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 $LSA(A[1:n])$

$last\_prefix\_sum \leftarrow 0$
$last\_largest\_sum \leftarrow 0$
$k \leftarrow n$
while $k \geq 1$ **do**
    $combined\_sum \leftarrow A[k] + last\_largest\_sum + last\_prefix\_sum$
    $max\_largest\_sum \leftarrow max(A[k], last\_largest\_sum, combined\_sum)$
    if $combined\_sum == max\_largest\_sum$ **then**
        $last\_prefix\_sum \leftarrow 0$
        $last\_largest\_sum \leftarrow combined\_sum$
    else if $A[k] == max\_largest\_sum$ **then**
        $last\_prefix\_sum \leftarrow 0$
        $last\_largest\_sum \leftarrow A[k]$
    else if $last\_largest\_sum == max\_largest\_sum$ **then**
        $last\_prefix\_sum \leftarrow last\_prefix\_sum + A[k]$
    end if
    $k \leftarrow k - 1$
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

3.3   Matroids

Matching Matroid

Disjoint Path Matroid: G = (V,E) + fixed vertex $S \subset V$

3.3.1   Cases where Matroids fail

Theorem X ([Eri19], appendix E): For any family of subsets I that do not form a matroid, there exists a weight function for which greedy fails.

Greedy fails on the **Set Cover Problem**: give a family I of subsets of S. Find that smallest subset of I that covers everything in S.

3.3.2   Reference

16.4, [CLRS22]

4   Randomized Algorithms

4.1   Definition

4.2   Application

1. Using randomization to improve run time
2. Guarantee the right solution but run time will be a random variable (Quicksort)
3. Allowed to make mistakes with low probability
4. Efficient testing (tesing sortedness)

4.3   Math

4.3.1   Arithmetic Series

$$a_k = a_0 + nd$$

$$\sum_{i=0}^{n} a_i = \frac{n(a_0 + a_n)}{2}$$

4.3.2   Geometric Series

The name geometric series indicates each term is the geometric mean of its two neighboring terms, similar to how the name arithmetic series indicates each term is the arithmetic mean of its two neighboring terms.

$$a_k = a_0 r^k$$

$$\sum_{i=0}^{n} a_i = 1 + r + r^2 + ... + r^n = \frac{1 - r^{n+1}}{1 - r}$$

For $-1 < r < 1$, the sum converges as $a \to \infty$,

$$\sum_{i=0}^{\infty} a_i = \frac{1}{1 - r}$$

4.3.3   Arithmetico-geometric Series

An arithmetic-geometric progression (AGP) is a progression in which each term can be represented as the product of the terms of an arithmetic progressions (AP) and a geometric progressions (GP).

$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{5}{32} + ... =?$$

The sequence has the form

$$a_n = (a + (n-1)d)r^{n-1}$$

$$S_\infty = \sum_{k=1}^{n} kr^k = \frac{r}{(1-r)^2}$$

4.3.4   Harmonic number

Harnomic series has the form

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$

According to [KT], harmonic number is bounded by $\Theta(logn)$

$$ln(n+1) < H(n) < 1 + ln(n)$$

4.4   Examples

Problem Randomized algorithm for global min cut
Input: undirected G=(V,E)
Find a min sized subset $C \subset E$ such that $G = (V, E - C)$ is disconnected.

Edge Contraction Operation: ???

If you contract $E - C$, then your graph looks like 2 nodes with many edges connecting them, how???

Problem Sortedness Testing
Given a array A of n items, create randomized an algorithm than can check if A has a 99% chance to be sorted in O(logn) time

---

To say with probability of 99% if A is nearly sorted: ignoring an ϵ fraction of elements, rest are sorted.

Idea 1: pick random indices i and j and compare a_i an a_j.

This won't work in case that A = 1,1,1,1,1,1,0,0,0,0,0,0. A is not nearly sorted.

p(success) = k/n for k tests. For P(success) ¿ 0.99, need O(n) tests.

Idea 2: pick 2 random indices $i \leq j$ and say that A not sorted if $x_i > x_j$.

This won't work in case A = [1,0,2,1,3,2,4,4,5], that has a subset of even-indexed numbers or a subset of odd-indexed numbers sorted. We will say A is sorted uness we pick i odd and j = i + 1.

P(success) = low ???

Algorithm 2:
For k iterations:
        Pick 2 items from A
        If they are not in sorted order
        output "not sorted"
        Else
        continue

Refer to "Testing if an Array is sorted".

4.4.1   Reference

Chapter 13, [KT]

References

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[Eri19]   Jeff Erickson. *Algorithms*. 2019.

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