Divide and Conquer

The most well known algorithm design strategy:
1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide and Conquer Examples

- Sorting: mergesort and quicksort
- Tree traversals
- Binary search
- Matrix multiplication- Strassen’s algorithm
- Convex hull- QuickHull algorithm

General Divide and Conquer recurrence:

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

where \( f(n) \in \Theta(n^k) \)

1. \( a < b^k \) \( T(n) \in \Theta(n^k) \)
2. \( a = b^k \) \( T(n) \in \Theta(n^k \log n) \)
3. \( a > b^k \) \( T(n) \in \Theta(n^{\log_b a}) \)

Note: the same results hold with \( O \) instead of \( \Theta \).

Mergesort

Algorithm:
1. Split array \( A[1..n] \) in two and make copies of each half in arrays \( B[1..n/2] \) and \( C[1..n/2] \)
2. Sort arrays \( B \) and \( C \)
3. Merge sorted arrays \( B \) and \( C \) into array \( A \) as follows:
   - Repeat the following until no elements remain in one of the arrays:
     - Compare the first elements in the remaining unprocessed portions of the arrays
     - Copy the smaller of the two into \( A \), while incrementing the index indicating the unprocessed portion of that array
   - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into \( A \).
Efficiency of mergesort

- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons is close to theoretical minimum for comparison-based sorting:
  - $\log n \approx \log_2 n \approx 1.44 \log n$
- Space requirement: $\Theta(n)$ (NOT in-place)
- Can be implemented without recursion (bottom-up)

Efficiency of quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

Improvements:
- Better pivot selection: median of three partitioning avoids worst case in sorted files
- Switch to insertion sort on small subfiles
- Elimination of recursion
  - These combine to 20-25% improvement
- Considered the method of choice for internal sorting for large files ($n > 10000$)

Quicksort

- Select a pivot (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot (See algorithm \texttt{Partition} in section 4.2)
- Exchange the pivot with the last element in the first sublist — the pivot is now in its final position
- Sort the two sublists

QuickHull Algorithm

Inspired by Quicksort compute Convex Hull:
- Assume points are sorted by $x$-coordinate values
- Identify extreme points $P_1$ and $P_2$ (part of hull)
- Compute upper hull:
  - Find point $P_{\text{max}}$ that is farthest away from line $P_1P_2$
  - Compute the hull of the points to the left of line $P_1P_{\text{max}}$
- Compute lower hull in a similar manner
Efficiency of QuickHull algorithm

- Finding point farthest away from line $P_1P_2$ can be done in linear time.
- This gives same efficiency as quicksort:
  * Worst case: $\Theta(n^2)$
  * Average case: $\Theta(n \log n)$
- If points are not initially sorted by x-coordinate value, this can be accomplished in $\Theta(n \log n)$ — no increase in asymptotic efficiency class.
- Other algorithms for convex hull:
  * Graham's scan
  * DC-Hull also in $\Theta(n \log n)$

Strassen's matrix multiplication

- Strassen observed [1969] that the product of two matrices can be computed as follows:

$$
\begin{bmatrix}
C_{00} & C_{01} \\
C_{10} & C_{11}
\end{bmatrix}
= \begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\begin{bmatrix}
B_{00} & B_{01} \\
B_{10} & B_{11}
\end{bmatrix}
$$

$$
C_{00} = A_{00}B_{00} + A_{01}B_{11}
$$

$$
C_{01} = A_{00}B_{10} + A_{01}B_{11}
$$

$$
C_{10} = A_{10}B_{00} + A_{11}B_{00}
$$

$$
C_{11} = A_{10}B_{10} + A_{11}B_{11}
$$

Efficiency of Strassen's algorithm

- If $n$ is not a power of 2, matrices can be padded with zeros.
- Number of multiplications:
  - $8n^2$ in the naive method
  - $O(n \log^2 n)$ in Strassen's method
- Number of additions:
  - $18n^2 - 12n$ in the naive method
  - $O(n \log^2 n)$ in Strassen's method
- Other algorithms have improved this result, but are even more complex.