CSCI211: Problem Set 2

Released 2025

Problem 1: Recurrence.

Suppose these are three algorithms which solve the same problem:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the those results in linear time.
2. Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining those results in linear time.
3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time

What is the recurrence for each algorithm? What are the running times of each of these algorithms (in asymptotic notation) and which is the fastest?

Solution 1

The following are the recurrences and the solutions to the recurrences.

- Algorithm A: $T(n) = 5T(n/2) + \Theta(n)$.

  This recurrence falls into Case 1 of the Master Theorem, since $f(n) = n$ is $O(n^{\log_2 5} - \epsilon)$ for some $\epsilon > 0$. Thus, $T(n)$ is $O(n^{\log_2 5})$
- Algorithm B: $T(n) = 2T(n-1) + \Theta(n)$

  This recurrence can be drawn as a recursion tree where the work in each level is increasing geometrically: $n, 2(n-1), 4(n-2), 8(n-3)\ldots$. From this, $T(n)$ is equal to the number of leaves at the base of tree of height n with a branching factor of 2, which is $O(2^n)$.
- Algorithm C: $T(n) = 9T(n/3) + \Theta(n^2)$

  This recurrence falls into Case 2 of the Master Theorem, since $f(n) = n^2$ is $\Theta(n^{\log_3 9}) \to \Theta(n^2)$. Thus, $T(n)$ is $\Theta(n^2 \log n)$.

Of the three, Algorithm C has the fastest running time since Algorithm B is exponential, and Algorithm A has a larger polynomial power than Algorithm C.

Problem 2: 2D Peak Finding.

Consider array A, a $n \times n$ array of distinct integers in arbitrary order. For example:

23	20	15	**58**
-15	32	30	45
16	**42**	34	**50**
39	5	-2	35
20	2	6	31

Let $A_{i,j}$ be a peak if it is larger than the 4 entries to its top, bottom, left and right.

For the entries at the edge of the array, where one of these 4 entries may not be available, only the available entries matter. In the example above, all the bold entries are peaks.

Describe an efficient algorithm which finds *one* peak in the array and returns its location. Make sure to analyze the running time.

Hint: The best possible running time is $\Theta(n)$, but you do not have to get that running time as *that algorithm* is quite intricate.

Solution 2

Let the 2D array A be notated as $A[1 \dots n, 1 \dots n]$.
Observation: Consider any row r of the input and let the maximum element within the row be item $A[r, M]$. Then, if $A[r, M]$ is not a peak, either $A[r-1, M]$ or $A[r+1, M]$ must be larger than it.

- If $A[(r-1), M]$ is larger, then there exists a peak within $A[1 \dots (r-1), 1 \dots n]$
- If $A[(r+1), M]$ is larger, then there exists a peak within $A[(r+1) \dots n, 1 \dots n]$
- In either case, it is a 2D Peak Finding problem with a reduced number of rows (but not columns!)
- Note that we do need to use the maximum value within a row in order to guarantee the above property

This leads to the following algorithm:

- We let $A[r_i \dots r_j, c_i \dots c_j]$ denote the slice of the array from row r_i to r_j and column c_i to c_j
- We choose the middle row and find the maximum, then check whether it is a peak to "eliminate" roughly half of the rows each time.

Algorithm 1 2D Peak Finding v1

procedure PEAK2D($A[r_i \dots r_j, 1 \dots n]$)
    if $r_j == r_i$ **then**
        Only one row.
        Loop through $A[r_i, 1 \dots n]$ and find the maximum, this is a peak.
    else
        $m \leftarrow \lfloor (r_j + r_i)/2 \rfloor$
        Loop through $A[m, 1 \dots n]$ and find the maximum of the row, let this be $A[m, \overline{x}]$
        if $A[m, \overline{x}] < A[m-1, \overline{x}]$ **then**
            Peak2D($A[r_i \dots (m-1), 1 \dots n]$)
        else if $A[m, \overline{x}] < A[m+1, \overline{x}]$ **then**
            Peak2D($A[(m+1) \dots r_j, 1 \dots n]$)
        else
            $A[m, \overline{x}]$ is a peak, so we are done.
        end if
    end if
end procedure

Running Time

- Let the input array be size $n \times n$. In each recursive call, we reduce the number of rows by a factor of 2, but the number of columns are never reduced.
- The procedure to find the maximum of a row takes $O(n)$ time always.
- The recurrence for an input of size $n \times m$ is $T(n, m) = T(n/2, m) + \Theta(m)$. Here, we carefully use separate sizes for rows and columns since the number of columns is never reduced by the algorithm.
- This recurrence solves to $\Theta(m \log n)$; basically do a $\Theta(m)$ procedure for $\log n$ times.
- The overall running time for an input of size $n \times n$ is then $\Theta(n \log n)$.

Sketch of the $\Theta(n)$ Time Algorithm

- The flaw in the previous algorithm is that we never reduce the number of columns, therefore, we should reduce the number of columns as well!
- The same logic for reducing the number of rows applies to the columns - we just need to find the maximum item in a column and check whether it is a peak.
- This, it turns out, leads to a simple but intricate modification.
  1. Suppose we are considering the middle row m of the input and found the maximum element within the row to be $A[m, \overline{x}]$.
  2. Suppose WLOG that $A[m, \overline{x}] < A[m-1, \overline{x}]$, then we know there exists a peak within $A[1 \ldots (m-1), 1 \ldots n]$.
  3. Now, instead of recursively solving this problem **immediately**, we instead find the maximum element in the middle **column** m_c of $A[1 \ldots (m-1), 1 \ldots n]$.
     - Note we only consider the element in rows $1 \ldots (m-1)$.
     - We check whether this element is a peak, if not, then we reduce the problem to either finding a peak in $A[1 \ldots (m-1), 1 \ldots (m_c - 1)]$ or $A[1 \ldots (m-1), (m_c) + 1, \ldots n]$.
     - This is a problem of half the size in both rows and columns, though we pay an additional $n/2 \to \Theta(n)$ time in finding the maximum in (half of a) column.
- The recurrence input of size $n \times m$ input becomes $T(n, m) = T(n/2, m/2) + \Theta(n + m)$, which solves to $\Theta(m + n)$.
- For input of size $n \times n$, this is then $\Theta(n)$.

Problem 3: Pancakes.

Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that the smaller pancakes are on top of larger pancakes.

The only allowed operation is a *flip* - insert a spatula under the top k pancakes (for some integer k between 1 and n), and flip them all over. I.e. Reversing the order.

Describe an algorithm to sort an arbitrary stack of n pancakes using $O(n)$ flips.

Fun Trivia: A $5n/3$-flips algorithm is given and a $15n/14$-flips lower bound (for any algorithm) is given in the paper *Bounds for Sorting by Prefix Reversal*[1]. Note the two bounds do not match. In 2011, it was shown that finding the optimal number of flips for any input size n is NP-Hard.

Solution 3

Observation: Once we flip the largest pancake to the bottom of the stack, we can treat the remaining problem as sorting the $n-1$ remaining pancakes while essentially ignoring the bottom pancake. Since that is one less pancake, we can recursively solve that problem.

Consider the following algorithm:

Pancakes($A[1...n]$)

1. If $n = 1$, then there is nothing to do.
2. Otherwise:
   (a) Identify the location of the largest pancake, let this be position x.
   (b) Flip(x), this will move the largest pancake to the top of the pile.
   (c) Flip(n), this will move the largest pancake to the bottom of the pile.
   (d) Pancakes($A[1 \ldots (n-1)]$)

[1] By William H. Gates and C. H. Papadimitriou, 1979. This is Bill Gates's only Computer Science research publication.

Number of Flips

In the worst case, each pancake is flipped twice (once to put to top, once to put to the bottom). Therefore, there are no more than $2n$ flips, which is $O(n)$.