COP 4531
Complexity & Analysis of Data Structures & Algorithms

Lecture 3
Recurrences
and Divide and Conquer

Thanks to people who contributed to these slides including the text authors and Piyush Kumar, Kevin Wayne, Harold Prokop.
Maximum subarray problem

**Input:** An array $A[1..n]$ of numbers.

**Output:** Indices $i$ and $j$ such that $A[i..j]$ has the greatest sum of any nonempty contiguous subarray of $A$, along with the sum of the values in $A[i..j]$. 
Example motivation - best interval to have bought and sold

- Note that best interval was buy day 7 and sell day 11. The best “change” subsequence was [18 20 -7 12]

<table>
<thead>
<tr>
<th>Day</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>100</td>
<td>113</td>
<td>110</td>
<td>85</td>
<td>105</td>
<td>102</td>
<td>86</td>
<td>63</td>
<td>81</td>
<td>101</td>
<td>94</td>
<td>106</td>
<td>101</td>
<td>79</td>
<td>94</td>
<td>90</td>
<td>97</td>
</tr>
<tr>
<td>Change</td>
<td>13</td>
<td>-3</td>
<td>-25</td>
<td>20</td>
<td>-3</td>
<td>-16</td>
<td>-23</td>
<td>18</td>
<td>20</td>
<td>-7</td>
<td>12</td>
<td>-5</td>
<td>-22</td>
<td>15</td>
<td>-4</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
How to solve this problem for an array $A[1..n]$?

- A “brute force” solution could involve looking at the sum of all possible subarrays - $\binom{n}{2}$ such choices, suggesting a $\Theta(n^2)$ solution.

- Is there a better solution, perhaps using divide and conquer?
Solution using divide and conquer

- Find a maximum subarray of $A[\text{low} .. \text{high}]$
  
  Original function call would have $\text{low} = 1$ and $\text{high} = n$.

- Divide the subarray into two subarrays of roughly equal sizes.
  Find midpoint $\text{mid}$ of the subarray and consider the two subarrays
  $A[\text{low} .. \text{mid}]$ and $A[\text{mid}+1 .. \text{high}]$

- Conquer by finding maximum subarrays of $A[\text{low} .. \text{mid}]$ and $A[\text{mid} + 1 .. \text{high}]$

- Combine?

  It could be the best of the left and right subarrays, but another possibility is that the best subarray crosses the midpoint. So need to find the best of these three cases.
Finding the cost of an array that crosses the midpoint

Can solve in linear time:

- Any subarray crossing the midpoint $A[mid]$ is made up of two subarrays $A[i .. mid]$ and $A[mid+1 .. j]$ where $low \leq i \leq mid$ and $mid < j \leq high$. 

\[ \text{crosses the midpoint} \]

\[ \text{entirely in } A[low .. mid] \quad \text{entirely in } A[mid + 1 .. high] \]

(a) \hspace{6cm} \hspace{6cm} \text{entirely in } A[mid + 1 .. high]

(b)
Find-Max-Cross-Subarray(A, low, mid, high)

// find max subarray of form A[i .. mid]
left-sum = -∞; sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i

// find max subarray of form A[mid + 1 .. high]
right-sum = -∞; sum = 0
for j = mid + 1 to high
    sum = sum + A[i]
    if sum > right-sum
        right-sum = sum
        max-right = j

// return the indices and the sum of the two subarrays
return (max-left, max-right, left-sum + right-sum)
Find-Max-Subarray(A, low, high)

if high == low
    return (low, high, A[low])  //base case 1 element
else mid = ⌊(low + high) / 2⌋
    (l-low, l-high, l-sum) = Find-Max-Subarray(A, low, mid)
    (r-low, r-high, r-sum) = Find-Max-Subarray(A, mid + 1, high)
    (c-low, c-high, c-sum) = Find-Max-Cross-Subarray(A, low, mid, high)
    if l-sum ≥ r-sum and l-sum ≥ c-sum
        return (l-low, l-high, l-sum)
    elseif r-sum ≥ l-sum and r-sum ≥ c-sum
        return (r-low, r-high, r-sum)
    else
        return (c-low, c-high, c-sum)

Note that the initial call is Find-Max-Subarray(A, 1, n)
Analysis of maximum subarray solution

- Let $T(n)$ be running time on array of $n$ elements
- Base case: when high equals low ($n = 1$).
  $T(n) = \Theta(1) = c$ (constant time).
- Recursive case when $n > 1$
  - Dividing takes $\Theta(1)$ time
  - Conquering solves two subproblems each of size $n/2$. Takes $2T(n/2)$ for these.
  - Combining takes $\Theta(n)$ time
Recurrence for this problem is:

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \smallskip \\
2T(n/2) + \Theta(n) & \text{if } n > 1 
\end{cases} \]

- We have seen that this recurrence has the solution \( T(n) = \Theta(n \log n) \).
- Note: there exists a non-recursive solution that works in \( \Theta(n) \)!
Solving recurrences of this type

• Substitution method
  - guess a bound and use mathematical induction to prove this bound is correct

• Recursion tree method
  - convert recursion into a tree and argue by summing up each level of the tree
  - this is not really a proof method but can be used for insight

• Master method
  - provides bounds for any recurrence of the form
  \[ T(n) = a \cdot T(n/b) + f(n) \]
Substitution method

\[ T(n) = \begin{cases} 1 & \text{if } n = 1. \\ 2T(n/2) + n & \text{if } n > 1. \end{cases} \]

- Guess \( T(n) = n \lg n + n \)

\[
T(n) = 2T(n/2) + n \\
= 2 ((n/2) \lg (n/2) + n/2) + n \\
= n(\lg n - 1) + 2n \\
= n \lg n + n
\]

- Can be applied to upper bounds and lower bounds as well
  - often use constants instead of specific values
  - should do base cases
  - often do not worry about boundary cases
Substitution method
upper bound

\[ T(n) = \begin{cases} 
  c & \text{if } n = 1, \\
  2T(n/2) + cn & \text{if } n > 1.
\end{cases} \]

- Guess \( T(n) \leq d \cdot n \cdot \log n \) for some positive constant \( d \).

\[
T(n) \leq 2T(n/2) + cn \\
= 2 \left( \left( \frac{dn}{2} \cdot \log \left( \frac{n}{2} \right) \right) + cn \right) \\
= dn \cdot \log \left( \frac{n}{2} \right) + cn \\
= dn \cdot \log n - dn + cn \\
\leq dn \cdot \log n \quad \text{if } d \geq c
\]

- Note: Have to use the exact form of the induction hypothesis. Cannot simply combine terms such as \( cn + n \) and call this \( O(n) \).
Recursion tree method
Another example

$T(n) = 3T(n/4) + cn^2$
Master method

\[ T(n) = a \cdot T(n/b) + f(n) \quad a \geq 1, b > 1, f(n) > 0 \]

Compare \( n^{\log_b a} \) with \( f(n) \):

**Case 1:** \( f(n) = O(n^{\log_b a - \epsilon}) \) for \( \epsilon > 0 \) \( f(n) \) polynomially smaller

\[ T(n) = \Theta(n^{\log_b a}) \]

**Case 2:** \( f(n) = \Theta(n^{\log_b a}) \) for \( \epsilon > 0 \) same

\[ T(n) = \Theta((n^{\log_b a}) \log n) \]

**Case 3:** \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for \( \epsilon > 0 \) \( f(n) \) polynomially greater

Solution: \( T(n) = \Theta(f(n)) \)
Some Problems Solved with Divide and Conquer

• Inversions – will look at this one
• Closest Point problems
• Integer multiplication
• Matrix Multiplication
Counting Inversions

- Music site tries to match your song preferences with others.
  - You rank n songs.
  - Music site consults database to find people with similar tastes.

- Similarity metric: number of inversions between two rankings.
  - My rank: 1, 2, ..., n.
  - Your rank: $a_1, a_2, ..., a_n$.
  - Songs $i$ and $j$ inverted if $i < j$, but $a_i > a_j$.

Brute force: check all $\Theta(n^2)$ pairs $i$ and $j$. 

<table>
<thead>
<tr>
<th>Songs</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>You</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Inversions
3-2, 4-2
• **Applications.**
  - Voting theory.
  - **Collaborative filtering.**
  - Measuring the "sortedness" of an array.
  - Sensitivity analysis of Google's ranking function.
  - Rank aggregation for meta-searching on the Web.
  - Nonparametric statistics (e.g., Kendall's Tau distance).
Counting Inversions: Divide-and-Conquer

• Divide-and-conquer.
Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - **Divide**: separate list into two pieces.

```
1  5  4  8  10  2  6  9  12  11  3  7
```

Divide: \(O(1)\).

```
1  5  4  8  10  2
6  9  12  11  3  7
```
Counting Inversions: Divide-and-Conquer

- **Divide-and-conquer.**
  - **Divide**: separate list into two pieces.
  - **Conquer**: recursively count inversions in each half.

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>11</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
</table>

**Divide**: $O(1)$.

5 blue-blue inversions
5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions
6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

**Conquer**: $2T(n/2)$
Counting Inversions: Divide-and-Conquer

• Divide-and-conquer.
  – Divide: separate list into two pieces.
  – Conquer: recursively count inversions in each half.
  – Combine: count inversions where \( a_i \) and \( a_j \) are in different halves, and return sum of three quantities.

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>11</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
</table>

Divide: \( O(1) \).

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>11</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
</table>

Conquer: \( 2T(n / 2) \)

5 blue-blue inversions
8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.
Counting Inversions: Combine

- **Combine:** count blue-green inversions
  - Assume each half is sorted.
  - Count inversions where \( a_i \) and \( a_j \) are in different halves.
  - **Merge** two sorted halves into sorted whole. Maintain invariant

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

\[
T(n) \leq T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lceil n/2 \right\rceil \right) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)
\]
Counting Inversions: Implementation

• Pre-condition. [Merge-and-Count] A and B are sorted.
• Post-condition. [Sort-and-Count] L is sorted.

Sort-and-Count(L) {
    if list L has one element
        return 0 and the list L

    Divide the list into two halves A and B
    \((r_A, A) \leftarrow \text{Sort-and-Count}(A)\)
    \((r_B, B) \leftarrow \text{Sort-and-Count}(B)\)
    \((r, L) \leftarrow \text{Merge-and-Count}(A, B)\)

    return \(r = r_A + r_B + r\) and the sorted list L
}
Summary

• Divide and Conquer has many applications, either alone as an approach or as part of an approach.
• Multiplication of matrices using Strassen’s algorithm is a classic example of an unexpected result using divide and conquer.
• Recurrence equations are a powerful tool for analysis.