Lecture: Divide and Conquer

- Divide and Conquer
  - Merge Sort
  - Binary Search
  - Prune and Search Method
  - Powering a Number

Note: The time-complexity analysis will be done when we study how to solve recurrences.

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.
- Analysis uses recurrences.

**Algorithm: Divide-and-Conquer**

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

Given: An array $A[1...n]$ of $n$ elements for which $<\geq$ 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: $10 \ 2 \ 5 \ 3 \ 7 \ 13 \ 1 \ 6$

Sorted: $1 \ 2 \ 3 \ 5 \ 6 \ 7 \ 10 \ 13$

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

1. if $p < r$
   - $q = \left\lfloor \frac{p+r}{2} \right\rfloor$
   - $\text{Divide: Trivial}$
   - $\text{MERGE-SORT}(A, p, q)$
   - $\text{MERGE-SORT}(A, q+1, r)$
   - $\text{CONQUER}$
   - $\text{MERGE}(A, p, q, r)$

Time: $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$
MERGE2(A, p, q, B, r, s) // merge two sorted arrays A[p...q] and B[r...s]
1. if p ≥ q return B
2. if r ≥ s return A
3. if A[k] ≤ B[j]
   4. return A[p] o MERGE2(A, p+1, q, B, r, s) // O in concatenation
   5. else return B[r] o MERGE2(A, p, q, B, r+1, s)

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, r)
1. C[p...q] ← MERGE2(A, p, q, A, r+1, r)
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.

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>6</th>
<th>9</th>
<th>10</th>
<th>14</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>$\color{red}{9}$</td>
<td>10</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>$\color{red}{4}$</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>$\color{red}{14}$</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$\color{red}{6}$</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td>17</td>
</tr>
</tbody>
</table>

**Search:** $x = 10$

**Exercise:** Give an iterative version of binary search.

**REC-BIN-SEARCH($A$, $x$, $i$, $j$)**

1. if $i > j$ return "key not found"
2. mid $\leftarrow (i+j)/2$
3. if $A[mid] = x$ 
   - divide: check the middle element
   - return mid
4. else if $A[mid] > x$
   - return REC-BIN-SEARCH($A$, $x$, $i$, mid-1)
5. else return REC-BIN-SEARCH($A$, $x$, mid+1, $j$)

**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 throws away a function 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 sub-section of the input and recurse on the remaining part of the input.
- Example: Binary search, where half of the array is pruned in each step giving us the recurrence
  \[ T(n) = T(n/2) + \Theta(1) \]
- General Prune and Search Recurrence:
  \[ T(n) = T(n^\alpha) + f(n) \Rightarrow T(n) = \begin{cases} 
\Theta(\lg n), & \text{if } f(n) = \Theta(1) \\
\Theta(f(n)), & \text{otherwise}.
\end{cases} \]
  \[ 0 < \alpha < 1 \]
- Cost of pruning: \( 1 - \alpha \)
- Input size for recursive calls: \( n, \alpha n, \alpha^2 n, \ldots, \Theta(1) \).
- 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
  \[ T(n) = \begin{cases} 
\Theta(\lg n), & \text{if } f(n) = \Theta(1) \\
\Theta(f(n)), & \text{otherwise}.
\end{cases} \]
- 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.

**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

**RECURSIVE-POWER($a, n$)**
1. if $n = 1$ return $a$
2. if $n$ is even
   
   $memo \leftarrow$ RECURSIVE-POWER($a, n/2$)
   
   return $memo \times memo$

else  // i.e. $n$ is odd

   $memo \leftarrow$ RECURSIVE-POWER($a, n/2$)

   return $memo \times memo \times a$

**Example:**

$$2^{19} = 2^9 \times 2^4 \times 2^2 \times 2$$

**Time:** $T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$.

# multiplications