CS 341: ALGORITHMS

Lecture 5: divide & conquer II

Readings: see website

Trevor Brown (co-taught with Anna Lubiw)
https://www.student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca
**PROBLEM: NON-DOMINATED POINTS**

- A point **dominates** everything to the **southwest**

  So, I am a non-dominated point

  No other point dominates me
MORE FORMALLY

- Given two points \((x_1, y_1)\) and \((x_2, y_2)\), we say \((x_1, y_1)\) dominates \((x_2, y_2)\) if \(x_1 > x_2\) and \(y_1 > y_2\).
- Input: a set \(S\) of \(n\) points,
- Output: all non-dominated points in \(S\), i.e., all points in \(S\) that are not dominated by any point in \(S\)

What’s an easy (brute force) algorithm for this?
BRUTE FORCE ALGORITHM

```plaintext
NDPoints(S)
for p in S
    dominated[p] = false
for q in S
    if q != p and q.x > p.x and q.y > p.y
        dominated[p] = true
if not dominated[p]
    print p
```

Running time? $O(n^2)$ Let's come up with a better algorithm
Observe that the non-dominated points form a **staircase** and all the other points are “under” this staircase.

The **treads** of the staircase are determined by the \( y \)-co-ordinates of the non-dominated points. The **risers** of the staircase are determined by the \( x \)-co-ordinates of the non-dominated points. The staircase descends from left to right.
Suppose we pre-sort the points in $S$ with respect to their $x$-co-ordinates. This takes time $\Theta(n \log n)$. 
PROBLEM DECOMPOSITION

**Divide:** Let the first $n/2$ points be denoted $S_1$ and let the last $n/2$ points be denoted $S_2$. 
PROBLEM DECOMPOSITION

Conquer: Recursively solve the subproblems defined by the two instances $S_1$ and $S_2$. 
PROBLEM DECOMPOSITION

Combine: Given the non-dominated points in $S_1$ and the non-dominated points in $S_2$, how do we find the non-dominated points in $S$?

Observe that no point in $S_1$ dominates a point in $S_2$.

Therefore we only need to eliminate the points in $S_1$ that are dominated by a point in $S_2$. It turns out that this can be done in time $O(n)$. 
COMBINING TO GET NON-DOMINATED POINTS

- Let $Q_1, Q_2, \ldots, Q_k$ be the non-dominated points in $S_1$
- Let $R_1, R_2, \ldots, R_m$ be the non-dominated points in $S_2$

Just need to find rightmost $Q_i$ that is not dominated (that has $y$-coordinate $\geq R_1.y$)
NDPoints(S[1..n])
    sort S by x-coord
    recurse(S)

Recurse(S[1..n]) // precondition: S sorted by x
    // base case
    if n == 1 then return S

    // divide
    S1 = S[1..\text{floor}(n/2)]
    S2 = S[\text{floor}(n/2)+1..n]

    // conquer
    Q[1..q] = Recurse(S1)
    R[1..r] = Recurse(S2)

    // combine
    i = 1
    while i <= q and Q[i].y > R[1].y
        i++

    // postcondition: return sorted by x
    return concat(Q[1..i-1], R)
Running time complexity?

NDPoints(S[1..n])
  sort S by x-coord
  recurse(S)

Recurse(S[1..n]) // precondition: S sorted by x
  // base case
  if n == 1 then return S

  // divide
  S1 = S[1..floor(n/2)]
  S2 = S[floor(n/2)+1..n]

  // conquer
  Q[1..q] = Recurse(S1)
  R[1..r] = Recurse(S2)

  // combine
  i = 1
  while i <= q and Q[i].y > R[1].y
    i++

  // postcondition: return sorted by x
  return concat(Q[1..i-1], R)

θ(1) or θ(n) depending on data structures...
either way doesn’t matter

Assume $n = 2^j$ for simplicity

$T(n) = 2T\left(\frac{n}{2}\right) + Θ(n)$

Same as merge sort recurrence: $Θ(n \log n)$

So total for sort & recursion is

$Θ(n \log n + T(n)) = Θ(n \log n)$
MULTIPRECISION MULTIPLICATION

- Input: two \textbf{k-bit} positive integers \(X\) and \(Y\)
  - With binary representations:
    \[
    X = [X[k-1], \ldots, X[0]] \\
    Y = [Y[k-1], \ldots, Y[0]]
    \]
- Output: The \(2k\)-bit positive integer \(Z = XY\)
  - With binary representation: \(Z = [Z[2k-1], \ldots, Z[0]]\)

Here, we are interested in the \textbf{bit complexity} of algorithms that solve \textbf{Multiprecision Multiplication}, which means that the complexity is expressed as a function of \(k\) (the size of the problem instance is \(2k\) bits).
**BRUTE FORCE ALGORITHM**

- One row per digit of $Y$
- For each row, copy the $k$ bits of $X$
- Add the $k$ rows together
  - $\Theta(k)$ binary additions of $\Theta(k)$ bit numbers
- Total runtime is $\Theta(k^2)$ bit operations

$$
\begin{array}{cccc}
X & 1 & 0 & 1 & 0 \\
\hline
Y & 1 & 0 & 1 & 1 \\
\hline
\hline
1 & 0 & 1 & 1 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
& 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
& 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
& 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
Z & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\end{array}
$$
A DIVIDE-AND-CONQUER APPROACH

Let $X_L$ be the integer formed by the $k/2$ high-order bits of $X$ and let $X_R$ be the integer formed by the $k/2$ low-order bits of $X$. Similarly for $Y$.

Thus

\[ X = 2^{k/2} X_L + X_R \quad \text{and} \quad Y = 2^{k/2} Y_L + Y_R. \]
EXPRESSING $k$-BIT MULT. AS $k/2$-BIT MULT.

- $X = 2^{k/2}X_L + X_R$ and $Y = 2^{k/2}Y_L + Y_R$
- So $XY = (2^{k/2}X_L + X_R)(2^{k/2}Y_L + Y_R)$
- $= 2^kX_LY_L + 2^{k/2}(X_LY_R + X_RY_L) + X_RY_R$
- Suggests a D&C approach...
  - **Divide** into four $k/2$-bit multiplication subproblems
  - **Conquer** with recursive calls
  - **Combine** with $k$-bit addition and bit shifting
Recall: $XY = 2^k X_L Y_L + 2^{k/2}(X_L Y_R + X_R Y_L) + X_R Y_R$
DnCMultiply(X, Y, k)

// base case
if k == 1 then return [[X[0]*Y[0]]]

// divide
XR = X[0..k/2-1]
XL = X[k/2..k-1]
YR = Y[0..k/2-1]
YL = Y[k/2..k-1]

// conquer
XLYL = DnCMultiply(XL, YL, k/2)
XRYR = DnCMultiply(XR, YR, k/2)
XLYR = DnCMultiply(XL, YR, k/2)
XRYL = DnCMultiply(XR, YL, k/2)

// combine
return (XLYL<<k) + (XLYR+XRYL)<<(k/2) + XRYR

- Assume $k = 2^j$ for ease
- $T(k) = 4T\left(\frac{k}{2}\right) + \Theta(k)$
- Master theorem says $T(k) \in \Theta(k^{\log_2 4}) = \Theta(k^2)$

Same complexity as brute force!
KARATSUBA’S ALGORITHM

- Let’s optimize from four subproblems to three

Recall: \( XY = 2^k X_L Y_L + 2^{k/2} (X_L Y_R + X_R Y_L) + X_R Y_R \)

- Idea: compute \( X_L Y_R + X_R Y_L \) with only one multiplication

- Note \( X_L Y_R + X_R Y_L \) appears in \((X_L + X_R)(Y_L + Y_R)\)

- \((X_L + X_R)(Y_L + Y_R) = X_L Y_L + X_L Y_R + X_R Y_L + X_R Y_R\)

- Let \( X_T = X_L + X_R \) and \( Y_T = Y_L + Y_R \)

- Then \( X_L Y_R + X_R Y_L = X_T Y_T - X_L Y_L - X_R Y_R \)

- And the other two terms \( X_L Y_L \) and \( X_R Y_R \) are already in \( XY \)

- So \( XY = 2^k X_L Y_L + 2^{k/2} (X_T Y_T - X_L Y_L - X_R Y_R) + X_R Y_R \)

Only three unique multiplications!
KaratsubaMultiply(X, Y, k)

// base case
if k == 1 then return [[X[0]*Y[0]]]

// divide
XR = X[0..k/2-1]
XL = X[k/2..k-1]
YR = Y[0..k/2-1]
YL = Y[k/2..k-1]
XT = XL + XR
YT = YL + YR

// conquer
XLYL = KaratsubaMultiply(XL, YL, k/2)
XRYR = KaratsubaMultiply(XR, YR, k/2)
XTYT = KaratsubaMultiply(XT, YT, k/2)

// combine
return (XLYL<<k) + (((XTYT - XLYL - XRYR)<<((k/2))) + XRYR

Input size increase by 10x causes runtime to 38x

Compare to $\Theta(k^2)$ algo:
- 10x input causes 100x time

Running time complexity?

- $T(k) = 3T\left(\frac{k}{2}\right) + \Theta(k)$
- Assume $k = 2^j$ for ease
- Master theorem:
  - $a = 3, b = 2, y = 1$
  - $x = \log_b a = \log_2 3$
  - $T(k) \in \Theta(k^{\log_2 3})$
  - $\approx \Theta(k^{1.58})$
Note that $X_L + X_R$ and $Y_L + Y_R$ could be $(k/2 + 1)$-bit integers. However, computation of $Z_3$ can be accomplished by multiplying $(k/2)$-bit integers and accounting for carries by extra additions.

Various techniques can be used to handle the case when $k$ is not a power of two. One possible solution is to pad with zeroes on the left. So let $m$ be the smallest power of two that is $\geq k$. The complexity is $\Theta(m^{\log_2 3})$. Since $m < 2k$ the complexity is $O((2k)^{\log_2 3}) = O(3k^{\log_2 3}) = O(k^{\log_2 3})$.

There are further improvements known:

- The **Toom-Cook algorithm** splits $X$ and $Y$ into three equal parts and uses five multiplications of $(k/3)$-bit integers. The recurrence is $T(k) = 5T(k/3) + \Theta(k)$, and then $T(k) \in \Theta(k^{\log_3 5}) = \Theta(k^{1.47})$.

- The 1971 **Schonhage-Strassen algorithm** (based on FFT) has complexity $O(n \log n \log \log n)$.

- The 2007 **Furer algorithm** has complexity $O(n \log n 2^{O(\log^* n)})$. 
MATRIX MULTIPLICATION

- **Input:** \( A \) and \( B \)
- **Output:** their product \( C = AB \)
- **Word-RAM model** (e.g., 64-bit int)
- Naïve algorithm for \( n \times n \) matrices:
  - For each output cell \( C_{ij} \)
  \[
  C_{ij} = \text{DotProd}(\text{row}_i(A), \text{col}_j(B^T))
  \]
  \[
  = \sum_{k=1}^{n} A_{ik} B_{kj}
  \]
- Running time?
ATTEMPTING A BETTER SOLUTION

- What if we first **partition** the matrix into **sub-matrices**
- Then **divide and conquer** on the **sub-matrices**
- Example of partitioning: 4x4 matrix into four 2x2 matrices

\[
\begin{bmatrix}
  a & b \\
  c & d \\
\end{bmatrix} =
\begin{bmatrix}
  a_{11} & a_{12} & b_{11} & b_{12} \\
  a_{21} & a_{22} & b_{21} & b_{22} \\
  c_{11} & c_{12} & d_{11} & d_{12} \\
  c_{21} & c_{22} & d_{21} & d_{22} \\
\end{bmatrix}
\]
MULTIPLYING PARTITIONED MATRICES

Let $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix}$

Let $B = \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \\ h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$

Note $C = AB = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$ where $a, b, \ldots, h$ are matrices
IDENTIFYING SUBPROBLEMS TO SOLVE

Recall $ae$, $bg$, etc., each represent **matrix multiplication**!

Can compute $C$ using 8 matrix multiplications
Suppose $A, B$ are $n \times n$ matrices

For simplicity assume $n$ is a power of 2

Then $a, b, c, d, e, f, g, h, r, s, t, u$ are $\frac{n}{2} \times \frac{n}{2}$ matrices

So we compute $C$ with 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices
Running time complexity?

- $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$
- Master theorem
  - $a = 8, \ b = 2, \ y = 2$
  - $x = \log_2 8 = 3$
  - $x > y \text{ so } T(n) \in \Theta(n^x)$
  - $T(n) \in \Theta(n^3)$

- Same time as brute force!

Intuition: to get speedup, must reduce the number of subproblems or their size.
STRASSEN

FAST MATRIX MULTIPLICATION ALGORITHM

\[
AB = \begin{bmatrix}
a & b \\
c & d \\
\end{bmatrix}
\begin{bmatrix}
e & f \\
g & h \\
\end{bmatrix} =
\begin{bmatrix}
ae + bg & af + bh \\
ce + dg & cf + dh \\
\end{bmatrix}
= C = \begin{bmatrix}
r & s \\
t & u \\
\end{bmatrix}
\]

Define

\[
P_1 = a(f - h) \\
P_3 = (c + d)e \\
P_5 = (a + d)(e + h) \\
P_7 = (a - c)(e + f).
\]

\[
P_2 = (a + b)h \\
P_4 = d(g - e) \\
P_6 = (b - d)(g + h)
\]

Each \(P_i\) requires one multiplication

Can combine these \(P_i\) terms with +/- to compute \(r, s, t, u\)!
### STRASSEN Fast Matrix Multiplication Algorithm

\[
AB = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix} = C = \begin{bmatrix} r & s \\ t & u \end{bmatrix}
\]

**Define**

- \( P_1 = a(f - h) \)
- \( P_3 = (c + d)e \)
- \( P_5 = (a + d)(e + h) \)
- \( P_7 = (a - c)(e + f) \).

**Claim**

- \( r = P_5 + P_4 - P_2 + P_6 \)
- \( t = P_3 + P_4 \)
- \( s = P_1 + P_2 \)
- \( u = P_5 + P_1 - P_3 - P_7 \)

- As an example, according to Strassen, \( t = P_3 + P_4 \)
- Plugging in \( P_3, P_4 \), we get \( t = (c + d)e + d(g - e) \)
- This simplifies to \( t = ce + de + dg - de = ce + dg \)
StrassenMatrixMult(A, B, n)
   // base case
   if n == 1 then return [[A[0][0]*B[0][0]]
       // divide
       [a, b, c, d] = Partition(A)
       [e, f, g, h] = Partition(B)

       // conquer
       P1 = StrassenMatrixMult(a, f-h, n/2)
       P2 = StrassenMatrixMult(a+b, h, n/2)
       P3 = StrassenMatrixMult(c+d, e, n/2)
       P4 = StrassenMatrixMult(d, g-e, n/2)
       P5 = StrassenMatrixMult(a+d, e+h, n/2)
       P6 = StrassenMatrixMult(b-d, g+h, n/2)
       P7 = StrassenMatrixMult(a-c, e+f, n/2)

       // combine (with *matrix* addition)
       return [[P5+P4-P2+P6, P1+P2],
               [P3+P4, P5+P1-P3-P7]]
Running time complexity?

\[ T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) \]

- Master theorem
  - \( a = 7, b = 2, y = 2 \)
  - \( x = \log_2 7 \)
  - \( x > y \) so \( T(n) \in \Theta(n^x) \)
  - \( T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81}) \)

```python
def StrassenMatrixMult(A, B, n):
    # base case
    if n == 1:
        return [[A[0][0] * B[0][0]]]
    # divide
    [a, b, c, d] = Partition(A)
    [e, f, g, h] = Partition(B)
    # conquer
    P1 = StrassenMatrixMult(a, f-h, n//2)
    P2 = StrassenMatrixMult(a+b, h, n//2)
    P3 = StrassenMatrixMult(c+d, e, n//2)
    P4 = StrassenMatrixMult(d, g-e, n//2)
    P5 = StrassenMatrixMult(a+d, e+h, n//2)
    P6 = StrassenMatrixMult(b-d, g+h, n//2)
    P7 = StrassenMatrixMult(a-c, e+f, n//2)
    # combine (with *matrix* addition)
    return [[P5+P4-P2+P6, P1+P2],
            [P3+P4, P5+P1-P3-P7]]
```

\( \Theta(n^2) \)
Strassen’s algorithm was improved in 1990 by Coppersmith-Winograd. Their algorithm has complexity $O(n^{2.376})$. Some slight improvements have been found more recently.

<table>
<thead>
<tr>
<th></th>
<th>How much better is $\Theta(n^{2.81})$ than $\Theta(n^3)$?</th>
<th>How much better is $\Theta(n^{2.376})$ than $\Theta(n^3)$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let $n=10,000$</td>
<td>$n^{2.81} \approx 174$ billion</td>
<td>$n^{2.376} \approx 3.2$ billion</td>
</tr>
<tr>
<td>$n^3 = 1$ trillion</td>
<td>$\approx 6x$ more</td>
<td>$\approx 312x$</td>
</tr>
</tbody>
</table>

Let $n=10,000$ $n^3 = 1$ trillion ($\approx 6x$ more)