Divide and Conquer

Christopher R. Genovese

Department of Statistics & Data Science

Tue 28 Oct 2025
Session #17

Plan

Finishing Greedy Algorithms

Plan

Finishing Greedy Algorithms

Divide and Conquer: Motivation

Plan

Finishing Greedy Algorithms

Divide and Conquer: Motivation

Divide and Conquer: Examples

Plan

Finishing Greedy Algorithms

Divide and Conquer: Motivation

Divide and Conquer: Examples

Divide and Conquer: A Framework

Plan

Finishing Greedy Algorithms

Divide and Conquer: Motivation

Divide and Conquer: Examples

Divide and Conquer: A Framework

A Quick Tour of Zippers

Announcements

- In documents: lecture notes, fpc updated draft
- **Reading**:
    - Zippers
        - You could have invented Zippers
        - Huet Zippers (optional but interesting)
        - Zippers as Derivatives (optional, for later)
    - Hughes, Why Functional Programming Matters
    - Commentary on Hughes with Python examples
    - Starting sequence on categories
        - What is Category Theory?
        - Definitions and Examples
        - What is a Functor? Part 1
        - What is a Functor? Part 2
        - Fibonacci Functor
        - Natural Transformations
    - Backus, Can Programming be Liberated from the Von Neumann Architecture
- **Homework**:
    - **zippers** assignment due Fri 31 Oct.
    - Push outstanding mini-assignments when complete

Plan

Finishing Greedy Algorithms

Divide and Conquer: Motivation

Divide and Conquer: Examples

Divide and Conquer: A Framework

A Quick Tour of Zippers

Recap: Greedy Algorithms

A **greedy algorithm** is one in which we make the choice that *looks best at each particular moment*.

We saw several examples and derived a condition implying that the greedy solution is globally optimal. *This need not hold.*

There are more general conditions that broaden the scope of greedy algorithms, but greedy algorithms are still limited.

We also saw the **fold fusion law**:

```
post (foldRight step init comps) = foldRight gstep ginit comps
```

when ginit = post init and for all c, a

```
post (step c a) = gstep c (post a)
```

This can be used to prove a variety of algorithmic claims.

Let's consider one more example before broadening our view.

Example: Huffman Coding

We have a stream of tokens drawn from some distribution. We want to encode that stream to compress the data. Specifically, we assign a binary codeword (a string of 0's and 1's) to each token such that more frequent tokens have shorter code words.

We want to come as possible to the *information bound* on data compression for this source.

Note: We will consider only **prefix codes**, in which no code word is a prefix of any other code word. This makes decoding efficient and (it turns out) costs us nothing in effectiveness.

We will represent binary prefix codes with "leafy" binary trees, with left branches corresponding to 0 and right branches corresponding to 1 and with tokens at the leaves.

```
data BTree t = Leaf t | Branch (BTree t) (BTree t)
  deriving (Display, Functor)
```

Aside: Making a Code

How do we build a code from a tree? Try a functional approach or use `fp-concepts`, if you can.

```
makeCode : BTree a -> NonEmptyList (a, Text)
```

Aside: Making a Code

In TL1:

```
makeCode : BTree a -> NonEmptyList (a, Text)
makeCode = go ""
  where go sofar (Leaf x) = [(x, sofar)]
        go sofar (Branch l r) = go (sofar <> "0") l ++ go (sofar <> "1") r
```

Using fp-concepts:

```python
def make_code[A](bt: LeafyBinaryTree[A]) -> List[tuple[A, str]]:
    tagged = bt.imap(lambda index, value: (value, "".join(index)))
    return foldMap(identity, tagged, Collect)
```

Now, given a code, how would you write `encode` and `decode`? Here, both have type `Code -> Text -> Text`, where `Code` is a data structure that you would build from the result of `make_code`/`makeCode`.

Implementation

We start with a forest, where each tree is a singleton node for one token (and its frequency). We then greedily combine them. The greedy algorithm is linear-logarithmic in general.

Consider how you would implement this.

Here's a simple example to use:

```
|-------+------+------+------+------+------+------|
| token |    a |    b |    c |    d |    e |    f |
| freq  | 0.45 | 0.13 | 0.12 | 0.16 | 0.09 | 0.05 |
|-------+------+------+------+------+------+------|
```

The types:

```
huffman : (Numeric f, Ord f) => NonEmptyList (f, a) -> NonEmptyList (a, Text)
huffman = makeCode . huffmanTree

huffmanTree : (Numeric f, Ord f) => NonEmptyList (f, a) -> BTree a

makeCode : BTree a -> NonEmptyList (a, Text)
```

Implementation

In TL2, this looks like

```
function huffman(tokens, freqs):
    q = PriorityQueue()
    for f, t in zip(freqs, tokens):
        q.insert((f, BLeaf(f, t)))

    n = len(tokens)
    for iteration in range(1,n):
        freq1, min1 = q.pop_min()
        freq2, min2 = q.pop_min()
        new_freq = freq1 + freq2
        q.insert((freq1 + freq2, BTree(new_freq, left=min1, right=min2)))
    return q.pop_min()[1]
```

Implementation

In TL1, this looks like

```
huffman : (Numeric f, Ord f) => NonEmptyList (f, a) -> NonEmptyList (a, Text)
huffman = makeCode . huffmanTree

huffmanTree : (Numeric f, Ord f) => NonEmptyList (f, a) -> BTree a
huffmanTree tokens =
  let queue : PriorityQueue
      queue = fromList $ map (fst `fork` Leaf . snd) tokens
  in
    snd $ findMin $ until isSingleton update queue
  where
    update : PriorityQueue f (BTree a) -> PriorityQueue f (BTree a)
    update pq = let ((freq1, best1), pq') = popMin pq
                    ((freq2, best2), pq'') = popMin pq'
                    freq' = freq1 + freq2
                in
                  insert freq' (Branch best1 best2) pq''

makeCode : BTree a -> NonEmptyList (a, Text)
makeCode = go ""
  where go sofar (Leaf x) = [(x, sofar)]
        go sofar (Branch left right) = go (sofar <> "0") left ++
                                       go (sofar <> "1") right
```

Plan

Recursive Thinking

Many seemingly intimidating problems can be solved by combining *the solution to a smaller versions of the same problem*.

There are several variants of this idea. Soon, we will see *dynamic programming*. Today, we look at problems that can be solved by a **divide and conquer** strategy.

This involves a recursive solution in three main steps:

1. **Divide** the problem into *smaller subproblems of the same type*, frequently just two subproblems of approximately the same size.

2. **Conquer** the subproblems:
   - Small enough subproblems can be solved directly. This is called the *base case*.
   - Otherwise, solve the subproblem recursively – applying the ***same algorithm*** to the subproblem. This is called the *recursive (or inductive) case*.

3. **Combine** the subproblem solutions into a solution for the original problem.

Recursive thinking is powerful but takes some getting used to!

Plan

Finishing Greedy Algorithms

Divide and Conquer: Motivation

Divide and Conquer: Examples

Divide and Conquer: A Framework

A Quick Tour of Zippers

Example: MergeSort

Problem: Given an array of objects $a_1, \ldots, a_n$ from an ordered set (e.g., numbers), find a permutation of the array such that $a_1 \leq a_2 \leq \cdots \leq a_n$.

One divide-and-conquer approach to this problem is **mergesort**:

- **Divide** the n-element array into two arrays of $n/2$ elements each.
- **Conquer** by sorting the two subarrays recursively using `mergesort`.
- **Combine** the sorted subarrays by merging into a single sorted array.

The base case here is when the subarrays have length 1 (or some small fixed length in practice).

MergeSort – The merge Utility

To merge the sorted subarrays, imagine having a function merge as follows.

Function merge(a, b):

Inputs:

- a :: Array or vector in sorted (not sordid) order
- b :: Another array or vector in sorted order

Output:

- An array or vector containing all the elements from a and b, combined into sorted order.

How should the computation time of merge vary with n, the total size of a and b?

We will write/sketch this later.

MergeSort – Sorting

Assume you have merge. Write mergesort(a) that sorts a using divide-and conquer.

MergeSort – Sorting, A Solution

```
mergesort <- function(a) {
    hi <- length(a)

    if (1 < hi) { # at least two elements in array
        mid <- floor((1 + hi) / 2)

        first_half <- mergesort(a[1:mid])
        second_half <- mergesort(a[(mid + 1):hi])

        a <- merge(first_half, second_half)
    }

    return(a)
}
```

MergeSort – Back to merge

How would you implement merge? (Hint: Think about two piles of cards.)

MergeSort – Performance

Thoughts to pursue fully another day:

- How could we estimate how long this algorithm takes, in big O notation?

- How could we estimate how much *memory* the implementation above takes, in big O notation?

- Can we fix the memory problem?

In practice, this algorithm can be made very effective.

Example: Multiplication

Multiplying two complex numbers seems to require *four* multiplications:

$$(a + ib)(c + id) = ac - bd + i(ad + bc),$$

but we can do it in three

$$(a + ib)(c + id) = ac - bd + i[(a + b)(c + d) - ac - bd].$$

This reduction of cost by $3/4$ might not be a big deal for a single multiplication, but when we apply this strategy **recursively** the $3/4$ gain is applied at each stage – to significant effect.

Example: Multiplication (cont'd)

Now consider multipying two n-bit integers x and y, where n is a *power of 2*. Thinking in terms of their binary representations, we can write

$$x = 2^{n/2}x_L + x_R$$
$$y = 2^{n/2}y_L + y_R,$$

where x_L, x_R, etc. are $n/2$-bit integers. For example, if $x = 10110110_2$ then $x_L = 1011_2$ and $x_R = 0110_2$.

We can apply the multiplication trick above to reduce four multiplications to three:

$$xy = 2^n x_L y_L + x_R y_R + 2^{n/2}[(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R].$$

The additions and multiplication by powers of 2 (shifts) are fast.

The arbitrary multiplications are the costly operation, but we have reduced the problem to three multiplications of half the size.

We can apply the same trick recursively to each of these three multiplications.

Example: Multiplication (cont'd)

TL2 pseudo-code:

```
fast_multiply(x,y):
    n = max(bitwidth(x), bitwidth(y))
    if n <= NATIVE_BITS:
        return x*y

    x_L, x_R = leftmost ceiling(n/2), rightmost floor(n/2) bits of x
    y_L, y_R = leftmost ceiling(n/2), rightmost floor(n/2) bits of y

    a = fast_multiply(x_L, y_L)
    b = fast_multiply(x_R, y_R)
    c = fast_multiply(x_L + x_R, y_L + y_R)
    return (a << n) + b + (c - a - b) << n/2
```

Practical Note: On real machines, it would not make sense to recurse down to the single bit level. Rather, the base case would be whatever size supports fast native multiplications on the given machine (e.g., 16-, 32-, or 64-bit).

Example: Multiplication (cont'd)

We can estimate the number of operations T_n required for this recursive algorithm on an n-bit number:

$$T_n \leq 3T_{n/2} + O(n),$$

which gives us $T_n = O(n^{\log_2 3}) \approx O(n^{1.59})$. This is notably better than the $O(n^2)$ performance we get if we use four multiplications.

Example: Multiplication – Extension to Semirings

We can extend this idea to matrix multiplication, and indeed to general semirings. This is called *Strassen's algorithm*.

The standard blockwise matrix multiplication gives eight multiplications of submatrices:

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

We can reduce this to *seven* multiplications with a bit of rearrangement.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} M_5 + M_4 - M_2 + M_6 & M_1 + M_2 \\ M_3 + M_4 & M_1 + M_5 - M_3 - M_7 \end{bmatrix}$$

where

$$\begin{aligned} M_1 &= A(F - H) & M_5 &= (A + D)(E + H) \\ M_2 &= (A + B)H & M_6 &= (B - D)(G + H) \\ M_3 &= (C + D)E & M_7 &= (A - C)(E + F) \\ M_4 &= D(G - E). \end{aligned}$$

Now recurse. This yields $O(n^{\log_2 7})$ versus $O(n^3)$.

Example: Selecting Order Statistics

Suppose you have a numeric vector with a very large dimension n (in the billions, say, or even more). You want to find the *median* of these values.

What's the most direct way to do this? What are some issues we might face with that approach?

Let's devise a divide-and-conquer method for this problem. To do this, we need to (at least):

1. Identify appropriate subproblems
2. Decide which subproblems to solve recursively and which to solve directly.
3. Determine how to combine subproblem solutions to get solutions to larger problems.

Number 1 is the tricky part here!

We will apply the same algorithm to the smaller problems. If you had a median value v already, what subproblems does this suggest? Ideas?

We will develop pseudo-code for a function median(X) that computes a median of the vector X by divide-and-conquer.

Example: Selecting Order Statistics

Given a list X, we want to pick the k-th smallest element from X in *linear time*.

Function `selection(X,k)`:

1. Pick a random "pivot" v from the list.
2. Partition X into X_L, X_v, X_R, values less than, equal to, and greater than v.
3. Let $k' = k - \text{size}(X_L) - \text{size}(X_v)$.
4. Return

$$\text{selection}(X, k) = \begin{cases} \text{selection}(X_L, k) & \text{if } k \leq \text{size}(X_L) \\ v & \text{if } \text{size}(X_L) < k \leq \text{size}(X_L) + \text{size}(X_v) \\ \text{selection}(X_R, k') & \text{if } k > \text{size}(X_L) + \text{size}(X_v). \end{cases}$$

Computation of the three sublists can be done in linear time (and in place, without copying the lists). At each stage, we reduce the size of the list from $\text{size}(X)$ to $\max(\text{size}(X_L), \text{size}(X_R))$.

If we pick v to split X in half, we get a linear time algorithm. But wait, that would need the median?!? Instead take v to be the median of a small random sample from the list. This is good with high probability and tends to work in practice.

The Fast Fourier Transform (FFT)

The FFT is a critically important algorithm for computing a critically important mathematical transform.

The FFT is based on a divide-and-conquer algorithm for **fast polynomial multiplication**, and it has other recursive representations as well.

If

$$A(x) = a_0 + a_1 x + \cdots + a_d x^d$$
$$B(x) = b_0 + b_1 x + \cdots + b_d x^d,$$

then

$$C(x) = A(x)B(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d},$$

where $c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0$. The coefficients of C are the **convolutions** of the coefficients of A and B.

The Fast Fourier Transform (FFT) (cont'd)

Key Ideas:

- A d-degree polynomial is determined by its value at $d + 1$ distinct points.

- This gives us two equivalent but distinct **representations** of a polynomial: the coefficients and the values at selected points.

- It is slow to multiply polynomials in the *coefficient representation*, but fast to multiply them in the *value representation*.

- Divide and conquer gives us a fast, recursive way to evaluate polynomials if we use special points called the roots of unity

- Divide and conquer gives us a fast way to move between coefficient and value representation, almost for free.

FFT: Multiplying in the Value Representation

1. Pick $n \geq 2d + 1$ distinct points $x_1, \ldots, x_n$

2. Given polynomials (value representation) $A(x_1), \ldots, A(x_n)$ and $B(x_1), \ldots, B(x_n)$, form products

$$C(x_1), \ldots, C(x_n) = A(x_1)B(x_1), \ldots, A(x_n)B(x_n)$$

3. Recover coefficient representation of C

FFT: Divide and Conquer

We can split A (and similarly B) as follows

$$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2).$$

Then to compute any pair of values $\pm x$, we can share calculations – giving a speed up.

$$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$$
$$A(-x) = A_{\text{even}}(x^2) - xA_{\text{odd}}(x^2),$$

requiring two evaluations of a smaller polynomial to compute both two evaluations of A.

Using pairs $\pm x_0, \ldots, \pm x_{n/2-1}$ seems to work fine at the first level. But the next set of points is $x_0^2, \ldots, x_{n/2-1}^2$. How can those be positive-negative pairs?

The answer: use well chosen complex numbers – **n-th roots of unity**!

FFT: Divide and Conquer (cont'd)

If $\omega^n = 1$, ω is an nth root of unity. The value $\omega = e^{2\pi i/n}$ is called the *principal* nth root of unity because all the others are derived from it. The nth roots of unity satisfy two properties that are important for our purposes:

- They come in positive-negative pairs: $\omega^{n/2+j} = -\omega^j$.

- Squaring them produces $(n/2)$-nd roots of unity: $(\omega^2)^{n/2} = 1$.

So if we start with the right roots of unity, our divide and conquer recursion keeps its special properties throughout.

FFT: Changing Representations

Using divide-and-conquer, we can change from the coefficient to the value representation:

```
Function fft(A, w):
  Inputs: A is a polynomial in the coefficient representation
          w is an nth root of unity
  Output: The values of A at selected points
```

Do:

1. If $\omega = 1$, return $A(1)$.

2. Write $A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2)$

3. Call $\texttt{fft}(A_{\text{even}}, \omega^2)$ to evaluate A_{even} at even powers of ω.

4. Call $\texttt{fft}(A_{\text{odd}}, \omega^2)$ to evaluate A_{odd} at even powers of ω.

5. for $j \in [0 \mathinner{.\,.} n-1]$, $A(\omega^j) = A_{\text{even}}(\omega^2 j) + \omega^j A_{\text{odd}}(\omega^2 j)$

Return $A(\omega^0), ..., A(\omega^{n-1})$.

FFT: Changing Representations (cont'd)

Amazingly, the FFT is invertible, so we can also translate from value to coefficient representations!

$$\text{values} = \text{fft}(\text{coefficients}, \omega)$$
$$\text{coefficients} = \frac{1}{n}\,\text{fft}(\text{values}, \omega^{-1}).$$

Polynomial multiplication becomes

1. Apply fft to convert from coefficient to value representation
2. Multiply value representations
3. Apply (inverse) fft to convert back to coefficient representation

FFT: Generalizing

It is useful to express this in matrix-vector terms. For instance, a polynomial A can be written as

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

with the matrix – call it $M(x)$ – known as the *Vandermonde* matrix.

As long as $x_0, \ldots, x_{n-1}$ are distinct, $M(x)$ is invertible.

Evaluation is multiplication by M; interpolation is multiplication by M^{-1}.

The FFT is just multiplication by $M(\omega)$ for an nth root of unity. The inverse FFT is obtained from the fact that $M^{-1}(\omega) = M(\omega^{-1})/n$.

Our general FFT uses divide-and-conquer to make this multiplication fast, similar to what we did above with the polynomials.

FFT: Generalizing

```
Function fft(a, w):
 Input: a = (a_0, ..., a_{n-1}) is an array, n is a power of 2
        w is a primitive nth root of unity
 Output: M_n(w) a

if w == 1:
    return a
(s_0, ..., s_{n/2 - 1}) = fft((a0, a_2, ..., a_{n-2}), w^2)
(u_0, ..., u_{n/2 - 1}) = fft((a1, a_3, ..., a_{n-1}), w^2)

for j from 0 to n/2 - 1:
    r_j = s_j + w^j u_j
    r_{j + n/2} = s_j - w^j u_j

return (r_0, ..., r_{n-1})
```

Plan

Finishing Greedy Algorithms

Divide and Conquer: Motivation

Divide and Conquer: Examples

Divide and Conquer: A Framework

A Quick Tour of Zippers

A General Framework

Here is a general framework for solving Divide and Conquer problems. (Only doing a split into two subproblems here, but easy to replace the pair by a list.)

```
-- when we can prove things
data DandC problem solution = record DandC where
    initial : problem
    isTrivial : problem -> Bool
    divide : problem -> Either problem (problem, problem)
    conquered : (p : problem ## isTrivial p === True) -> solution
    conquer : solution -> solution -> solution

-- in practice in most languages:
-- problem must be trivial in conquered or error thrown
data DandC problem solution = record DandC where
    initial : problem
    isTrivial : problem -> Bool
    divide : problem -> Either problem (problem, problem)
    conquered : problem -> solution    -- trivial case or error thrown
    conquer : solution -> solution -> solution
```

Try defining this record for the various examples we've seen.

A General Framework

```
data DandC problem solution = record DandC where
  initial : problem
  isTrivial : problem -> Bool
  divide : problem -> Either problem (problem, problem)
  conquered : problem -> solution    -- trivial case or error thrown
  conquer : solution -> solution -> solution

BinaryTree a = Leaf a | Branch (BinaryTree a) a (BinaryTree a)

unfoldBTree : (b -> Either a (b, a, b)) -> b -> BinaryTree a




foldBTree : (a -> s) -> (s -> s -> s) -> BinaryTree a -> s
```

A General Framework

```
data DandC problem solution = record DandC where
  initial : problem
  isTrivial : problem -> Bool
  divide : problem -> Either problem (problem, problem)
  conquered : problem -> solution    -- trivial case or error thrown
  conquer : solution -> solution -> solution

BinaryTree a = Leaf a | Branch (BinaryTree a) a (BinaryTree a)

unfoldBTree : (b -> Either a (b, a, b)) -> b -> BinaryTree a
unfoldBTree gen seed =
  case gen seed of
    Left base         -> Leaf base
    Right (sl, p, sr) -> Branch (unfoldBTree gen s1) p (unfoldBTree gen s1)

foldBTree : (a -> s) -> (s -> s -> s) -> BinaryTree a -> s
foldBTree trivial combine (Leaf p) = trivial p
foldBTree trivial combine (Branch p1 p2) =
  combine (foldBTree trivial combine p1) (foldBTree trivial combine p2)
```

A General Framework

And now we can put it together into a general form:

```
divideAndConquer (DandC {initial, divide, conquered, conquer}) =
  unfoldBTree divide initial |> foldBTree conquered conquer
```

Here, |> is the pipeline operator as in R.

Can we avoid building an explicit tree? Yes!

A General Framework

And now we can put it together into a general form:

```
divideAndConquer (DandC {initial, divide, conquered, conquer}) =
  unfoldBTree divide initial |> foldBTree conquered conquer
```

Here, |> is the pipeline operator as in R.

Can we avoid building an explicit tree? Yes!

Later...

Plan

Zippers for Trees

Zippers are a data structure that enable the exploration and modification of other data structures (like trees) in a functional way.

Goals: Move freely through a tree, allow local modifications that can maintain the original tree while efficiently sharing structure.

Metaphor: Moving through the directory/folder tree on your computer.

THE END