Chapter 8

Maximum Contiguous Subsequences

In this chapter, we consider a well-know problem and apply the algorithm-design techniques that we have learned thus far to this problem. While applying these techniques, we going to be careful in identifying the techniques being used carefully, sometimes at a level of detail that may, especially in subsequent reads, feel pedantic. This is intentional.

8.1 The Problem

We start by defining contiguous subsequences.

Definition 8.1. *Contiguous subsequence For any sequence s of n elements, the subsequence $S' = S[i \dots j]$, $0 \leq i \leq j < n$, which consists of the elements at positions $i, i+1, \dots, j$ is a contiguous subsequence of S.*

Example 8.2. *For $S = \langle 1, -5, 2, -1, 3 \rangle$, here are some contiguous subsequences:*

- $\langle 1 \rangle$,
- $\langle 2, -1, 3 \rangle$, *and*
- $\langle -5, 2 \rangle$.

The sequence $\langle 1, 2, 3 \rangle$ is not a contiguous subsequence, even though it is a subsequence (if we don't say "contiguous", then it is allowed to have gaps in it).

The maximum-contiguous-subsequence problem requires finding the subsequence of a sequence of integers with maximum total sum.

Definition 8.3 (The Maximum Contiguous-Subsequence-Sum (MCS2) Problem)**.** *Given a sequence of numbers, the maximum contiguous-subsequence-sum problem is to find*

$$\mathit{MCS2}(S) = \max\left\{\sum_{k=i}^{j} S[k] \ : \ 0 \leq i, j \leq |S| - 1\right\}.$$

(i.e., the sum of the contiguous subsequence of s that has the largest value).
For an empty sequence, the maximum contiguous subsequence sum is $-\infty$.

Example 8.4. *For $S = \langle 1, -5, 2, -1, 3\rangle$, the maximum contiguous subsequence is, $\langle 2, -1, 3\rangle$. Thus $\mathit{MCS2}(S) = 4$.*

8.2 Algorithm 1: Using Brute Force

Let's start by using the brute force algorithm to solve this problem.

Question 8.5. *To apply the brute-force technique, where do we start?*

We start by identifying the structure of the output. In this case, this is just a number. So technically speaking, we will need to enumerate all numbers and for each number check that there is a subsequence that matches that number until we find the largest number with a matching subsequence.

Question 8.6. *Would such an algorithm terminate?*

Unfortunately such an algorithm would not terminate because we may never know when to stop unless we know the result a priori, which we don't.

Question 8.7. *Can we bound the result to guarantee non-termination?*

We can, however bound the sum, by adding up all positive numbers in the sequence and using that bound. But this can still be a very large bound. Furthermore the cost bounds would depend on the elements of the sequence rather than its length. We thus have already encountered our first challenge.

We can tackle this challenge by changing the result type.

Question 8.8. *How can we change the result type to something that we can enumerate in finite time?*

One natural choice is to consider the contiguous subsequences directly by reducing this problem to another closely related problem: ***maximum-contiguous-subsequence***, in short ***MCS*** , problem. This problem requires not finding the sum but the sequence itself.

Question 8.9. *Can you see how we can solve the MCS2 problem by reducing it to the maximum-contiguous-subsequence MCS problem?*

We can reduce MCS2 problem to MCS problem trivially: since they both operate on the same input, there is no need to convert the input, to compute the output all we have to do is sum up the elements in the sequence returned by the MCS problem.

Question 8.10. *What is the work and span of the reduction?*

Since all we have to do is compute the sum, which we know by using `reduce` requires $O(n)$ work and $O(\log n)$ span, the work and span of the reduction is $O(n)$ and $O(\log n)$ respectively.

Thus, all we have to do now is to solve the MCS problem. We can again apply the brute-force-technique by enumerate all possible results.

Question 8.11. *What are all possible results for the MCS problem? How do we pick the best?*

This time, however, it is easier to enumerate all possible results, which are contiguous subsequences of the input sequence. Since such sequences can be represented by a pair of integers (i, j), $0 \leq i \leq j < n$, we can generate all such integer pairs, compute the sum for each sequence, and pick the largest.

We thus completed our first solution. We used the reduction and the brute-force techniques.

Question 8.12. *Do you something strange about this algorithm?*

Our algorithm for solving the maximum-contiguous-subsequence problem has a strange property: it already computes the result for the MCS2 problem to find the subsequence with the largest sum. In other words, the reduction does redundant work by computing the sum again at the end.

Question 8.13. *Can you see how we may eliminate this redundancy?*

We can eliminate this redundancy by strengthening the problem to require it to return the sum in addition to the subsequence. This way we can reduce the problem to the strengthened problem and compute the result in constant work.

The resulting algorithm can be specified as follows:

$$\texttt{AlgoMCS2}(S) = \overline{\sum}^{\,max\ -\infty} \left\langle \left(\overline{\sum}^{\,+\,0} S[i \ldots j] \right) : 0 \le i \le j < n \right\rangle.$$

Question 8.14. *What is the work and span of the algorithm?*

We can analyze the work and span of the algorithm by appealing to our cost bounds for `reduce`,`subseq`, and `tabulate`.

$$\begin{aligned} W(n) &= 1 + \sum_{1 \le i \le j \le n} W_{\text{reduce}}(j-i) \le 1 + n^2 \cdot W_{\text{reduce}}(n) = 1 + n^2 \cdot O(n) = O(n^3) \\ S(n) &= 1 + \max_{1 \le i \le j \le n} S_{\text{reduce}}(j-i) \le 1 + S_{\text{reduce}}(n) = O(\log n) \end{aligned}$$

These are cost bounds for enumerating over all possible subsequences and computing their sums. The final step of the brute-force solution is to find the max over these $O(n^2)$ combinations. Since max reduce for this step has $O(n^2)$ work and $O(\log n)$ span[1], the cost of the final step is subsumed by other costs analyzed above. Overall, we have an $O(n^3)$-work $O(\log n)$-span algorithm.

Summary 8.15. *In summary, when trying to apply the brute-force technique, we have encountered a problem, which we solved by first reducing MCS2 problem to another problem, MCS. We then realized a redundancy in the resulting algorithm and eliminated that redundancy by strengthening MCS. This is a quite common route when designing a good algorithm: we may often find ourselves refining the problem and the solution until it is (close to) perfect.*

8.2.1 Algorithm 2: Refining Brute Force with a Reduction

Using the brute-force technique, we developed an algorithm that has low span but large work. In this section, we will reduce the work performed by the algorithm by a linear factor by using a reduction. Let's first notice that the algoritm does in fact perform a lot of redundant work, because algorithm repeats the same work many times.

Question 8.16. *Can you see where the redundancy is?*

[1] Note that it takes the maximum over $\binom{n}{2} \le n^2$ values, but since $\log n^a = a \log n$, this is simply $O(\log n)$

To see this let's consider the subsequences that start at some location, for example in the middle. For each position the algorithm considers sequnces that differ by "one" element in their ending positions. In other words many sequences actually overlap but the algorithm does not take advantage of such overlaps.

We can take advantage of such overlaps by computing all subsequences that start at a given position. Let's call this problem the ***Maximum-Contiguous-Sum-with-Start*** problem, abbreviated ***MCS3*** .

Question 8.17. *Can you think of an algorithm for solving the MCS3 problem?*

We can solve this problem by starting at the given position and scanning over the elements of the array to the right as we compute a running sum and take the maximum. The algorithm can be written as follows.

$$\texttt{AlgoMCS3}(S,i) = \overline{\sum}^{max\ -\infty} \left(\overline{\oint}^{+\ 0} \ S[i..(|S|-1)] \right)$$

Question 8.18. *What is the work and span of this algorithm?*

Since the algorithm performs a scan and a reduce, it performs linear work in logarithmic span.

Question 8.19. *Can you improve the brute-force algorithm by reducing the MCS2 problem to MCS3 problem?*

We can use this algorithm to find a more efficient brute-force algorithm for MCS2 by reducing that problem to it: we can try all possible start positions, solve the MCS3 problem for each, and pick the maximum of all the solutions. This would give us a quadratic work and logarithmic span algorithm, which can be expressed succinctly as follows:

$$\texttt{AlgoMCS2}(S) = \overline{\sum}^{max\ -\infty} \ \langle \texttt{AlgoMCS3}(S,i) : 0 \leq i < n \rangle.$$

8.3 Algorithm 3: Using Scan

Let's consider how we might use the scan function to solve the MCS2 problem.

Question 8.20. *Why do you think that we can use scan?*

Recall that the function `scan` returns a reduction over all of the prefixes of a sequence. While the prefixes include some of the contiguous subsequences, they don't include all. We need to find a way to consider all contiguous subsequences.

Question 8.21. *Can you see how?*

The key observation is that any contiguous subsequence of the original sequence can be expressed in terms of the difference between two prefixes. More precisely, the subsequence $S[i..j] = S[0..j] - S[0..(i-1)]$, where the operation $-$ (minus) is left intentionally vague to refer to the difference between the two prefixes. In the context of the MCSS problem, we can find the sum of the elements in a contiguous subsequence $\overline{\sum}^{+\,0} S[i..j]$ in terms of the sum for the corresponding prefixes:

$$\overline{\sum}^{+\,0} S[i..j] = \overline{\sum}^{+\,0} S[0..j] - \overline{\sum}^{+\,0} S[0..(i-1)],$$

where the "-" operation is the usual substraction operation on integers.

But how can we use this property? Let's suppose that we can somehow use this property to solve the problem of finding the ***Maximum-Contiguous-Subsequence Ending*** at any given position, i.e., the ***MCS2E*** problem.

Question 8.22. *How can you reduce MCS2 to MCS2E?*

We can easily reduce the MCS2 problem to MCS2E problem by solving the MCS2E problem for each position and taking the maximum over all solutions.

Of the different problems that we could have reduced MCS2 to, we chose MCS2E because it can be solved easily using scan. To see this consider an ending position j and suppose that you have the sum for each prefix that ends at $i < j$.

Question 8.23. *Can you solve MCS2E using this information?*

Recall that we can express any subsequence ending at the position by subtracting the corresponding prefixes. More importantly, the sum for all such subsequences can be found by subtracting the value for the prefix ending at j from the prefix ending at i. Thus the maximum sequence ending at position j starts at position i with a minimal prefix sum. Thus all we have to compute is the minumum prefix that comes before i, which requires just another scan. One more insight is that we can perform such a scan to find the minumum for all end positions.

These are the main insight but there is some work to be done to work out the details, which may be best demonstrated by considering an example.

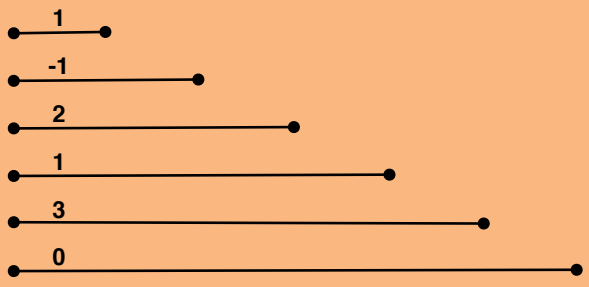
Example 8.24. *Let the sequence S be defined as*

$$S = \langle\, 1, -2, 3, -1, 2, -3 \,\rangle\ .$$

Compute

$$P = \overline{\oint}^{+\,0}\ S = \langle\, 1, -1, 2, 1, 3, 0 \,\rangle\,.$$

The sequence P contains the prefix sums at each position in the sequence.



Using P, we can find the minimum prefix up to any position k (excluding k), as follows. Compute

$$(M, \_) = \overline{\int}^{\,min\ \infty}\ P = \langle\, \infty, -1, -1, -1, -1, -1 \,\rangle\,.$$

We can now find the maximum subsequence ending at any position $i > 0$ by subtracting the value for j in P from the value for all the prior prefixes calculated in M. We have to special case position 0 because there are no prefixes that come before it.
Compute

$$X = \mathtt{append}\,\langle\, P[0] \,\rangle\,\langle\, P[i] - M[i] : 0 < i < |S| \,\rangle = \langle\, 1, -1, 3, 2, 4, 1 \,\rangle\,.$$

It is not difficult to verify in this small example that the values in X are indeed the maximum contigous subsequences ending in each position of the original sequence. Finally, we take the maximum of all the values is X to compute the result:

$$\overline{\sum}^{\,max\ -\infty}\ X = 4.$$

It is not difficult to generalize this example to obtain the following very simple algorithm.

Algorithm 8.25 (Scan-based MCSS)**.**

function `ScanAlgMCSS` (S) $=$
let
$P = \left(\overline{\oint}^{+\ 0}\ S\right)$
$(M, \_) = \overline{\int}^{min\ \infty}\ P$
$X = append\ \langle P[0] \rangle\ \langle P[i] - M[i] : 0 < i < |S| \rangle$
in
$\overline{\sum}^{max\ -\infty}\ X$
end

Given the costs for scan and the fact that addition and minimum take constant work, this algorithm has $O(n)$ work and $O(\log n)$ span.

Question 8.26. *Can we do better than this? In general, how do we know that we have a work-optimal algorithm?*

We can determine whether we have made enough progress or not by comparing the work to a lower bound.

Question 8.27. *What is a lower bound for this problem?*

To find the maximal contiguous subsequences, we have to inspect each element of the sequence at least once to determine whether it would contribute to the result. Since this requires Ωn work, we have a lower bound of Ωn.

8.4 Algorithm 4: Divide And Conquer

Let's now consider the divide-and-conquer technique. Before we do that it might be helpful to re-consider the brute-force algorithm and ask why it performs so poorly, compare for example to the scan based algorithm.

Question 8.28. *Why does the improved brute-force algorithm performs poorly?*

The reason is that it performs much redundant work by considering seperately subsequences that overlap significantly. To apply the divide-and-conquer technique, we first need to figure out how to divide the input.

Question 8.29. *Can you think of ways of dividing the input?*

There are many possibilities, but cutting the input in two halves is often a good starting point because it reduces the input for both subproblems equally, which reduces both the overall work and the overall span by reducing the size of the largest component. Correctness is usually independent of the particular splitting position. So let us cut the sequence and recursively solve the problem on both parts, and combine the solutions to solve the original problem.

Example 8.30. *Let* $s = \langle -2, 2, -2, -2, 3, 2\rangle$. *By using the approach, we cut the sequence into two sequences* L *and* R *as follows*

$$L = \langle -2, 2, -2\rangle$$

and

$$R = \langle -2, 3, 2\rangle.$$

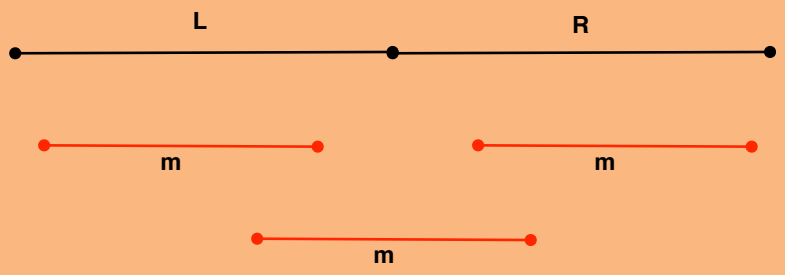
We can now solve each part to obtain 2 *and* 5 *as the two solutions.*

Question 8.31. *How can we combine the solutions to two halves to solve the original problem?*

To obtain the solution to the original problem from those of the subproblems, let's consider where the solution subsequence might come from. There are three possibilities.

1. The maximum sum lies completely in the left subproblem.
2. The maximum sum lies completely in the right subproblem.
3. The maximum sum overlaps with both halves, spanning the cut.

Example 8.32. *The three cases illustrated*



The first two cases are already solved by the recursive calls, but not the last. Assuming we can find the largest subsequence that spans the cut, we can write our algorithm as shown in Algorithm 8.33.

Algorithm 8.33 (Simple Divide-and-Conquer MCSS)**.**

```
1 fun DCAlgoMCS2(S) =
2   case (showt S)
3     of EMPTY = −∞
4      | ELT(x) = x
5      | NODE(L, R) = let
6           val (m_L, m_R) = (mcss(L) || mcss(R))
7           val m_a = bestAcross(L, R)
8         in max{m_L, m_R, m_A}
9         end
```

Question 8.34. *Can you find an algorithm for finding the subsequence with the largest sum that spans the cut (i.e.,* $\texttt{bestAcross}(L, R)$*)? Hint: try the problem-reduction technique to reduce the problem to another one that we know.*

The problem of finding the maximum subsequence spanning the cut is actually closely related to a problem that we have seen already: Maximum-Contiguous-Subsequence Sum with Start, MCS3. The maximum sum spanning the cut is the sum of the largest suffix on the left plus the largest prefix on the right. The prefix of the right part is easy as it directly maps to the solution of MCS3 problem at position 0. For the left part, we reverse the sequence and again solve for MCS3 at position 0.

Example 8.35. *In Example 8.30 the largest sum of a suffix on the left is* 0*, which is given by the maximum of the sums of* $\langle -2, 2, -1\rangle$*,* $\langle 2, -1\rangle$*,* $\langle -1\rangle$*, and* $\langle\rangle$*. The largest sum of a prefix on the right is* 3*, given by summing all the elements. Therefore the largest sum that crosses the middle is* $0 + 3 = 3$*.*

Correctness. Does the algorithm always find the maximum contiguous subsequence sum? Before we show a proof of correctness, it is important to determine the level of the precision of the proof. In 15-150, you familiarized yourself with writing detailed proofs that reason about essentially every step down to the most elementary operations. You would prove your algorithm correct by considering each line. Although proving you code is correct is still important, in this class we will step up a level of abstraction and prove that the algorithms are correct. We still expect your proof to be rigorous but rigorous enough to convince a fellow computer scientist. In other words, in this course, we adopt the mathematical notion of proof, which is based on social agreement. Concretely, we are more interested in seeing the critical steps highlighted and the standard or obvious steps summarized, with sufficient detail that makes it possible for *somebody else* to fill in the remaining detail if needed. The idea is that we want to make key ideas in an algorithm stand out as much as we can. It will be difficult for us to specify exactly how detailed we expect the proof to be, but you will pick it up by example.

Question 8.36. *What technique can we use to show that the algorithm is correct?*

As we briefly mention in Chapter 7, we can use the technique of strong induction, which enables us to assume that the theorem that we are trying to prove stands correct for all smaller subproblems.

Well now prove that the divide-and-conquer algorithm, `DCAlgoMCS2`, computes the maximum contiguous subsequence sum by proving the following theorem.

Theorem 8.37. *Let S be a sequence. The algorithm `DCAlgoMCS2` returns the maximum contiguous subsequence sum in a gives sequence—and returns $-\infty$ if S is empty.*

Proof. The proof will be by (strong) induction on length. We have two base cases: one when the sequence is empty and one when it has one element. On the empty sequence, it returns $-\infty$ as we stated. On any singleton sequence $\langle x \rangle$, the MCSS is x, for which

$$\max\left\{\sum_{k=i}^{j} S[k] \;:\; 0 \leq i < 1, 0 \leq j < 1\right\} = \sum_{k=0}^{0} S[0] = S[0] = x\,.$$

For the inductive step, let s be a sequence of length $n \geq 1$, and assume inductively that for any sequence S' of length $n' < n$, the algorithm correctly computes the maximum contiguous subsequence sum. Now consider the sequence S and let L and R denote the left and right subsequences resulted from dividing S into two parts (i.e., `NODE(L, R) = showt S`). Furthermore, let $S[i..j]$ be any contiguous subsequence of S that has the largest sum, and this value is v. Note that the proof has to account for the possibility that there may be many other subsequences with equal sum. Every contiguous subsequence must start somewhere and end after it. We consider the following 3 possibilities corresponding to how the sequence $S[i..j]$ lies with respect to L and R:

- If the sequence $S[i..j]$ starts in L and ends R. Then its sum equals its part in L (a suffix of L) and its part in R (a prefix of R). If we take the maximum of all suffixes in R and prefixes in L and add them this must equal the maximum of all contiguous sequences bridging the two since $\max\{a+b : a \in A, b \in B\}\} = \max\{a \in A\} + \max\{b \in B\}$. By assumption this equals the sum of $S[i..j]$ which is v. Furthermore by induction m_L and m_R are sums of other subsequences so they cannot be any larger than v and hence $\max\{m_L, m_R, m_A\} = v$.
- If $S[i..j]$ lies entirely in L, then it follows from our inductive hypothesis that $m_L = v$. Furthermore m_R and m_A correspond to the maximum sum of other subsequences, which cannot be larger than v. So again $\max\{m_L, m_R, m_A\} = v$.
- Similarly, if $s_{i..j}$ lies entirely in R, then it follows from our inductive hypothesis that $m_R = \max\{m_L, m_R, m_A\} = v$.

We conclude that in all cases, we return $\max\{m_L, m_R, m_A\} = v$, as claimed. □

Cost analysis. What is the work and span of this algorithm? Before we analyze the cost, let's first remark that it turns out that we can compute the max prefix and suffix sums in parallel by using a primitive called `scan`. For now, we will take it for granted that they can be done in $O(n)$ work and $O(\log n)$ span. dividing takes $O(\log n)$ work and span. This yields the following recurrences:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

Using the definition of big-O, we know that

$$W(n) \leq 2W(n/2) + k_1 \cdot n + k_2,$$

where k_1 and k_2 are constants.

We have solved this recurrence using the recursion tree method. We can also arrive at the same answer by mathematical induction. If you want to go via this route (and you don't know the answer a priori), you'll need to guess the answer first and check it. This is often called the "substitution method." Since this technique relies on guessing an answer, you can sometimes fool yourself by giving a false proof. The following are some tips:

1. Spell out the constants. Do not use big-O—we need to be precise about constants, so big-O makes it super easy to fool ourselves.
2. Be careful that the inequalities always go in the right direction.
3. Add additional lower-order terms, if necessary, to make the induction go through.

Let's now redo the recurrences above using this method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 8.38. *Let a constant $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot n$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then we can find constants κ_1 and κ_2 such that*

$$W(n) \leq \kappa_1 \cdot n \log n + \kappa_2.$$

Proof. Let $\kappa_1 = 2k$ and $\kappa_2 = k$. For the base case ($n = 1$), we check that $W(1) = k \leq \kappa_2$. For the inductive step ($n > 1$), we assume that

$$W(n/2) \leq \kappa_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + \kappa_2,$$

And we'll show that $W(n) \leq \kappa_1 \cdot n \log n + \kappa_2$. To show this, we substitute an upper bound for $W(n/2)$ from our assumption into the recurrence, yielding

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot n \\ &\leq 2(\kappa_1 \cdot \tfrac{n}{2} \log(\tfrac{n}{2}) + \kappa_2) + k \cdot n \\ &= \kappa_1 n(\log n - 1) + 2\kappa_2 + k \cdot n \\ &= \kappa_1 n \log n + \kappa_2 + (k \cdot n + \kappa_2 - \kappa_1 \cdot n) \\ &\leq \kappa_1 n \log n + \kappa_2, \end{aligned}$$

where the final step follows because $k \cdot n + \kappa_2 - \kappa_1 \cdot n \leq 0$ as long as $n > 1$. □

Question 8.39. *Using divide and conquer, we were able to reduce work to $O(n \log n)$. Can you see where the savings came from by comparing this algorithm to the refined brute-force algorithm that we have considered?*

8.5 Algorithm 5: Divide And Conquer with Strengthening

Our first divide-and-conquer algorithm performs $O(n \log n)$ work, which is $O(\log n)$ factor more than the optimal. In this section, we shall reduce the work to $O(n)$ by being more careful about not doing redundant work.

Question 8.40. *Is there some redundancy in our first divide-and-conquer algorithm?*

Our divide-and-conquer algorithm has an important redundancy: the maximum prefix and maximum suffix are computed recursively to solve the subproblems for the two halves. Thus, by finding them again, the algorithm does redundant work.

Question 8.41. *Can we avoid re-computing the maximum prefix and suffix?*

Since these should be computed as part of solving the subproblems, we should be able to return them from the recursive calls. In other words, we want to strengthen the problem so that it returns the maximum prefix and suffix. Since this problem, called MCS2PS, matches the original MCS2 problem in its input and returns strictly more information, solving MCS2 using MCS2PS is trivial. We can thus focus on solving the MCS2PS problem.

Question 8.42. *Can you see how we can update our divide and conquer algorithm to solve the MCS2PS problem, i.e., to return also the maximum prefix and suffix in addition to maximum contiguous subsequence?*

We need to return a total of three values: the max subsequence sum, the max prefix sum, and the max suffix sum. At the base cases, when the sequence is empty or consists of a single element, this is easy to do. For the recursive case, we need to consider how to produce the desired return values from those of the subproblems. Suppose that the two subproblems return (m_1, p_1, s_1) and (m_2, p_2, s_2).

Question 8.43. *How can we compute the result from the solutions to the subproblems?*

One possibility to compute as result

$$(\max(s_1 + p_2, m_1, m_2), p_1, s_2).$$

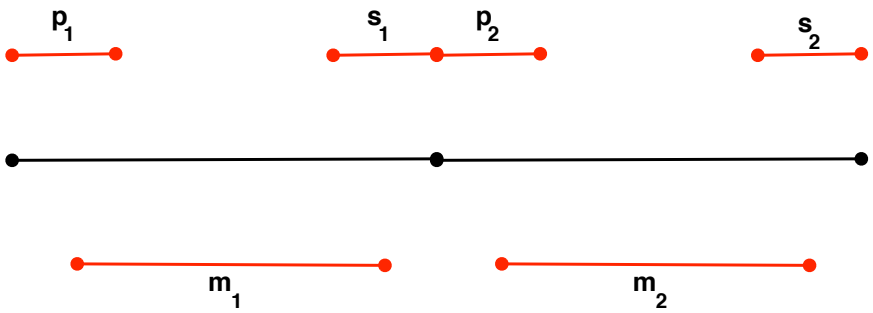


Figure 8.1: Solving the MCS2PS problem with divide and conquer.

Question 8.44. *Don't we have consider the case when s_1 or p_2 is the maximum?*

Note that we don't have to consider the case when s_1 or p_2 is the maximum, because that case is checked in the computation of m_1 and m_2 by the two subproblems.

Question 8.45. *Are our prefix's and suffixes correct? Can we not have a bigger prefix that contains all of the first sequence?*

This solution misses to account for the case when the suffix and prefix can span the whole sequence.

Question 8.46. *How can you fix this problem?*

This problem is easy to fix by returning the total for each subsequence so that we can compute the maximum prefix and suffix correctly. Thus, we need to return a total of four values: the max subsequence sum, the max prefix sum, the max suffix sum, and the overall sum. Having this information from the subproblems is enough to produce a similar answer tuple for all levels up, in constant work and span per level. Thus what we have discovered is that to solve the strengthened problem efficiently we have to strengthen the problem once again. Thus if the recursive calls return (m_1, p_1, s_1, t_1) and (m_2, p_2, s_2, t_2), then we return

$$(\max(s_1 + p_2, m_1, m_2), \max(p_1, t_1 + p_2), \max(s_1 + t_2, s_2), t_1 + t_2)$$

This gives the following algorithm:

Algorithm 8.47 (Linear Work Divide-and-Conquer MCSS)**.**

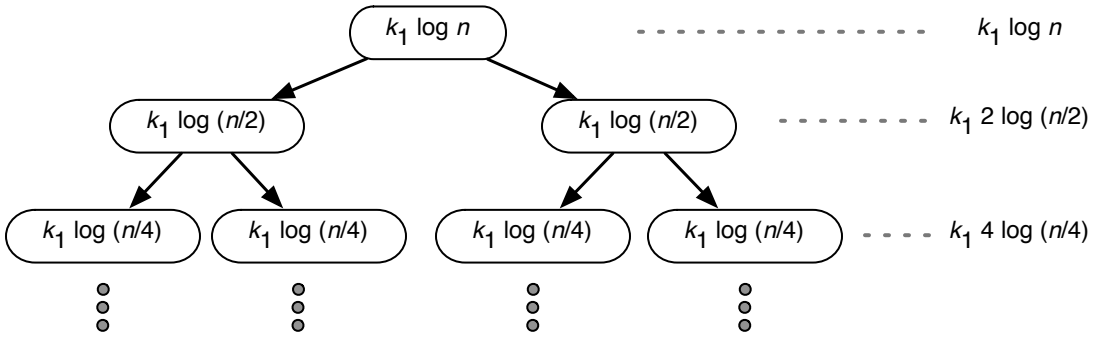
```
 1 function mcss(a) = let
 2    function mcss'(a)
 3       case (showt a)
 4          of EMPTY ⇒ (−∞, −∞, −∞, 0)
 5           | ELT(x) ⇒ (x, x, x, x)
 6           | NODE(L, R) =
 7                let
 8                   val ((m1, p1, s1, t1), (m2, p2, s2, t2)) = (mcss'(L) || mcss'(R))
 9                in
10                   (max(s1 + p2, m1, m2),     % overall mcss
11                    max(p1, t1 + p2),         % maximum prefix
12                    max(s1 + t2, s2),         % maximum suffix
13                    t1 + t2)                  % total sum
14                end
15    val (m,_,_,_) = mcss'(a)
16 in m end
```

You should verify the base cases are doing the right thing.

Cost Analysis. Assuming `showt` takes $O(\log n)$ work and span, we have the recurrences

$$\begin{aligned} W(n) &= 2W(n/2) + O(\log n) \\ S(n) &= S(n/2) + O(\log n) \end{aligned}$$

Note that the span is the same as before, so we'll focus on analyzing the work. Using the tree method, we have



Therefore, the total work is upper-bounded by

$$W(n) \leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i)$$

It is not so obvious to what this sum evaluates. The substitution method seems to be more convenient. We'll make a guess that $W(n) \leq \kappa_1 n - \kappa_2 \log n - k_3$. More formally, we'll prove the following theorem:

Theorem 8.48. *Let $k > 0$ be given. If $W(n) \leq 2W(n/2) + k \cdot \log n$ for $n > 1$ and $W(n) \leq k$ for $n \leq 1$, then we can find constants κ_1, κ_2, and κ_3 such that*

$$W(n) \;\leq\; \kappa_1 \cdot n - \kappa_2 \cdot \log n - \kappa_3.$$

Proof. Let $\kappa_1 = 3k$, $\kappa_2 = k$, $\kappa_3 = 2k$. We begin with the base case. Clearly, $W(1) = k \leq \kappa_1 - \kappa_3 = 3k - 2k = k$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot \log n \\ &\leq 2(\kappa_1 \tfrac{n}{2} - \kappa_2 \log(n/2) - \kappa_3) + k \cdot \log n \\ &= \kappa_1 n - 2\kappa_2(\log n - 1) - 2\kappa_3 + k \cdot \log n \\ &= (\kappa_1 n - \kappa_2 \log n - \kappa_3) + (k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) \\ &\leq \kappa_1 n - \kappa_2 \log n - \kappa_3, \end{aligned}$$

where the final step uses the fact that $(k \log n - \kappa_2 \log n + 2\kappa_2 - \kappa_3) = (k \log n - k \log n + 2k - 2k) = 0 \leq 0$ by our choice of κ's. □

Finishing the tree method. It is possible to solve the recurrence directly by evaluating the sum we established using the tree method. We didn't cover this in lecture, but for the curious, here's how you can "tame" it.

$$\begin{aligned} W(n) &\leq \sum_{i=0}^{\log n} k_1 2^i \log(n/2^i) \\ &= \sum_{i=0}^{\log n} k_1 \left(2^i \log n - i \cdot 2^i\right) \\ &= k_1 \left(\sum_{i=0}^{\log n} 2^i\right) \log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i \\ &= k_1(2n-1) \log n - k_1 \sum_{i=0}^{\log n} i \cdot 2^i. \end{aligned}$$

We're left with evaluating $s = \sum_{i=0}^{\log n} i \cdot 2^i$. Observe that if we multiply s by 2, we have

$$2s = \sum_{i=0}^{\log n} i \cdot 2^{i+1} = \sum_{i=1}^{1+\log n} (i-1)2^i,$$

so then

$$\begin{aligned} s &= 2s - s = \sum_{i=1}^{1+\log n} (i-1)2^i - \sum_{i=0}^{\log n} i \cdot 2^i \\ &= ((1+\log n) - 1)\, 2^{1+\log n} - \sum_{i=1}^{\log n} 2^i \\ &= 2n \log n - (2n-2). \end{aligned}$$

Substituting this back into the expression we derived earlier, we have $W(n) \leq k_1(2n-1)\log n - 2k_1(n\log n - n + 1) \in O(n)$ because the $n \log n$ terms cancel.

.