Chapter 6  
Divide and Conquer

CS3330: Algorithms

Divide and Conquer

- Consider the problem of finding both the minimum and maximum in an array of integers $A[1...n]$ and assume for simplicity that $n$ is a power of 2.

- Divide the input array into two halves $A[1...n/2]$ and $A[(n/2)+1...n]$, find the minimum and maximum in each half and return the minimum of the two minima and the maximum of the two maxima.

**Divide and Conquer**

- A divide-and-conquer algorithm divides the problem instance into a number of subinstances (in most cases 2), recursively solves each subinstance separately, and then combines the solutions to the subinstances to obtain the solution to the original problem instance.

- Input: An array $A[1...n]$ of $n$ integers;
- Output: $(x, y)$: the minimum and maximum integers in $A$;
  - 1. minmax(1, $n$);
    - $\text{minmax}(\text{low}, \text{high})$
      - 1. if high $=$ low $\quad$ // there is only 1 number
      - 2. return $A[\text{low}], A[\text{low}]$;
      - 3. if high $>$ low + 1 $\quad$ // there are only 2 numbers
      - 4. if $A[\text{low}]$ $<$ $A[\text{high}]$: return $A[\text{low}], A[\text{high}]$;
      - 5. else return $A[\text{high}], A[\text{low}]$;
      - 6. $\text{mid} = (\text{low} + \text{high})/2$; // more than two numbers
      - 7. $(x_1, y_1) = \text{minmax}(\text{low}, \text{mid})$;
      - 8. $(x_2, y_2) = \text{minmax}(\text{mid} + 1, \text{high})$;
      - 9. return($\text{min}(x_1, x_2), \text{max}(y_1, y_2)$);

**Divide and Conquer**

- How many comparisons does this algorithm need?

- Given an array $A[1...n]$ of $n$ elements, where $n$ is a power of 2, it is possible to find both the minimum and maximum of the elements in $A$ using only $1.5n - 2$ element comparisons.

- Let $C(n)$ be the maximum number of comparisons the algorithm needs for $n$ numbers.
  - $C(1) = 0$, $C(2) = 1$
  - $C(n) = 2C(n/2) + 2$.
  - Then $C(n) = 1.5n - 2$ for (power of 2) $n > 1$.

**The Divide and Conquer Paradigm**

- The divide step: the input is partitioned into $p \geq 1$ parts, each of size strictly less than $n$.

- The conquer step: performing $p$ recursive call(s) if the problem size is greater than some predefined threshold $n_p$.

- The combine step: the solutions to the $p$ recursive call(s) are combined to obtain the desired output.
Sorting algorithms

- Insertion, selection and bubble sort have quadratic worst-case performance
- The faster comparison based algorithm O(nlogn)
- Heapsort, Mergesort, Quicksort

Merge Sort

- An example of divide-and-conquer technique
- Problem: Given n elements, sort elements into non-decreasing order
- Divide-and-Conquer:
  - If n=1 terminate (every one-element list is already sorted)
  - If n>1, partition elements into two or more sub-collections; sort each; combine into a single sorted list
- How do we partition?

Partitioning - Choice 1

- First n-1 elements into set A, last element set B
- Sort A using this partitioning scheme recursively
  - B already sorted
- Combine A and B using method Insert() (= insertion into sorted array)
- Leads to recursive version of InsertionSort()
  - Number of comparisons: O(n²)
    - Best case = n-1
    - Worst case = \( \sum_{i=2}^{n} \frac{n(n-1)}{2} \)

Divide-Conquer View of Insertion Sort

```java
insertion_sort(int[] A, int n) {
    // Pre: 0 <= n < A.length
    // Post: A[0..n] is sorted
    if (n == 0) return;
    // divide:
    // do nothing
    // Now we have two parts: A[0..n-2] and A[n-1]
    // conquer:
    insertion_sort(A, n-1);
    // combine:
    // do nothing.
}
```

Partitioning - Choice 2

- Put element with largest key in B, remaining elements in A
- Sort A recursively
- To combine sorted A and B, append B to sorted A
  - Use Max() to find largest element \rightarrow recursive SelectionSort()
  - Use bubbling process to find and move largest element to right-most position \rightarrow recursive BubbleSort()
- All O(n²)

Divide-Conquer View of Selection Sort

```java
selection_sort(int[] A, int n) {
    // Pre: 0 <= n < A.length
    // Post: A[0..n] is sorted
    if (n == 0) return;
    // divide:
    int max = maxPosition(A, n);
    // it returns the position of max element exchange(A, n, max);
    // Now we have two parts: A[0..n-2] and A[n-1]
    // conquer:
    selection_sort(A, n-1);
    // combine:
    // do nothing.
}
```
Partitioning - Choice 3

- Let’s try to achieve balanced partitioning
- A gets n/2 elements, B gets the rest half
- Sort A and B recursively
- Combine sorted A and B using a process called *merge*, which combines two sorted lists into one
  - How? We will see soon

Mergesort

- Divide the array into two equally-sized halves
- Recursively sort the halves
- Merge the sorted halves

Example

- Partition into lists of size n/2

```
[10, 4, 6, 8, 2, 5, 7]
[10, 4, 6, 3]
[10, 4] [6, 3]
[4, 10] [3, 6]

[8, 2, 5, 7]
[8, 2]
[2, 8]
[5, 7]
```

Example Cont’d

- Merge

```
[2, 3, 4, 5, 6, 7, 8, 10]
[2, 5, 7, 8]
[2, 5] [7, 8]
[4, 10] [3, 6]
[4, 10] [3, 6]

[10, 4] [6, 3]
[10, 4] [6, 3]
```

Merging

- The key to Merge Sort is merging two sorted lists into one, such that if you have two lists X \((x_1 \leq x_2 \leq \cdots \leq x_m)\) and Y \((y_1 \leq y_2 \leq \cdots \leq y_n)\) the resulting list is \(z_1 \leq z_2 \leq \cdots \leq z_{m+n}\)
- Example:
  \(X = \{3, 8, 9\} \quad Y = \{1, 5, 7\}\)
  \(\text{merge}(X, Y) = \{1, 3, 5, 7, 8, 9\}\)
Merging (cont.)

X: [ ] [ ] [ ] [ ] Y: [ ] [ ] [ ] 75
Result: [ ] [ ] [ ] [ ] 23 25 54

Merging (cont.)

X: [ ] [ ] [ ] [ ] Y: [ ] [ ] [ ] [ ]
Result: [ ] [ ] [ ] [ ] 23 25 54 75

Static Method mergeSort()

```java
public static void mergeSort(Comparable[] a, int left, int right)
{
    // sort a[left:right] using b[left:right]
    if (left < right) // at least two elements
    {
        int mid = (left + right) / 2; // midpoint
        mergeSort(a, left, mid);
        mergeSort(a, mid + 1, right);
        merge(a, b, left, mid, right); // merge from a to b
        copy(b, a, left, right); // copy result back to a
    }
}
```

Evaluation

- Recurrence equation:
- Assume \( n \) is a power of 2

\[
T(n) = \begin{cases} 
    c_1 & \text{if } n=1 \\
    2T(n/2) + c_2n & \text{if } n>1, \; n=2^k 
\end{cases}
\]

Solution

By Substitution:

\[
\begin{align*}
T(n) &= 2T(n/2) + c_2n \\
T(n/2) &= 2T(n/4) + c_2n/2 \\
T(n) &= 4T(n/4) + 2c_2n \\
T(n) &= 8T(n/8) + 3c_2n \\
T(n) &= 2T(n/2) + kc_2n
\end{align*}
\]

Assuming \( n = 2^k \), expansion halts when we get \( T(1) \) on right side; this happens when \( k=\log n \)
Since \( 2^k = n \), we know \( k=\log n \); since \( T(1) = c_1 \), we get
\[
T(n) = c_1n + c_2n\log n
\]
thus an upper bound for \( T_{\text{mergeSort}}(n) \) is \( O(n\log n) \)

Implementing Merge Sort

- There are two basic ways to implement merge sort:
  - In Place: Merging is done with only the input array
    - Pro: Requires only the space needed to hold the array
    - Con: Takes longer to merge because if the next element is in the right side then all of the elements must be moved down.
  - Double Storage: Merging is done with a temporary array of the same size as the input array.
    - Pro: Faster than In Place since the temp array holds the resulting array until both left and right sides are merged into the temp array, then the temp array is appended over the input array.
    - Con: The memory requirement is doubled.
Variants and Applications

There are other variants of Merge Sorts including bottom-up merge sort, k-way merge sorting, but the common variant is the Double Memory Merge Sort. Though the running time is $O(n \log n)$ and runs much faster than insertion sort and bubble sort, merge sort’s large memory demands makes it not very practical for main memory sorting.

Merge Sort is the major method for external sorting, parallel algorithms, and sorting circuits.

Bottom-up Merge Sort

• Sublists are always of size $2^k$, except the last

```
[2, 3, 4, 5, 6, 7, 8, 10]

[3, 6, 10] [2, 5, 8]

[4, 10] [3, 6] [2, 8]

```

Quicksort Algorithm

Given an array of $n$ elements (e.g., integers):

• If array only contains one element, return

• Else
  
  — pick one element to use as pivot.
  
  — Partition elements into two sub-arrays:
    
    • Elements less than or equal to pivot
    
    • Elements greater than pivot
  
  — Quicksort two sub-arrays
  
  — Return results
Example
We are given array of n integers to sort:

```
40  20  10  80  60  50  7  30  100
```

Pick Pivot Element
There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

```
40  20  10  80  60  50  7  30  100
```

Partitioning Array
Given a pivot, partition the elements of the array such that the resulting array consists of:
1. One sub-array that contains elements $\geq$ pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

```
0. too_big_index = left+1; too_small_index = right;
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
```

```
pivot_index = 0
```

```
0. too_big_index = left+1; too_small_index = right;
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
```

```
pivot_index = 0
```

```
0. too_big_index = left+1; too_small_index = right;
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
```

```
pivot_index = 0
```

```
0. too_big_index = left+1; too_small_index = right;
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
```
0. too_big_index = left+1; too_small_index = right;
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
   too_big_index = left+1; too_small_index = right;
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
0. too_big_index = left+1; too_small_index = right;
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

Partition Result

```
[0] 7 20 10 30 50 60 80 100
```

Recursive calls on two sides to get a sorted array.

QuickSort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. QuickSort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. QuickSort each sub-array
  - Depth of recursion tree?
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
    – Depth of recursion tree? O(log₂n)

Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
    – Depth of recursion tree? O(log₂n)
    – Number of accesses in partition?

Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: O(n log₂n)
• Worst case running time?

Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: O(n log₂n)
• Worst case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays:
      • one sub-array of size 0
      • the other sub-array of size n-1
    2. Quicksort each sub-array
    – Depth of recursion tree?
### Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays: 
       - one sub-array of size 0
       - the other sub-array of size $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$

### Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays: 
       - one sub-array of size 0
       - the other sub-array of size $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$
  - Number of accesses per partition? $O(n)$

### Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!

### Quicksort Analysis

- Bad divide: $T(n) = T(1) + T(n-1) + n$ $-- O(n^2)$
- Good divide: $T(n) = T(n/2) + T(n/2) + n$ $-- O(n \log_2 n)$
- Random divide: Suppose on average one bad divide followed by one good divide.
  - $T(n) = T(1) + T(n-1) + n = T(1) + 2T((n-1)/2) + 2n$
  - $T(n) = 2n + 2T((n-1)/2)$ is still $O(n \log_2 n)$
Improved Pivot Selection

Pick median value of three elements from data array:
\[ \text{data}[0], \text{data}[n/2], \text{data}[n-1]. \]

Use this median value as pivot.

For large arrays, use the median of three medians from
\( \{\text{data}[0], \text{data}[1], \text{data}[2]\}, \{\text{data}[n/2-1], \text{data}[n/2], \text{data}[n/2+1]\}, \) and \( \{\text{data}[n-3], \text{data}[n-2], \text{data}[n-1]\}. \)

Improving Performance of Quicksort

• Improved selection of pivot.

For sub-arrays of size 100 or less, apply brute force search, such as insert-sort.
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    • if(data[first] > data[second]) swap them
  - Sub-array of size 3: left as an exercise.

0. too_big_index = left; too_small_index = right;
1. while (data[too_big_index] <= data[pivot] ++too_big_index;
2. while (data[too_small_index] > data[pivot]) --too_small_index;
3. if (too_big_index < too_small_index)
   swap data[too_big_index] and data[too_small_index];
4. if (too_small_index > too_big_index) go to 1.
5. return too_small_index;

Divide and Conquer

<table>
<thead>
<tr>
<th></th>
<th>Simple Divide</th>
<th>Fancy Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uneven Divide</td>
<td>Insert Sort</td>
<td>Selection Sort</td>
</tr>
<tr>
<td>Even Divide</td>
<td>Merge Sort</td>
<td>Quick Sort</td>
</tr>
</tbody>
</table>

Stable Sorting

• Maintains the original relative positioning of equivalent values
  • InsertSort, SelectSort, BubbleSort are stable
  • Heapsort is not stable
  • Mergesort is stable
  • Quicksort is not stable
How Fast Can We Sort?

• Selection Sort, Bubble Sort, Insertion Sort: \(O(n^2)\)
• Heap Sort, Merge sort: \(O(n \log n)\)
• Quicksort: \(O(n \log n)\) - average
• What is common to all these algorithms?
  – Make comparisons between input elements
  \[ a_i < a_j, \ a_i \leq a_j, \ a_i = a_j, \ a_i > a_j, \] or \( a_i > a_j \)

Comparison-based Sorting

• **Comparison sort**
  – Only comparison of pairs of elements may be used to gain order information about a sequence.
  – Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
• All our sorts have been comparison sorts
• The best **worst-case complexity** so far is \(\Theta(n \log n)\) (e.g., merge sort).
• We prove a lower bound of \(\Omega(n \log n)\) for any comparison sort: merge sort and heapsort are optimal.
• The idea is simple: there are \(n!\) outcomes, so we need a tree with \(n!\) leaves, and therefore \(\log(n!) = n \log n\).

Decision Tree

For insertion sort operating on three elements.

\[
\begin{array}{c}
1.2 \\
2.3 \\
1.3 \\
(1,2,3) \\
(1,3,2) \\
(2,1,3) \\
(2,3,1) \\
(3,1,2) \\
(3,2,1)
\end{array}
\]

Simply unroll all loops for all possible inputs.
Node \((i,j)\) means compare \(a_i\) to \(a_j\).
Leaves show outputs; No two paths go to same leaf!

Contains \(3! = 6\) leaves.

A Lower Bound for Worst Case

• **Worst case no. of comparisons for a sorting algorithm** is
  – Length of the longest path from root to any of the leaves in the decision tree for the algorithm.
  – Which is the height of its decision tree.
• **A lower bound on the running time of any comparison sort** is given by
  – A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf.

Decision Tree (Contd.)

• Execution of sorting algorithm corresponds to **tracing a path from root to leaf**.
• The tree models all possible execution traces.
• **At each internal node**, a comparison \(a_i \leq a_j\) is made.
  – If \(a_i \leq a_j\), follow left subtree, else follow right subtree.
  – View the tree as if the algorithm splits in two at each node, based on information it has determined up to that point.
• When we come to a **leaf**, ordering \(a_{i(1)} \leq a_{i(2)} \leq ... \leq a_{i(n)}\) is established.
  – A correct sorting algorithm must be able to produce any permutation of its input.
  – Hence, each of the \(n!\) permutations must appear at one or more of the leaves of the decision tree.

Optimal sorting for three elements

Any sort of six elements has 5 internal nodes.

\[
\begin{array}{c}
1.2 \\
2.3 \\
1.3 \\
(1,2,3) \\
(1,3,2) \\
(2,1,3) \\
(2,3,1) \\
(3,1,2) \\
(3,2,1)
\end{array}
\]

There must be a worst-case path of length \(\geq 3\).
**A Lower Bound for Worst Case**

**Theorem:** Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

**Proof:**
- Suffices to determine the height of a decision tree.
- The number of leaves is at least $n!$ (# outputs)
- The number of internal nodes $\geq n! - 1$
- The height is at least $\log(n!-1) = \Omega(n \lg n)$

**Can we do better?**

- Linear time sorting algorithms
  - Counting Sort
  - Bucket sort
  - Radix Sort
- Make certain assumptions about the data
- Linear sorts are NOT “comparison sorts”

**Counting Sort**

- **Assumptions:**
  - $n$ integers which are in the range $[0 ... r]$
  - $r$ is in the order of $n$, that is, $r=O(n)$
- **Idea:**
  - For each element $x$, find the number of occurrences of $x$ and store it in the counter
  - Place $x$ into its correct position in the output array using the counter.

**Step 1**

1. int index = 0;
2. for $i = 1$ to max_value
3. for $j = 1$ to counter[$i$]
4. values[index++] = $i$;
   // Copy value $i$ into the array counts[$j$] times

**Step 2**

1. int index = 0;
2. for $i = 0$ to max_value
3. for $j = 1$ to counter[$i$]
4. values[index++] = removeFirst(queue[$i$]);

**When an element has other info**

- **Step 1:** Create a queue for each key. When an element is inserted, the key counter increases by one and the element is stored in the queue.
- **Step 2:** For each key $i$ from small to big, the elements in the queue $i$ is inserted into the array.
Analysis of Counting Sort

- Overall time: $O(n + r)$
  - Step 1: $O(n)$
  - Step 2: $O(n + r)$
- Space: $O(n+r)$
- In practice we use COUNTING sort when $r = O(n)$ \iff running time is $O(n)$

Bucket Sort

- Assumption:
  - the input is generated by a random process that distributes elements uniformly over a range $R = [a..b]$
- Idea:
  - Divide $R$ into $k$ equal-sized buckets ($k = \Theta(n)$)
  - Distribute the $n$ input values into the buckets
  - Sort each bucket (e.g., using quicksort or bucket sort)
  - Go through the buckets in order, listing elements in each one
  - Similar idea to sort post office mails, where $k = 10$.
- Input: $A[0 \ldots (n-1)]$, where $a \leq A[i] \leq b$ for all $i$
- Output: elements $A[i]$ sorted

Example - Bucket Sort $R = [0..0.99]$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>78</td>
<td>/</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>/</td>
</tr>
<tr>
<td>2</td>
<td>38</td>
<td>/</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>/</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>/</td>
</tr>
<tr>
<td>5</td>
<td>39</td>
<td>/</td>
</tr>
<tr>
<td>6</td>
<td>94</td>
<td>/</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>/</td>
</tr>
<tr>
<td>8</td>
<td>78</td>
<td>/</td>
</tr>
<tr>
<td>9</td>
<td>72</td>
<td>/</td>
</tr>
<tr>
<td>10</td>
<td>34</td>
<td>/</td>
</tr>
</tbody>
</table>

Distribute into buckets

Sort within each bucket

Concatenate the lists from 0 to $k - 1$ together, in order

Analysis of Bucket Sort

\textbf{Alg.:} BUCKET-SORT($A$, $n$)
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad \text{do insert } A[i] \text{ into list } B[\lfloor k*A[i] \rfloor] \\
&\text{for } i \leftarrow 0 \text{ to } k - 1 \\
&\quad \text{do sort list } B[i] \text{ with quicksort sort} \\
&\quad \text{concatenate lists } B[0], B[1], \ldots, B[n-1] \text{ together in order} \\
&\text{return the concatenated lists}
\end{align*}

$O(n)$

$O(k)$ ($if$ $k=\Theta(n)$)
Counting Sort vs Bucket Sort

• Counting sort uses queues while Bucket Sort uses buckets — similar data structures.
• The queues used by Counting sort store the elements of the same key — they are already sorted.
• The buckets used by Bucket Sort store the elements sharing one feature of the keys — they need be further sorted.
• Both are stable sorting algorithms if Bucket Sort uses stable sorting for each bucket.

Radix Sort

• Represents keys as $d$-digit numbers in some base-$k$
  \[ key = x_1 x_2 \ldots x_d \text{ where } 0 \leq x_i \leq k-1 \]
• Example: $key = 154$
  \[ key_{10} = 15, d=3, k=10 \text{ where } 0 \leq x_i \leq 9 \]

Radix-Sort

Alg.: RADOX-SORT($A$, $d$)
  \[
  \text{for } i ← 1 \text{ to } d \text{ do use a stable sort to sort array } A \text{ on digit } i
  \]
  \[
  \text{(stable sort: preserves order of identical elements)}
  \]

Analysis of Radix Sort

• Given $n$ numbers of $d$ digits each, where each digit may take
  up to $k$ possible values, RADOX-SORT correctly sorts the
  numbers in $O(d(n+k))$, or $O(n)$ if $d = O(1)$ and $k = O(n)$.
  – One pass of sorting per digit takes $O(n+k)$ assuming that
    we use counting sort
  – There are $d$ passes (for each digit)

Integer Addition

Addition: Given two $n$-bit integers $a$ and $b$, compute $a + b$. Grade-school $O(n)$ bit operations.

Remark: Grade-school addition algorithm is optimal.
Q. Is grade-school multiplication algorithm optimal?

Divide-and-Conquer Multiplication: Warmup

To multiply two \( n \)-bit integers \( a \) and \( b \):
- Multiply four \( \frac{n}{2} \)-bit integers, recursively.
- Add and shift to obtain result.

\[
a = 2^{n/2} a_1 + a_0 \\
b = 2^{n/2} b_1 + b_0 \\
ab = (2^{n/2} a_1 + a_0)(2^{n/2} b_1 + b_0) = 2^n a_1 b_1 + 2^{n/2} (a_1 b_0 + a_0 b_1) + a_0 b_0
\]

Ex. \( a = 10001101 \) \( b = 11100001 \)

\[
T(n) = 4T(n/2) + O(n) \implies T(n) = \Theta(n^2)
\]

Karatsuba Multiplication

To multiply two \( n \)-bit integers \( a \) and \( b \):
- Add two \( \frac{n}{2} \)-bit integers.
- Multiply three \( \frac{n}{2} \)-bit integers, recursively.
- Add, subtract, and shift to obtain result.

\[
a = 2^{n/2} a_1 + a_0 \\
b = 2^{n/2} b_1 + b_0 \\
ab = 2^n a_1 b_1 + 2^{n/2} (a_1 b_0 + a_0 b_1) + a_0 b_0
\]

Theorem. [Karatsuba-Ofman 1962] Can multiply two \( n \)-bit integers in \( O(n^{\log_2 3}) \) bit operations.

Karatsuba: Recursion Tree

\[
T(n) = \sum_{j=0}^{\log_2 n} \left[ T\left( \frac{n}{2^j} \right) + T\left( \frac{n}{2^{j+1}} \right) + \Theta\left( \frac{n}{2^j} \right) \right]
\]
Dot Product

Dot product: Given two length $n$ vectors $a$ and $b$, compute $c = a \cdot b$.

Grade-school: $\mathcal{O}(n^2)$ arithmetic operations.

$$a = \begin{bmatrix} 70 & 20 & 10 \end{bmatrix}$$
$$b = \begin{bmatrix} 30 & 40 & 30 \end{bmatrix}$$

$$a \cdot b = (70 \times 30) + (20 \times 40) + (10 \times 30) = 32$$

Remark. Grade-school dot product algorithm is optimal.

Matrix Multiplication

Matrix multiplication: Given two $n \times m$ matrices $A$ and $B$, compute $C = AB$.

Grade-school: $\mathcal{O}(n^3)$ arithmetic operations.

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Fast Matrix Multiplication

Key idea. Multiply 2-by-2 blocks with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$C_{11} = P_{11} + P_{22} - P_{12} - P_{21}$
$C_{12} = P_{12} + P_{22}$
$C_{21} = P_{11} + P_{21}$
$C_{22} = P_{11} + P_{22} - P_{12} - P_{21}$

7 multiplications.
18 = 8 + 10 additions and subtractions.

Matrice Multiplication: Warmup

To multiply two $n \times m$ matrices $A$ and $B$:

- Divide: partition $A$ and $B$ into $\frac{n}{2} \times \frac{m}{2}$ blocks.
- Conquer: multiply 8 pairs of $\frac{n}{2} \times \frac{m}{2}$ matrices, recursively.
- Combine: add appropriate products using $4$ matrix additions.

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Analysis:

- Assume $n$ is a power of 2.
- $T(n) = \#$ arithmetic operations.

$$T(n) = 7T(n/2) + 6n^2 \implies T(n) = \Theta(n^\omega)$$
Fast Matrix Multiplication

To multiply two \( n \times n \) matrices \( A \) and \( B \): [Strassen 1969]
\[
T(n) = 7T(n/2) + O(n^3) \quad \Rightarrow \quad T(n) = O(n^{\log_2 7}) = O(n^{2.807})
\]

Table 6.2: The number of arithmetic operations done by the three algorithms.

<table>
<thead>
<tr>
<th>( n \times n )</th>
<th>Multiplications</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional alg.</td>
<td>( n^3 )</td>
<td>( n^3 )</td>
</tr>
<tr>
<td>Recursive version</td>
<td>( n^{3+\epsilon} )</td>
<td>( n^{3+\epsilon} )</td>
</tr>
<tr>
<td>Strassen’s alg.</td>
<td>( n^{\log_2 7} )</td>
<td>( n^{\log_2 7} )</td>
</tr>
</tbody>
</table>

Table 6.3: Comparisons between Strassen’s algorithm and the traditional algorithms.

Fast Matrix Multiplication: Practice

Implementation issues.
- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around \( n = 128 \).

Common misperception. "Strassen is only a theoretical curiosity."
- Apple reports 8x speedup on G4 Velocity Engine when \( n \approx 2,500 \).
- Range of instances where it’s useful is a subject of controversy.

Remark. Can "Strassenize" \( Ax = b \), determinant, eigenvalues, SVD, ...

Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen 1969]

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr 1971]

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible.

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, ...]
- Two 20-by-20 matrices with 4,460 scalar multiplications.
- Two 48-by-48 matrices with 47,361 scalar multiplications.
- A year later.

Best known. \( O(n^{2.376}) \) [Coppersmith-Winograd, 1987]

Conjecture. \( O(n^{2+\epsilon}) \) for any \( \epsilon > 0 \).

Caveat. Theoretical improvements to Strassen are progressively less practical.