Chapter 7

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.

7.1 Example I: Sequence Reduce

As a simple example let's start with how we may implement `reduce` using divide and conquer.

Question 7.1. *How can we do this?*

Algorithm 7.2 (Reduce via divide and conquer)**.**

```
1 fun reduce_danc f I (A) =
2   case (showt A)
3     of EMPTY = I
4      | ELT(x) = x
5      | NODE(L, R) = let
6            val (a, b) = (reduce_dandc(L) || reduce_dandc(R))
7          in
8            f(a, b)
9          end
```

Question 7.3. *Can you write the recursions for work and span?*

We can actually write the recursion for work and span as follows:

$$\begin{aligned} W(n) &= 2W(n/2) + O(1) \in O(n) \\ S(n) &= SW(n/2) + O(1) \in O(\log n). \end{aligned}$$

7.2 Example II: Merge Sort

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

Definition 7.4 (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 9 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 7.5 (Mergesort)**.**

```
1 fun sort(S) =
2   case (showt S)
3     of EMPTY = EMPTY
4      | ELT(_) = S
5      | NODE(L, R) = let
6                       val (L', R') = (sort(L) || sort(R))
7                     in
8                       merge(L', R')
9                     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 7.6. *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.

Since in divide-and-conquer algorithms, 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 $W(n) = 2W(n/2) + O(n)$. We will encounter many such recurrences in this course .

It is interesting to note that the work and span recurrence for a divide-and-conquer algorithm usually follows the recursive structure of the algorithm, but is a function of size of the arguments instead of the actual values.

For example the MergeSort algorithm leads to a recurrence of the form $W(n) = 2W(n/2)+ O(n)$. This corresponds to the fact that for an input of size n, mergeSort makes two recursive calls of size $n/2$, and also does $O(n)$ other work. In particular the merge itself requires $O(n)$ work. Similarly for span we can write a recurrence of the form $S(n) = \max(S(n/2), S(n/2))+ O(\log n) = S(n/2) + O(\log n)$. This is because the merge has spand $O(\log n$, and since the two recursive calls are made in parallel we take the maximum instead of summing them.

7.3 Example III: Sequence Scan

Let's consider a divide-and-conquer solution to the problem of scanning over a sequence.

Question 7.7. *How can we divide?*

We can divide the sequence in two halves, solve each half, and then put the results together. Putting the results together is the tricky part.

Question 7.8. *But how can we put the results together?*

Going back to our example,

$$\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$$

if we divided in the middle and scanned over the two resulting sequences we obtain (S_l, t_l) and (S_r, t_r), such that $((S_l, t_l) = (\langle 0, 2, 3, 6 \rangle, 8)$ and $(S_r, t_r) = (\langle 0, 2, 7, 11 \rangle, 12)$. Note that S_l already gives us the first half of the solution.

Question 7.9. *How do we get the second half?*

To get the second half, note that in calculating S_r in the second half we started with the identity instead of the sum of the first half, t_l. Therefore if we add the sum of the first half, t_l, to each element of S_r, we get the desired result. This leads to the following algorithm:

Algorithm 7.10 (Scan using divide and conquer)**.**

```
1  function scan f I S =
2    case showt(S) of
3      EMPTY ⇒ (⟨ ⟩, I)
4    | ELT(v) ⇒ (⟨ I ⟩, v)
5    | NODE(L, R) ⇒ let
6          val ((S_l, t_l), (S_r, t_r)) = (scan f I L || scan f I R)
7          val X_r = ⟨ f(t_l, y) : y ∈ S_r ⟩
8        in
9          (append(S_l, X_r), t_l + t_r)
10       end
```

One caveat about this algorithm is that it only works if I is really the "identity" for f, i.e. $f(I, x) = x$, although it can be fixed to work in general.

We now consider the work and span for the algorithm. Note that the joining step requires a map to add t_l to each element of S_r, and then an append. Both these take $O(n)$ work and $O(1)$ span, where $n = |S|$. This leads to the following recurrences for the whole algorithm:

$$W(n) = 2W(n/2) + O(n) \in O(n \log n)$$

$$S(n) = S(n/2) + O(1) \in O(\log n)$$

Although this is much better than $O(n^2)$ work, we can do better.

7.4 Example IV: 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 7.11 (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.*

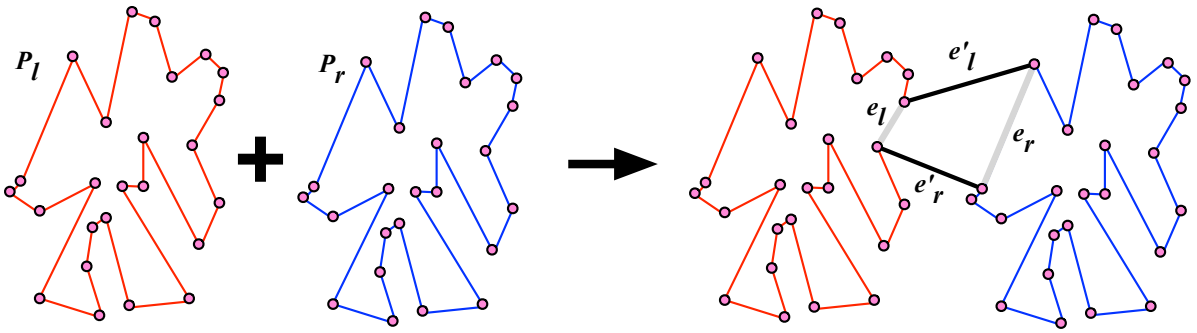
Not counting bridges, this is the problem we would want to solve to find a minimum length route visiting your favorite places in Pittsburgh. As with the TSP, it is **NP**-hard, but this problem is easier[1] to approximate.

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.

[1] 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

Algorithm 7.12.

```
1  fun eTSP(P) =
2    case (|P|) of
3       0,1 ⇒ raise TooSmall
4     | 2 ⇒ ⟨(P[0], P[1]), (P[1], P[0])⟩
5     | n ⇒ let
6           val (P_ℓ, P_r) = splitLongestDim(P)
7           val (L, R) = (eTSP(P_ℓ) || eTSP(P_r))
8           val (c, (e, e')) = minVal_first {(swapCost(e, e'), (e, e')) : e ∈ L, e' ∈ R}
9         in
10          swapEdges(append(L, R), e, e')
11        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 7.13. *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 $W(n) \in O(n^{1+\varepsilon})$.

7.5 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 7.14. *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.

7.5.1 Divide and Conquer with Reduce

Let's look back at divide-and-conquer algorithms you have encountered so far. Many of these algorithms have a "divide" step that simply splits the input sequence in half, proceed to solve the subproblems recursively, and continue with a "combine" step. This leads to the following structure where everything except what is in boxes is generic, and what is in boxes is specific to the particular algorithm.

```
1 fun myDandC(S) =
2   case showt(S) of
3     Empty ⇒ [emptyVal]
4   | Elt(v) ⇒ [base](v)
5   | Node(L, R) ⇒ let val (L', R') = (myDandC(L) || myDandC(R))
6                  in
7                    [someMessyCombine](L', R')
8                  end
```

Algorithms that fit this pattern can be implemented in one line using the sequence `reduce` function. Turning a divide-and-conquer algorithm into a reduce-based solution is as simple as invoking `reduce` with the following parameters:

$$\overline{\sum_{x \in S}}^{\boxed{\text{someMessyCombine}}\ \boxed{\text{emptyVal}}} (\boxed{\text{base}}\ x)$$

or equivalently as

`reduce` $\boxed{\texttt{someMessyCombine}}$ $\boxed{\texttt{emptyVal}}$ (`map` $\boxed{\texttt{base}}$ S)

We will take a look two examples where `reduce` can be used to implement a relatively sophisticated divide-and-conquer algorithm. Both problems should be familiar to you.

Stylistic Notes. We have just seen that we could spell out the divide-and-conquer steps in detail or condense our code into just a few lines that take advantage of the almighty `reduce`. *So which is preferable*, using the divide-and-conquer code or using reduce? We believe this is a matter of taste. Clearly, your reduce code will be (a bit) shorter, and for simple cases easy to write. But when the code is more complicated, the divide-and-conquer code is easier to read, and it exposes more clearly the inductive structure of the code and so is easier to prove correct.

Restriction. You should realize, however, that this pattern does not work in general for divide-and-conquer algorithms. In particular, it does not work for algorithms that do more than a simple split that partitions their input in two parts in the middle. For example, it cannot be used for implementing quick sort as the divide step partitions the data with respect to a pivot. This step requires picking a pivot, and then filtering the data into elements less than, equal, and greater than the pivot. It also does not work for divide-and-conquer algorithms that split more than two ways, or make more than two recursive calls.