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. The idea is to "Compare the partial solutions that the greedy algorithm constructs to initial segments of O, and show that the greedy algorithm is doing better in a step-by-step fashion", [KT].

Steps:
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 the assumption that it won't and come to an contradiction.

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

Problem Interval Scheduling Problem
Input: a list of intervals I = {(s,e)} with s and e being start and end time respectively. Two intervals overlap when they occur at the same time, excluding start and end time. For example, (1,2) does not overlap (2,3).

Problem: Find the maximum number of non-overlapping intervals in I.

Greedy Algorithm
1. Sort I by start time.
2. Iterate through I, add the interval to the solution if it does not overlap with any intervals. If some intervals overlap, pick the one with the smallest end time.

Let $A = a_1, ..., a_k$ be the solution constructed by the greedy algorithm and $O = o_1, ..., o_m$ be the optimal solution, both of which are sorted by end time. We will prove that $k >= m$.

Lemma 1. $a_i.e < o_i.e$ for any $i \leq k$.
Proof. We prove this by induction. For base case $i = 1$, the statement is clearly true because the algorithm picks that smallest end time.

Induction hypothesis: $a_i.e < o_i.e$ for any $i > 1$.

Induction step: $a_{i+1}.e < o_{i+1}.e$? This is always true because either $a_{i+1}.e$ or $o_{i+1}.e$ is compatible with A, therefore we can always pick o_{i+1} if it has smaller end time. Proof completes!

Theorem. The greedy algorithm returns an optimal set A.
Proof. Assume that A is not optimal, followed by $m > k$. Since $m > k$, there must be some o_{k+1} in O that starts after $o_k.e$, and hence after $a_k.e$ (Lemma 1). Therefor, o_{k+1} is compatible with the current set A, and the algorithm would have picked it. This contradicts the assumption that A is not optimal. Proof completes!

Problem Minimum interval removals for non-overlapping
Input: a list of intervals I = {(s,e)} with s and e being start and end time respectively. Two intervals overlap when they occur at the same time, excluding start and end time. For example, (1,2) does not overlap (2,3).

Problem: Find the minimum number of intervals in I that need to be removed so that I is non-overlapping.

Greedy Algorithm
1. Sort I by end time.
2. Iterate through I. If the interval overlaps with the previous interval, remove the one with later end time. Otherwise, keep it.

3.2 An Exchange Argument

Let A be the algorithm's solution and O be the optimal solution. Gradually modify O, **preserving its optimality** at each step, but eventually transforming it into A.

1. Derive some useful properties from the greedy choice
2. Assume that there exists an O solution without these properties. Gradually tweak O until it has these properties. Prove that each tweak results in another optimal solution.

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

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

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

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 $\Theta(logn)$

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

4.3.5 Probability

Theorem 13.7
If we repeatedly perform independent trials of an experiment, each of which succeeds with probability $p > 0$, then the expected number of trials we need to perform until the first success is 1/p [KT].

Theorem 13.8
Linearity of Expectation. Given two random variables X and Y defined over the same probability space, we can define X + Y to be the random variable equal to $X(\omega) + Y(\omega)$ on a sample point ω. For any X and Y, we have $E[X + Y] = E[X] + E[Y]$ [KT].

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.