Chapter 4

Divide and Conquer

Divide and conquer (DC) is one of the most important algorithmic techniques and can be used to solve a variety of computational problems. The structure of a divide-and-conquer algorithm applied to a given problem P has the following form.

Base Case: When the instance I of the problem P is sufficiently small, return the answer $P(I)$ directly, or resort to a different, usually simpler, algorithm that is well suited for small instances.

Inductive Step:

1. **Divide** I into some number of smaller instances of the same problem P.
2. **Recurse** on each of the smaller instances to obtain their answers.
3. **Combine** the answers to produce an answer for the original instance I.

Divide-and-Conquer has several nice properties. Firstly it very closely follows the structure of an inductive proof, and therefore most often leads to rather simple proofs of correctness. As in induction, one first needs to prove the base case is correct. Then one can assume by strong (or structural) induction that the recursive solutions are correct, and needs to show that given correct solutions to each smaller instance the combined solution is a correct answer. A second nice property is that divide-and-conquer can lead to quite efficient solutions to a problem. However, to be efficient one needs to be sure that the divide and combine steps are efficient, and that they do not create to many sub instances. This brings us to the third nice property, which is that the work and span for divide-and-conquer algorithms can be expressed as a form of mathematical equations called recurrences. Often these recurrences can be solved without too much difficulty making analyzing the work and span of many divide-and-conquer algorithms reasonably straightforward. Solving such recurrences will be a major topic of this chapter.

Finally, divide-and-conquer is a naturally parallel algorithmic technique. Most often we can solve the sub instances in parallel. This can lead to significant amount of parallelism since at each level of can create more instances to solve in parallel. Even if we only divide our instance into two sub instances, each of those sub instances will themselves generate two more sub-instances, and this repeats.

In this chapter we will look at how to apply divide-and-conquer to a variety of problems, how to convert divide-and-conquer algorithms into recurrences for analyzing costs, how to apply divide-and-conquer to a variety of problems, how to solve recurrences, and an approach called strengthening that allows us to apply to a wider variety of problems.

Example: Mergesort Mergesort and Quicksort are perhaps the canonical examples of divide-and-conquer. They both solve the sorting problem:

Definition 4.1 (The (Comparison) Sorting Problem)**.** *Given a sequence S of elements from a universe U, with a total ordering given by $<$, return the same elements in a sequence R in sorted order, i.e.* $R_i \leq R_{i+1}, 0 < i \leq |S| - 1$.

Both Mergesort and Quicksort use $\Theta(n \log n)$ work, which is optimal for the comparison sorting problem. What is interesting is that one of them, Mergesort, has an trivial divide step and interesting combine step, while the other, Quicksort, has an interesting divide step but trivial combine step. We will cover Quicksort in Chapter 7 on randomized algorithms, since it involves randomization. In Mergesort the divide step simply consists of splitting the input sequence. As we will see this is actually common in several divide-and-conquer algorithms. In this book we use the `showt` function to split a sequence in approximately half. Mergesort can be defined as follows.

Algorithm 4.2 (Mergesort)**.**

```
fun sort(S) =
  case (showt S)
    of EMPTY = EMPTY
     | ELT(_) = S
     | NODE(L, R) = let
           val (L', R') = (sort(L) || sort(R))
         in
           merge(L', R')
         end
```

In this algorithm the base case is when the sequence is empty or contains a single element. In practice, however, instead of using a single element or empty sequence as the base case, some implementations use a larger base case consisting of perhaps ten to twenty keys.

Question 4.3. *Why might someone choose to use a larger base case?*

In the code, if the sequence is larger than one it is split in two approximately equal sized parts, each part is recursively sorted, and the results are merged. Recall that merging takes two sorted sequences and merges them into a single sorted sequence with the same elements. Also note that the two recursive calls are made in parallel. To prove correctness we first note that the base case is certainly correct. Then by induction, roughly, we note that L and R together contain exactly the same elements as S, that by induction L' and R' are sorted versions of L and R, and finally that `merge`(L', R') will therefore be a sorted version of S.

Work and Span. Since the subproblems can be solved independently (by assumption), the work and span of divide-and-conquer algorithms can be described using simple recurrences. In particular for a problem of size n is broken into k subproblems of size $n_1, \ldots, n_k$, then the work is

$$W(n) \;=\; W_{\text{divide}}(n) \;+\; \sum_{i=1}^{k} W(n_i) \;+\; W_{\text{combine}}(n) + 1$$

and the span is

$$S(n) \;=\; S_{\text{divide}}(n) \;+\; \max_{i=1}^{k} S(n_i) \;+\; S_{\text{combine}}(n) + 1$$

Note that the work recurrence is simply adding up the work across all components. More interesting is the span recurrence. First, note that a divide and conquer algorithm has to split a problem instance into subproblems *before* these subproblems are recursively solved. We therefore have to add the span for the divide step. The algorithm can then execute all the subproblems in parallel. We therefore take the maximum of the span for these subproblems. Finally *after* all the problems complete we can combine the results. We therefore have to add in the span for the combine step.

Applying this formula often results in familiar recurrences such as $W(n) = 2W(n/2)+O(n)$. In the rest of this lecture, we will see other recurrences.

Strengthening. In most divide-and-conquer algorithms you have encountered so far, the subproblems are occurrences of the problem you are solving. For example, in sorting we subproblems are smaller sorting instances. This is not always the case. Often, you will need more information from the subproblems to properly combine the results. In this case, you'll need to **strengthen** the problem definition. If you have seen the approach of strengthening an inductive hypothesis in a proof by induction, it is very much an analogous idea. Strengthening involves defining a problem that solves more than what you ultimately need, but makes it easier or even possible to use solutions of subproblems to solve the larger problem.

Question 4.4. *You have recently seen an instance of strengthening when solving a problem with divide and conquer. Can you think of the problem and how you used strengthening?*

In the recitation you looked at how to solve the Parenthesis Matching problem by defining a version of the problem that returns the number of unmatched parentheses on the right and left. This is a stronger problem than the original, since the original is the special case when both these values are zero (no unmatched right or left parentheses). This modification was necessary to make divide-and-conquer work—if the problem is not strengthened, it is not possible to combine the results from the two recursive calls (which tell you only whether the two halves are matched or not) to conclude that the full string is matched. This is because there can be an unmatched open parenthesis on one side that matches a close parenthesis on the other.

4.1 Example I: The Maximum Contiguous Subsequence Sum Problem

For a sequence s of n elements, let's write s_i, $0 \leq i < n$, to denote the i'th element of that sequence and use the "angle bracket" notation for writing the sequence, i.e., $s = \langle s_0, \ldots, s_{n-1} \rangle$.

We say that a sequence s' is a *contiguous subsequence* of s if $s' = s_i, s_{i+1}, \ldots, s_{i+k}$ for some k.

Example 4.5. *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).

As the name suggests, the maximum-contiguous-subsequence problem requires finding the subsequence of a sequence of integers with maximum total sum. We can make this problem precise as follows.

Definition 4.6 (The Maximum Contiguous Subsequence Sum (MCSS) Problem)**.** *Given a sequence of numbers, the* maximum contiguous subsequence sum *problem is to find*

$$\textsf{mcss}(s) = \max\left\{\sum_{k=i}^{j} s_k \ : \ 0 \le i \le n-1, 0 \le j \le n-1\right\}.$$

(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 4.7. *For $s = \langle 1, -5, 2, -1, 3\rangle$, the maximum contiguous subsequence is, $\langle 2, -1, 3\rangle$. Thus $\textsf{mcss}(s) = 4$.*

4.1.1 Algorithm 1: Using Brute Force

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

Question 4.8. *To apply brute force, where do we start?*

We first start by identifying the structure of the output. In this case, this is just a number. So shall we enumerate all numbers (sums) and check that there is a subsequence that matches than number and we continue until we can no longer find a larger sum?

That is indeed what a literal interpretation of the brute-force technique would suggest.

Question 4.9. *Would such an algorithm terminate?*

No, because we may never know when to stop unless we know the result a priori, which we don't.

Question 4.10. *Can we bound the result to guarantee non-termination?*

We can by adding up all positive numbers in the sequence and using that bound but this can still be a very large bound. Furthermore our 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 4.11. *How can we change the result type to something that we can enumerate in finite time?*

One natural choice would be to consider the contiguous subsequences directly. Indeed, we can change the result type by reducing this problem to another closely related problem: maximum contiguous subsequence. This problem requires not finding the sum but the sequence itself.

Question 4.12. *Can you see how we can solve this problem reducing it to the maximum-contiguous-subsequence problem?*

Reducing the problem requires no work because both operate on the same input, a sequence of the same type. If we have a solution to the maximum-contiguous-subsequence problem, then we can solve the maximum-contiguous-subsequence-sum problem, by simply computing the sum.

Question 4.13. *What is the work and span of the reduction?*

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

We can now apply the brute-force-technique to the problem. Again, we have to enumerate all possible results.

Question 4.14. *What are all possible results for the maximum-contiguous-subsequence problem? How do we pick the best?*

They are all contiguous subsequences, which can be represented by a pair of integers (i, j), $0 \leq i \leq j < n$. To pick the best, we simply compute their sum and pick the one with the largest. That is our algorithm for solving the maximum-contiguous-subsequence problem. Since we know how to solve the maximum-contiguous-subsequence-sum problem from this problem, we are done.

Question 4.15. *There is something strange about this algorithm. Do you see what?*

Our algorithm for solving the maximum-contiguous-subsequence problem, already computes the result for the maximum-contiguous-subsequence-sum problem. So the reduction does redundant work.

Question 4.16. *Can you see how we may eliminate this redundancy? (Perhaps by using another technique that we briefly talked about.)*

We can strengthen the problem by requiring it to return the subsequence in addition to the sum. This makes it possible to apply the brute-force technique directly to the problem, without having to go through the reduction.

Summary 4.17. *In summary, when trying to apply the brute-force technique, we have encountered a problem, which we solved by first reducing it to another problem. We then realized a redundancy in the resulting algorithm and eliminated that redundancy by strengthening the problem itself. 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.*

Specifying the algorithm and analyzing its cost. The pseudocode for this algorithm using our notation exactly matches the definition of the problem.

Question 4.18. *What is the work and span of the algorithm?*

For each subsequence $i..j$, we can compute its sum using `reduce`. This does $O(j-i)$ work and has $O(\log(j-i))$ span. Furthermore, all the subsequences can be examined independently in parallel. This leads to the following bounds:

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

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.

As you might have noticed already, this algorithm is clearly inefficient. We will apply divide and conquer to come up with a more efficient solution.

[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 $O(\log n)$

4.1.2 Algorithm 2: Refining Brute Force with a Reduction

The brute-force algorithm does a lot of redundant work.

Question 4.19. *Can you see where the redundancy is?*

The 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 many ending positions that differ by one, i.e., the sequences are all contained in each other and thus can potentially be computed much more efficiently, avoiding the redundancy.

Question 4.20. *Can you think of an algorithm for computing the maximum contiguous subsequence that starts at some particular position, say i.*

Simply scan to the right computing the sum as you go, and also maintaining the maximum sum that has been seen so far.

Question 4.21. *What is the work and span of your algorithm?*

The algorithm requires linear work and span.

Question 4.22. *Can you now see a way to improve the brute-force algorithm by reducing our original problem to the problem of finding the problem "maximum contiguous subsequence with a given start"?*

Yes, we can reduce the original problem to the "with-start" problem by considering all possible start positions and picking the best. The algorithm results in a significant decrease in work.

Exercise 4.23. *Complete this algorithm and analyze its work and span carefully.*

Question 4.24. *Can this algorithm be optimal?*

No, because we have only eliminated the aforementioned redundancy in one dimension, when solving the problem that starts at a particular position. There is still redundancy when solving for, for example, neighboring positions.

4.1.3 Algorithm 3: Divide And Conquer

To apply the divide-and-conquer technique, we first need to figure out how to divide the input.

Question 4.25. *Can you think of ways of dividing the input?*

There are many possibilities, but splitting the input in two based on the ordering is often a good first try. Note that we often split the ordering in approximately half for the purpose of getting good efficiency, but for the purpose of correctness, we can typically split anywhere, e.g. pick 10 elements off the front, and then the rest.

Question 4.26. *Why is splitting approximately in half typically a good approach.*

By splitting in approximately half we more quickly reduce the size of the largest component. This then reduces both the overall work and the overall span.

So let us split the sequence and recursively solve the problem on both parts, as illustrated by the picture below.

$$\langle \text{——}\ s_l\ \text{——} \,\|\, \text{——}\ s_r\ \text{——} \rangle$$

$$\Downarrow$$

$$s_l = \underbrace{\langle \quad \cdots \quad \rangle}_{\textsf{mcss}(s_l)} \qquad\qquad s_r = \underbrace{\langle \quad \ldots \quad \rangle}_{\textsf{mcss}(s_r)}$$

Example 4.27. *Let* $s = \langle -2, 2, -2, -2, 3, 2 \rangle$. *By using the approach, we divide the sequence into*

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

and

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

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

Now that we have the solutions from both parts, can we combine them to generate the solution for the whole problem?

Question 4.28. *Can you think of a way to solve the problem by using the solution for the two halves?*

As a first guess, you might consider simply calculating the maximum of the MCSS of the two parts, e.g., $\max(\text{mcss}(s_l), \text{mcss}(s_r))$. This answer is incorrect, unfortunately. We need a better understanding of the problem to devise a correct combine step.

Notice that the subsequence we're looking for has one of the three forms: (1) the maximum sum lies completely in the left subproblem, (2) the maximum sum lies completely in the right subproblem, and (3) the maximum sum spans across the divide point. We want to find whichever is the largest of these. The first two cases are easy and have already been solved by the recursive calls. The more interesting case is when the largest sum goes between the two subproblems. Our algorithm so far is shown in Algorithm 4.29.

Algorithm 4.29 (Simple Divide-and-Conquer MCSS)**.**

```
fun mcss(S) =
  case (showt S)
    of EMPTY = $-\infty$
     | ELT($x$) = $x$
     | NODE($L, R$) = let
         val ($m_L, m_R$) = (mcss($L$) || mcss($R$))
         val $m_a$ = bestAcross($L, R$)
       in $\max\{m_L, m_R, m_A\}$
       end
```

Question 4.30. *Can you find an algorithm for finding the subsequence with the largest sum that cuts across the divide (i.e., bestAcross(L, R))? Hint: try the problem-reduction technique to reduce the problem to another one that we know.*

We can reduce this problem to the problem of "maximum contiguous sum with a start" by noticing that the maximum sum going across the divide is the largest sum of a suffix on the left plus the largest sum of a prefix on the right.

Example 4.31. *In Example 4.27 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 $0 + 3 = 3$.*

The prefix of the right part is easy as it directly maps to the solution at position 0. For the left part, we have to reverse the sequence and ask for the position 0.

Is this algorithm correct? Does it always find the maximum contiguous subsequence sum? Before we show a proof of correctness, what do we consider a proof that an algorithm is correct.

Question 4.32. *What technique can we use to show that the algorithm is correct?*

As we briefly talked about in a previous class, 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.

Proofs of correctness. As was the case in 15-150, you are familiar with writing detailed proofs that reason about essentially every step down to the most elementary operations. You would prove your ML code was correct line by 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 we are more interested in seeing the critical steps highlighted and the standard or obvious steps summarized, with the idea being that if probed, you can easily provide detail on demand. 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.

We'll now prove that the $mcss$ algorithm above computes the maximum contiguous subsequence sum. Specifically, we're proving the following theorem:

Theorem 4.33. *Let s be a sequence. The algorithm* $\textsf{mcss}(s)$ *returns the maximum contiguous subsequence sum in s—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

$$\max\left\{\sum_{k=i}^{j} s_k \;:\; 0 \le i < 1, 0 \le j < 1\right\} = \sum_{k=0}^{0} s_0 = s_0 = x\,.$$

For the inductive step, let s be a sequence of length $n \ge 1$, and assume inductively that for any sequence s' of length $n' < n$, $\textsf{mcss}(s')$ 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. $\square$

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 4.34. *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 4.35. *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?*

4.1.4 Algorithm 4: Divide And Conquer with Strengthening

We have progressively improved our algorithm from cubic to near linear work. Is it possible to do better than $O(n \log n)$ work using divide and conquer?

Question 4.36. *In general, how do we know that we have a good algorithm?*

We can determine whether we have made enough progress or not by comparing the cost of our algorithm to a lower bound.

Question 4.37. *What is a lower bound for this problem?*

We cannot do less than linear work because we have to inspect each element of the sequence at least once to determine whether it would contribute to the result or not.

Question 4.38. *Can you see a way that we might be able to reduce the work further? Is there some redundancy in our divide-and-conquer algorithm?*

Our divide-and-conquer algorithm has another redundancy: the maximum prefix and maximum suffix must have been computed recursively to solve the subproblems. Thus, we should be avoid re-computing them.

Question 4.39. *How 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.

Question 4.40. *Let's take a step back and think about what we did. We now changed the problem slightly so that we return more information. What is the name of this technique?*

This is another application of the strengthening technique same. Now that we have figured out how to strengthen the problem let's go and solve it.

Question 4.41. *Can you see how we can update our divide and conquer algorithm to return also the maximum prefix and suffix in addition to maximum contiguous subsequence.*

So, we need to return a total of three values: the max subsequence sum, the max prefix sum, and the max suffix sum.

Having this information from the subproblems, we can now produce a similar answer tuple for all levels up, in constant work and span per level. More specifically, we strengthen our problem to return a 3-tuple `(mcss, max-prefix, max-suffix)`, and if the recursive calls return (m_1, p_1, s_1, t_1) and (m_2, p_2, s_2), then we return

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

Let's check that our answer makes sense:

Question 4.42. *Don't we have consider the case when s_1 or p_2 is the maximum?*

No, because that case is included in m_1 and m_2.

Question 4.43. *Are our prefix's and suffixes correct? Can we not have a bigger prefix that contains all of the fist sequence?*

Yes, we can. Indeed, we have a bug in our algorithm. We are not returning the prefix and the suffix correctly.

Question 4.44. *Can you think of a way to fix this problem?*

We also need to return 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. More precisely, we strengthen our problem to return a 4-tuple `(mcss, max-prefix, max-suffix, total)`, and 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 4.45 (Linear Work Divide-and-Conquer MCSS).

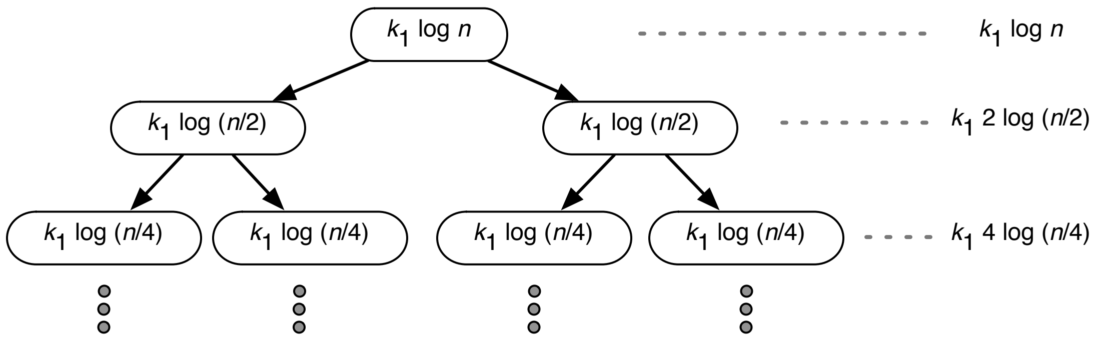
```
function mcss(a) = let
   function mcss'(a)
      case (showt a)
         of EMPTY ⇒ $(-\infty, -\infty, -\infty, 0)$
          | ELT(x) ⇒ $(x, x, x, x)$
          | NODE(L, R) =
             let
                val $((m_1, p_1, s_1, t_1), (m_2, p_2, s_2, t_2))$ = (mcss'(L) ‖ mcss'(R))
             in
                (max($s_1 + p_2, m_1, m_2$),      % overall mcss
                 max($p_1, t_1 + p_2$),           % maximum prefix
                 max($s_1 + t_2, s_2$),           % maximum suffix
                 $t_1 + t_2$)                     % total sum
             end
   val (m,_,_,_) = mcss'(a)
in m end
```

You should verify the base cases are doing the right thing. The SML code for this algorithm is at the end of these notes.

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 4.46. *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.

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

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 $W(n) \le k_1(2n - 1)\log n - 2k_1(n \log n - n + 1) \in O(n)$ because the $n \log n$ terms cancel.

4.2 Example II: The Euclidean Traveling Salesperson Problem

We'll now turn to another example of divide and conquer. In this example, we will apply it to devise a heuristic method for an **NP**-hard problem. The problem we're concerned with is a variant of the traveling salesperson problem (TSP) from Lecture 1. This variant is known as the Euclidean traveling salesperson (eTSP) problem because in this problem, the points (aka. cities, nodes, and vertices) lie in a Euclidean space and the distance measure is the Euclidean measure. More specifically, we're interested in the planar version of the eTSP problem, defined as follows:

Definition 4.47 (The Planar Euclidean Traveling Salesperson Problem)**.** *Given a set of points P in the 2-d plane, the* planar Euclidean traveling salesperson *(eTSP) problem is to find a tour of minimum total distance that visits all points in P exactly once, where the distance between points is the Euclidean (i.e. ℓ_2) distance.*

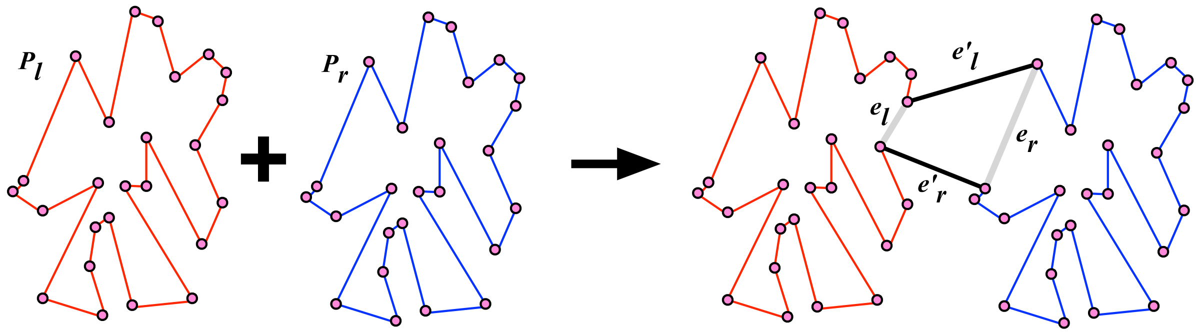
Not counting bridges, this is the problem we would want to solve to find a minimum length route visiting your favorite places in Pittsburgh. As with the TSP, it is **NP**-hard, but this problem is easier[2] to approximate.

Here is a heuristic divide-and-conquer algorithms that does quite well in practice. In a few weeks, we will see another algorithm based on Minimum Spanning Trees (MST) that gives a constant-approximation guarantee. This divide-and-conquer algorithm is more interesting than the ones we have done so far because it does work both before and after the recursive calls. Also, as we will see, the recurrence it generates is root dominated.

The basic idea is to split the points by a cut in the plane, solve the TSP on the two parts, and then somehow merge the solutions. For the cut, we can pick a cut that is orthogonal to the coordinate lines. In particular, we can find in which of the two dimensions the points have a larger spread, and then find the median point along that dimension. We'll split just below that point.

To merge the solutions we join the two cycles by making swapping a pair of edges.

[2]Unlike the TSP problem, which only has constant approximations, it is known how to approximate this problem to an arbitrary but fixed constant accuracy ε in polynomial time (the exponent of n has $1/\varepsilon$ dependency). That is, such an algorithm is capable of producing a solution that has length at most $(1+\varepsilon)$ times the length of the best tour.



To choose which swap to make, we consider all pairs of edges of the recursive solutions consisting of one edge $e_\ell = (u_\ell, v_\ell)$ from the left and one edge $e_r = (u_r, v_r)$ from the right and determine which pair minimizes the increase in the following cost:

$$\texttt{swapCost}((u_\ell, v_\ell), (u_r, v_r)) = \|u_\ell - v_r\| + \|u_r - v_\ell\| - \|u_\ell - v_\ell\| - \|u_r - v_r\|$$

where $\|u - v\|$ is the Euclidean distance between points u and v.

Here is the pseudocode for the algorithm

```
function  $\texttt{eTSP}(P)$ =
   case  $(|P|)$
      of  $0, 1$  ⇒  raise  $TooSmall$
       | $2$  ⇒  $\{(P[0], P[1]), (P[1], P[0])\}$
       | $n$  ⇒  let
            val  $(\texttt{P}_\ell, \texttt{P}_r) = \texttt{splitLongestDim}(P)$
            val  $(\texttt{L}, \texttt{R}) = (\, \texttt{eTSP}(P_\ell) \;\|\; \texttt{eTSP}(P_r) \,)$
            val  $(c, (e, e')) = \texttt{minVal}_{\texttt{first}}\, \{(\texttt{swapCost}(e, e'), (e, e')) : e \in L, e' \in R\}$
         in
            $\texttt{swapEdges}(\texttt{append}(L, R), e, e')$
         end
```

The function $\texttt{minVal}_{\texttt{first}}$ uses the first value of the pairs to find the minimum, and returns the (first) pair with that minimum. The function $\texttt{swapEdges}(E, e, e')$ finds the edges e and e' in E and swaps the endpoints. As there are two ways to swap, it picks the cheaper one.

Cost analysis Now let's analyze the cost of this algorithm in terms of work and span. We have

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

We have already seen the recurrence $S(n) = S(n/2) + O(\log n)$, which solves to $O(\log^2 n)$. Here we'll focus on solving the work recurrence.

In anticipation of recurrences that you'll encounter later in class, we'll attempt to solve a more general form of recurrences. Let $\varepsilon > 0$ be a constant. We'll solve the recurrence

$$W(n) \;=\; 2W(n/2) + k \cdot n^{1+\varepsilon}$$

by the substitution method.

Theorem 4.48. *Let $\varepsilon > 0$. If $W(n) \leq 2W(n/2) + k \cdot n^{1+\varepsilon}$ for $n > 1$ and $W(1) \leq k$ for $n \leq 1$, then for some constant κ,*

$$W(n) \;\leq\; \kappa \cdot n^{1+\varepsilon}.$$

Proof. Let $\kappa = \frac{1}{1-1/2^{\varepsilon}} \cdot k$. The base case is easy: $W(1) = k \leq \kappa_1$ as $\frac{1}{1-1/2^{\varepsilon}} \geq 1$. For the inductive step, we substitute the inductive hypothesis into the recurrence and obtain

$$\begin{aligned} W(n) &\leq 2W(n/2) + k \cdot n^{1+\varepsilon} \\ &\leq 2\kappa\left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} \\ &= \kappa \cdot n^{1+\varepsilon} + \left(2\kappa\left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon}\right) \\ &\leq \kappa \cdot n^{1+\varepsilon}, \end{aligned}$$

where in the final step, we argued that

$$\begin{aligned} 2\kappa\left(\frac{n}{2}\right)^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon} &= \kappa \cdot 2^{-\varepsilon} \cdot n^{1+\varepsilon} + k \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon} \\ &= \kappa \cdot 2^{-\varepsilon} \cdot n^{1+\varepsilon} + (1 - 2^{-\varepsilon})\kappa \cdot n^{1+\varepsilon} - \kappa \cdot n^{1+\varepsilon} \\ &\leq 0. \end{aligned}$$

□

Solving the recurrence directly. Alternatively, we could use the tree method and evaluate the sum directly. As argued before, the recursion tree here has depth $\log n$ and at level i (again, the root is at level 0), we have 2^i nodes, each costing $k \cdot (n/2^i)^{1+\varepsilon}$. Thus, the total cost is

$$\begin{aligned} \sum_{i=0}^{\log n} k \cdot 2^i \cdot \left(\frac{n}{2^i}\right)^{1+\varepsilon} &= k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\log n} 2^{-i\cdot\varepsilon} \\ &\leq k \cdot n^{1+\varepsilon} \cdot \sum_{i=0}^{\infty} 2^{-i\cdot\varepsilon}. \end{aligned}$$

But the infinite sum $\sum_{i=0}^{\infty} 2^{-i\cdot\varepsilon}$ is at most $\frac{1}{1-1/2^{\varepsilon}}$. Hence, we conclude $W(n) \in O(n^{1+\varepsilon})$.