Divide and Conquer

Christopher R. Genovese

Department of Statistics & Data Science

08 Oct 2024
Session #13

Plan

Recursive Thinking

Plan

Recursive Thinking

Examples

Plan

Recursive Thinking

Examples

Activity: Making Change

Announcements

- **Reading**:
  - – You could have invented Zippers
  - – Huet Zippers
  - – Zippers as Derivatives (optional)
- **Homework**: `monoidal-folds` or `zippers` due Tuesday 22 Oct.
- Rough schedule added to the syllabus on line
- `migit-2` ready imminently

Plan

Recursive Thinking

Examples

Activity: Making Change

Divide and Conquer

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. Next time, we will see *dynamic programming*. Today, we look at problems that can be solved recursively in three main steps:

1. **Divide** the problem into *smaller subproblems of the same type*.
2. **Conquer** the subproblems:
   - Small enough subproblems can be solved directly. This is called the *base case*.
   - Otherwise, solve the subproblem *recursively* – using the original algorithm. 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

Recursive Thinking

Examples

Activity: Making Change

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

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

Recursive Thinking

Examples

Activity: Making Change

Making Change

Problem: Given a monetary value `amount` in cents and a set of coin denominations `coins`, also in cents, generate *all* the distinct ways to make change for the former in terms of the latter.

Write this as a function `change_for` (or `change-for` or `changeFor` depending on your language conventions). Also write a function `best_change_for`, with the same arguments, that gives the way to make change with the smallest number of coins required.

The function `change_for` should return a list of vectors, where each vector contains the coins used to make change. The returned list should contain no repeats (accounting for order). The function `best_change_for` should return any single way of making change (a vector of denominations) of minimal size.

For example:

```
change_for(17, [1, 5, 10, 25])

# should return these results:
[[10, 5, 1, 1],
 [5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [10, 1, 1, 1, 1, 1, 1, 1],
 [5, 5, 1, 1, 1, 1, 1, 1, 1],
 [5, 5, 5, 1, 1]]
```

Making Change: Discussion Questions

Don't write any code for now. Discuss these questions **in order** with your group:

1. What are the "base cases", i.e. trivially small problems that we can solve directly?
2. How can I reduce this problem into simpler/smaller problems?
3. How are the subproblem results combined?
4. How can you ensure that there are no repeats in the generated list?

Generate some test ideas with a partner!

With a partner, sketch out the pseudocode for a recursive function that solves the problem.

The End