Warm up
Show that finding the minimum of an unordered list requires $\Omega(n)$ comparisons

Find Min, Lower Bound Proof
Show that finding the minimum of an unordered list requires $\Omega(n)$ comparisons
Suppose (toward contradiction) that there is an algorithm for Find Min that does fewer than $\frac{n}{2} = \Omega(n)$ comparisons.
This means there is at least one "uncompared" element.
We can’t know that this element wasn’t the min!

Announcements
• HW4 due Monday 3/4 at 11pm
  — Sorting
  — Written (use LaTeX)
• No Instructor Office Hours this week
  — I’ll be at SIGCSE
  — Available on Piazza and Email!
• HW1 solutions in-class on Wednesday
• Midterm next Wednesday
  — Covers material through today
  — Review session M or Tu evening
Today’s Keywords

- Sorting
- Linear time Sorting
- Counting Sort
- Radix Sort
- Maximum Sum Continuous Subarray

CLRS Readings

- Chapter 8

Sorting in Linear Time

- Cannot be comparison-based
- Need to make some sort of assumption about the contents of the list
  - Small number of unique values
  - Small range of values
  - Etc.
Counting Sort

- **Idea:** Count how many things are less than each element

1. **Range:** $[1, k]$ (here $[1, 6]$)
   - make an array $C$ of size $k$ to populate with counts of each value
   - For $i$ in $L$:
     - $C[i] += 1$
   - Take "running sum" of $C$ to count things less than each value
   - For $i = 1$ to len($C$):
     - $C[i] = C[i-1] + C[i]$

2. To sort: last item of value $v$ goes at index $v - 1$
   - For $% = len(L)$ down to 1:
     - $B[\%] = v$
     - $C[v] = C[v] - 1$

$B = [6, 2, 3, 1, 4, 1, 6, 1, 2, 3, 4, 5, 6, 7, 8, 2, 0, 2, 1, 0, 3, 1, 2, 3, 4, 5, 6, 7]$

**Run Time:** $O(n + k)$

**Memory:** $O(n + k)$
Counting Sort

• Why not always use counting sort?
• For 64-bit numbers, requires an array of length $2^{64} > 10^{19}$
  - 5 GHz CPU will require > 116 years to initialize the array
  - 18 Exabytes of data
    • Total amount of data that Google has

Radix Sort

• Idea: Stable sort on each digit, from least significant to most significant

Place each element into a "bucket" according to its 1's place
Radix Sort

- **Idea:** Stable sort on each digit, from least significant to most significant

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>513</td>
<td>514</td>
<td>515</td>
<td>516</td>
<td>517</td>
<td>518</td>
<td>519</td>
</tr>
</tbody>
</table>

Place each element into a "bucket" according to its 10's place

Run Time: $O(d(n + b))$

- $d =$ digits in largest value
- $b =$ base of representation

Maximum Sum Continuous Subarray Problem

The maximum-sum subarray of a given array of integers $A$ is the interval $[a, b]$ such that the sum of all values in the array between $a$ and $b$ inclusive is maximal.

Given an array of $n$ integers (may include both positive and negative values), give a $O(n \log n)$ algorithm for finding the maximum-sum subarray.
Divide and Conquer $\Theta(n \log n)$

\[
\begin{array}{cccccccccccc}
5 & 8 & 4 & 3 & 7 & 10 & 2 & 8 & 22 & 19 & 12 & 82 & 3 & 22
\end{array}
\]

Divide in half

Recursively Solve on Left

Recursively Solve on Right

Divide and Conquer $\Theta(n \log n)$

\[
\begin{array}{cccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
5 & 8 & -4 & 3 & 7 & -15 & 2 & 8 & 20 & -3 & 1 & 30 & 5 & 22
\end{array}
\]

Recursively Solve on Left

Divide in half

Find Largest sum that spans the cut

Recursively Solve on Right

Divide and Conquer $\Theta(n \log n)$

\[
\begin{array}{cccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
5 & 8 & -4 & 3 & 7 & -15 & 2 & 8 & 20 & -3 & 1 & 30 & 5 & 22
\end{array}
\]

Recursively Solve on Left

Divide in half

Find Largest sum that spans the cut

Recursively Solve on Right

Return the Max of Left, Right, Center

\[
T(n) = 2T\left(\frac{n}{2}\right) + n
\]
Divide and Conquer Summary

- **Divide**
  - Break the list in half
  - Typically multiple subproblems. Typically all roughly the same size.
- **Conquer**
  - Find the best subarrays on the left and right
- **Combine**
  - Find the best subarray that "spans the divide"
  - I.e. the best subarray that ends at the divide concatenated with the best that starts at the divide

Typically multiple subproblems.

Generic Divide and Conquer Solution

```python
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem)  # brute force if necessary
        return solution
    subproblems = Divide(problem)
    for sub in subproblems:
        subsolutions.append(myDCalgo(sub))
    solution = Combine(subsolutions)
    return solution
```

MSCS Divide and Conquer $\Theta(n \log n)$

```python
def MSCS(list):
    if list.length < 2:
        return list[0]  # list of size 1 the sum is maximal
    {listL, listR} = Divide(list)
    for list in (listL, listR):
        subSolutions.append(MSCS(list))
    solution = max(solutionL, solutionR, span(listL, listR))
    return solution
```
Types of “Divide and Conquer”

• Divide and Conquer
  – Break the problem up into several subproblems of roughly equal size, recursively solve
  – E.g. Karatsuba, Closest Pair of Points, Mergesort...
• Decrease and Conquer
  – Break the problem into a single smaller subproblem, recursively solve
  – E.g. Gotham City Police, Quickselect, Binary Search

Pattern So Far

• Typically looking to divide the problem by some fraction (½, ¼ the size)
• Not necessarily always the best!
  – Sometimes, we can write faster algorithms by finding unbalanced divides.

Unbalanced Divide and Conquer

• Divide
  – Make a subproblem of all but the last element
• Conquer
  – Find best subarray on the left \(BSL(n-1)\)
  – Find the best subarray ending at the divide \(RED(n-1)\)
• Combine
  – New Best Ending at the Divide:
    • \(RED(x) = \max(RED(n-1) + arr[x], 0)\)
  – New best on the left:
    • \(BSL(x) = \max(BSL(n-1), RED(x))\)
Divide Recursively

Solve on Left

Find Largest sum ending at the cut

0
Divide Recursively
Solve on Left

Find Largest sum ending at the cut

Divide

Recursively Solve on Left

Find Largest sum ending at the cut

Divide

Recursively Solve on Left

Find Largest sum ending at the cut

Divide
Unbalanced Divide and Conquer

- **Divide**
  - Make a subproblem of all but the last element

- **Conquer**
  - Find best subarray on the left ($BSL(n - 1)$)
  - Find the best subarray ending at the divide ($BED(n - 1)$)

- **Combine**
  - New Best Ending at the Divide:
    - $BED(n) = \max(BED(n - 1) + arr[n], 0)$
    - New best on the left:
      - $BSL(n) = \max(BSL(n - 1), BED(n))$

Was unbalanced better? **YES**

- **Old:**
  \[ T(n) = 2T\left(\frac{n}{2}\right) + n \]
  - We divided in Half
  - We solved 2 different problems:
    - Find the best overall on BOTH the left/right
    - Find the best which end/start on BOTH the left/right respectively
  - Linear time combine

- **New:**
  \[ T(n) = 17(n - 1) + 1 \]
  - We divide by 1, n-1
  - We solve 2 different problems:
    - Find the best overall on the left ONLY
    - Find the best which ends on the left ONLY
  - Constant time combine
Maximum Sum Continuous Subarray Problem Redux

- Solve in $\Theta(n)$ by increasing the problem size by 1 each time.
- Idea: Only include negative values if the positives on both sides of it are “worth it”

### $\Theta(n)$ Solution

- Begin here
- Remember two values: Best So Far Best ending here

### $\Theta(n)$ Solution

- Remember two values: Best So Far Best ending here

Remember two values:  
Best So Far: 13  
Best ending here: 9
Remember two values:
- Best So Far: 19
- Best ending here: 4

Remember two values:
- Best So Far: 19
- Best ending here: 14

Remember two values:
- Best So Far: 19
- Best ending here: 0
End of Midterm Exam Materials!
**Mid-Class Stretch**
How many ways are there to tile a $2 \times n$ board with dominoes?

<table>
<thead>
<tr>
<th>How many ways to tile this:</th>
<th>With these?</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Tile Options" /></td>
<td><img src="image" alt="Domino Options" /></td>
</tr>
</tbody>
</table>