1A Lent Algorithms
Supervision 2: Algorithm Design

Hayk Saribekyan

February 7, 2019

Where you are asked to describe an algorithm, also analyse its time and space complexity. Mention how you would implement the solution and give recurrence relations for dynamic programming algorithms. You should also always prove that your algorithms are correct. This is, in particular, important for greedy algorithms.

In this supervision we will concentrate on few important approaches for algorithm design: dynamic programming and its implementation techniques, greedy algorithms and their proofs, divide and conquer.

1 Refresher and Warm-up

1. What does the statement "The running time of an algorithm is at least $O(n)$" mean, if anything?

   Solution This is problem 3.1-3 from CLRS3. The O-notation gives a an upper bound for a function and the statement is relevant for lower bounds. It does not make any sense. Either Ω or *'at most'* should be used

2. Show that for any two real a and $b > 0$

   $$(n + a)^b = \Theta(n^b)$$

   Solution This is problem 3.1-2 from CLRS3.

   When $n \geq a$, $n^b \leq (n + a)^b \leq (2n)^b$, thus $(n + a)^b = \Theta(n^b)$.

   Notice that we cannot use binomial expansion because b is real.

3. Write a pseudocode of a binary search algorithm that finds element x in a sorted array a_i in $O(\log n)$ time.

```
l = 0;
r = n - 1;
while (l < r) {
    m = (l + r) / 2;
```

```
        if (a[m] <= x)
            l = m + 1;
        else
            r = m;
    }
    if (a[l] == x)
        return l;
    else // not found
```

Pay attention on how l and r are modified. In binary search always think of the base cases when $r - l = 0$ or 1.

4. When would you prefer dynamic programming using tabulation vs memoisation and vice versa? Give an example of a problem, which can be decomposed into smaller instances of itself but no DP is necessary.

   Solution Tabulation (bottom-up) makes sense if all the subproblems have to be calculated eventually and there is a convenient order (e.g. knapsack). When there is no repetition of subproblems DP does not make sense.

5. You are playing a game on a map represented by an $n \times n$ matrix. You start in the upper left corner, and your objective is to reach the bottom right corner. At each step, you are only allowed to go right or down. Each cell contains some amount of coins. Explain the algorithm you would use to determine the path that will maximise your number of coins.

   Below is an example of an optimal path:

1	3	5	3	1
4	2	5	3	4
2	2	2	4	5
4	4	1	5	3
5	1	2	3	1

   Credit: this problem is from Petar Velickovic's notes.

   Solution Use DP. Let the matrix be called a. Let d_{ij} be the maximum score we achieve for the submatrix $a[0..i, 0..j]$. Then

$$d_{ij} = max\{d_{i-1,j}, d_{i,j-1}\} + a_{ij}$$

   The answer is in the last element of the d.

2 Examples

1. The Knapsack problem. The hiker has to choose some of the n items to take to a trip in his knapsack of capacity W pounds. Item i weighs w_i pounds and has a value v_i.

   (a) Describe an algorithm that will maximize the value the hiker can carry.

Solution Let $d_{i,m}$ be the maximum value we can get with a *full* backpack of size m using the first i items. Then, omitting the base cases,

$$d_{i,m} = \max\{d_{i-1,m}; d_{i-1,m-w_i} + v_i\}$$

In the first case we assume item i was not used, in the second case we assume it was used. The answer is $\max_{m=0,\ldots,W} d_{n,m}$.

(b) Modify your solution for the case when the hiker has infinite supply of each item.

Solution Now, let $d_{i,m}$ denote the same thing as before. Then

$$d_{i,m} = \max\{d_{i-1,m}; d_{i,m-w_i} + v_i\}$$

Note that here the second term is $d_{i,m-w_i}$ instead of $d_{i-1,m-w_i}$. Questions: why does the first solution not allow repetitions and this one does?

(c) Give an example, where a simple greedy algorithm that picks items with largest v/w ratio does not work.

Solution Two items: $v_1 = 10$, $w_1 = 2$, $v_2 = 12$, $w_2 = 3$. The backpacks size is $W = 3$. The first item has larger value to weight ratio, but we should pick the second one.

(d) Can you solve the problem using just $O(W)$ additional memory?

Solution Notice that we never need to use more than two rows of d, so we can use two arrays of size W that will act alternatively as d_i and d_{i-1}.
Question: Can you do the same using only one array of size W? Do (b) first and then (a).

2. The shoemaker has to complete n orders. Order i takes t_i days. For every day of delay of order i, the shoemaker gets fined s_i pence. Unfortunately, the shoemaker cannot work on more than one order a day. Write an algorithm that will help the shoemaker decide the order in which he should do the jobs so that the total amount of fine is minimised.

   If you come up with a simple solution make sure to prove its correctness.

   Solution `http://algorithmist.com/index.php/UVa_10026`

3. Given an array of integers. Implement a divide-and-conquer algorithm that returns the maximum sum of a contiguous subsequence of the integers. Your algorithm should run in $O(n \log n)$ time or better.

   Solution `https://en.wikipedia.org/wiki/Maximum_subarray_problem`. See the Divide and Conquer method, which is not the best but is instructive. For the most practical algorithm, look at the other algorithms in Wikipedia.

4. I have n supervisions to schedule. Supervision i starts at time a_i and ends at time b_i. Help me find the minimum number of supervisions that I will have to reschedule so that I do not have any overlapping ones. It would be wonderful if your algorithm could also give me the list of the supervisions that should be rescheduled!

Solution

A greedy attempt 1: Keep removing the supervision that conflicts with largest number of other supervisions. Consider the following supervisions: $[0, 3]; [2, 5]; [4, 7]; [6, 9]; [8, 11]$ i.e. i conflicts with $i - 1$ and $i + 1$.

The middle three supervisions each have 2 conflicts, so the greedy algorithm can remove any of those. Let's say it removes $[4, 7]$. In this case, we will still have to reschedule 2 supervisions, making the total number 3. But we could remove just two of the supervisions and have no conflicts.

Notice that the problem started from the fact that the greedy algorithm could have multiple 'best local steps' and then it picked a bad one among those. This is a common way to find counterexamples to greedy algorithms: think about what would your algorithm do if it had multiple choices and picked an arbitrary one. Of course, this is not the only way greedy algorithms fail.

A greedy algorithm (correct): There is, however, a greedy algorithm that works. Assume that $b_{i-1} < b_i$. Then it can be proven that we can always keep the first supervision (prove it). This results in a simple algorithm: (1) keep the first supervision, (2) remove all the supervisions that overlap with it, (3) repeat.

We need $O(n \log n)$ time for sorting and $O(n)$ time for the greedy approach.

A DP solution: Once again assume that $b_{i-1} < b_i$ (we can sort in $O(n \log n)$ time). It is easier to think about the maximum number of supervisions that we can keep, rather than the minimum number of rescheduled ones. Let d_k be the maximum number of the supervisions that can keep if only the first k were given (ignore supervisions $k+1, \ldots, n$). We have two options

- Reschedule k: then $d_k = d_{k-1}$.
- Keep k as is: then $d_k = d_{j_k} + 1$, where $j_k = \max_{b'_j < a_j} j'$ i.e. j_k is the last supervision that ends before k starts.

Thus

$$d_k = \max\{d_{k-1}, d_{j_k} + 1\}$$

for j as defined above. The answer is d_n (the answer to the original problem is $n - d_n$).

We can find j_k using binary search, therefore the algorithm runs in $O(n \log n)$ time overall.

Note: it helps to sort segments on problems such as this one.

5. Given n line segments on an axis described by their endpoints a_i and b_i. Describe an algorithm that picks the smallest number of them so that their union covers the interval $[0, t]$.

Solution If a segment $[0, x]$ is covered, then we should pick a segment k with largest b_k given $a_k \leq x$. As a result we cover the segment $[0, \max\{x, b_k\}]$.

To implement this efficiently we can sort the segments according to a_i. And do one pass over the array updating x as we go.

6. A string is called palindrome if it reads the same from both ends e.g. `madam`. Given a string of length n. Describe an algorithm that determines the size of its longest palindrome

subsequence? The subsequence does not have to be contiguous. Example: if the string is `baobab`, then we can remove `o` and get `babab`, which is palindrome.

Explain how can we find this subsequence.

Hint: what can you say if the first and last letters of the string are the same. What if they are different?

Solution Problem source: `https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1558`.

Let the string be called s and let d_{ij} be the length of the longest subpalindrome of the string $s[i \dots j]$. There are three cases:

- If $s_i = s_j$, then $d_{ij} = d_{i+1,j-1} + 2$, because in this case if the best sub-palindrome does not contain both s_i and s_j, we can modify it to contain without making it shorter.
- Otherwise either s_i or s_j is not in the best sub-palindrome of $s[i, \dots, j]$. Then $d_{ij} = \max\{d_{i+1,j}, d_{i,j-1}\}$.
- The base cases are trivial.

The DP implementation can either use memoisation or fill in a matrix in a diagonal order.

There is another DP solution (found by Scarlet Cox). The longest subpalindrome's length is equal to the longest common subsequence of the given string and its reverse. The longest common subsequence of two strings can be found using DP in $O(n^2)$ time. Of course, the above statement needs proving.