CSE 421

Divide and Conquer: Median Approximation Algorithms

Shayan Oveis Gharan
Integer Multiplication
Integer Arithmetic

Add: Given two n-bit integers $a$ and $b$, compute $a + b$.

\[ \begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{array} \]

$O(n)$ bit operations.

Multiply: Given two n-bit integers $a$ and $b$, compute $a \times b$.

The “grade school” method:

$O(n^2)$ bit operations.
How to use Divide and Conquer?

Suppose we want to multiply two 2-digit integers (32, 45). We can do this by multiplying four 1-digit integers. Then, use add/shift to obtain the result:

\[
x = 10x_1 + x_0 \\
y = 10y_1 + y_0 \\
xy = (10x_1 + x_0)(10y_1 + y_0)
\]

\[
= 100 x_1 y_1 + 10(x_1 y_0 + x_0 y_1) + x_0 y_0
\]

Same idea works when multiplying n-digit integers:

• Divide into 4 n/2-digit integers.
• Recursively multiply
• Then merge solutions
A Divide and Conquer for Integer Mult

Let \( x, y \) be two \( n \)-bit integers

Write \( x = 2^{n/2} x_1 + x_0 \) and \( y = 2^{n/2} y_1 + y_0 \)
where \( x_0, x_1, y_0, y_1 \) are all \( n/2 \)-bit integers.

\[
x = 2^{n/2} \cdot x_1 + x_0 \\
y = 2^{n/2} \cdot y_1 + y_0 \\
xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) \\
\quad = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0
\]

Therefore,

\[
T(n) = 4T \left( \frac{n}{2} \right) + \Theta(n)
\]

So,

\[
T(n) = \Theta(n^2).
\]

We only need 3 values \( x_1 y_1, x_0 y_0, x_1 y_0 + x_0 y_1 \)
Can we find all 3 by only 3 multiplication?
Key Trick: 4 multiplies at the price of 3

\[ x = 2^{n/2} \cdot x_1 + x_0 \]
\[ y = 2^{n/2} \cdot y_1 + y_0 \]
\[ xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) \]
\[ = 2^n \cdot x_1 y_1 + 2^{n/2} (x_1 y_0 + x_0 y_1) + x_0 y_0 \]

\[ \alpha = x_1 + x_0 \]
\[ \beta = y_1 + y_0 \]
\[ \alpha \beta = (x_1 + x_0)(y_1 + y_0) \]
\[ = x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 \]
\[ (x_1 y_0 + x_0 y_1) = \alpha \beta - x_1 y_1 - x_0 y_0 \]
Key Trick: 4 multiplies at the price of 3

Theorem [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585...})$ bit operations.

$$x = 2^{n/2} \cdot x_1 + x_0 \Rightarrow \alpha = x_1 + x_0$$
$$y = 2^{n/2} \cdot y_1 + y_0 \Rightarrow \beta = y_1 + y_0$$
$$xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0)$$
$$= 2^n \cdot x_1y_1 + 2^{n/2} \cdot (x_1y_0 + x_0y_1) + x_0y_0$$

To multiply two n-bit integers:
- Add two $n/2$ bit integers.
- Multiply three $n/2$-bit integers.
- Add, subtract, and shift $n/2$-bit integers to obtain result.

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O\left(n^{\log_2 3}\right) = O(n^{1.585...})$$
Integer Multiplication (Summary)

• Naïve: \( \Theta(n^2) \)

• Karatsuba: \( \Theta(n^{1.585 \ldots}) \)

• Amusing exercise: generalize Karatsuba to do 5 size \( n/3 \) subproblems
  This gives \( \Theta(n^{1.46 \ldots}) \) time algorithm

• Best known algorithm runs in \( \Theta(n \log n) \) using fast Fourier transform
  but mostly unused in practice (unless you need really big numbers - a billion digits of \( \pi \), say)

• Best lower bound \( O(n) \): A fundamental open problem
Median
Selecting k-th smallest

**Problem:** Given numbers $x_1, ..., x_n$ and an integer $1 \leq k \leq n$, output the $k$-th smallest number

$$\text{Sel}(\{x_1, ..., x_n\}, k)$$

A simple algorithm: Sort the numbers in time $O(n \log n)$ then return the $k$-th smallest in the array.

Can we do better?

Yes, in time $O(n)$ if $k = 1$ or $k = 2$.

Can we do $O(n)$ for all possible values of $k$?

Assume all numbers are distinct for simplicity.
An Idea

Choose a number $w$ from $x_1, \ldots, x_n$

Define
\[
\begin{align*}
S_<(w) &= \{x_i : x_i < w\} \\
S_\&(w) &= \{x_i : x_i = w\} \\
S_>(w) &= \{x_i : x_i > w\}
\end{align*}
\]

Can be computed in linear time

Solve the problem recursively as follows:
\[
\begin{align*}
&\text{If } k \leq |S_<(w)|, \text{ output } Sel(S_<(w), k) \\
&\text{Else if } k \leq |S_<(w)| + |S_\&(w)|, \text{ output } w \\
&\text{Else output } Sel(S_>(w), k - |S_<(w)| - |S_\&(w)|)
\end{align*}
\]

Ideally want $|S_<(w)|, |S_>(w)| \leq n/2$. In this case ALG runs in $O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + \cdots + O(1) = O(n)$. 

How to choose w?

Suppose we choose w uniformly at random similar to the pivot in quicksort.
Then, \( \mathbb{E}[|S_<(w)|] = \mathbb{E}[|S_>(w)|] = n/2 \). Algorithm runs in \( O(n) \) in expectation.

Can we get \( O(n) \) running time deterministically?
- Partition numbers into sets of size 3.
- Sort each set (takes \( O(n) \))
- \( w = Sel(midpoints, n/6) \)
How to lower bound $|S_{<}(w)|, |S_{>}(w)|$?

- $|S_{<}(w)| \geq 2 \left( \frac{n}{6} \right) = \frac{n}{3}$
- $|S_{>}(w)| \geq 2 \left( \frac{n}{6} \right) = \frac{n}{3}$.

So, what is the running time?

\[ n \frac{3}{3} \leq |S_{<}(w)|, |S_{>}(w)| \leq \frac{2n}{3} \]
Asymptotic Running Time?

- If $k \leq |S_<(w)|$, output $Sel(S_<(w), k)$
- Else if $k \leq |S_<(w)| + |S_=(w)|$, output $w$
- Else output $Sel(S_>(w), k - S_<(w) - S_=(w))$

Where $\frac{n}{3} \leq |S_<(w)|, |S_>(w)| \leq \frac{2n}{3}$

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n) \Rightarrow T(n) = O(n \log n)$$
**An Improved Idea**

Partition into \( \frac{n}{5} \) sets. Sort each set and set \( w = \text{Sel}(\text{midpoints}, \frac{n}{10}) \)

- \( |S_{<}(w)| \geq 3 \left( \frac{n}{10} \right) = \frac{3n}{10} \)
- \( |S_{>}(w)| \geq 3 \left( \frac{n}{10} \right) = \frac{3n}{10} \)

\[
T(n) = T \left( \frac{n}{5} \right) + T \left( \frac{7n}{10} \right) + O(n) \Rightarrow T(n) = O(n)
\]
An Improved Idea

\[
\text{Sel}(S, k) \{ \\
n \leftarrow |S| \\
\text{If } (n < ??) \text{ return } ?? \\
\text{Partition } S \text{ into } n/5 \text{ sets of size 5} \\
\text{Sort each set of size 5 and let } M \text{ be the set of medians, so } |M|=n/5 \\
\text{Let } w=\text{Sel}(M,n/10) \\
\text{For } i=1 \text{ to } n\{ \\
\text{If } x_i < w \text{ add } x \text{ to } S_<(w) \\
\text{If } x_i > w \text{ add } x \text{ to } S_>(w) \\
\text{If } x_i = w \text{ add } x \text{ to } S_=w) \\
\} \\
\text{If } (k \leq |S_<(w)|) \\
\text{return } \text{Sel}(S_<(w),k) \\
\text{else if } (k \leq |S_<(w)| + |S_=w|) \\
\text{return } w; \\
\text{else} \\
\text{return } \text{Sel}(S_>(w),k - |S_<(w)| - |S_=w|) \\
\}
\]

We can maintain each set in an array
D&C Summary

Idea:

“Two halves are better than a whole”
  • if the base algorithm has super-linear complexity.

“If a little's good, then more's better”
  • repeat above, recursively

• Applications: Many.
  • Binary Search, Merge Sort, (Quicksort),
  • Root of a Function
  • Closest points,
  • Integer multiplication
  • Median
Approximation Algorithms
How to deal with NP-complete Problem

Many of the important problems in real world are NP-complete.  
SAT, Set Cover, Graph Coloring, TSP, Max IND Set,  
Vertex Cover, …

So, we cannot find optimum solutions in polynomial time.  
What to do instead?

• Find optimum solution of special cases (e.g., random inputs)

• Find near optimum solution in the worst case
Approximation Algorithm

Polynomial-time Algorithms with a guaranteed approximation ratio.

\[ \alpha = \frac{\text{Cost of computed solution}}{\text{Cost of the optimum}} \]

worst case over all instances.

Goal: For each NP-hard problem find an approximation algorithm with the best possible approximation ratio.
Vertex Cover

Given a graph $G=(V,E)$, Find smallest set of vertices touching every edge
Greedy algorithms are typically used in practice to find a (good) solution to NP-hard problems

**Strategy (1):** Iteratively, include a vertex that covers most new edges

Q: Does this give an optimum solution?
A: No,
Greedy (1): Pick vertex that covers the most
Greedy (1): Pick vertex that covers the most
Greedy (1): Pick vertex that covers the most
Greedy (1): Pick vertex that covers the most $\mathcal{B}$
Greedy (1): Pick vertex that covers the most

\[ B \]
Greedy (1): Pick vertex that covers the most
Greedy (1): Pick vertex that covers the most
Greedy (1): Pick vertex that covers the most
Greedy (1): Pick vertex that covers the most

\[ \mathcal{B} \]
Greedy (1): Pick vertex that covers the most

Greedy Vertex cover = 20
OPT Vertex cover = 8
Greedy (1): Pick vertex that covers the most

$n$ vertices. Each vertex has one edge into each $B_i$

Each vertex in $B_i$ has $i$ edges to top

Greedy pick bottom vertices = $n + \frac{n}{2} + \frac{n}{3} + \cdots + 1 \approx n \ln n$

OPT pick top vertices = $n$
A Different Greedy Rule

Greedy 2: Iteratively, pick both endpoints of an uncovered edge.

Vertex cover = 6
Greedy 2: Pick Both endpoints of an uncovered edge

Greedy vertex cover = 16

OPT vertex cover = 8
**Greedy (2) gives 2-approximation**

**Thm**: Size of greedy (2) vertex cover is at most twice as big as size of optimal cover.

**Pf**: Suppose Greedy (2) picks endpoints of edges $e_1, ..., e_k$. Since these edges do not touch, every valid cover must pick one vertex from each of these edges! i.e., $OPT \geq k$.

But the size of greedy cover is $2k$. So, Greedy is a 2-approximation.