Chapter 9

Maximum Contiguous Subsequences

In this chapter, we apply the algorithm-design techniques considered thus far to a well-known problem, that of finding contiguous subsequence of a string that sums of to the maximal value. In order to exercise our vocabulary for algorithm design, we intentionally identify the techniques being used, sometimes at a level of precision that may, especially in subsequent reads, feel pedantic.

9.1 The Problem

A ***subsequence*** b of a sequence a is a sequence that can be derived from a by deleting elements of a without changing the order of remaining elements. For example, $\langle 0, 2, 4 \rangle$ is a subsequence of $\langle 0, 1, 2, 2, 3, 4, 5 \rangle$. A contiguous subsequence is a subsequence that appears contiguously in the original sequence. For example, $\langle 0, 2, 4 \rangle$ is a not subsequence of $\langle 0, 1, 2, 2, 3, 4, 5 \rangle$ but $\langle 2, 3, 4 \rangle$ is.

Definition 9.1. [Contiguous subsequence] For any sequence a of n elements, the subsequence $b = a[i, \ldots, j]$, $0 \leq i \leq j < n$, consisting of the elements of a at positions $i, i+1, \ldots, j$ is a contiguous subsequence of b.

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

Definition 9.2. [The Maximum Contiguous-Subsequence-Sum (MCS2) Problem] Given a sequence of numbers, the maximum contiguous-subsequence-sum problem requires finding the sum of the contiguous subsequence with the largest value, i.e.,

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

For an empty sequence, the maximum contiguous subsequence sum is $-\infty$.

While we only consider sequences of numbers and the addition operation to compute the sum, the techniques that we describe should apply to sequences of other types and other associative sum operations.

Before we proceed to design algorithms for the MCS2 problem, let's first establish a lower bound for the amount of work required to solve this problem. Observe that to solve the MCS2 problem, we need to inspect, at the very least, every element of the sequence. This requires linear work in the length of the sequence. We thus have a lower bound of $\Omega(n)$ for the work needed to solve MCS2.

Remark 9.3. [History of the Problem] The study of maximum contiguous subsequence problem goes to 1970's. The problem was first proposed in by Ulf Grenander, a Swedish statistician and a professor of applied mathematics at Brown University, in 1977. The problem has several names, such maximum subarray sum problem, or maximum segment sum problem, the former of which appears to be the name originally used by Grenander. Grenander intended it to be a simplified model for maximum likelihood estimation of patterns in digitized images, whose structure he wanted to understand. According to Jon Bentley [a] in 1977, Grenander described the problem to Michael Shamos of Carnegie Mellon University who overnight designed a divide and conquer algorithm, which corresponds to our first-divide-and-conquer algorithm. When Shamos and Bentley discussed the problem and Shamos' solution, they thought that it was probably the best possible. A few days later Shamos described the problem and its history at a Carnegie Mellon seminar attended by statistician Joseph (Jay) Kadane, who designed the work efficient algorithm within a minute.

[a] Jon Bentley, Programming Pearls (1st edition), page 76.

9.2 Brute Force

To solve the MCS2 problem, we can apply our most basic algorithm-design technique, the brute force method, which requires trying out all possible solutions. To apply this technique, we first identify the structure of the output, which in this case, is just a number. Thus technically speaking, we will enumerate all numbers and, for each number, check that there is a contiguous subsequence that matches that number until we find the largest number with a matching subsequence. 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.

We can, however bound the sum, by adding up all positive numbers in the sequence and using that bound; clearly the sum of the maximum contiguous subsequence cannot exceed such an optimistic bound. But unfortunately such a bound can be large and cause the cost bounds to depend on the elements of the sequence rather than its length.

We can overcome this small challenge by changing our problem slightly to return a different result. More precisely, we can reduce MCS2 problem to another closely related problem: ***maximum-contiguous-subsequence***, in short ***MCS***, problem, which requires not finding the sum but the sequence itself. This reduction is quite simple: 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. Since all we have to do is to compute the sum, which can find by using `reduce` in $O(n)$ 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 enumerating all possible results. 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. But our algorithm for solving the maximum-contiguous-subsequence problem has an important redundancy: to find the solution, it computes the result for the MCS2 problem and takes a sum, which is already computed by the MCS2 algorithm. We can eliminate this redundancy by strengthening the problem and requiring 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

Algorithm 9.4. [Brute Force 1]

```
BFMCS2 a =
  let
    b = ⟨ reduce + 0 a[i,...,j]  : 0 ≤ i ≤ j < n ⟩
  in
    reduce max −∞ b
  end.
```

We can analyze the work and span of the algorithm by appealing to our cost bounds for `reduce`,`subseq`, and `tabulate`. Since `subseq` is somewhat more expensive with three sequences, we can use the array-sequence cost specification, where is has constant work and span.

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

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 maximum over these $\Theta(n^2)$ combinations. Since the reduce for this step has $\Theta(n^2)$ work and $\Theta(\log n)$ span[1], the cost of the final step is subsumed by other costs analyzed above. Overall, we have an $\Theta(n^3)$-work $\Theta(\log n)$-span algorithm.

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

Summary 9.5. When trying to apply the brute-force technique to the MCS2 problem, we encountered a problem. We solved this problem by 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 find ourselves refining the problem and the solution until it is (close to) perfect.

9.3 Brute Force Refined with a Reduction

Using the brute-force technique, we developed an algorithm that has low span. The algorithm, however, performs $\Theta(n^3)$ work, which is large. Our goal in this section is to reduce the work performed by the algorithm by a linear factor. Let's first notice that the algorithm does in fact perform a lot of redundant work, because algorithm repeats the same work many times. To see this let's consider the subsequences that start at some location, for example in the middle. For each position the algorithm considers a number of subsequnces 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 together. Let's call the problem of computing the maximum contiguous subsequence of a sequence that start a given position as the ***Maximum-Contiguous-Sum-with-Start***, abbreviated ***MCS3***, problem.

We can solve this problem by first computing the sum for all subsequences that start at the given position by using a scan and then computing their maximum. This algorithm can be written as follows

Algorithm 9.6. [Brute Force MCS3 Algorithm]

```
BFMCS3 a i =
  let
    b = scan + 0 a[i, ..., (|a| − 1)]
  in
    reduce max −∞ b
  end.
```

Since the algorithm performs a scan and a reduce, it performs (wost-case) $\Theta(n)$ work in $\Theta(\log n)$ span, for any starting position.

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:

Algorithm 9.7. [Brute Force 2]

```
RBFMCS2 a =
  reduce max −∞ ⟨ BFMCS3 a i : 0 ≤ i < n ⟩.
```

This algorithm performs worst-case $\Theta(n^2)$ work in $\Theta(\log n)$ span, delivering a linear-factor improvement in work.

9.4 Reduction and Iteration

In the previous section, we reduced the MCS2 problem to the MCS3 problem, which we solved using scan. Let's now consider a closely related problem: ***Maximum-Contiguous-Subsequence at Ending***, i.e., the ***MCS2E*** problem, which requires finding the maximum contiguous subsequence ending at a specified end position. We can reduce the MCS2 problem to MCS2E problem by solving the MCS2E problem for each position and taking the maximum over all solutions.

We can solve the MCS2E problem by following a similar strategy to the MCS3 problem but we would have to reverse the list and assume furthermore that our addition operation, with which we compute sums, is commutative. Reversing is easy but we might not want to make further assumptions about the addition operation.

There is a better way. Suppose that we are given the maximum contiguous sequence, M_i ending at position i. We can compute the maximum contiguous sequence ending at position $i+1$, M_{i+1}, from this by noticing that $M_{i+1} = M_i +\!\!+ \langle\, a[i] \,\rangle$ or $M_{i+1} = \langle\, a[i] \,\rangle$, depending on the sum for each.

We can now use this insight to solve MCS2 problem using iteration: we can iterate over the sequence and solve the MCS2E problem for each ending position and take the maximum over all positions. The SPARC code for this algorithm is shown in Algorithm 9.8. Note that we use the function `iteratePrefixes` to iterate over the input sequence and construct a sequence whose i^{th} position contains the solution to the MCS2E problem at that position.

Algorithm 9.8. [MCS2 with iteration]

```
IterativeMCS2 a =
  let
    f (sum, x) =
      if sum + x ≥ x then
        sum + x
      else
        x
    b = iteratePrefixes f −∞ a
  in
    reduce max −∞ b
  end.
```

Let's analyze the work and span of this algorithm. To do this, we first have to decide the cost specification of sequences that we want to use. Since the algorithm only uses `iteratePrefixes` and `reduce`, we can use array sequences. Since the function f is constant work and span function, we have $W(n) = O(n)$ and $S(n) = O(n)$. Using iteration, we designed an algorithm that is work efficient, which performs asymptotically optimal work. But unfortunately the algorithm has no parallelism.

9.5 Contraction

We have been able to obtain a work-efficient algorithm for the MCS2 problem by using the brute force and reductions techniques but the algorithm has a large span. We will now design a work-efficient and low-span algorithm for the MCS2 problem using the `scan` operation.

We start with a key observation: any contiguous subsequence of a given sequence can be expressed in terms of the difference between two prefixes of the sequence. More precisely, the subsequence $A[i, \ldots, j]$ is equivalent to the subsequence $A[0, \ldots, j]$ "minus" $A[0, \ldots, i-1]$, where the operation "minus" can be specified precisely depending on the nature of the problem. For example, in MCS2 problem, we can find the sum of the elements in a contiguous subsequence `reduce + 0` $a[i, \ldots, j]$ as follows

$$\texttt{reduce + 0}\ a[i, \ldots, j] = (\texttt{reduce + 0}\ a[0, \ldots, j]) - (\texttt{reduce + 0}\ a[0, \ldots, i-1])$$

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

Based on this observation, we can solve the MCS2E problem. To see how, consider an ending position j and suppose that you have the sum for each prefix that ends at $i < j$. Since we can express any subsequence ending at position j by subtracting the corresponding prefix, we can compute the sum for the subsequence $A[i, \ldots, j]$ by subtracting the sum for the prefix ending at j from the prefix ending at $i - 1$. Thus the maximum contiguous sequence ending at position i starts at position j which has the minimum of all prefixes up to i. We can compute the

minimum prefix that comes before i by using just another scan. Furthermore, we can compute the minimum preceeding prefix for all positions in a single scan. Example **??** illustrates an example. The algorithm below shows a scan-based algorithm based on these insights.

Algorithm 9.9. [Scan-based MCSS]

```
ScanMCSS a =
  let
    (b, v) = scan + 0 a
    c = append b ⟨v⟩
    (d, _) = scan min ∞ c
    e = ⟨c[i] − d[i] : 0 < i < |a|⟩
  in
    reduce max −∞ e
  end
```

Given the costs for scan and the fact that addition and minimum take constant work, this algorithm has $\Theta(n)$ work and $\Theta(\log n)$ span. Since, we have to inspect each element of the sequence at least once to solve the MCS2 problem, this algorithm is work optimal.

Example 9.10. Consider the sequence a

$$a = \langle\, 1, -2, 0, 3, -1, 0, 2, -3 \,\rangle\,.$$

Compute

$$\begin{array}{lcl} (b, v) & = & \texttt{scan} + 0\ a \\ c & = & \texttt{append}\ b\ \langle\, v \,\rangle\,. \end{array}$$

We have $c = \langle\, 0, 1, -1, -1, 2, 1, 1, 3, 0 \,\rangle$.

The sequence c contains the prefix sums ending at each position, including the element at the position; it also contains the empty prefix.

Using the sequence c, we can find the minimum prefix up to all positions as

$$(d, \_) = \texttt{scan min}\ \infty\ c$$

to obtain

$$d = \langle\, \infty, 0, 0, -1, -1 - 1, -1, -1, -1 \,\rangle\,.$$

We can now find the maximum subsequence ending at any position i by subtracting the value for i in c from the value for all the prior prefixes calculated in d.

Compute

$$\begin{array}{lcl} e & = & \langle\, c[i] - d[i] : 0 < i < |a| \,\rangle \\ & = & \langle\, 1, -1, 0, 3, 2, 2, 4, 1 \,\rangle\,. \end{array}$$

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

$$\texttt{reduce max}\ -\infty\ e = 4.$$

9.6 Divide And Conquer

To apply the divide-and-conquer technique, we first need to figure out how to divide the input. There are many possibilities, but dividing the input in two halves is usually a good starting point, because it reduces the input for both subproblems equally, reducing thus the size of the largest component, which is important in bounding the overall span. Correctness is usually independent of the particular strategy of division.

Let us divide the sequence into two halves, recursively solve the problem on both parts, and combine the solutions to solve the original problem.

Example 9.11. Let $a = \langle\, 1, -2, 0, 3, -1, 0, 2, -3 \,\rangle$. By using the approach, we divide the sequence into two sequences b and c as follows

$$b = \langle\, 1, -2, 0, 3 \,\rangle$$

and

$$c = \langle\, -1, 0, 2, -3 \,\rangle$$

We can now solve each part to obtain 3 and 2 as the solutions to the subproblems. Note that there are multiple sequences that yield the maximum sum.

To construct a solution for 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.

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 9.12.

Algorithm 9.12. [Simple Divide-and-Conquer MCSS]

```
DCMCS2 a =
  if |a| = 0 then
    −∞
  else if |a| = 1 then
    a[0]
  else
    let
      (b, c)   = splitMid a
      (m_b, m_c) = (DCMCS2 b || DCMCS2 c)
      m_bc = bestAcross (b, c)
    in
      max{m_b, m_c, m_bc}
    end
```

The problem of finding the maximum subsequence spanning the cut is closely related to two problems that we have seen already: Maximum-Contiguous-Subsequence Sum with Start, MCS3, and Maximum-Contiguous-Subsequence Sum at Ending, MCS2E.

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. Similarly, the suffix for the left part is exactly an instance of MCS2E problem.

Example 9.13. In Example 9.11 the largest suffix on the left is 3, which is given by the sequence $\langle\ 3\ \rangle$ or $\langle\ 0, 3\ \rangle$. The largest prefix on the right is 1 given by the sequence $\langle\ -1, 0, 2\ \rangle$. Therefore the largest sum that crosses the middle is $3 + 1 = 4$.

Correctness. As described in Chapter 8, to prove a divide-and-conquer algorithm correct, we can use the technique of strong induction, which enables to assume that the theorem that we are trying to prove remains correct for all smaller subproblems. We now present such a correctness proof for the algorithm `DCMCS2`.

Theorem 9.14. *Let a be a sequence. The algorithm `DCMCS2` returns the maximum contiguous subsequence sum in a gives sequence—and returns $-\infty$ if a is empty.*

Proof. The proof will be by (strong) induction on length of the input sequence. Our induction hypothesis is that the theorem above holds for all inputs smaller than the current input.

We have two base cases: one when the sequence is empty and one when it has one element. On the empty sequence, the algorithm returns $-\infty$ and thus the theorem holds. On any singleton sequence $\langle\ x\ \rangle$, the MCS2 is x, because

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

The theorem therefore holds.

For the inductive step, let a be a sequence of length $n \ge 1$, and assume inductively that for any sequence a' of length $n' < n$, the algorithm correctly computes the maximum contiguous subsequence sum. Now consider the sequence a and let b and c denote the left and right subsequences resulted from dividing a into two parts (i.e., `(b, c) = splitMid a`). Furthermore, let $a[i, \ldots, j]$ be any contiguous subsequence of a 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 $a[i, \ldots, j]$ lies with respect to b and c:

- If the sequence $a[i, \ldots, j]$ starts in b and ends c. Then its sum equals its part in b (a suffix of b) and its part in c (a prefix of c). If we take the maximum of all suffixes in c and prefixes in b and add them this is equal the maximum of all contiguous sequences bridging the two, because $\max\{x + y : x \in X, y \in Y\} = \max\{x \in X\} + \max\{y \in Y\}$. By assumption this equals the sum of $a[i, \ldots, j]$ which is v. Furthermore by induction m_b and m_c are sums of other subsequences so they cannot be any larger than v and hence $\max\{m_b, m_c, m_{bc}\} = v$.

- If $a[i, \ldots, j]$ lies entirely in b, then it follows from our inductive hypothesis that $m_b = v$. Furthermore m_c and m_{bc} correspond to the maximum sum of other subsequences, which cannot be larger than v. So again $\max\{m_b, m_c, m_{bc}\} = v$.
- Similarly, if $a_{i..j}$ lies entirely in c, then it follows from our inductive hypothesis that $m_c = \max\{m_b, m_c, m_{bc}\} = v$.

We conclude that in all cases, we return $\max\{m_b, m_c, m_{bc}\} = v$, as claimed. □

Cost analysis. What is the work and span of our divide-and-conquer algorithm? Before we analyze the cost, let's first remark that it turns out that we can compute the maximum prefix and suffix sums in parallel by using a primitive called `scan` in $O(n)$ work and $O(\log n)$ span. Note also that `splitMid` requires $O(\log n)$ work and span using array or tree sequences. We thus have the following recurrences with array-sequence or tree-sequence specifications

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

Using the definition of big-Θ, we know that

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

where k_1 and k_2 are constants. By using the tree method, we can conclude that $W(n) = \Theta(n \lg n)$ and $S(n) = \log^2 n$.

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 the asymptotic notation—we need to be precise about constants, the asymptotic notation 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 the substitution method. Specifically, we'll prove the following theorem using (strong) induction on n.

Theorem 9.15. *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 $n > 1$. □

9.7 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 avoiding redundant work. 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, the algorithm does redundant work by computing them again.

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, which we shall call ***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.

We can solve this problem by strengthening our divide-and-conquer algorithm from the previous section. 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).

One possibility to compute as result

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

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

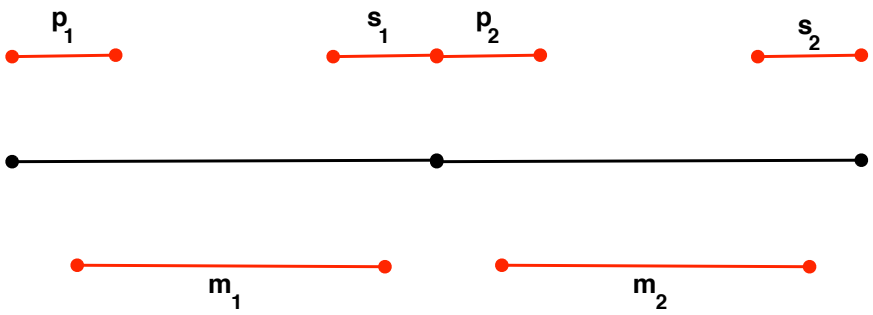


Figure 9.1: Solving the MCS2PS problem with divide and conquer.

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 9.16. [Linear Work Divide-and-Conquer MCSS]

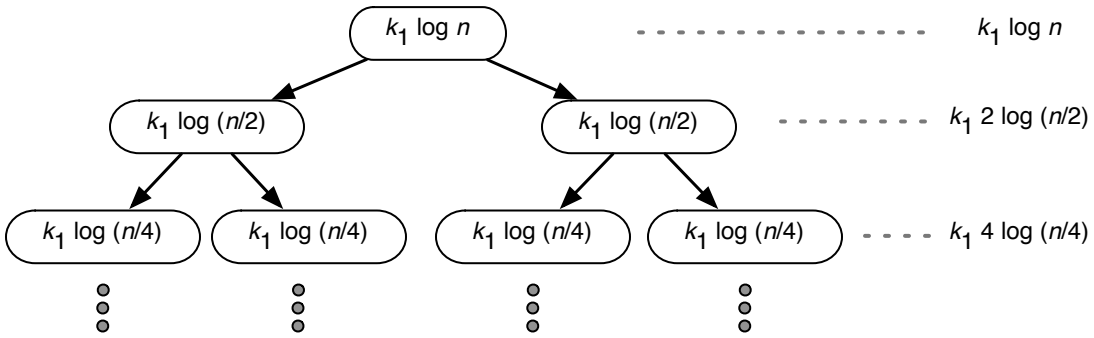
```
DCSMCS2' a =
  if |a| = 0 then
    (-∞, -∞, -∞, 0)
  else if |a| = 1 then
    (a[0], a[0], a[0], a[0])
  else
    let
      (b, c) = splitMid a
      ((m1, p1, s1, t1), (m2, p2, s2, t2)) = (DCSMCS2' b || DCSMCS2' c)
    in
      (max (s1 + p2, m1, m2),    (* overall mcss *)
       max (p1, t1 + p2),        (* maximum prefix *)
       max (s1 + t2, s2),        (* maximum suffix *)
       t1 + t2)                  (* total sum *)
    end
  (m, _, _, _) = DCSMCS2' a
in m end

DCSMCS2 a =
  let (m, _, _, _) = DCSMCS2' a
  in m end
```

Cost Analysis. Since `splitMid` requires $O(\log n)$ work and span in both array and tree sequences, we have

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

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 9.17. *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.

.