CSE 101 Homework 6

Brian Masse, Taira Sakamoto, Emily Xie, Annabelle Coles

November 25, 2025

1. **Recurrence Relations**

(a) $T(n) = 4T(\lfloor n/4 \rfloor) + cn^3, n \geq 2, T(1) = c$
The master theorem applies.

$$\begin{aligned} T(n) &= 4T(\lfloor n/4 \rfloor) + cn^3 \\ &\in 4T(n/4) + O(n^3) \end{aligned}$$

$a = 4, b = 4, d = 3 \implies a < b^d \implies T(n) \in O(n^3)$

(b) $T(n) = 2 * T(n-3)$ for $n \geq 4$, $T(1) = c$
The master theorem does not apply.

$1 : 2T(n-3)$
$2 : 2 \cdot 2T(n-6)$
$\vdots k : 2^k \cdot T(n-3k)$

Solving for the value of k that yields $T(1) : n - 3k = 1 \implies k = 1/3(n-1)$. Thus,

$$T(n) = c \cdot 2^{1/3(n-1)} \in O(2^{n/3})$$

(c) $T(n) = T(\lfloor n/4 \rfloor) + c, T(1) = c$
The master theorem applies.

$$\begin{aligned} T(n) &= T(\lfloor n/4 \rfloor) + C \\ &\in T(n/4) + O(1) \end{aligned}$$

$a = 1, b = 4, d = 0 \implies a = b^d \implies T(n) \in O(n^d \cdot logn) = O(logn)$

(d) $T(n) = (\log n)T(\lfloor n/2 \rfloor), n \geq 2; T(1) = c.$
The master theorem does not apply. ($logn$ not constant). Solving via Recurrence:

$1 : (logn) \cdot T(\lfloor n/2 \rfloor)$
$2 : log(n) \cdot log(\lfloor n/2 \rfloor) \cdot T(\lfloor n/4 \rfloor)$
$\vdots$
$k : \prod_{i=0}^{k-1} (log(\lfloor n/2^i \rfloor)) \cdot T(\lfloor n/2^k \rfloor)$

$n/2^k = 1/ \implies 2^k = n \implies k = log_2(n)$

$$
\begin{aligned}
T(n) &= \prod_{i=0}^{log_2(n)-1} log(n/2^i) \cdot c \\
&= \prod_{i=0}^{log_2(n)-1} (logn - i \cdot log(2)) \cdot c \\
&= \prod_{i=0}^{k-1} ((k-i) \cdot log(2)) \cdot c \\
&= k! \cdot (log2)^k \cdot c \\
&= (log_2(n))! \cdot (log2)^{log_2(n)} \cdot c \\
&\in O(log_2(n)! \cdot (log2)^{log_2(n)})
\end{aligned}
$$

2. **High Frequency Element**

Say we have an array of numbers $A[1...n]$. x is *high frequency* in A if it occurs more than $n/2$ times in A, i.e., there are strictly more than $n/2$ many indices $1 \leq i \leq n$ with $A[i] = x$. Here's an algorithm that finds a high frequency element if one exists (it may return an element if no high frequency element exists):

HF(A[1..n])

(a) If $n = 0$ return "No HF element".

(b) If $n = 1$ return $A[1]$.

(c) If n is odd do:

(d) $Count = 0$

(e) FOR $I = 1$ to n IF $A[I] = A[n]$ THEN $Count++$

(f) IF $Count > n/2$ THEN Return $A[n]$

(g) Initialize an array $B[1..\lfloor n/2 \rfloor]$.

(h) $I = 1, J = 1$.

(i) While $I < n$ do:

(j) IF $A[I] == A[I+1]$ THEN $B[J] = A[I], J++$

(k) $I = I + 2$.

(l) Return $HF(B[1..J-1])$.

Questions:

(a) Give the recursive calls the algorithm would make on input $A[1..7] = (1, 4, 3, 3, 2, 3, 3)$.

Call 1:

- Arr $= [1, 4, 3, 3, 2, 3, 3]$
- n $= 7$
- count $= 4$

$4 > floor(7/2) = 3 \implies return3;$

(b) Prove that if x is a high-frequency element in $A[1..n]$, then x is a high frequency element in $B[1...J-1]$ at the end of the loop.

The only way to reach $J++$ is to have a consecutive pair in A.

There must be at least 1 pair $(x, x) \in A$:

- If n is even:
  the number of times x appears in a, $|x| > \lfloor n/2 \rfloor = n/2$. Thus you cannot put a $y \neq x$ between every x, and thus there is a consecutive pair (x, x)

- If n is odd:
  If there is a space between every occurrence of x, then $A[1] = x = A[n]$, which would have been detected in the count section of the algorithm and returned.

  Otherwise ($A[n] \neq x$), you cannot space apart $\lfloor n/2 \rfloor + 1$ xs in $\frac{n-1}{1}$ spaces

Therefore B will always have at least 1 x in it. The only other way to add an element to B is if there is another consecutive pair, (y, y) such that $y \neq x$. However, for any (y, y) there exists at least another (x, x)

Consider placing the required (x, x) at the beginning of A:

- Place a (y, y). There are now $n - 4$ remaining spots
- For x to be a highest frequency item, you must place $\geq n/2 - 1$ more xs.
- $\frac{n-4}{2} = \frac{n}{2} - 2 < n/2 - 1$

Thus, you must fill more than half of the remaining spots with x, meaning that there must be another consecutive pair of xs. Therefore, for every added y in B, there is another x in B. So at most $A \ni |(x, x)| > |(y, y)| \implies B \ni |x| > |y|$

Therefore x is the highest frequency in B as well.

(c) Use this to prove that if x is a high-frequency element in $A[1..n]$, then $HF(A[1..n]) = x$.

Note that every recursive call has a smaller B: $|B_{i+1}| \leq 1/2|B_i|$. Thus B is always decreasing, and by the previous question, x is always the highest frequency item in B. During each iteration there are several possibilities:

- $|B_i|$ is odd and $B.last() = x$:
  return x.
  This cannot return any other value other than x, since no other value appears over half the time (by problem statement)

- $|B_i|$ is even or $B.last() \neq x$
  Iterate to the next B_{i+1}. By the previous question, if x is the highest frequency item in B_i, then B_{i+1} must have at least 1 item, namely x.

Thus after each recursive call, B must get smaller, can never be empty, and must contain x. The only value the function is able to return is x, which happens in the smallest (end) case, of $B = \{x\}$

(d) Give a worst-case time analysis for this algorithm.

The time analysis can be solved via the master theorem. First, each call only takes $O(n)$ time, since, at worst, it loops through every element twice (once for cunt, and the second to construct B.). Then:

- Each iteration only spawns 1 sub problem
- Each subproblem is, at worst $n/2$.
  This happens when the array only has consecutive pairs, and thus adds an element to B.

Thus,

$$T(n) = 1 \cdot T(n/2) + O(n)$$

By the master theorem: $a = 1, b = 2, d = 1 \implies a < b^d \implies T(n) \in O(n)$

3. **Weighted Median**

Say that we are given a list of pairs of values and weights, with weights greater than 0, $(v_1, w_1), ..(v_n, w_n)$. Let $W = \sum_{1 \leq i \leq n} w_i$ be the total weight. The weighted median is a value v_j with $\sum_{1 \leq i \leq n, v_i < v_j} w_i \leq W/2$ and $\sum_{1 \leq i \leq n, v_i > v_j} w_i \leq W/2$.

For example, if the list had elements $(2, .1), (4, .2), (-3, .2), (1, .4), (5, .1)$,we can compute $W = 1$, so $W/2 = .5$. A weighted median is 1, because the sum of the weights of values less than 1 is .2 and the sum of the weights larger than 1 is $.1 + .2 + .1 = .4$.

Give an efficient algorithm (faster than sorting the list) to compute a weighted median.

```
// pass in the list of pairs, the size of the list, and half the total weight
// The final piece can be precomputed in linear time.
weighted_median(list L, size n, WM):

    // Find the median of the given list (by value, not weight)
    // Using the select algorithm from class
    median = select_lowest(L, n / 2)

    // create 3 sub array for pairs with a value <, =, > the median
    // keep track of their cumulative weight
    SL = []; WL = 0;
    SM = []
    SR = []; WR = 0;

    // loop through the list to build sub arrays.
    for pair in L:
        if pair.value < median.value:
            SL.add(pair)
            WL += pair.weight
        elif pair.value > median.value:
            SR.add(pair)
            WR += pair.weight
        else:
            SM.add(pair)

    // if both sides are less than the target weight, that value is the weighted median
    if (WL < WM && WR < WM):
        return median

    // run the algorithm recursively on whichever subset was greater than the target
        weight
    if (WL > WM):
        return weighted_median(WL, WL.size(), WM)
    else:
        return weighted_median(WR, WR.size(), WM)
```

Short Correctness Proof

The algorithm recursively calls the select median algorithm discussed in class. It is assumed that, for any set, it correctly returns the median. Once the median is found, the algorithm uses it as a pivot to create 3 partitions: values less than the median, equal to the median, and greater than the median, and calculates the total weight of each partition.

There are 2 outcomes:

- The weight of both subsets are less than $W/2$. This implies the median is the weighted median and returns
- The weight of one of the subsets is greater than $W/2$. In this case, the weighted median must be in the heavier subset, since being in either other subset implies values above or below it sum above $W/2$, violating the definition of the weighted median. Thus, the algorithm runs itself on the smaller subset.

  Note: Only 1 subset can have a weight $> W/2$, since all others must have weight $w < W/2$ to all sum to W

  Note: A recursive call will never contain an empty set. If there are no numbers above or below the median returned by select, that number is the weighted median.

Analysis / Efficiency

This algorithm can be determined using the master theorem. First note:

- Calculating target weight $\in O(n)$
- Select (from class) $\in O(n \cdot log(n))$
- partitioning into subsets $\in O(n)$
- The partitions will always be roughly $n/2$ elements large, since they use the median as the pivot, which by definition is in the center of the set.

Together this implies the total time complexity, $T(n)$:

$$T(n) = 1 \cdot T(n/2) + O(n \cdot log(n))$$

(a) Only 1 subset is recursively called on $\implies$ 1 subroutine
(b) With median as input, subroutines are roughly $n/2$
(c) The total time complexity of each call is dominated by the select median $\implies \in O(n \cdot logn)$

By the master theorem:

$$\begin{aligned} f(n) &= n \cdot log(n) \\ &\in \Omega(n^{log_b(a)}) = \Omega(1) \\ &\implies T(n) \in O(f(n)) \end{aligned}$$

Thus, the total runtime complexity is $O(n \cdot logn)$

4. **Weighted Independent Set for Trees**

A subset S of vertices in an undirected graph is an *independent set* if it doesn't contain both ends of any edge. If we assign every vertex a weight $w(x) > 0$, the *maximum weight independent set* is to find the independent set of the graph that has maximum possible total weight of its vertices. While the maximum weight independent set problem is NP-hard, some special cases can be solved efficiently. In particular, consider the special case when the underlying graph is a complete binary tree of depth k, so $|V| = 2^{k+1} - 1$. Give a divide-and-conquer algorithm for this problem that runs in polynomial time. (5 points clear algorithm description, 5 points for short correctness argument, 5 points for correct time analysis, and 5 points for efficiency; the best algorithm is $O(|V|)$ time.)

```
// pass in the starting node. In the specialized problem, this will always be the
   root.
// The function returns 2 things:
//   1. The highest weight independent set with the root
//   2. The highest weight independent set without the root
// The answer is the set with the larger weight at the end.
def WeightedSets(node):

    if node has no children:
        return ( [node], [] )

    if node has 1 child:
        return ( [node], [child] )

    left_result = WeightedSets(leftChild)
    right_result = WeightedSets(rightChild)

    with_left = left_result[0]; without_left = left_result[1]
    with_right = right_result[0]; without_right = right_result[1]

    with_node = [], without_node = []

    // Generate the optimal set containing the current node
    with_node += without_left + without_right + node

    // generate the optimal set without the current node
    if weight(with_right) > weight(without_right):
        without_node += with_right
    else:
        without_node += without_right
    if weight(with_left) > weight(without_left):
        without_node += with_left
    else:
        without_node += without_left

    return with_node, without_node
```

```
// calculate the actual maxiumum independent set
result = WeightedSets(root)
max_set = max(weight(result[0]), weight(result[1]))
```

Short Correctness Proof

This algorithm can be proved via induction on the size on the input.

Base Case:

- n = 1.
  There is only one node. The algorithm returns a set containing the node and an empty set. The non-empty set is trivially the max independent set.
- n = 2
  There are 2 nodes, 1 with 1 child. The algorithm returns 2 sets each with one of the nodes. The set with the bigger weight is trivially the max independent set.

Induction step:

Assume that for any tree of size $n-1, n > 2$ the algorithm returns the largest sets with and without the root node. Then for an input of size n:

- Returning the largest set with the node
  In this case, it is not possible to include either of the node's children, since that would complete the edge. Given that all weights are positive, the largest weight this set could have is te union of $without\_left$ and $without\_right$
- Returning the largest set without the node
  In this case, it is possible to include or not include either of the node's children. Thus the answer will be the union of the biggest set on the left (with or without the left child), and the biggest set on the right (with or without the right child)

Thus for an input of size n, the function returns the largest set for both cases, and thus the largest independent set.

Time Analysis / Efficiency

The time complexity of this problem can be solved via the master theorem. Note:

- The runtime of each subroutine is $O(1)$.

  Note: It may be more than O(1) to find the weight of a set. However, it is trivially easy modify the algorithm to build up and return the weight of the 2 sets with each function call to avoid having to compute the weight from the set itself.

- Each call generates, at most, 2 sub problems.
- Each sub problem is roughly $n/2$ big.

This implies that total runtime, $T(n)$ is:

$$T(n) = 2 \cdot T(n/2) + O(1)$$

By the master theorem: $2 = a > b^d = 2^0 = 1 \implies T(n) \in O(n^{log_b(a)}) = O(n^1) = O(n) = O(|V|)$

5. **Karatsuba implementation**

This problem is designed to teach you an idea that makes divide-and-conquer algorithms that only improve time for huge inputs into ones that give substantial improvements even for moderate sized inputs.

Implement both the Karatsuba method for multiplying polynomials from the ungraded problems above and a straight-forward $O(n^2)$ polynomial multiplication algorithm. Collect average running times for a wide range of input sizes n, say powers of 2 until the algorithm is taking over 15 minutes to run, for random polynomials with co-efficients 0 or 1. Graph these on a log-log scale, log n vs. log (algorithm time). How big are inputs where the Karatsuba method is faster, or how big would you interpolate such inputs to be from your data (if there are no actual data points where Karatsuba is better)? Then try a hybrid algorithm, where we use the Karatusba recursion when $n > T$ and the quadratic time method in recursive calls when $n \leq T$. For a wide range of possible T's, starting with rather small values such as $T = 16$, graph the hybrid's performance and compare to the other two. What do you conclude from this experiment?

(2 points clearly describing features of implementation and experiment, such as PL, libraries, type of computer, etc. For each of Karatsuba, quadratic algorithm and hybrid, 2 points for running experiment with adequate variety of input sizes and 2 points for clearly presenting results. 6 points for conclusions based on comparing results.)

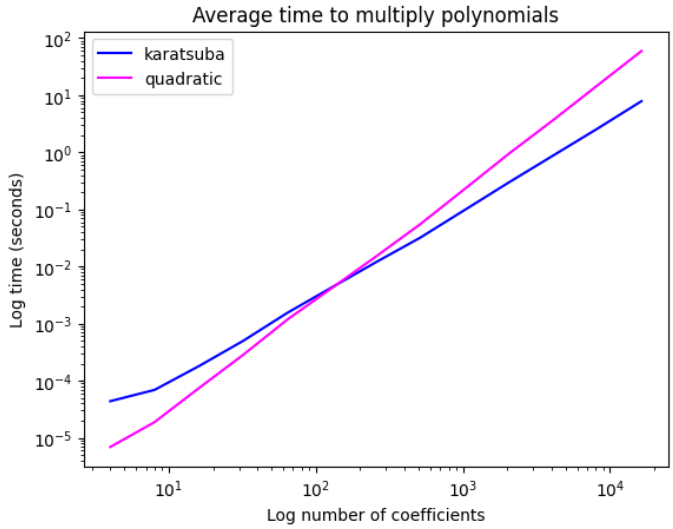
Programming language: Python 3.11. Packages: Numpy, Matplotlib, Time. Computer: Macbook pro.

Functions:

```
def quadratic_time_algorithm(A, B, n):
    start = time.time()
    out = np.zeros(2*n-1)

    for i in range(n):
        for j in range(n):
            out[i+j] += A[i]* B[j]
    time_taken = time.time() - start
    return out, time_taken

def karatsuba_algorithm(A, B, n):
    start = time.time()
    if n == 1:
        return [A[0] * B[0]], 0

    size = int(n//2)

    p_h = A[:size]
    p_l = A[size:]

    q_h = B[:size]
    q_l = B[size:]
```

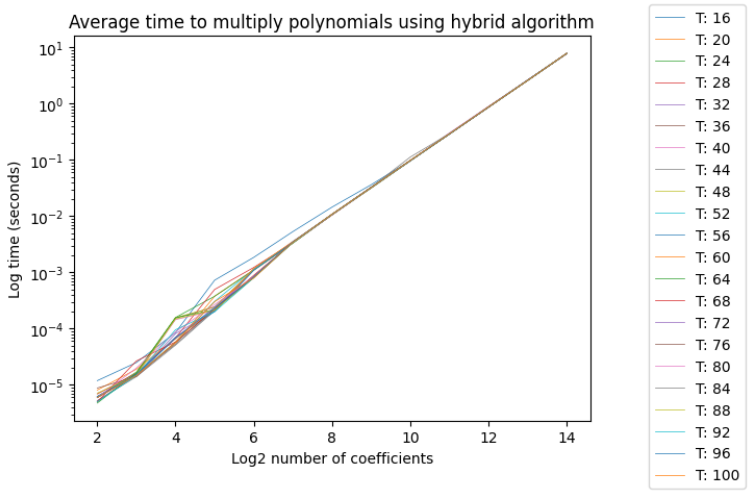
```
    r_x, t = karatsuba_algorithm(p_h, q_h, size)
    s_x, t = karatsuba_algorithm(p_l, q_l, size)
    t_x, t = karatsuba_algorithm(([p_h[i] + p_l[i] for i in range(size)]), ([q_h[i] +
        q_l[i] for i in range(size)]), size)

    r_3 = [t_x[i] - r_x[i] - s_x[i] for i in range(len(t_x))]

    out = np.zeros(2*n-1)

    for i, val in enumerate(r_x):
        out[i] += val

    for i, val in enumerate(r_3):
        out[size + i] += val

    for i, val in enumerate(s_x):
        out[2 * size + i] += val

    time_taken = time.time() - start

    return out, time_taken


def hybrid_algorithm(A, B, n, T):
  if (n <= T):
    return quadratic_time_algorithm(A, B, n)
  else:
    return karatsuba_algorithm(A, B, n)
```

To quantify the runtimes, we ran each algorithm with array sizes with powers of 2 in range $(2^4, 2^{14})$. Graph comparing Karatsuba and quadratic multiplication algorithms:



Once array lengths are above around 300 elements, the Karatsuba algorithm performs faster than the quadratic algorithm.

Graph comparing Hybrid multiplication algorithm with different T thresholds:



From this experiment, it shows that with a smaller number of coefficients ¡ 300, the quadratic algorithm $O(n^2)$ performs faster than Karatsuba. However, once the number of coefficients exceeds this, the Karatsuba algorithm performs better. Using the hybrid algorithm, the run times of the algorithms converge as the number of coefficients increase. By setting different thresholds T, we see that the run time continually increase at the same rate as Karatsuba.