,j+1], SA[i+1,j], SA[i+1,j+1]), otherwise
- Runtime and space are both $\Theta(mn)$

Improving the space requirements

To calculate the matrix, we only need to remember the current and previous column.

- Therefore, we can reduce the memory requirements to $\Theta(m+n)$
- The cost is that we cannot reconstruct the solution.
- We will attempt to get the best of all worlds, using Divide & Conquer: reconstructing the answer in $\Theta(mn)$ time and $\Theta(m+n)$ space

The high level idea

Some prefix of Y will match with the first half of X. The rest of Y will match with the second half of X.

- We don't know where the optimal split point of Y is, so, dynamic programming-style, we will try all possible splits and take the best one.
- We will run Sequence Alignment, using our memory-saving technique, on the first half of X and all prefixes of Y.
- We will also run it on the second half of X and all suffixes of Y.
- Then we will inspect the results and figure out where the optimal split point is.

The high level idea: visually

Prefix of Y	First half of X	Suffix of Y	Second half of X	Total
ε	5	$y_1 \ldots y_m$	4	9
y_1	4	$y_2 \ldots y_m$	3	7
$y_1 y_2$	3	$y_3 \ldots y_m$	2	5
$y_1 y_2 y_3$	2	$y_4 \ldots y_m$	4	6
…	…	…	…	…
$y_1 \ldots y_{m-1}$	3	y_m	5	8
$y_1 \ldots y_m$	4	ε	6	10

Divide & Conquer

We will have determined the total edit distance, as well as the optimal split point, using only $\Theta(m+n)$ space.

$x_1x_2x_3x_4$	$x_5x_6x_7x_8$
$y_1y_2y_3$	$y_4y_5y_6y_7y_8y_9$

Now we repeat the process on the left-subproblem, and the right-subproblem

x_1x_2	x_3x_4	x_5x_6	x_7x_8
y_1	y_2y_3	y_4y_5	$y_6y_7y_8y_9$

We repeat until we are matching substrings of Y with individual characters of X, at which point we have the full reconstruction.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
y_1	ε	y_2	y_3	y_4y_5	ε	y_6y_7	y_8y_9

Sequence Alignment on the last half of X, and suffixes of Y

Run Sequence Alignment on the last half of X and **all** of Y (using our space-saving technique).

Runtime = $\Theta(mn)$

	$i = \frac{n}{2}$	$i = \frac{n}{2} + 1$	…	i = n
j=1	4	5	We	are
j=2	3	4	not	saving
…	…	…	these	columns,
j=m	5	4	to	save
j=m+1	6	5	space	!

The first column stores the edit distances of all possible suffixes of Y with the last half of X.

Sequence Alignment on the first half of X, and prefixes of Y

Run Sequence Alignment on the first half of X and **all** of Y (using our space-saving technique).

	i = 1	i = 2	…	i = $\frac{n}{2}$ -1
j=1	4	5	We	are
j=2	3	4	not	saving
…	…	…	these	columns,
j=m	5	4	to	save
j=m+1	6	5	space	!

Does the first column store the edit distances of all possible prefixes of Y with the first half of X?

- Nope, it's still suffixes of Y.

Sequence Alignment on the first half of X, and prefixes of Y, Take two

Run Sequence Alignment on the first half of X and the reverse of Y (using our space-saving technique).

	i = 1	i = 2	…	i = $\frac{n}{2}$ -1
j=m	4	5	We	are
j=m-1	3	4	not	saving
…	…	…	these	columns,
j=1	5	4	to	save
j=0	6	5	space	!

Does this work now?

- Nope, now we're matching the first character of X with the last character of Y.

Sequence Alignment on the first half of X, and prefixes of Y, Take three

Run Sequence Alignment on the first half of X (reversed) and the reverse of Y (using our space-saving technique).

	$i = \frac{n}{2} - 1$	$i = \frac{n}{2} - 2$	…	i = 1
j=m	4	5	We	are
j=m-1	3	4	not	saving
…	…	…	these	columns,
j=1	5	4	to	save
j=0	6	5	space	!

Now this works!

Analysis

Each individual call takes only $\Theta(m+n)$ space.

- Additionally, we need to store the reconstruction, but that also takes only $\Theta(m+n)$ space.

Our recurrence relation is

$f(n,m) = f(\frac{n}{2}, k) + f(\frac{n}{2}, m-k) + \Theta(mn)$.

- The next level of recursion will get runtime

$$\frac{n}{2}k + \frac{n}{2}(m-k) = \frac{n}{2}m$$

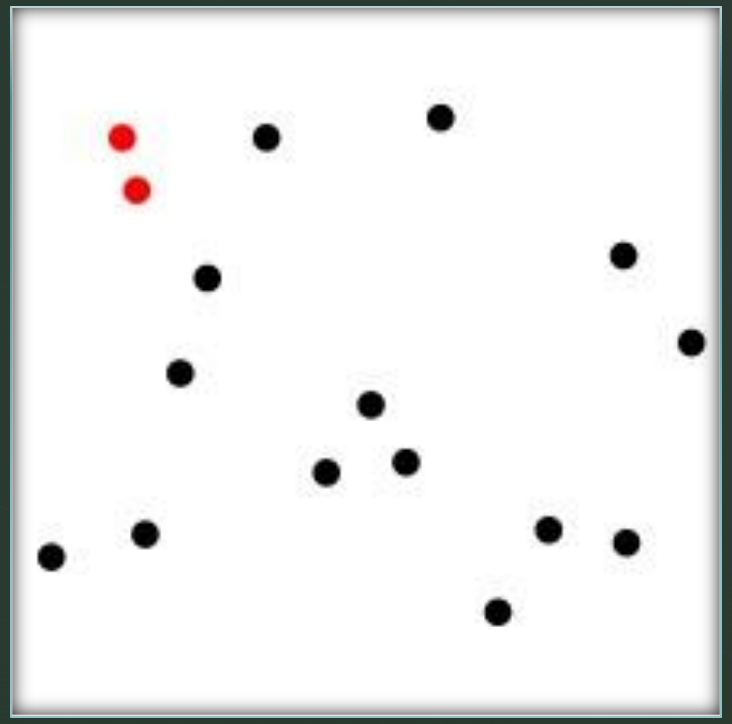
- Each successive level divides the runtime in half, so the total runtime is $\Theta(mn)$

Closest Points on a Plane

Given n points on a plane (specified by their x and y coordinates), find the pair of points with the smallest Euclidean distance between them.

What would be the runtime for a Brute-Force algorithm?

- $\Theta(n^2)$



Divide & Conquer

How should we divide the problem?

- We can divide at the median x-coordinate, so that half of the points are on the left, and half of the points are on the right.
- If we also try to split on the y-coordinate, there is no guarantee we will be able to split the problem into equal quarters.

Recursion

We will continually split the problem until we have 2-3 points, and then we will brute-force identify the closest pair of points among them.

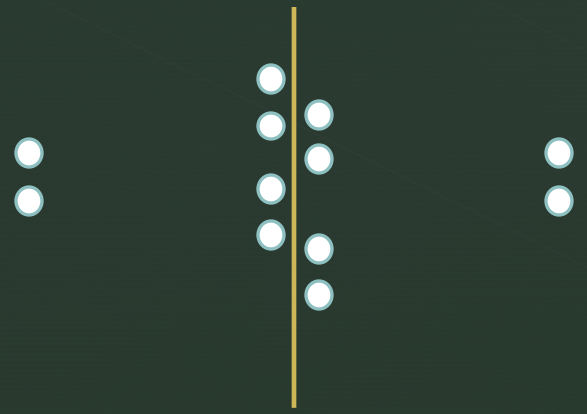
- So, if I've found the closest pair of points on the left, with distance δ, and the closest pair of points on the right, with distance ε, do I just return min(δ, ε)?
- We also need to check points that span the divide. If we brute force this, we'll get the following recurrence:
- $f(n) = 2f(\frac{n}{2})+\Theta(n^2)$

The Combine Phase, Take Two

Do I really need to compare all pairs of points that span the divide?

- Assume δ is the smallest distance over both subproblems. Then we only need to look at points that are within δ of the dividing line.

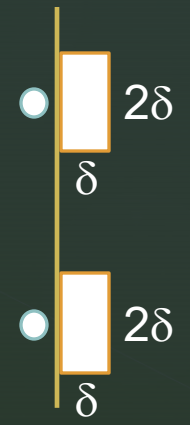
Does this improve our runtime?



The Combine Phase, Take Three

Do I really need to compare all pairs of points within δ of the dividing line?

- For each point, I only need to look at the points on the other side that are within δ of the given y-coordinate.
- Does this improve the runtime?
- It does! Since the smallest pair of points on the right side are at least δ apart, we can only fit a constant number of points in the rectangle!
- The new recurrence is $f(n) = 2f(\frac{n}{2})+\Theta(n)$



Details of Implementation

1. Sort the list of points (list 1) by x-coordinate: $\Theta(n \log n)$
2. Sort the list of points (list 2) by y-coordinate: $\Theta(n \log n)$
3. Use list 1 to figure out the division in constant time, and recursively call the algorithm.
4. Walk through list 2, and pull out the points that are within δ of the dividing line (list 3).
5. For each point in list 3, compare it with each successive point in list 3 until you are over δ away from that point: this is a constant number of comparisons.

Matrix Multiplication

The standard way to multiply matrices:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

$$c_{ij} = \sum_{k} a_{ik} b_{kj}$$

Using this method, how long will it take to multiply two n by n matrices?

- $\Theta(n^3)$

Divide & Conquer

We can divide our n by n matrices into 4 quadrants:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Then we need to figure out how to multiply two $\frac{n}{2}$ by $\frac{n}{2}$ matrices together (AE, BG, AF, BH, CE, DG, CF, DH).

We continue to break these down until we're multiplying scalars.

What's the recurrence relation?

- $f(n) = 8f(\frac{n}{2}) + \Theta(n^2)$

Divide & Conquer, Take Two

It is possible to reduce the number of multiplies:

1. $M_1 = A(F-H)$
2. $M_2 = (A+B)H$
3. $M_3 = (C+D)E$
4. $M_4 = D(G-E)$
5. $M_5 = (A+D)(E+H)$
6. $M_6 = (B-D)(G+H)$
7. $M_7 = (A-C)(E+F)$

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} E & F \\ G & H \end{pmatrix} =$$

$$\begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} =$$

$$\begin{pmatrix} M_5 + M_4 - M_2 + M_6 & M_1 + M_2 \\ M_3 + M_4 & M_1 + M_5 - M_3 - M_7 \end{pmatrix}$$

$f(n) = 7f(\frac{n}{2}) + \Theta(n^2)$

$f(n) = \Theta(n^{\log 7}) \approx \Theta(n^{2.81})$

For n = 2500, this has a speedup factor of 8

Improvements

This is known as Strassen Multiplication, and was discovered in 1969.

- In 1971, Hopcroft and Kerr showed that 6 multiplies is impossible
- In 1980, Pan showed that if you split the matrix up into 70^2 sectors, you can achieve 143640 multiplies $\approx \Theta(n^{2.80})$
- In December 1979, we got a general $O(n^{2.521813})$ algorithm.
- In January 1980 this was improved to $O(n^{2.521801})$
- In 1987, Coppersmith and Winograd achieved $O(n^{2.376})$
- There have been small improvements starting in 2010
- Hypothesis: $O(n^{2+\varepsilon})$ is possible

Take-Home Practice

Chapter 5

Exercises 1, 2, 3, 6, 7