

03. Divide-and-Conquer

CPSC 535

Kevin A. Wortman



This work is licensed under a Creative Commons Attribution 4.0 International License.

Divide-and-Conquer

One of the *big ideas* of computer science problem solving

1. **Divide** a problem into smaller parts
2. **Conquer** the smaller problems recursively
3. **Combine** the smaller solutions into one solution for the original problem

(The term carries some baggage from the age of imperialism.)

Divide-and-conquer, outside of algorithm design

- ▶ Software design; breaking features into classes, functions
- ▶ Networking; OSI seven layer model
- ▶ Parallel processing; MapReduce
- ▶ Software process; agile methods; sprints

Divide-and-conquer at a high level

```
1: function DIVIDE-AND-CONQUER(INPUT)
2:   if INPUT is base case then
3:     return trivial base case solution
4:   else
5:      $x_1, x_2, \dots, x_k = \text{divide INPUT into } k \text{ pieces (often 2)}$ 
6:      $s_1 = \text{DIVIDE-AND-CONQUER}(x_1)$ 
7:     ...
8:      $s_k = \text{DIVIDE-AND-CONQUER}(x_k)$ 
9:      $S = \text{combine } s_1, \dots, s_k \text{ into one solution}$ 
10:    return  $S$ 
11:   end if
12: end function
```

Time complexity recurrences

Recursive pseudocode leads to recurrences in run-time functions

Suppose base case is $n = 1$ and takes $\Theta(1)$ time; in the recursive case we divide evenly into k pieces of size $\approx n/k$, recurse once on each, and spend $f(n)$ time in the *divide* and *conquer* phases:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ kT(n/k) + f(n) & \text{if } n > 1. \end{cases}$$

Recall merge sort divides into $k = 2$ pieces, merge takes $\Theta(n)$ time:

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

Taking liberties with recurrences

General math: bound recurrences precisely including constant factors

Algorithm analysis: ordinarily bounding asymptotically; Θ notation will hide constant factors anyway; drop math details that can only impact constants and add clutter

- ▶ drop ceilings/floors, so write e.g. $n/2$ in lieu of $\lceil n/2 \rceil$ or $\lfloor n/2 \rfloor$ is more precise
- ▶ when the base case is $\Theta(1)$ time for $n < c$ for some $c \in \Theta(1)$, don't bother writing it explicitly; so

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

is abbreviated as

$$T(n) = 2T(n/2) + \Theta(n)$$

Maximum subarray problem

Maximum subarray problem

input: an array $\langle p_1, p_2, \dots, p_n \rangle$ where each $p_i \in \mathbb{R}$ is a *profit* (or loss) on day i

output: indices s, e with $s \leq e$, maximizing the total profit

$$\sum_{i=s}^e p_i$$

Applications

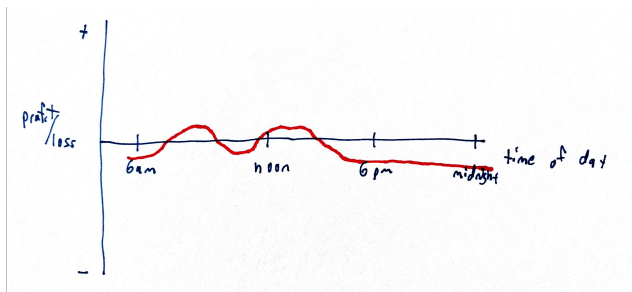
- ▶ buy then sell a stock/security
- ▶ pick opening/closing time of a retail store with slow periods
- ▶ computer vision, data mining: identify region most consistent with a pattern e.g. street striping

Examples

The optimal subarray may involve negative elements:

$$\langle 100, -1, -1, -1, 5, 3 \rangle$$

Application: when to open/close a cafe:



Greedy fails

Straightforward greedy algorithm would be:

- ▶ buy at the lowest price or sell at the highest price
- ▶ incorrect; best "run" could be elsewhere
- ▶ example: $\langle 0, 1, 10, 4, 4, 4, 4 \rangle$
 - ▶ $\langle 1, 10 \rangle$ is the biggest trough-to-peak; sum 11
 - ▶ but slow-and-steady $\langle 4, 4, 4, 4 \rangle$ has sum 12
- ▶ not always correct \implies not actually an algorithm

Brute force

Exhaustive search: try every legal start/end

```
1: function BRUTE-FORCE-MAX-SUBARRAY(P)
2:    $s = e = 1$ 
3:   for  $i$  from 1 to  $n$  do
4:     for  $j$  from  $i$  to  $n$  do
5:       if  $(\sum p_i \dots p_j) > (\sum p_s \dots p_e)$  then
6:          $s = i, e = j$ 
7:       end if
8:     end for
9:   end for
10:  return  $(s, e)$ 
11: end function
```

$\Theta(n^3)$ time as written; can cache sums to achieve $\Theta(n^2)$

Divide-and-conquer brainstorm

Divide: chop array in half into two smaller arrays L, R

Conquer: recursively compute maximum subarray in L and in R

Combine: maximum subarray of entire P could be

1. subarray entirely in L ;
2. subarray entirely in R ; or
3. *crossing* subarray that starts in L and ends in R

(exhaustive case analysis)

Theme with **combine**: choose best among small solutions (easy)
or a distinct solution that crosses boundaries (trickier)

Identify crossing subarray — try 1

Suppose the two pieces of P are $P[low \dots mid]$ and $P[mid + 1 \dots high]$

Tempting to try all pairs of $s \in \{low, \dots, mid\}$ and $e \in \{mid + 1, \dots, high\}$

Would work, but

- ▶ time becomes $T(n) = 2T(n/2) + \Theta(n^2)$ which is $\Theta(n^2)$ by master theorem
- ▶ same time as brute force, but more complicated \implies not a win

Identify crossing subarray — insight

Theme in algorithm design: in general, a more specific problem admits a faster and/or simpler algorithm

First try is not using the fact that a *crossing* subarray must cross *mid*

- ▶ substantially simplifies the search
- ▶ *s* is how far before *mid*; separately, *e* is how far after *mid*?
- ▶ two separate 1D searches \implies two linear loops
- ▶ $\Theta(n) + \Theta(n) = \Theta(2n) = \Theta(n)$ time
- ▶ versus: *s* is where, and *e* is how much later?
- ▶ 2D search \implies two nested loops $\implies \Theta(n^2)$ time
- ▶ location of the "2" is profound; $\Theta(2n) \ll \Theta(n^2)$

Identify crossing subarray — try 2

```
1: function MAX-CROSSING-SUBARRAY(P, low, mid, high)
2:   leftsum = rightsum =  $-\infty$ 
3:   sum = 0
4:   for i from mid down to low do
5:     sum = sum + P[i]
6:     if sum > leftsum then
7:       leftsum = sum
8:       maxleft = i
9:     end if
10:  end for
11:  sum = 0
12:  for i from mid + 1 to high do
13:    sum = sum + P[i]
14:    if sum > rightsum then
15:      rightsum = sum
16:      maxright = i
17:    end if
18:  end for
19:  return (maxleft, maxright, leftsum + rightsum)
20: end function
```

$\Theta(n)$ time

(Note scoping of *maxleft*, *maxright*, and that they are inevitably initialized.)

Maximum subarray algorithm

```

1: function MAX-SUBARRAY(P, low, high)
2:   if low == high then
3:     return (low, high, P[low])
4:   else
5:     mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
6:     (leftlow, lefthigh, leftsum) = MAX - SUBARRAY(P, low, mid)
7:     (rightlow, righthigh, rightsum) = MAX - SUBARRAY(P, mid + 1, high)
8:     (crosslow, crosshigh, crosssum) = MAX - CROSSING - SUBARRAY(A, low, mid, high)
9:     if leftsum ≥ rightsum and leftsum ≥ crosssum then
10:      return (leftlow, lefthigh, leftsum)           ▷ entirely-left subarray
11:     else if rightsum ≥ leftsum and rightsum ≥ crosssum then
12:      return (rightlow, righthigh, rightsum)       ▷ entirely-right subarray
13:     else
14:      return (crosslow, crosshigh, crosssum)       ▷ mid-crossing subarray
15:     end if
16:   end if
17: end function

```

Maximum subarray analysis

D&C runtime is

$$T(n) = 2T(n/2) + \Theta(n)$$

Solves to $\Theta(n \log n)$, by master theorem, same as merge sort.

Brute force was $\Theta(n^2)$

- ▶ D&C is much faster
- ▶ perhaps counterintuitive due to recursion's reputation for sloth
- ▶ D&C benefits from observation that subarrays are contiguous, so extend in two directions from a middle
- ▶ brute force is oblivious to this
- ▶ human mathematical insight eliminates wasted effort

Matrix multiplication

Matrix multiplication problem

input: A, B each an $n \times n$ matrix

output: matrix product $C = AB$

Recall notation: element at row i and column j of matrix A is denoted a_{ij}

Definition of matrix multiplication:

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}.$$

Naïve matrix multiplication

```
1: function MATRIX-MULTIPLY(A, B)
2:    $C = \text{new } n \times n \text{ matrix}$ 
3:   for  $i$  from 1 to  $n$  do
4:     for  $j$  from 1 to  $n$  do
5:        $c_{ij} = 0$ 
6:       for  $k$  from 1 to  $n$  do
7:          $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8:       end for
9:     end for
10:  end for
11:  return  $C$ 
12: end function
```

$\Theta(n^3)$ time

Is naïve optimal?

The definition of matrix multiplication involves a sum that is iterated n times, for each of the $n \times n$ elements of C , which might seem to require exactly n^3 scalar multiply instructions, and imply an $\Omega(n^3)$ lower bound for matrix multiplication.

Surprise! Strassen's algorithm (1969) takes $O(n^{\lg 7}) = O(n^{2.81})$ time; more complicated Williams-Le Gall algorithm (2014) takes $O(n^{2.37})$ time

Insight: per the definition of matrix multiplication, some elements of A and B are multiplied together more than once; avoid duplicating these efforts.

Moving to divide-and-conquer

Suppose n is an even power of 2, i.e. $n = 2^k$ for $k \geq 0$
(Can preprocess A, B by adding padding zeroes, then trim the zeroes out of C .)

Divide A into four equal-size submatrices, and same for B, C .

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

so we can compute C as

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Moving to divide-and-conquer (continued)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

can be broken down into four separate computations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

each of which can be performed recursively.

Divide-and-conquer matrix multiplication — try 1

```
1: function MMR(A, B)
2:    $C = \text{new } n \times n \text{ matrix}$ 
3:   if  $n == 1$  then
4:      $c_{11} = a_{11} \cdot b_{11}$ 
5:   else
6:     quadrisect  $A, B, C$ 
7:      $C_{11} = \text{MMR}(A_{11}, B_{11}) + \text{MMR}(A_{12}, B_{21})$ 
8:      $C_{12} = \text{MMR}(A_{11}, B_{12}) + \text{MMR}(A_{12}, B_{22})$ 
9:      $C_{21} = \text{MMR}(A_{21}, B_{11}) + \text{MMR}(A_{22}, B_{21})$ 
10:     $C_{22} = \text{MMR}(A_{21}, B_{12}) + \text{MMR}(A_{22}, B_{22})$ 
11:   end if
12:   return  $C$ 
13: end function
```

Analysis

- ▶ each of the submatrices A_{11} , etc. has size $n/2$
- ▶ quadrisecting A, B is $\Theta(n^2)$ time; same for assembling C
- ▶ each matrix + takes $\Theta((\frac{n}{2})^2) = \Theta(\frac{n^2}{4}) = \Theta(n^2)$ time
- ▶ 8 recursive calls

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Solves to $T(n) \in \Theta(n^3)$ by master theorem; same as naïve

Observe: the 8 factor is meaningful, but the $\frac{1}{4}$ isn't

\implies it's a win to have fewer recursive calls, but more work (by a constant factor) in the **combine** step

Strassen's insight

Use algebra to refactor into 7 recursive multiplies instead of 8

1. quadrisect A, B, C as before
2. create $10 (n/2) \times (n/2)$ submatrices S_1, \dots, S_{10} using matrix $+$ and $-$
3. recursively compute 7 submatrix products P_1, \dots, P_7 in terms of the matrices from steps 1, 2
4. compute $C_{11}, C_{12}, C_{21}, C_{22}$ using matrix $+$ and $-$

$$\begin{aligned}T(n) &= \Theta(n^2) + \Theta(10 \frac{n}{4}) + 7T(n/2) + T(4 \frac{n}{4}) \\&= 7T(n/2) + \Theta(n^2) \\&\in \Theta(n^{\lg 7})\end{aligned}$$

by master theorem

Divide-and-conquer matrix multiplication — try 2

```
1: function MMS(A, B)
2:   C = new  $n \times n$  matrix
3:   if  $n == 1$  then
4:      $c_{11} = a_{11} \cdot b_{11}$ 
5:   else
6:     quadrisect A, B, C
7:     form  $S_1, \dots, S_{10}$  as shown on next slide
8:      $P_1 = \text{MMS}(A_{11}, S_1)$ 
9:      $P_2 = \text{MMS}(S_2, B_{22})$ 
10:     $P_3 = \text{MMS}(S_3, B_{11})$ 
11:     $P_4 = \text{MMS}(A_{22}, S_4)$ 
12:     $P_5 = \text{MMS}(S_5, S_6)$ 
13:     $P_6 = \text{MMS}(S_7, S_8)$ 
14:     $P_7 = \text{MMS}(S_9, S_{10})$ 
15:     $C_{11} = P_5 + P_4 - P_2 + P_6$ 
16:     $C_{12} = P_1 + P_2$ 
17:     $C_{21} = P_3 + P_4$ 
18:     $C_{22} = P_5 + P_1 - P_3 - P_7$ 
19:   end if
20:   return C
21: end function
```


Details of Strassen's algorithm

$$S_1 = B_{12} - B_{22}$$

$$S_2 = A_{11} + A_{12}$$

$$S_3 = A_{21} + A_{22}$$

$$S_4 = B_{21} - B_{11}$$

$$S_5 = A_{11} + A_{22}$$

$$S_6 = B_{11} + B_{22}$$

$$S_7 = A_{12} - A_{22}$$

$$S_8 = B_{21} + B_{22}$$

$$S_9 = A_{11} - A_{21}$$

$$S_{10} = B_{11} + B_{12}$$

$$P_1 = A_{11} \cdot S_1$$

$$P_2 = S_2 \cdot B_{22}$$

$$P_3 = S_3 \cdot B_{11}$$

$$P_4 = A_{22} \cdot S_4$$

$$P_5 = S_5 \cdot S_6$$

$$P_6 = S_7 \cdot S_8$$

$$P_7 = S_9 \cdot S_{10}$$

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

Editorial Commentary

- ▶ proof that 7 recursive multiplies suffice, instead of 8, is surprising and therefore interesting
- ▶ equations on previous slide are relatively uninteresting (though not unimportant) **technical detail**
- ▶ $o(n^3)$ matrix multiply is of great theoretical interest (because surprise)
- ▶ but the naïve alg. has substantially better constant factors, and the gap between $\Theta(n^3)$ and $\Theta(n^{2.81})$ is narrow
- ▶ Strassen (and descendants) are only practical for very large n
- ▶ in practice: naïve alg. for base case $n < 128$ (say)

Takeaways

Recall

- ▶ insertion sort is $\Theta(n^2)$; D&C merge sort is $\Theta(n \log n)$
- ▶ brute force maximum subarray is $\Theta(n^2)$; D&C alg. is $\Theta(n \log n)$
- ▶ naïve matrix multiply is $\Theta(n^3)$; Strassen's alg. is $\Theta(n^{2.81})$

In each case study,

- ▶ first try was no faster; just using D&C isn't an automatic improvement
- ▶ master method analyses hinted at the bottleneck
- ▶ shift work around to decrease asymptotic time complexity (but increase constant factors); beneficial trade-off
- ▶ optimization comes from human insight into the problem
- ▶ unclear how to make these insights w/o the D&C framing

Master Method

- ▶ **Master method:** plug-and-chug process for solving some recurrences
 - ▶ doesn't work for *all*
 - ▶ but works for *typical* D&C recurrences
- ▶ **Master theorem:** proof that the method is sound

Master Theorem

Let $a \geq 1$, $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ be the recurrence

$$T(n) = aT(n/b) + f(n).$$

Then

1. If $\exists \epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $\exists \epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n , then $T(n) = \Theta(f(n))$.

Step 1: Identify Relevant Case

3 cases: $f(n)$ is asymptotically

1. less than,
2. equal,
3. greater than

the benchmark $n^{\log_b a}$.

So identify a and b , substitute into $n^{\log_b a}$, simplify, and decide among the cases.

If unsure, take the limit

$$\lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a}}.$$

Step 1: Identify Relevant Case

Example:

$$T(n) = 2T(n/2) + 7$$

Identify: $a = 2$, $b = 2$

Plug and chug:

$$n^{\log_b a} = n^{\log_2(2)} = n^{(1)} = n$$

Is 7 less than, equal, or greater than n ?

Intuition: less than

Check:

$$\lim_{n \rightarrow \infty} \frac{7}{n} = 0$$

Step 2 alternative 1: Justify Case 1

Need to Prove: If $\exists \epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

Prove by showing an example of a ϵ that makes $f(n) = O(n^{\log_b a - \epsilon})$.

Continuing example: show ϵ s.t. $7 = O(n^{1-\epsilon})$ Choose $\epsilon = 1$, so we have $7 = O(n^{1-(1)}) = O(n^0) = O(1)$

Justification: "We have $f(n) = 7$ and $n^{\log_b a} = n^{\log_{(2)}(2)} = n^{(1)} = n$. Let $\epsilon = 1$; then $f(n) = O(n^{\log_b a - \epsilon}) = O(n^{(1)-(1)}) = O(n^{(0)}) = O(1)$, and by case 1 of the master theorem, $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$."

Step 2 alternative 2: Justify Case 2

Need to Prove: If $f(n) = O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

Case 2 is true without qualification; don't need to show anything else.

Example: $T(n) = 2T(n/2) + 5n$, so $a = 2$, $b = 2$, and $f(n) = 5n$.

$n^{\log_b a} = n^{\log_{(2)}(2)} = n^{(1)} = n$ is asymptotically equal to $f(n) = 5n$ so case 2 applies and $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.

Justification: "We have $f(n) = 5n$ and $n^{\log_b a} = n^{\log_{(2)}(2)} = n^{(1)} = n$. Case 2 of the master theorem applies, so $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^{(1)} \log n) = \Theta(n \log n)$."

Step 2 alternative 3: Justify Case 3

Need to Prove: If $\exists \epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n , then $T(n) = \Theta(f(n))$.

Prove by showing examples of ϵ, c that makes $f(n) = O(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ for large n .

Example: $T(n) = 2T(n/2) + n^2$, so $a = 2$, $b = 2$, and $f(n) = n^2$.

$n^{\log_b a} = n^{\log_{(2)}(2)} = n^{(1)} = n$. Can choose $\epsilon = 1$ to have $\Omega(n^{1+(1)})$.

Step 2 alternative 3: Justify Case 3 (cont'd)

Need

$$af(n/b) \leq cf(n)$$

$$(2)f\left(\frac{n}{2}\right) \leq c(n^2)$$

$$2 \cdot \left(\frac{n}{2}\right)^2 \leq cn^2$$

$$2 \cdot \frac{n^2}{4} \leq cn^2$$

$$\frac{1}{2} \leq c$$

Any c satisfying $\frac{1}{2} \leq c < 1$ can work; arbitrarily choose $c = \frac{3}{4}$.

Step 2 alternative 3: Justify Case 3 (cont'd)

Justification: "We have $f(n) = n^2$ and $n^{\log_b a} = n^{\log_2(2)} = n^{(1)} = n$. Let $\epsilon = 1$; then $f(n) = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{(1)+(1)}) = \Omega(n^2)$. Let $c = \frac{3}{4}$; then $af(n/b) = (2)f(n/2) = 2(n/2)^2 = 2(n^2/4) = n^2/2 \leq cf(n) = (\frac{3}{4})n^2$ for sufficiently large n . Case 3 of the master theorem applies, so $T(n) = \Theta(f(n)) = \Theta(n^2)$."

Limitations Of The Master Method

Master Theorem is a big “if/then”

Theorem does not apply when:

- ▶ $T(n)$ not in the necessary form, or
- ▶ none of cases 1, 2, 3 apply

There are gaps between the cases.

However, when an algo. is designed according to the D&C pattern, the master theorem almost always applies.