Medians

Definition

Suppose we have a multi-set: \( \{ x_1 \leq x_2 \leq \cdots \leq x_n \} \).

The median of a finite multi-set \( S \) is an element \( v \) splits the multiset into two equal-sized (±1) halves:

- one with elements \( \leq \) the median, and
- the other with elements \( \geq \) the median.

- The median is often a better estimator than the average.

  *Bill Gates walks into a bar.*

  *He thus raises the average salary in the bar by several billion dollars.*

  *However, the median salary is not likely to move that much.*

- How to compute: Sort the input, take the middle element. This takes \( \Theta(n \log n) \) time.

  Can we do better?

Selection, 1

The Selection Problem

**Input:** A list of \( n \)-many numbers, \( S \), and \( k \in \{1, \ldots, n\} \).

**Output:** The \( k \)th smallest element of \( S \).

- For \( k = 1 \), this gives the minimum.
- For \( k = n \), this gives the maximum.
- For \( k = \lfloor n/2 \rfloor \), this gives the median.

**function** selection(\( S, k \)) // returns \( k \)-th smallest elm of \( S \)

Pick \( v \) randomly from \( S \).

Split \( S \) into three lists:

- \( S_L \leftarrow \) the elements of \( S \) that are \(< v \)
- \( S_v \leftarrow \) the elements of \( S \) that are \( = v \)
- \( S_R \leftarrow \) the elements of \( S \) that are \( > v \)

if \( k \leq |S_L| \) then return selection(\( S_L, k \))

else if \( |S_L| < k \leq |S_L| + |S_v| \) then return \( v \)

else return selection(\( S_R, k - |S_L| - |S_v| \))

Best case: If we are lucky \( |S_L| \approx |S_R| \approx |S|/2 \).

So we have the recurrence \( T(n) = T(n/2) + O(n) \).

Hence, \( T(n) \in O(n) \). (Why?)

Worst case: If we are unlucky, \( S \) gets smaller by 1 each time.

Hence, \( T(n) \in O(n^2) \). (How?)

Average case: ???

Selection, 2

**function** selection(\( S, k \)) // returns \( k \)-th smallest elm of \( S \)

Pick \( v \) randomly from \( S \).

Split \( S \) into three lists:

- \( S_L, S_v, S_R \leftarrow \) the elements of \( S \) that are \(<, =, > v \)

if \( k \leq |S_L| \) then return selection(\( S_L, k \))

else if \( |S_L| < k \leq |S_L| + |S_v| \) then return \( v \)

else return selection(\( S_R, k - |S_L| - |S_v| \))

Best case: If we are lucky \( |S_L| \approx |S_R| \approx |S|/2 \).

So we have the recurrence \( T(n) = T(n/2) + O(n) \).

Hence, \( T(n) \in O(n) \). (Why?)

Worst case: If we are unlucky, \( S \) gets smaller by 1 each time. (How?)

Hence, \( T(n) \in O(n^2) \). (Why?)

Average case: ???
Selection, 3: Average Case Analysis

**Function**

```c
function selection(S, k) // returns k-th smallest elm of S
    Pick v randomly from S.
    (S_L, S_v, S_R) ← the elements of S that are (<, =, >) v
    if k ≤ |S_L| then return selection(S_L, k)
    else if |S_L| < k ≤ |S_L| + |S_v| then return v
    else return selection(S_R, k − |S_L| − |S_v|)
```

**Definition**

v is **good** when it is within the 25th and 75th percentile of the array’s values. (A random v has a 50% chance of being good.)

**Lemma**

On average a fair coin needs two flips before a head is seen. Proof on board

---

Selection, 4: Average Case Analysis Continued

Time taken on an size-n array

\[ = (\text{time on an } (3n/4)-\text{size array}) + (\text{time to reduce to an } (3n/4)-\text{size array}) \]

- Let \( T(n) = \) the expected time for a size-\( n \) array.
- **Fact:** Expectation \[ \sum iX_i = \sum i \text{ Expectation}[X_i] \]
- So \( T(n) = T(3n/2) + O(n) \).

\( a = 1, b = 2/3, d = 1 > 0 = \log_{2/3} 1. \)

\[ \therefore T(n) \in O(n) \]

---

Multiplying Polynomials

\[
(a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d) \cdot (b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d}
\]

where \( c_i = a_0 \cdot b_i + a_1 \cdot b_{i-1} + \cdots + a_i \cdot b_0 \) and, for \( i > d, a_i = b_i = 0 \).

- Using the above formula, computing the \( c_i \)'s from the \( a_i \)'s and \( b_i \)'s takes \( 1 + 2 + 3 + \cdots + d + (d + 1) + d + \cdots + 2 + 1 = (d + 1)^2 \) many multiplies.
- Why count multiplies?
- Can we do better? (Yes)
- Why do we care?

---

Representing Polynomials

Suppose \( A(\cdot) \) is a \( d \)-degree polynomial function.

**Coefficient representation**

\( A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d \).

**Value representation**

\( A \) is determined by its value on \( (d + 1) \)-many distinct points.

\( A(p_0), \ldots, A(p_d) \).

Both representations have their advantages/disadvantages.
Multiplying Polynomials, again

- Suppose $A(\cdot)$ and $B(\cdot)$ are $d$-degree polynomial functions.
- Suppose $p_0, \ldots, p_{2d}$ are $(2d + 1)$-many distinct points.
- Then $A(p_0) \cdot B(p_0), \ldots, A(p_{2d}) \cdot B(p_{2d})$ completely determines $C = A \cdot B$.
- This takes $(2d + 1)$-many multiplies.

As compared to the $(d + 1)^2$-many multiplies for the coefficient representation.

Evaluating Polynomials by Horner’s Rule

$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_d x^d$

\[= a_0 + x(a_1 + a_2 x + a_3 x^2 + \cdots + a_d x^{d-1})\]

\[= a_0 + x(a_1 + x(a_2 + a_3 x + \cdots + a_d x^{d-2}))\]

\[= a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + a_d x^{d-3})))\]

\[\vdots\]

function horner([a_0, \ldots, a_d], v)  
  // Compute: $a_0 + a_1 v + a_2 v^2 + \cdots + a_d v^d$, using $d$-many multiplies  
  val ← a_d  
  for $i \leftarrow d - 1, \ldots, 0$ do  
    val ← $a_i + v \cdot$val  
  return val

An Idea for Divide-and-Conquer Evaluation

- We pick $\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$ as the evaluation points.
- Since $x_i^2 = (-x_i)^2$, the computations for $A(x_i)$ and $A(-x_i)$ have lots of overlap. That is: (Suppose $d$ is even.)

\[A(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_d x_i^d = A_e(x_i^2) + x_i A_o(x_i^2).\]

\[A(-x_i) = a_0 - a_1 x_i + a_2 x_i^2 + \cdots + a_d x_i^d = A_e(x_i^2) - x_i A_o(x_i^2).\]

\[A_e(z) = a_0 + a_2 z + a_4 z^2 + \cdots + a_d z^{d/2}.\]

\[A_o(z) = a_1 + a_3 z + a_5 z^2 + \cdots + a_{d-1} z^{d/2-1}.\]

A size-$d$ problem $\rightarrow$ two size-$\frac{d}{2}$ problems

Q: Where to find more $\pm$ pairs?
A: $C$, the complex numbers.

Changing representations

Coefﬁcient representation $a_0, \ldots, a_d$

Evaluation $A(p_0), \ldots, A(p_d)$

Value representation

Interpolation $A(p_0), \ldots, A(p_d)$

Idea for Polynomial-multiplication

// Input: the coefficients of two $d$-degree polynomials $A(\cdot)$ and $B(\cdot)$.
// Output: The coefficients of $C$ where $C = A \times B$.

Selection: Pick some points $x_0, \ldots, x_{n-1}$ where $n \geq 2d + 1$.

Evaluation: Compute $A(x_i)$ and $B(x_i)$ for $i = 0, \ldots, n-1$.

Multiplication: Compute $C(x_i) = A(x_i) \cdot B(x_i)$ for $i = 0, \ldots, n-1$.

Interpolation: Recover $c_0, \ldots, c_{2d}$, the coefficients of $C(\cdot)$.

Naïvely, evaluation takes $O(n^2)$-time. We can do better.
Review: Complex Numbers

- $z = a + ib$ is a complex number, where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}$.
- $z$ has co-ordinates $(a, b)$ in the complex plane.
- $z$ can also be written as $r(\cos \theta + i \sin \theta) = re^{i\theta}$, where $r = \sqrt{a^2 + b^2}$.
- $\cos \theta = a/r$, $\sin \theta = b/r$, and $0 \leq \theta < 2\pi$.
- $z$ has polar co-ordinates $(r, \theta)$.
- Multiplying in polar co-ordinates is easy: $(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 \cdot r_2, (\theta_1 + \theta_2) \mod (2\pi))$.
- Also $-z = (r, (\theta + \pi) \mod (2\pi))$ for $z = (r, \theta)$.
- $z^* = a - ib = re^{-i\theta}$ is the complex conjugate of $z$.

Review: Roots of Unity, 1

- An $n$-th root of unity is a complex number $z$ such that $z^n = 1$.
- A primitive $n$-th root of unity is a $n$-th root of unity $z$ such that $z^k \neq 1$ for $k = 1, \ldots, n - 1$.
- $e^{\frac{2\pi i}{n}}$ (for $i = 0, 1, \ldots, n - 1$) are the $n$-th roots of unity.
- For $k = 1, \ldots, n - 1$, $(e^{\frac{2\pi i}{n}})^k \neq 1$.

The Fast Fourier Transform Algorithm

The values = FFT(the coefficients, $\omega$)

function FFT($A, \omega$)

// Input: The co-efficients of a polynomial $A$ and an $n$th root of unity where $n$ is a power of 2 and $d < n$.
// Output: [$A(\omega), \ldots, A(\omega^{n-1})$]  Runtime: $O(n \log n)$ (Why?)

if $\omega = 1$ then return [$A(1)$]

Express $A(x)$ as $A_0(x^2) + xA_1(x^2)$

Call FFT($A_0$, $\omega^2$) to evaluate $A_0$ at even powers of $\omega$

Call FFT($A_0$, $\omega^2$) to evaluate $A_0$ at even powers of $\omega$

for $j = 0 \rightarrow n - 1$

compute $A(\omega^j) \leftarrow A_0(\omega^{2j}) + \omega^j A_1(\omega^{2j})$

compute $A(\omega^{j+n/2}) \leftarrow A_0(\omega^{2j}) - \omega^j A_1(\omega^{2j})$

return [$A(\omega), \ldots, A(\omega^{n-1})$]

Amazing Fact

the coefficients = $\frac{1}{n}$ FFT(the values, $\omega^{-1}$)
Evaluation as a Matrix Multiplication

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

The matrix, \( M \), is a Vandermonde matrix.

**Theorem**

If \( x_0, \ldots, x_{n-1} \) are distinct, \( M \) is invertible.

Evaluation is multiplication by \( M \).

Interpolation is multiplication by \( M^{-1} \).

### The FFT Algorithm in Matrix Terms, 1

\[
M_n(\omega) = \begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{j-1} & \omega^{2j} & \cdots & \omega^{j(n-1)} \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\]

**Lemma**

The columns of \( M_n(\omega) \) are orthogonal.

Note: \((\omega^m)^* = \omega^{-m}\).

**Proof.**

\[
\begin{align*}
[1, \omega, \omega^2, \ldots, \omega^{(n-1)}] \cdot [1, \omega^k, \omega^{2k}, \ldots, \omega^{k(n-1)}] &= 1 + \omega + \omega^2 + \cdots + \omega^{n-1} \\
&= 1 + \omega + \omega^2 + \cdots + \omega^{n-1} \\
&= (1 - \omega^{n-1}) / (1 - \omega).
\end{align*}
\]

### The FFT Algorithm in Matrix Terms, 2

\[
M_n(\omega) = \begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{j-1} & \omega^{2j} & \cdots & \omega^{j(n-1)} \\
1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{bmatrix}
\]

### The FFT Algorithm in Matrix Terms, 3

#### The first step of D&C in matrix terms

![Figure from DVP](image)

- **Row 2**
- **Column 2k**
- **2k + 1**
- Even columns
- Odd columns

- **Row**
- **Column 2k**
- **2k + 1**
- Even columns
- Odd columns

**Corollary**

(a) \( M_n(\omega) \cdot (M_m(\omega))^* = n \cdot I \).

(b) \((M_n(\omega))^* = M_n(\omega^{-1})\).

(c) \((M_n(\omega))^{-1} = 1/n \cdot M_n(\omega^{-1})\).
The FFT Algorithm in Matrix Terms, 4

function FFT(a, ω)
// Input: An array $a = (a_0, \ldots, a_{n-1})$ for $n$ a power of 2 and
// $ω$, a primitive $n$th root of unity
// Output: $M_n(ω)a$
// Runtime: $O(n \log n)$ (Why?)
if $ω = 1$ then return $a$
$(s_0, s_1, \ldots, s_{n/2-1}) \leftarrow$ FFT$((a_0, a_2, \ldots, a_{n-2}), ω^2)$$
(s_0', s_1', \ldots, s_{n/2-1}') \leftarrow$ FFT$((a_1, a_3, \ldots, a_{n-1}), ω^2)$
for $j \leftarrow 0$ to $n/2 - 1$ do
\hspace*{1em}$r_j \leftarrow s_j + ω^j s'_j$
\hspace*{1em}$r_{j+n/2} \leftarrow s_j - ω^j s'_j$
return $(r_0, r_1, \ldots, r_{n-1})$

The basic step of the for-loop has a particular form:

$r_j \leftarrow s_j + ω^j s'_j$
$r_{j+n/2} \leftarrow s_j - ω^j s'_j$

The Butterfly

All the real work of the FFT algorithm happens in:

$r_j \leftarrow s_j + ω^j s'_j$
$r_{j+n/2} \leftarrow s_j - ω^j s'_j$


Where: $x_0 \equiv s_j, x_1 \equiv s'_j, y_0 \equiv r_j,$
and $y_1 \equiv r_{j+n/2}$.

Unfolding the FFT recursion, 1

Diagram from: http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html

Unfolding the FFT recursion, 2

Diagram from: http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html
Unfolding the FFT recursion, 3

Diagram from: DPV

Facts about the FFT/Butterfly Circuit

1. For \( n \) inputs:
   - \( \log_2 n \)-many levels,
   - each level has \( n \)-many nodes,
   - a total of \( n \log_2 n \) many nodes

<table>
<thead>
<tr>
<th>Input order_{10}</th>
<th>0</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>1</th>
<th>5</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output order_{10}</td>
<td>000</td>
<td>100</td>
<td>010</td>
<td>110</td>
<td>001</td>
<td>101</td>
<td>011</td>
<td>111</td>
</tr>
</tbody>
</table>

2. There is a unique path from any input \( a_j \) to each output \( A(\omega^k) \):
   - follow the binary representation of the input.

3. On the path from \( a_j \) to \( A(\omega^k) \) the labels add up to \( jk \mod n \).

4. The circuit shows up in parallel computing and is often directly implemented in hardware.

Next time: Graphs, graphs, and more graphs.