# CS515 - Algorithms & Data Structures

# Notes

Vy Bui - 934370552

Instructor: Professor Glencora Borradaile

The School of Electrical Engineering and Computer Science
Oregon State University

# 1 Divide and Conquer

## 1.1 Pattern

There are 3 strategies:

A. Simply divide input to 2 equal parts, then recurse on each side and combine the solutions (merge sort, binary search).

B. **Find a clever way** to reduce the number of subproblems so that we can bypass some of the subproblems. For example, in binary search, at each iteration, we only consider a half of the problem and circumvent the other half. This helps save lots of time.

C. **Find something useful** for dividing the input. For example, quick sort relies on

## 1.2 Proof Skeleton

## 1.3 Common Recurrence

Merge sort:

$$T(n) = 2T(n/2) + O(n) = O(nlogn)$$

Integer Multiplication:

$$T(n) = 3T(\frac{n}{2}) + O(n) = O(n^{log_2 3}) = O(n^{1.59})$$

Median Selection:

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n) = T(\frac{9n}{10}) + O(n) = O(n)$$

# 2 Dynamic Programming

## 2.1 Example Problems

**Problem 1**
**Longest Increasing Sequence**: Given an array of integers, find the longest increasing sequence.

Example 1: $LIS([6,3,9,12,4,7,10,3]) = len([3,4,7,10]) = 4$

Example 2: $LIS([7,8,9,2,3]) = len([7,8,9]) = 3$

### Recursive Formulation

$$LIS(k) = \begin{cases} 0 & k < 0 \\ 1 & k = 0 \\ max() & otherwise \end{cases}$$

### Proof/Explanation

For simplicity, in this section, let $LSA(k)$ denote the subarray of interest only, but not its prefix. This problem's optimal substructure property can be described as follows. The optimal solution to the problem with $A[k:n]$ can be found based on the optimal solution to the problem with $A[k+1:n]$. In particular, assume that we know the optimal solution to the problem of $A[k+1:n]$, adding $A[k]$ to the problem results in two new candidate largest subarrays. First, $A[k]$ can be combined with $LSA(k+1)$ to create a new solution. Note that this comes with the cost of the sum of the prefix of $LSA(k+1)$ because the new solution has to be contiguous. The second candidate is $A[k]$ itself. $LSA(k)$ has the largest sum among these two new candidates and $LSA(k+1)$. Note that the combination solution is prioritized when a tie happens in order to open up opportunities for further combination.

We can use proof by contradiction to prove this algorithm. Assume that there exists a better solution $LSA'(k)$ that is not one of the three candidates, $A[k]$, the combination, and $LSA(k+1)$. First, because $LSA'(k)$ is not $A[k]$, it must be a subarray of $A[k+1:n]$. This makes $LSA'(1)$ the subarray of the largest sum of $A[k+1:n]$ because $sum(LSA'(k)) > sum(LSA(k+1))$ based on our assumption. However, this contradicts another assumption that $LSA(2)$ is the optimal solution to the problem with $A[2:n]$. The proof completes!

**Pseudocode**

Observe that $LSA(k)$ only depends on $LSA(k+1)$, we can implement this algorithm iteratively from $LSA(n)$ to $LSA(0)$. The solution is $LSA(0)$.

**Algorithm 1** $LSA(A[1:n])$

```
last_prefix_sum ← 0
last_largest_sum ← 0
k ← n
while k ≥ 1 do
    combined_sum ← A[k] + last_largest_sum + last_prefix_sum
    max_largest_sum ← max(A[k], last_largest_sum, combined_sum)
    if combined_sum == max_largest_sum then
        last_prefix_sum ← 0
        last_largest_sum ← combined_sum
    else if A[k] == max_largest_sum then
        last_prefix_sum ← 0
        last_largest_sum ← A[k]
    else if last_largest_sum == max_largest_sum then
        last_prefix_sum ← last_prefix_sum + A[k]
    end if
    k ← k − 1
end while
    return last_largest_sum
```

**Runing Time and Space Analysis** There are n iteration, each of which has constant number of operations, hence the algorithm has $O(n)$ time complexity. Furthermore, the algorithm uses $O(1)$ space.

# 3 Greedy Algorithms

## 3.1 Greedy Stays Ahead

Let A be the algorithm's solution and O be the optimal solution.

1. Align A and O in some ways so that we can compare $a_i$ with $o_i$.
2. Use proof by induction to prove that $a_i$ is better than or of the same quality as $o_i$.
3. Use proof by contradiction and (2) to prove the algorithm will eventually lead to the optimal solution, starting off with assume that it won't and come to an contradiction.

Refer to section 4.1 in [KT] for more details.

## 3.2 An Exchange Argument

# References

[KT] Jon Kleinberg and Éva Tardos. Algorithm design.