CSE 421 Winter 2026: Section 4

January 29, 2026

Instructions: This section worksheet is designed to assist you in working through some example problems and developing your basics. You are encouraged to collaborate on the problems on this section as well as use the course's generative AI.

1 Mechanical calculations: master theorem

Rank the following algorithms by asymptotic runtime. Give the answer and a brief rationale.

1. Algorithm $\mathcal{A}$ solves the problem by dividing it into seven problems of half the size, plus linear-time pre- and post-processing.
2. Algorithm $\mathcal{B}$ solves the problem by dividing it into 25 problems each of size $n/5$, plus quadratic-time pre- and post-processing.
3. Algorithm $\mathcal{C}$ solves the problem of size n by dividing it into 4 problems each of size $n-4$, plus constant-time pre- and post-processing.

2 Maximum subarray sum

Consider as input an array A of integers of length n. We define a *subarray* any contiguous block of entries $A[i], A[i+1], \cdots, A[j]$. The empty subarray is allowed. The subarray sum is the value $A[i]+A[i+1]+\ldots+A[j]$ with the empty subarray sum.

Design a divide-and-conquer algorithm that computes the maximum possible subarray sum. Your algorithm should run in $O(n \log n)$ time.

Instructions: You are free to solve this problem however you like — the following "step-by-step" procedure is our general technique for thinking about algorithms.

1. **What is the problem asking?** It is hard to solve a problem if you don't know what you are supposed to do. Reread the problem slowly, highlighting key parameters, requirements, and the format of the "return type" that the answer needs to be provided in. Ask yourself the following questions:
    - Are they are technical terms in the problem that you don't understand?

- What is the input type? (ex., Array, Graph (directed/undirected/weighted), Integer, something else)?
- What is the output type? Do I need to keep track of decisions I make (ex. assignments, weights, values, etc.) as I run the algorithm?
- For a word problem, can you extract the core algorithmic question?

2. **What are some good examples?** Come up with some examples in an effort to ensure that you truly understand the problem. Generate a few sample instances and calculate the optimal solution.

   Did you notice any patterns while coming up with the solution? If so, what were they, and can those "patterns" be expressed algorithmically? If so, what properties might we need to prove? At this stage, the goal is to just come up with a few (2-3) examples, but it is important to keep an eye on the goal.

   In terms of examples, our suggestion is to **not** focus on base or edge cases. Rather, we want to understand the general behavior of the algorithm. Come up with examples that are decently large ($n \approx 5 - 10$) and are sufficiently different. For example, if the problem is described by a graph, make sure that at least one example is *non-planar*.

3. **What is a baseline?** In a time-constrained setting (ex., interview or exam), it is good to come up with a "baseline" or "brute-force" algorithm. It may not be optimal, but it can help give you a baseline to compare any future improvement against.

4. **What problems does this remind me of?** A great starting point is identifying all the problems you have seen that are similar. Try thinking through the following questions:

   - Does this remind me of any prior problems? What techniques did we use there?
   - Does the problem look like a *standard* algorithm when turned on its head? For example, do I need to create a graph or adjust the data and *then* run a standard algorithm on the new graph/data?
   - Is there some structure to the input? For example, is it 1-dimensional?
   - Is there some structure to the output? If we need to make multiple selections or decisions, can we make them *greedily*?

5. **Write an algorithm.** Try running your algorithms against your examples. Did something click? If one of your algorithms seems to be working, construct an example problem that bucks your algorithm – meaning make sure every path through your example is explored. If you have an "if" argument in your algorithm, make sure there exist examples that apply to both sides of the "if" statement. If you cannot come up with such an example, perhaps the "if" statement wasn't needed.

6. **Prove that your algorithm is correct.** As we explore more algorithms, we will also come up with more techniques for proving correctness. For this problem, there are a couple of equivalent proofs of correctness. Any will do; come up with the one that feels most natural for you to write.

3 Additional Problems

3.1 Mountains

An array $A[1..n]$ is called a *mountain* if there exists an index i (called the peak) such that

$$\forall\, 1 \le j < i, \quad A[j] < A[j+1] \qquad \text{and} \qquad \forall\, i \le j < n, \quad A[j] > A[j+1].$$

1. Suppose you are promised that A is a mountain. Design a divide-and-conquer algorithm that finds the peak index in $O(\log n)$ time.
2. Explain why no algorithm can determine whether an arbitrary array is a mountain in $o(n)$ time.

3.2 Squaring

Professor Sue Spiscious tells her class that it is asymptotically faster to square an n-bit integer than to multiply two n-bit integers. Should they believe her?

3.3 Squaring matrices

1. Show that squaring an 2×2 matrix A can be expressed in terms of 5 multiplications with additions.
2. What's wrong with the following argument?

   > Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$. Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in time $O(n^{\log_2 5})$.

3. Show that a $O(f(n))$ algorithm for squaring $n \times n$ matrices for $n^2 \le f(n) \le n^3$ implies a $O(f(n))$ algorithm for multiplying matrices.

4 Solutions

4.1 Master theorem calculations

We analyze each algorithm using standard recurrences. Algorithm $\mathcal{A}$ satisfies the recurrence

$$T(n) = 7T(n/2) + \Theta(n).$$

Here $a = 7$, $b = 2$, and $k = 1$ so $a > b^k$. And $n^{\log_b a} = n^{\log_2 7}$. The master theorem states that the work is leaf-dominated, for a runtime of $T(n) = \Theta(n^{\log_2 7})$.

Algorithm $\mathcal{B}$ satisfies the recurrence

$$T(n) = 25T(n/5) + \Theta(n^2).$$

Here $a = 25$, $b = 5$, and $k = 2$ so $a = b^k$. This is the balanced case of the master theorem, giving $T(n) = \Theta(n^2 \log n)$.

Algorithm C satisfies the recurrence

$$T(n) = 4T(n-4) + \Theta(1).$$

This is not a Master Theorem recurrence. The recursion has linear depth and exponential branching, yielding $T(n) = \Theta(4^{n/4})$, which is exponential.

Thus, from fastest to slowest, the algorithms are $\mathcal{B}$, then $\mathcal{A}$, then C.

4.2 Maximum Subarray Sum

1. **What is the problem asking?** We are given an array A of n integers, and a subarray is defined as any contiguous block $A[i], A[i+1], \ldots, A[j]$, with the empty subarray allowed and having sum 0. The goal is to compute the maximum possible subarray sum. The input is a one-dimensional array, and the output is a single integer value. No auxiliary information about which subarray achieves the maximum is required. The algorithm must be designed using divide and conquer and must run in $O(n \log n)$ time.

2. **What are some good examples?** For the array $A = [2, -4, 3, 5, -1]$, the subarray $[3, 5]$ has sum 8, which is larger than any other contiguous block, so the answer is 8. For $A = [-3, 4, -1, 2, -6, 5]$, the subarray $[4, -1, 2]$ has sum 5, and the single element subarray $[5]$ also has sum 5, so the answer is 5. If $A = [-5, -2, -8, -1]$, then every non-empty subarray has negative sum, and since the empty subarray is allowed, the correct answer is 0. These examples show that the optimal subarray may lie strictly inside the array, may include negative values, and may even be empty.

3. **What is a baseline?** A simple baseline solution is to enumerate all possible subarrays and compute their sums. There are $\Theta(n^2)$ subarrays, and with prefix sums each subarray sum can be computed in constant time, yielding an $O(n^2)$ algorithm. This approach is correct but does not satisfy the required time bound.

4. **What problems does this remind me of?** This problem closely resembles the Euclidean closest-pair problem from lecture. In both cases, the input is divided into two halves, the problem is solved recursively on each half, and then solutions that cross the dividing boundary are handled separately. In the closest-pair problem, these are point pairs that lie on opposite sides of the vertical split, while here they are subarrays that cross the midpoint of the array. In both settings, once the left and right solutions are known, the remaining candidates have a restricted structure that allows them to be checked in linear time.

5. **Write the algorithm.** Given a subarray $A[\ell \dots r]$, we split it at the midpoint $m = \lfloor (\ell + r)/2 \rfloor$. We recursively compute the maximum subarray sum entirely contained in the left half $A[\ell \dots m]$ and the maximum subarray sum entirely contained in the right half $A[m+1 \dots r]$. To account for subarrays that cross the midpoint, we compute the maximum suffix sum of the left half ending at m by scanning leftward, and the maximum prefix sum of the right half starting at $m+1$ by scanning rightward. The sum of these two values gives the maximum subarray sum that crosses the midpoint. The algorithm returns the maximum of the left solution, the right solution, and the crossing solution, with the empty subarray of sum 0 implicitly allowed.

   Let $T(n)$ denote the running time on an array of length n. The algorithm makes two recursive calls on arrays of size $n/2$ and performs $O(n)$ work to compute the crossing sum, giving the recurrence

   $$T(n) = 2T(n/2) + O(n).$$

   By the Master Theorem, this solves to $T(n) = O(n \log n)$.

6. **Prove that your algorithm is correct.** We prove correctness by induction on the length of the subarray. When the subarray has length 1, the only possible subarrays are the empty subarray and the single element itself, so returning $\max\{0, A[\ell]\}$ is correct. Assume the algorithm correctly computes the maximum subarray sum for all subarrays of length strictly less than $r - \ell + 1$, and consider the subarray $A[\ell \dots r]$ with midpoint m. Any subarray of $A[\ell \dots r]$ must either lie entirely in the left half, entirely in the right half, or cross the midpoint. By the inductive hypothesis, the algorithm correctly handles the first two cases, and by explicitly combining the best suffix of the left half with the best prefix of the right half, it correctly handles all subarrays that cross the midpoint. Therefore, the algorithm considers all possible subarrays and returns the maximum of their sums.

4.3 Additional problems

4.3.1 Mountain Array Peak

To find the peak, compare three consecutive elements at a midpoint index. If the array is increasing at that point, the peak lies to the right. If it is decreasing, the peak lies to the left. Otherwise, the midpoint itself

is the peak. Each recursive step halves the search interval, giving $O(\log n)$ time.

Second, no algorithm can determine whether an arbitrary array is a mountain in sublinear time. If even one index is unexamined, its value can be chosen to either preserve or violate the mountain property. Thus, every element must be inspected in the worst case, requiring $\Omega(n)$ time.

4.3.2 Squaring

No. Clearly, squaring is a special case of multiplication so the runtime of squaring is $\leq$ that of multiplication. However, we can show that they are asymptotically linked: Calculating ab can be achieved through addition and squaring as:

$$ab = \frac{(a+b)^2 - a^2 - b^2}{2}.$$

Therefore, the complexity of multiplication is at most $O(n) + f(n+1) + 2f(n)$ where $f(n)$ is the complexity of squaring n-bit integers. Since $f(n) \leq O(n^{\log_2 3})$ (by Karatsuba's method), then the complexity of multiplication and squaring is asymptotically the same.

4.3.3 Squaring matrices

1. Let A be an 2×2 matrix written as

   $$A = \begin{pmatrix} x & y \\ z & w \end{pmatrix},$$

   where each block is of size $n/2 \times n/2$. Then

   $$A^2 = \begin{pmatrix} x^2 + yz & y(x+w) \\ z(x+w) & yz + w^2 \end{pmatrix}.$$

   There are 5 multiplications involved: x^2, w^2, yz, $y(x+w)$, and $z(x+w)$.

2. The argument incorrectly assumes that expressing matrix squaring using five recursive multiplications immediately yields a divide-and-conquer recurrence identical in structure to Strassen's algorithm. The issue is that the seven subproblems are of *matrix multiplication*, which is not the same problem as the original, *matrix squaring*.

   Furthermore, the prior calculation relied on commutation to rearrange the matrix into 5 multiplications. However, in general, submatrices do not need to commute, which adds another thorn as to why this proof is incorrect.

3. **From fast squaring to fast multiplication and determinants.**

   Suppose we have an algorithm that squares an $n \times n$ matrix in $O(f(n))$ time, where $n^2 \leq f(n) \leq n^3$. We first show how to multiply two matrices A and B in $O(f(n))$ time. Consider the block matrix

   $$M = \begin{pmatrix} 0 & A \\ B & 0 \end{pmatrix}.$$

   Then

   $$M^2 = \begin{pmatrix} AB & 0 \\ 0 & BA \end{pmatrix}.$$

   Thus, computing M^2 using the squaring algorithm yields the product AB as a submatrix. Since M has size $2n \times 2n$, this takes $O(f(2n)) = O(f(n))$ as $f \leq n^3$. Hence, fast squaring implies fast matrix multiplication.