Lecture 6

Divide and conquer (cont.), master theorem, integer multiplication, maxima set

CS 161 Design and Analysis of Algorithms

Ioannis Panageas
Application: Maxima Sets

- We can visualize the various trade-offs for optimizing two-dimensional data, such as points representing hotels according to their pool size and restaurant quality, by plotting each as a two-dimensional point, \((x, y)\), where \(x\) is the pool size and \(y\) is the restaurant quality score.
- We say that such a point is a **maximum point** in a set if there is no other point, \((x', y')\), in that set such that \(x \leq x'\) and \(y \leq y'\).
- The maximum points are the best potential choices based on these two dimensions and finding all of them is the **maxima set** problem.

We can efficiently find all the maxima points by divide-and-conquer. Here the set is \{A,H,I,G,D\}. 

![Diagram showing pools and restaurants with points marked as A, H, I, G, D, B, F, E, J, C, D]
Divide-and-Conquer

Divide-and conquer is a general algorithm design paradigm:

- **Divide**: divide the input data $S$ in two or more disjoint subsets $S_1$, $S_2$, ...
- **Conquer**: solve the subproblems recursively
- **Combine**: combine the solutions for $S_1$, $S_2$, ..., into a solution for $S$

The base case for the recursion are subproblems of constant size

Analysis can be done using recurrence equations
Merge-Sort Review

Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:

- **Divide**: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
- **Conquer**: recursively sort $S_1$ and $S_2$
- **Combine**: merge $S_1$ and $S_2$ into a unique sorted sequence

---

**Algorithm** $mergeSort(S)$

**Input** sequence $S$ with $n$ elements

**Output** sequence $S$ sorted according to $C$

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

$mergeSort(S_1)$

$mergeSort(S_2)$

$S \leftarrow merge(S_1, S_2)$
Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with \( n/2 \) elements and implemented by means of a doubly linked list, takes at most \( bn \) steps, for some constant \( b \).
- Likewise, the basis case (\( n < 2 \)) will take at \( b \) most steps.
- Therefore, if we let \( T(n) \) denote the running time of merge-sort:

\[
T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}
\]

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has \( T(n) \) only on the left-hand side.
Iterative Substitution

In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

\[ T(n) = 2T(n/2) + bn \]

\[ = 2(2T(n/2^2)) + b(n/2)) + bn \]

\[ = 2^2 T(n/2^2) + 2bn \]

\[ = 2^3 T(n/2^3) + 3bn \]

\[ = 2^4 T(n/2^4) + 4bn \]

\[ = \ldots \]

\[ = 2^i T(n/2^i) + ibn \]

Note that base, \( T(n)=b \), case occurs when \( 2^i=n \). That is, \( i = \log n \).

So,

\[ T(n) = bn + bn \log n \]

Thus, \( T(n) \) is \( O(n \log n) \).
The Recursion Tree

Draw the recursion tree for the recurrence relation and look for a pattern:

\[ T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases} \]

<table>
<thead>
<tr>
<th>depth</th>
<th>T’s size</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>(bn)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>(bn)</td>
</tr>
<tr>
<td>(i)</td>
<td>(2^i)</td>
<td>(bn)</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

Total time = \(bn + bn \log n\)
(last level plus all previous levels)
Guess-and-Test Method

In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

\[ T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
\end{cases} \]

Guess: \( T(n) < cn \log n \)

Wrong: we cannot make this last line be less than \( cn \log n \)
Guess-and-Test Method, (cont.)

- Recall the recurrence equation:
  \[ T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn \log n & \text{if } n \geq 2 
  \end{cases} \]

- **Guess #2:** \( T(n) < cn \log^2 n \).

  \[
  T(n) = 2T(n/2) + bn \log n \\
  = 2(c(n/2) \log^2 (n/2)) + bn \log n \\
  = cn(\log n - \log 2)^2 + bn \log n \\
  = cn \log^2 n - 2cn \log n + cn + bn \log n \\
  \leq cn \log^2 n
  \]

  - if \( c > b \).

- So, \( T(n) \) is \( O(n \log^2 n) \).

- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.
Master Method

Many divide-and-conquer recurrence equations have the form:

\[ T(n) = \begin{cases} 
    c & \text{if } n < d \\
    aT(n/b) + f(n) & \text{if } n \geq d
\end{cases} \]

The Master Theorem:

1. if \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).
Master Method, Example 1

The form:

\[
T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases}
\]

The Master Theorem:

1. if \( f(n) = O(n^{\log_b a - \varepsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. if \( f(n) = \Theta(n^{\log_b a \log^k n}) \), then \( T(n) = \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:

\[
T(n) = 4T(n/2) + n
\]

Solution: \( \log_b a = 2 \), so case 1 says \( T(n) \) is \( O(n^2) \).
Master Method, Example 2

The form:

\[
T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d
\end{cases}
\]

The Master Theorem:

1. if \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \),
   provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:

\[
T(n) = 2T(n/2) + n \log n
\]

Solution: \( \log_b a = 1 \), so case 2 says \( T(n) \) is \( \Theta(n \log^2 n) \).
Master Method, Example 3

The form:

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d
\end{cases} \]

The Master Theorem:

1. If \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \),
   provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:

\[ T(n) = T(n/3) + n \log n \]

Solution: \( \log_b a = 0 \), so case 3 says \( T(n) \) is \( O(n \log n) \).
Master Method, Example 4

The form:
\[ T(n) = \begin{cases} 
    c & \text{if } n < d \\
    aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:

1. if \( f(n) \) is \( O(n^\log_b a - \varepsilon) \), then \( T(n) \) is \( \Theta(n^\log_b a) \)
2. if \( f(n) \) is \( \Theta(n^\log_b a \log^k n) \), then \( T(n) \) is \( \Theta(n^\log_b a \log^{k+1} n) \)
3. if \( f(n) \) is \( \Omega(n^\log_b a + \varepsilon) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:
\[ T(n) = 8T(n/2) + n^2 \]
Solution: \( \log_b a = 3 \), so case 1 says \( T(n) \) is \( O(n^3) \).
Master Method, Example 5

The form:

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:

1. if \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:

\[ T(n) = 9T(n/3) + n^3 \]

Solution: \( \log_b a = 2 \), so case 3 says \( T(n) \) is \( O(n^3) \).
Master Method, Example 6

The form:

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\ 
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:

1. if \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) = \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:

\[ T(n) = T(n/2) + 1 \] (binary search)

Solution: \( \log_b a = 0 \), so case 2 says \( T(n) \) is \( O(\log n) \).
Master Method, Example 7

The form:
\[
T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d
\end{cases}
\]

The Master Theorem:
1. if \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \),
   provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:
\[
T(n) = 2T(n/2) + \log n
\]
Solution: \( \log_b a = 1 \), so case 1 says \( T(n) \) is \( O(n) \).
Sketch of Proof of the Master Theorem

Using iterative substitution, let us see if we can find a pattern:

\[ T(n) = aT(n/b) + f(n) \]

\[ = a(aT(n/b^2) + f(n/b)) + bn \]

\[ = a^2 T(n/b^2) + af(n/b) + f(n) \]

\[ = a^3 T(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \]

\[ = \ldots \]

\[ = a^{\log_b n} T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \]

\[ = n^{\log_b a} T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \]

We then distinguish the three cases as:

- The first term is dominant
- Each part of the summation is equally dominant
- The summation is a geometric series
Integer Multiplication

Algorithm: Multiply two n-bit integers I and J.

- Divide step: Split I and J into high-order and low-order bits
  \[ I = I_h 2^{n/2} + I_l \]
  \[ J = J_h 2^{n/2} + J_l \]

- We can then define I*J by multiplying the parts and adding:
  \[ I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l) \]
  \[ = I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l \]

- So, \( T(n) = 4T(n/2) + n \), which implies \( T(n) \) is \( O(n^2) \).
- But that is no better than the algorithm we learned in grade school.
An Improved Integer Multiplication Algorithm

Algorithm: Multiply two n-bit integers I and J.

- Divide step: Split I and J into high-order and low-order bits
  \[ I = I_h 2^{n/2} + I_l \]
  \[ J = J_h 2^{n/2} + J_l \]

- Observe that there is a different way to multiply parts:
  \[
  I \times J = I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l]2^{n/2} + I_l J_l
  
  = I_h J_h 2^n + [(I_h J_l - I_l J_h - I_h J_h + I_l J_l) + I_h J_h + I_l J_l]2^{n/2} + I_l J_l
  
  = I_h J_h 2^n + (I_h J_l + I_l J_h)2^{n/2} + I_l J_l
  
- So, \( T(n) = 3T(n/2) + n \), which implies \( T(n) \) is \( O(n^{\log_2 3}) \), by the Master Theorem.

- Thus, \( T(n) \) is \( O(n^{1.585}) \).
Maxima Set Problem Statement

- We have a database of hotels.
- Each hotel has:
  - a pool size (x-coordinate)
  - quality of restaurant (y-coordinate)
  - Assume all coordinates distinct
- Want hotel with largest pool and best restaurant
  Might not be a unique hotel.
  - One might have largest pool, other best restaurant.
  - Return the set that aren’t wrong.
    Any where no other hotel has both larger pool and better restaurant.
Maxima Set Example
Sort hotels along any dimension

for $i = 1 \rightarrow n - 1$ do

    for $j = i + 1 \rightarrow n$ do

        if $A_i$ has larger pool and better food than $A_j$
            Remove $A_j$

return All hotels that we did not remove

This is $O(n^2)$. 
MaximaSet($S$)

if $n \leq 1$ then
    return $S$

$p \leftarrow$ median point in $S$ by x-coordinate
$L \leftarrow$ points less than $p$
$G \leftarrow$ points greater than or equal to $p$

$M_1 \leftarrow$ MaximaSet($L$)
$M_2 \leftarrow$ MaximaSet($G$)

return $M_1 \cup M_2$
Example revisited

From $M_1 \cup M_2$, which point(s) belong for sure?
Finding a correct recombine

MaximaSet($S$)

\[
\text{if } n \leq 1 \text{ then} \\
\quad \text{return } S \\
\text{\hspace{1cm}} p \leftarrow \text{median point in } S \text{ by } x\text{-coordinate} \\
\text{\hspace{1cm}} L \leftarrow \text{points less than } p \\
\text{\hspace{1cm}} G \leftarrow \text{points greater than or equal to } p \\
\text{\hspace{1cm}} M_1 \leftarrow \text{MaximaSet}(L) \\
\text{\hspace{1cm}} M_2 \leftarrow \text{MaximaSet}(G) \\
\]

\[
\text{\hspace{1cm}} \text{return } M_1 \cup M_2? \\
\]

\[
\text{\hspace{1cm}} \text{How do I recombine correctly?}
\]
Improved Recombine

\[ M_1 \leftarrow \text{MaximaSet}(L) \]
\[ M_2 \leftarrow \text{MaximaSet}(G) \]
\[ \text{for each } a \in M_1 \text{ do} \]
\[ \quad \text{for each } b \in M_2 \text{ do} \]
\[ \quad \text{if } a \text{ better than } b \text{ then} \]
\[ \quad \quad \text{remove } b \text{ from } M_2 \]

▶ How can we improve the “recombine” step?

▶ What is the resulting running time?
Example for the Combine Step

Dominance point from the right
Analysis

In either case, the rest of the non-recursive steps can be performed in $O(n)$ time, so this implies that, ignoring floor and ceiling functions (as allowed by the analysis of Exercise C-11.5), the running time for the divide-and-conquer maxima-set algorithm can be specified as follows (where $b$ is a constant):

$$T(n) = \begin{cases} 
  b & \text{if } n < 2 \\
  2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}$$

Thus, according to the Master Theorem, this algorithm runs in $O(n \log n)$ time.