Chapter 8

# Divide and Conquer

Divide and conquer is one of the most important algorithm-design techniques that can be used to solve a variety of computational problems. The structure of a divide-and-conquer algorithm for a problem $P$ has the following form.

**Base Case:** When the instance $I$ of the problem $P$ is sufficiently small, compute the answer $P(I)$ perhaps by using a different algorithm.

**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. First, it follows the structure of an inductive proof, and therefore usually leads to relatively simple proofs of correctness. To prove a divide-and-conquer algorithm correct, we first prove the base case is correct. Then, we we assume by strong (or structural) induction that the recursive solutions are correct, and show that, given correct solutions to smaller instances, the combined solution is a correct answer. Second, divide-and-conquer technique can lead to efficient algorithms. To ensure efficiency, we need to make sure that the divide and combine steps are efficient, and that they do not create too many sub instances. This brings us to the third nice property, which is that the work and span for a divide-and-conquer algorithm can be expressed as a mathematical equation called recurrence. Such recurrences can be solved without too much difficulty, making analyzing the work and span of many divide-and-conquer algorithms reasonably straightforward (Chapter 4). Finally, divide-and-conquer is naturally yields parallel algorithms, because we can solve the sub-instances in parallel. This can lead to significant amount of parallelism, because each inductive step can create more instances to solve in parallel. For example, even if we only divide the problem instance into two sub-instances, each of those sub-instances will themselves generate two more sub-instances, leading to a geometric progression, which quickly accumulates.

In this chapter we apply the divide-and-conquer design technique to several problems and analyze their costs by using recurrences. We also discuss another design-technique, called strengthening, that allows us to apply divide-and-conquer to a wider variety of problems.

## 8.1 Example I: Sequence Reduce

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

**Algorithm 8.1.** [Reduce via divide and conquer]

```
reduce_dc $f$ $id$ $A$ =
  if isEmpty $A$ then
    $I$
  else if isSingleton $A$ then
    $A[0]$
  else
    let
      $(L, R)$ = splitMid $A$
      $(a, b)$ = (reduce_dc $f$ $id$ $L$ || reduce_dc $f$ $id$ $R$)
    in
      $f(a, b)$
    end
```

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

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

## 8.2 Example II: MergeSort

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

**Definition 8.2.** [The comparison sorting problem] Given a sequence $A$ 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. $R_i \leq R_{i+1}, 0 < i \leq |A| - 1$.

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 a 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 11 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. Mergesort can be defined as follows.

**Algorithm 8.3.** [Mergesort]

```
merge_sort $A$ =
  **if** isEmpty $A$ or isSingleton $A$ **then**
    $A$
  **else**
    **let**
      $(L, R)$ = splitMid $A$
      $(L', R')$ = (merge_sort L || merge_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. 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 $A$, 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 $A$.

Since in divide-and-conquer algorithms, the subproblems can be solved independently, 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.

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. The resulting formulas are usually in the form of familiar recurrences such as $W(n) = 2W(n/2) + O(n)$. We will encounter many such recurrences in this book.

Assuming a linear-work, logarithmic-span algorithm for merging sorted sequences, 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 $O(n)$ work to merge the resulting sorted sequences. Similarly we can write a recurrence for the span as $S(n) = \max(S(n/2), S(n/2)) + O(\log n) = S(n/2) + O(\log n)$. This accounts for the $O(\log n)$ span of the merge algorithm, and the the fact that two recursive calls are made in parallel.

## 8.3 Example III: Sequence Scan

Let's consider a divide-and-conquer solution to the problem of scanning over a sequence. 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. Consider the sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1 \rangle$, if we divide in the middle and scan over the two resulting sequences we obtain $(L, sL)$ and $(R, sR)$, such that $((L, sL) = (\langle 0, 2, 3, 6 \rangle, 8)$ and $(R, sR) = (\langle 0, 2, 7, 11 \rangle, 12)$. Note that $L$ already gives us the first half of the solution. To compute the second half, observe that in calculating $R$ in the second half, we started with the identity instead of the sum of the first half, $sL$. Therefore, if we add the sum of the first half, $sL$, to each element of $R$, we get the desired result. This leads to the following algorithm:

**Algorithm 8.4.** [Scan using divide and conquer]

```
scan $f$ $id$ $A$ =
  if isEmpty $A$ then
    $(\langle\,\rangle, I)$
  else if isSingleton $A$ then
    $(\langle id \rangle, A[0])$
  else
    let
      $(AL, AR)$ = splitMid $A$
      $((L, sL), (R, sR))$ = (scan $f$ $id$ $AL$ || scan $f$ $id$ $AR$)
      $R'$ = $\langle f(sL, x) : x \in R \rangle$
    in
      (append $(L, R')$, $sL + sR$)
    end
```

Observe that this algorithm takes advantage of the fact that $id$ is really the "identity" for $f$, i.e. $f(id, x) = x$.

We now consider the work and span for the algorithm. Note that the combine step requires a map to add $sL$ to each element of $R$, and then an append. Both these take $O(n)$ work and $O(1)$ span, where $n = |A|$. 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 know that we can do better by using contraction (Chapter 7).

## 8.4 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 Chapter 5. 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 8.5.** [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. $\ell_2$) distance.

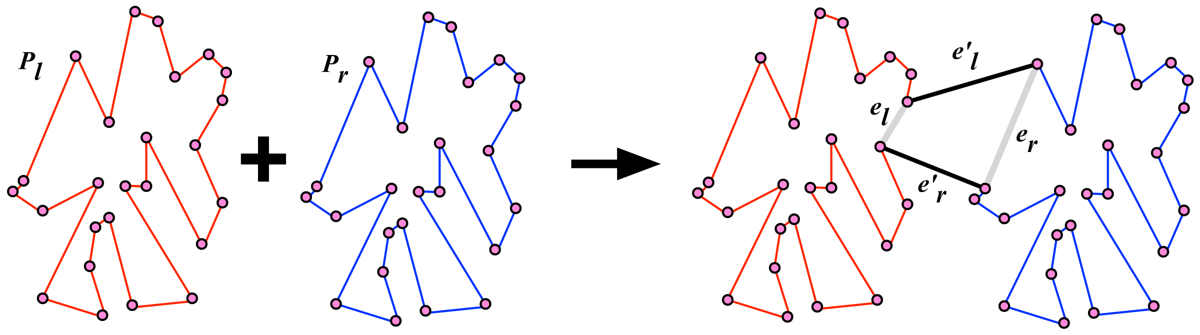
Not counting bridges, this is the problem we would want to solve to find a minimum length route visiting your favorite places in Pittsburgh. As with the TSP, it is **NP**-hard, but this problem is easier[1] to approximate.

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 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 $\varepsilon$ 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 $v$.

The pseudocode for the algorithm is shown below.

**Algorithm 8.6.**

```
eTSP($P$) =
  case ($|P|$)
  | 0,1 => raise TooSmall
  | 2 => $\langle (P[0], P[1]), (P[1], P[0]) \rangle$
  | n =>
    let
      $(\mathtt{P}_\ell, \mathtt{P}_r) = \texttt{splitLongestDim}(P)$
      $(L, R) = (\texttt{eTSP}(P_\ell) \;||\; \texttt{eTSP}(P_r))$
      $(c, (e, e')) = \texttt{minVal}_{\texttt{first}} \{(\texttt{swapCost}(e, e'), (e, e')) : e \in L, e' \in R\}$
    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

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

by the substitution method.

**Theorem 8.7.** *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 $\kappa$,*

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

*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 $W(n) \in O(n^{1+\varepsilon})$.

## 8.5 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 the 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.

### 8.5.1 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.

```
myDC $A$ =
  if isEmpty $A$ then
    $\boxed{\texttt{emptyVal}}$
  else if isSingleton $A$ then
    $\boxed{\texttt{base}}(A[0])$
  else
    let $(L, R)$ = splitMid $A$ in
        $(L', R')$ = (myDC $L$ || myDC $R$)
    in
      $\boxed{\texttt{my\_combine}}$ $(L', R')$
    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:

$$\texttt{reduce}\ \boxed{\text{my\_combine}}\ \boxed{\text{emptyVal}}\ (\texttt{map}\ x \in S\ (\boxed{\text{base}}\ x))$$

or equivalently as

$$\texttt{reduce}\ \boxed{\texttt{my\_combine}}\ \boxed{\texttt{emptyVal}}\ (\texttt{map}\ \boxed{\texttt{base}}\ S)$$

**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 Quicksort 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.

## 8.6 Problems

**8-1 Parallel merge**
Give a parallel algorithm for merging two sorted sequences. If the sequences have length $m$ and $n$, then your algorithm should perform $O(n + m)$ work in $O(\log (n + m)$ span.

**8-2 Binary expression trees**

A binary expression tree is a tree where each internal node is a binary operation such as multiply or add and each leaf is a natural number.

```
type binop = Plus | Minus | Multiply | Divide
type exp_tree = Leaf of int
              | Node of (exp_tree * binop * exp_tree)

function evaluate (t: exp_tree) = ... (* your code here *)
```

- Using divide-and-conquer design an algorithm for evaluating a given expression tree to find the value represented by the expression.
- What is the worst-case work and span of your algorithm?
- What is the best-case work and span of your algorithm?