Lecture 04: Divide and Conquer

- Divide and Conquer
  - Merge Sort
  - Binary Search
  - Prune and Search Method
  - Powering a Number
  - Computing Fibonacci Numbers
  - Matrix Multiplication
  - Strassen's Matrix Multiplication

These notes are hand-written, unedited and sketchy. They are primarily used for, and based on my lectures.

If you find any bug, impreciseness, or a rare poor-/mis-interpretation of facts, please let me know. I will be grateful for any additional comments you have that are intended to make the quality of the notes better.

Please note that I will provide my hand-written lecture notes only for a subset of my lectures, not for all lectures. Therefore, it is your responsibility to attend all the lectures, take notes regularly, and ask me and/or the TAs if you have any questions.

Thank you!
--Chittu
Divide and Conquer: In politics: Divide and rule/Divide et impera.
-One of the powerful techniques for algorithm design
-Leads to naturally recursive algorithms
-Analyzes uses recursion

Algorithm: Divide-and-Conquer

Step 1: Divide the problem instance into a set of smaller subproblems
Step 2: Conquer the subproblems by solving them recursively
Step 3: Combine the solutions to the subproblems to obtain solution
to the original problem
Merge Sort:

Given: An array \( A[p...r] \) of \( n \) elements for which \( <, =, > \) relations are defined.

Goal: Sort \( A \) in nondecreasing order.

Idea: Divide and Conquer

1. If \( n = 1 \) then do nothing
2. Divide \( A \) into two halves and recursively sort each half
3. Merge two sorted arrays

Unsorted: \( \begin{array}{ccccccccc} 10 & 2 & 5 & 3 & 7 & 13 & 1 & 6 \end{array} \)

Sorted: \( \begin{array}{ccccccccc} 1 & 2 & 3 & 5 & 6 & 7 & 10 & 13 \end{array} \)

\( \text{MERGE-SORT}(A, p, r) \)

1. if \( p < r \)
   
   \[ q = \lfloor \frac{p+r}{2} \rfloor \]  // Divide: Trivial
   
   \( \text{MERGE-SORT}(A, p, q) \)  // Conquer
   
   \( \text{MERGE-SORT}(A, q+1, r) \)

   \( \text{MERGE}(A, p, q, r) \)  // Combine: \( O(n) \) for MERGE

Time: \( T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n) \)
MERGE2(C+r, p, q, q+r, y)  // Merge two sorted arrays A[p...q] and B[q+r]
1. if p < r, return B
2. if r > b, return A
3. if A[p] ≤ B[q+r]
4. return A[p] 0 MERGE2(A, p+1, q, B, q+r, y)
5. else return B[q] 0 MERGE2(A, p, q, B, q+r+1, y)

Note: MERGE2 is implemented using an auxiliary array of size which is the sum of the sizes of A and B.
We can use MERGE2 to implement our MERGE procedure.
MERGE(A, p, q, y)
1. C[p...y] ← MERGE2(A, p, q, A, p+1, y)
2. Copy C back to A

Exercise: MERGE-SORT can be implemented in a bottom-up iterative manner using a queue. Design an algorithm for the iterative MERGE-SORT.

Exercise: The MERGE2 procedure can be implemented as a simple iterative procedure. It is helpful to recall the two-fingers algorithm we briefly discussed during the lecture. Design an iterative algorithm for MERGE2.
**Binary Search:**

**Given:** a sorted array $A$ of $n$ elements, and a key $x$.

**Goal:** check if the key $x$ is present in $A$. If so, then return its corresponding index of the element in $A$; otherwise, report error.

Given array: $2, 4, 6, 9, 10, 14, 17$

Search: $x = 10$

Exercise: Give an iterative version of binary search.

```plaintext
REC-BIN-SRCH($A, x, i, j$)

1. if $i > j$ return "key not found"
2. mid = [(i+j)/2]
3. if $A[mid] = x$
   ...
4. return mid
5. else if $A[mid] > x$
   return REC-BIN-SRCH($A, x, i, mid-1$)
else return REC-BIN-SRCH($A, x, mid+1, j$)
```

**Divide:** check the middle element

**Conquer:** throw away half of the array, and search for $x$ in the other half.

**Combine:** trivial. Do nothing

**Time:** $T(n) = T(n/2) + O(1) = O(n)$

**Note:** in binary search, we always throw away half of the array and search in the other half.

- A divide and conquer algorithm which always away a fraction of the input and relatively work on the other part of the input is called a "Prune and Search" algorithm.
- Binary search is an example.
Prune and Search:
- A special type of divide and conquer in which we throw away (i.e., eliminate from further consideration, a.k.a. PRUNE) a fraction of the input and recurse on the remaining part of the input.
- Example: Binary search, where half of the array is found in each step giving us the recurrence
  \[ T(n) = T(n/2) + \Theta(1) \]

- General Prune and Search Recurrence:
  \[ T(n) = T(\alpha n) + f(n) \Rightarrow T(n) = \begin{cases} 
\Theta(\log n), & \text{if } f(n) = \Theta(1) \\
\Theta(f(n)), & \text{otherwise}.
\end{cases} \]
  
  - Pruning factor: \( 1 - \alpha \)
  - Input size for recursive calls: \( n, \alpha n, \alpha^2 n, \ldots \)

- More General Prune and Search Recurrence:

Theorem: Let \( T(n) = T(\alpha_1 n) + T(\alpha_2 n) + \cdots + T(\alpha_k n) + f(n) \)
for constants \( 0 < \alpha_1, \ldots, \alpha_k < 1 \) s.t. \( \alpha_1 + \alpha_2 + \cdots + \alpha_k < 1 \).
Then the solution to \( T(n) \) is given by
  \[ T(n) = \begin{cases} 
\Theta(\log n), & \text{if } f(n) = \Theta(1) \\
\Theta(f(n)), & \text{otherwise}.
\end{cases} \]

- We will see one more example of prune and search when we design an algorithm for finding median (or i-th smallest element in an array).
**Powering a Number: Computing \( a^n \), \( n \in \mathbb{N} \).**

\[ a^n = a \times a \times \cdots \times a \]

\( n \) a's

**Algorithm 1:** Naively multiply from left to right.

\[ \text{ITERATIVE-POWER}(a, n) \]

1. result \( \leftarrow 1 \)  
2. for \( i \leftarrow 1 \) to \( n \)  
3. result \( \leftarrow \) result \( \times a \)  
4. return result  

*Time: \( T(n) = \Theta(n) \).*

**Perpetual Question:** Can we do better?

**Idea:** Let's try divide-and-conquer — split \( n \) into two halves, compute \( a^{n/2} \), save it to use once more to compute \( a^n \).

i.e. Memoize

**Algorithm 2:** Use the above idea

\[ \text{RECURSIVE-POWER}(a, n) \]

1. if \( n = 1 \) return \( a \)
2. if \( n \) is even  
   
   \[ \text{memo} \leftarrow \text{RECURSIVE-POWER}(a, n/2) \]
   
   return \( \text{memo} \times \text{memo} \)

else if \( n \) is odd  
   
   \[ \text{memo} \leftarrow \text{RECURSIVE-POWER}(a, \frac{n-1}{2}) \]
   
   return \( \text{memo} \times \text{memo} \times a \)

*Example:*

\[ 2^{13} \]

\[ 2^6 \times 2^3 \times 2 \]

\[ 2^4 \times 2^4 \times 2^1 \]

\[ 2^2 \times 2 \]

\[ 2 \]

*Time: \( T(n) = T(\frac{n}{2}) + \Theta(1) = \Theta(n) \).*

*\#multiplications*
Computing Fibonacci Numbers:

- We have seen in the last lecture:
  \[ F_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right] \]
  \[ \text{Time: } T(n) = \Omega \left( \left( \frac{1 + \sqrt{5}}{2} \right)^n \right) \quad \leftarrow \text{Exponential!} \]

- Previous lectures: we’ve seen
  one exponential-time algorithm,
  one linear-time algorithm with memoization,
  and one linear-time algorithm without memoization
  for computing n-th Fibonacci number.

Perpetual Question: Can we do better?

From Homework \( \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \) (proof by induction)

Is it not equivalent to computing \( a^n \)?

Since two 2x2 matrices can be multiplied in \( \Theta(1) \) time,
the time complexity of computing n-th Fibonacci number is:

\[ T(n) = 2T(n/2) + \Theta(1) = \Theta \left( 4^n \right), \]
which follows directly from how we computed \( a^n \) previously.
Matrix Multiplication:

Input: \[
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix} \times \begin{pmatrix}
e & f \\
g & h
\end{pmatrix} = \begin{pmatrix}
(ae+bg) & (af+bh) \\
(cf+dg) & (cf+dh)
\end{pmatrix}
\]

\[A_{n \times n} \times B_{n \times n} = C_{n \times n}\]

\[C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}\]

Algorithm 1: High School Matrix Multiplication Coded

\[\text{MULT}(A_{n \times n}, B_{n \times n})\]

for \(i \leftarrow 1\) to \(n\)

for \(j \leftarrow 1\) to \(n\)

\(c_{ij} \leftarrow 0\)

for \(k \leftarrow 1\) to \(n\)

\(c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}\)

Perpetual Question: Can we do better?

Idea: Try divide-and-conquer.

\[
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}_{n/2 \times n/2} \times \begin{pmatrix}
e & f \\
g & h
\end{pmatrix}_{n/2 \times n/2} = \begin{pmatrix}
(a*e+b*g) & (a*f+b*h) \\
(c*e+d*g) & (c*f+d*h)
\end{pmatrix}_{n \times n}
\]

D-S-C-MULT(\(A, B\))

Exercise for you!

\[\text{Obs:} \ # \text{multiplications} = 8\]
\[\text{#additions} = 4\]

Running Time: \(T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^3)\). \(\circ\)

What can be a possible way of bringing down the running time?

Somehow can we make it 7 or less at the expense of doing more (constant #)

number of additions? \(\circ\)

Volker Strassen showed it is possible in 1969!
Strassen's Matrix Multiplication: (1969)

Idea: Replace a multiplication by additions.

\[
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\times
\begin{pmatrix}
e & f \\
g & h
\end{pmatrix}
=
\begin{pmatrix}
\gamma = ae + bg & \beta = af + bh \\
t = ce + dg & u = cf + dh
\end{pmatrix}
\]

\[
A \times B = C
\]

\[
p_1 = a(f - h) \\
p_2 = (a + b)h \\
p_3 = (c + d)e \\
p_4 = d(g - e) \\
p_5 = (a + d)(e + h) \\
p_6 = (b - d)(g + h) \\
p_7 = (a - c)(e + f)
\]

\[
\gamma = p_5 + p_4 - p_2 + p_6 \\
\beta = p_1 + p_2 \\
t = p_3 + p_4 \\
u = p_1 + p_5 - p_3 - p_7
\]

Total: \#multiplication = 7

\#addition/Subtraction = 18

So intricate that it makes us wonder how Strassen discovered it!

Algorithm:

- Divide: Divide A and B into 8 submatrices \(a_1\ldots h\) of dimension \(\frac{n}{2} \times \frac{n}{2}\).
- Conquer: Recursively multiply the 7 submatrices.
- Combine: Write C by doing the add/sub operations to compute \(r, s, t, u, v\).

Time: \(T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807})\)

Aside: Coppersmith–Winograd bound: \(O(n^{2.376})\)

Recently, in 2011, Vassilevska Williams gave a bound of \(O(n^{2.3727})\).

Trivial lower bound: \(\Omega(n^2)\). Can this gap be closed?