Chapter 8

Sorting and Sequencing

If you keep proving stuff that others have done, getting confidence, increasing the complexities of your solutions—for the fun of it—then one day you'll turn around and discover that nobody actually did that one! And that's the way to become a computer scientist.

Richard Feynman, *Lectures on Computation*

In this chapter, we present two extended examples that use the programming techniques from Chapters 2–5 and the analysis ideas from Chapter 6 and 7 in the context of two interesting problems. First, we consider the problem of arranging a list in order. Then, we consider the problem of aligning genome sequences. Both examples involve quite challenging problems, and incorporate many of the ideas we have seen up to this point in the book. If you are able to understand them well now, you are well on your way to thinking like a computer scientist!

8.1 Sorting

The sorting problem takes two inputs: a list of elements, and a comparison procedure. As output, it produces a list consisting of the same elements as the input list, but ordered according to the comparison procedure. For example, if we sort a list of numbers using < as the comparison procedure, the output is the list of numbers sorted in order from least to greatest. Sorting is one of the most commonly considered problems in computer science, and many different sorting algorithms

have been developed and analyzed.[1] In this section, we will explore a few different ways of implementing a sorting procedure. We encourage curious readers to attempt to develop their own sorting procedure before continuing further.

8.1.1 Best-First Sort

A simple sorting strategy is to find the best element (that is, the one for which the comparison procedure evaluates to true when applied to that element and every other element) in the list and put that at the front. Then, find the best element from the remaining elements and put that next. Continue finding the best element of the remaining elements until no more elements remain. To define our best-first sort procedure, we will first define procedures for finding the best element in the list, and for removing an element from a list.

Finding the Best. The best element in the list is either the first element, or the best element from the rest of the list. Hence, we can define `find-best` recursively. If the list is empty, however, there is no best element. So, the base case is for a list that has one element instead of the empty list. When the input list has only one element, then that element must be the best element in the list.

```
(define (pick-better proc p1 p2)
  (if (proc p1 p2) p1 p2))

(define (find-best proc lst)
  (if (null? (cdr lst))
      (car lst)
      (pick-better proc
                   (car lst)
                   (find-best proc (cdr lst)))))
```

Assuming the procedure passed as `proc` has constant running time, the running time of `pick-better` is constant. To evaluate an application of `find-best`, there are $n - 1$ recursive applications of `find-best`, since each one passes in `(cdr lst)` as the new `lst` operand, and the base case stops when the list has

[1] Donald Knuth's *The Art of Computer Programming* devotes an entire 780-page volume to the problem of sorting and searching

one element left. The running time for each application (excluding the recursive application) is constant: it involves only applications of procedures with running times in $O(1)$. So, the total running time for `find-best` is in $\Theta(n-1)$ which is equivalent to $\Theta(n)$.

Deleting an Element. To implement best first sorting, we need to produce a list that contains all the elements of the original list except for the best element, which will be placed at the front of the output list. We do this using a procedure, `delete`, that takes as inputs a list and a value, and produces as output a list that is identical to the input list except with the first element that is equal to the value parameter removed.

```
(define (delete lst el)
  (if (null? lst) null
      (if (eq? (car lst) el) (cdr lst)
          (cons (car lst)
                (delete (cdr lst) el)))))
```

The worst case running time for `delete` occurs when no element in the list matches the value of `el` (note that in the best case, the first element matches and the running time does not depend on the length of the input list at all). In the worst case, there will be n recursive applications of `delete` where n is the number of elements in the input list. Each application has constant running time since other than the recursive application, all the other applications are of constant time procedures. Hence, the total running time for `delete` is in $\Theta(n)$ where n is the length of the input list.

Sorting. Now, we can define `best-first-sort` using `find-best` and `delete`:[2]

```
(define (best-first-sort lst cf)
  (if (null? lst) null
      (cons (find-best cf lst)
            (best-first-sort
             (delete lst
```

[2]We follow the convention of the `sort` library procedure by making the input list the first parameter, and the sorting procedure the second parameter. This is the opposite order from most of our other procedures, in which the procedure parameter is first.

```
                    (find-best cf lst))
                cf))))
```

Assuming the procedure passed as the comparison function has constant running time, the running time of the `best-first-sort` procedure grows with the length of the input list, n. There are n recursive applications of `best-first-sort`, since each application of `delete` produces an output list that is one element shorter than its input list. In addition to the constant time procedures (`null?` and `cons`), it involves two applications of `find-best` on the input list, and one application of `delete` on the input list. As analyzed earlier, each of these applications has running time in $\Theta(m)$ where m is the length of the input lists to `find-best` and `delete` (we use m here to avoid confusion with n, the length of the original input list). In the first application, this input list will be a list of length n, but in later applications it will be involve lists of decreasing length: $n-1, n-2, \cdots, 1$. Hence, the *average* length of the input lists to `find-best` and `delete` is $\frac{n}{2}$. The average running time for each of these applications is in $\Theta(\frac{n}{2})$, which is equivalent to $\Theta(n)$. There are three applications (two of `find-best` and one of `delete`) for each application of `best-first-sort`, so the total running time for each application is in $\Theta(3n)$, which is equivalent to $\Theta(n)$. There are n recursive applications, each with average running time in $\Theta(n)$, so the running time for `best-first-sort` is in $\Theta(n^2)$. Doubling the length of the input list will quadruple the expected running time.

Let. Each application of the `best-first-sort` procedure involves two evaluations of `(find-best cf lst)`, a procedure with running time in $\Theta(n)$ where n is the length of the input list. The result of both evaluations is the same, so there is no need to evaluate this expression twice.

We could avoid this by defining a helper procedure and passing in the result of the evaluation:

```
(define (best-first-sort lst cf)
  (define (combine-helper lst best cf)
    (cons best
          (best-first-sort
           (delete lst best) cf)))
  (if (null? lst) null
      (combine-helper
       lst (find-best cf lst) cf)))
```

A more natural way to avoid this duplication is to use a *let expression*.

The grammar rule for the let expression is:

Expression	::⇒	*LetExpression*
LetExpression	::⇒	`(let (`*Bindings*`)` *Expression*`)`
Bindings	::⇒	*Binding Bindings*
Bindings	::⇒	ϵ
Binding	::⇒	`(`*Name Expression*`)`

The evaluation rule is:

> **Evaluation Rule 6: Let.** To evaluate a let expression, evaluate each binding in order. To evaluate each binding, evaluate the binding expression and bind the name to the value of that expression. Then, the value of the let expression is the value of the body expression evaluated with the names in the expression that match binding names substituted with their bound values.[3]

A let expression can be transformed into an equivalent application expression. The let expression

```
(let ((Name1 Expression1)
      (Name2 Expression2)
      ...
      (Namek Expressionk))
    Expression)
```

is equivalent to the application expression:

```
((lambda (Name1
          Name2
```

[3]Note that the grammar and evaluation rule for let here are slightly different from the version in Problem Set 3. Here, we use a simpler version of the let rules, where the body is a single expression instead of a list of one or more expressions.

```
          ...
          Namek)
    Expression)
 Expression1
 Expression2
 ...
 Expressionk)
```

The advantage of the let expression syntax is it puts the expressions next to the names to which they are bound. For example, the let expression:

```
(let ((a 2)
      (b (* 3 3)))
  (+ a b))
```

is easier to understand than the corresponding application expression:

```
((lambda (a b) (+ a b)) 2 (* 3 3))
```

Using a let expression, we can now define `best-first-sort` to avoid the duplicate evaluations of `find-best` without needing a helper procedure:

```
(define (best-first-sort lst cf)
  (if (null? lst) null
      (let ((best (find-best cf lst)))
        (cons best
              (best-first-sort
               (delete lst best) cf)))))
```

This should improve the running time, but it does not change the asymptotic growth rate. The running time is still in $\Theta(n^2)$ since there are n recursive applications of `best-first-sort` and each application involves linear time applications of `find-best` and `delete`.

Exercise 8.1. Use the `time` special form (see Chapter 6) to measure the actual evaluation times for applications of the different `best-first-sort` procedures. See if the results in your interpreter match the expected running times based

on the analysis that the running time of the procedure is in $\Theta(n^2)$. Also, compare the running times of the first and final `best-first-sort` procedures. Are the timing results consistent with the analysis? $\diamondsuit$

Exercise 8.2. Define the `find-best` procedure using `accumulate` (from Section 5.3.2). $\diamondsuit$

Exercise 8.3. ($\star\star$) Instead of sorting the elements by finding the best element first and putting at the front of the list, we could sort by finding the worst element first and putting it at the end of the list. Define a `worst-last-sort` procedure that sorts this way and analyze its running time. $\diamondsuit$

8.1.2 Insertion Sort

The `best-first-sort` procedure seems to be quite inefficient. For every output element, we are searching the whole remaining list to find the best element, but do nothing of value with all the comparisons that were done to find the best element. An alternate approach is to build up a sorted list as we go through the elements.

Insertion sort works by putting the first element in the list in the right place in the result of sorting the rest of the elements. First, we define the `insert-one` procedure that takes three inputs: an element, a list, and a comparison function. The input list must be sorted according to the comparison function. As output, `insert-one` produces a list consisting of the elements of the input list, with the input element inserts in the right place according to the comparison function.

```
(define (insert-one el lst cf)
  (if (null? lst) (list el)
      (if (cf el (car lst)) (cons el lst)
          (cons (car lst)
                (insert-one el (cdr lst) cf)))))
```

The running time for `insert-one` is in $\Theta(n)$ where n is the number of elements in the input list. In the worst case, the input element belongs at the end of the list

and we need to make n recursive applications of `insert-one`. Each application involves constant work, so the overall running time is in $\Theta(n)$.

To sort the whole list, we need to insert each element in the sorted list that results from sorting the rest of the elements:

```
(define (insert-sort lst cf)
  (if (null? lst) null
      (insert-one (car lst)
                  (insert-sort (cdr lst) cf) cf)))
```

Evaluating an application of `insert-sort` on a list of length n involves evaluating n recursive applications of `insert-sort`, for lists of length n, $n-1$, $n-2$, ..., 0. Each application includes an application of `insert-one` which has running time in $\Theta(n)$ where n is the number of elements in the input list to `insert-one`. The average length of the input list over all the applications is $\frac{n}{2}$, so the average running time of the `insert-one` applications is in $\Theta(n)$. Since there are n applications of `insert-one`, the total running time is in $\Theta(n^2)$.

Exercise 8.4. We analyzed the worst case running time of `insert-sort` above. Analyze the best case running time. Your analysis should describe the input for which `insert-sort` runs fastest, and explain what the asymptotic running time is for the best case input. $\diamondsuit$

Exercise 8.5. Both the `best-first-sort` and `insert-sort` have running times in $\Theta(n^2)$. How do their actual running times compare? Are there any inputs for which `best-first-sort` is faster than `insert-sort`? For sorting a long list of n random elements, how long would each procedure take?

To answer these questions, you will need to time the actual running times of the procedures in your Scheme interpreter. You may also find it helpful to define a procedure that constructs a list of n random elements. To select random elements, you may use the library procedure, `(random` k`)`, that evaluates to a random number between 0 and $k-1$. Be careful in your time measurements that you do not include the time required to generate the input list. Instead, use a `define` to

bind the input list to a name, and use that name as the operand expression in your timing tests. ♢

8.1.3 Quicker Sorting

Although insertion sort is generally faster than best-first sort, its running time is still in $\Theta(n^2)$ which is woefully inefficient for sorting a long list. For example, if it takes $\frac{1}{10}$ of a second to sort a list of 1000 elements using `insert-sort` (on my laptop, the time to sort a random list of 1000 elements averages 0.13 seconds), we would expect it to take 1000^2 as long (over a day!) to sort a list containing one million $(1000 * 1000)$ elements.

The problem with our insertion sort is it divides the work into inserting one element, and sorting the rest of the list. This is a very unequal division. As long as we sort by considering one element at a time and putting it in the sorted position as is done by `find-best-sort` and `insert-sort`, our sort procedures will have running times in $\Omega(n^2)$. We cannot do better than this with this strategy since there are n elements, and the time required to figure out where each element goes is in $\Omega(n)$. To do better, we need to either reduce the number of recursive applications needed to sort the list (this would mean each recursive call results in more than one element being sorted), or reduce the time required for each application.

The approach we take is to use each recursive application to divide the list into two approximately equal-sized parts, but to do the division in such a way that the results of sorting the two parts can be combined directly to form the result. This means, we should divide the elements in the list so all elements in the first part are less than (according to the comparison function) all elements in the second part.

Our first attempt is to modify `insert-one` to divide the list in two parts. First, we define the `sublist` procedure that takes three inputs: a list, and two numbers indicating the start and end positions. As output, it produces a sublist of the input list, between the start and end position.

```
(define (sublist lst start end)
  (if (= start 0)
      (if (= end 0)
          null
```

```
          (cons (car lst)
                (sublist (cdr lst) start (- end 1))))
      (sublist (cdr lst) (- start 1) (- end 1))))
```

The running time of the `sublist` procedure is in $\Theta(n)$ where n is the number of elements in the input list. The worst case input is when the value of `end` is the length of the input list, which means there will be n recursive applications, each involving a constant amount of work.

We use `sublist` to define procedures for obtaining lists of the first and second halves of the elements of an input list (when the list has an odd number of elements, we define the `first-half` to contain one more element than the `second-half`).

```
(define (first-half lst)
  (sublist lst 0
           (floor (/ (+ 1 (length lst)) 2))))

(define (second-half lst)
  (sublist lst (floor (/ (+ 1 (length lst)) 2))
           (length lst)))
```

The `first-half` and `second-half` procedures apply `sublist`, so they have running times in $\Theta(n)$ where n is the number of elements in the input list.

Using `first-half` and `second-half`, we can define the `insert-one` procedure to only consider the appropriate half of the list.

```
(define (insert-one el lst cf)
  (if (null? lst)
      (list el)
      (if (null? (cdr lst))
          (if (cf el (car lst))
```

```
      (cons el lst)
      (list (car lst) el))
  (let ((front (first-half lst))
        (back (second-half lst)))
    (if (cf el (car back))
        (append (insert-one el front cf) back)
        (append front (insert-one el back cf)))))))
```

In addition to the normal base case, we need a special case for when the input list has one element. If the element to be inserted is before this element, the output is produces using `cons`; otherwise, we can list the first (only) element in the list followed by the inserted element. In the recursive case, the `first-half` and `second-half` procedures are used to divide the input list elements. We use a let expression to bind the results of the first and second halves to the `front` and `back` variables.

Since the list passed to `insert-one` must be sorted, the elements in `front` must all be less than the first element in `back`. Hence, we can determine into which of the sublists contains the element should be inserted using just one comparison: if the element is before the first element in `back`, then it is in the first half, so we produce the result by appending the result of inserting the element in the front half (the recursive call) with the back half unchanged; if the element is not before the first element in `back`, then it must be in the second half, so we produce the result by keeping the front half as it is, and appending it with the result of inserting the element in the back half.

We have not changed `insert-sort` at all, so there are still n applications of `insert-one`, and the average length of the input list is $\frac{n}{2}$. The running time for the new `insert-one` procedure may be different, however (in fact, we will see that grows asymptotically at the same rate as the previous `insert-one` procedure, but analyzing it will lead to a faster alternative).

Unlike other recursive list procedures we have analyzed, the number of recursive applications of `insert-one` does not scale linearly with the length of the input list. The reason for this is instead of using `(cdr lst)` in the recursive application, `insert-one` passes in either the `front` or `back` value. This is the result of an application of `(first-half lst)` or `(second-half lst)`. The length of the list produced by these procedures is approximately $\frac{1}{2}$ the length of the input list. With each recursive application, the size of the input list is halved.

This means, doubling the size of the input list only adds one more recursive application.

Before proceeding further, we briefly review logarithms, since we will need them in our analysis.

Logarithms. The *logarithm* ($\log_b$) of a number n is the number x such that $b^x = n$ where b is the *base* of the logarithm. If the base is 10, then the value of $\log_{10} n$ is the number x such that $10^x = n$. For example, $\log_{10} 10 = 1$ and $\log_{10} 1000 = 3$. In computer science, we will most commonly encounter logarithms with base 2. Doubling the input value, increases the value of its logarithm base two by one: $\log_2 2n = 1 + \log_2 n$. This corresponds to the situation with `insert-one`, where doubling the size of the input increases the number of recursive applications by one.

We can change the base of a logarithm using the following formula:

$$\log_b n = \frac{1}{\log_k b} \log_k n$$

Changing the base of a logarithm from k to b changes the value by the constant factor, $\frac{1}{\log_k b}$. Hence, inside our asymptotic operators (O, Ω, and Θ) the base of the logarithm (as long as it is constant) does not matter. Thus, we use $O(\log n)$ without specifying the base of the logarithm.

The `log` primitive procedure computes natural logarithms (logarithms where the base is $e \approx 2.71828$). To compute a logarithm of any base we can use the formula above. The `logb` procedure below takes one parameter, the base of the logarithm, and evaluates to a procedure that computes logarithms using that base.

```
(define (logb b)
  (lambda (n)
    (/ (log n) (log b))))
```

For example, `((logb 10) 10)` evaluates to 1.0.[4]

[4]Because of rounding errors, the values produced by this function will not match the correct logarithm values exactly. For example, evaluating `((logb 10) 1000)` should produce 3.0, but actually produces 2.9999999999999996. Computers cannot represent all floating point numbers exactly since there are infinitely many floating point numbers but computers have only

Analysis. Returning to our analysis of `insert-one`, the number of recursive applications is in $\Theta(\log n)$ since doubling the size of the input requires one more recursive application. Each application of `insert-one` involves an application of `append` where the first parameter is either the front half of the list, or the result of inserting the element in the front half of the list. In either case, it is a list of length approximately $\frac{n}{2}$, and `append` has running time in $\Theta(m)$ where m is the length of the first input list. So, the time required for each `insert-one` application is in $\Theta(n)$ where n is the length of the input list to `insert-one`. The lengths of the input lists to `insert-one` are $\frac{n}{2}$, $\frac{n}{4}$, $\frac{n}{8}$, ..., 1. There are $\log_2 n$ terms, and the sum of the list is n, so the average length input is $\frac{n}{\log_2 n}$. Hence, the total running time for each application of `insert-one` is in $\Theta(\log_2 n * \frac{n}{\log_2 n}) = \Theta(n)$.

The analysis of the applications of `first-half` and `second-half` is similar: each of these requires running time in $\Theta(n)$ where n is the length of the input list, which averages $\frac{n}{\log_2 n}$ where n is the length of the input list of `insert-one`.

Since `insert-sort` involves $\Theta(n)$ applications of `insert-one` (with average input list length of $\frac{n}{2}$, the total running time for `insert-sort` using the new `insert-one` procedure is still in $\Theta(n^2)$. Because of the cost of evaluating the `append`, `first-half`, and `second-half` applications, the change to dividing the list in halves has not improved the asymptotic performance; in fact, because of all the extra work in each application, the actual running time has been increased.

8.1.4 Binary Trees

The problem with our `insert-one` procedure is that it has to `cdr` down the whole list to get to the middle of the list, hence the work for each application will be in $\Omega(n)$. What we need is some way of getting to the middle of the list quickly. With the standard list data structure this is impossible: it requires one `cdr` application to get to the next element in the list, so getting to the middle of the list requires approximately $\frac{n}{2}$ applications of `cdr`. There is no way to do better

---

finite memory. Hence, calculations involving floating point numbers are prone to such rounding errors. These errors may be small, but frequently cause serious problems in critical systems. One infamous example is the failure of the Patriot missile system (see the *GAO Patriot Missile Defense Report*).

than this without changing the way we represent our data. What we need is a data structure where a single application is enough to get to the middle of the list.

The data structure we will use is known as a *sorted binary tree*. Whereas a list provides constant time procedures for accessing the first element and the rest of the elements, with a binary tree we have constant time procedures for accessing the *current* element, the *left* side of the tree, and the *right* side of the tree. The left and right sides of the tree are themselves trees. So, like a list, a binary tree is a recursive data structure. We define a tree as:

tree ::⇒ `null`
tree ::⇒ `(make-tree` *element tree tree*`)`

The `make-tree` procedure can be defined using `cons` to package the three inputs into a tree:

```
(define (make-tree left element right)
  (cons element (cons left right)))
```

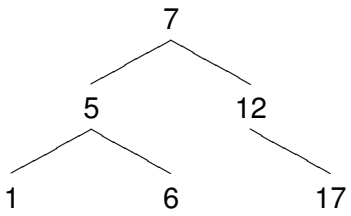
Then, we can define selector procedures for extracting the parts of a non-null tree:

```
(define (tree-element tree)
  (car tree))

(define (tree-left tree)
  (car (cdr tree)))

(define (tree-right tree)
  (cdr (cdr tree)))
```

The `tree-left` and `tree-right` procedures are constant time procedures that evaluate to the left or right halves respectively of a tree.

We will use a tree where the elements are maintained in a sorted structure. All elements in the left side of a node are less than (according to the comparison function) the value of the element of the node; all elements in the right side of a

node are greater than or equal to the value of the element of the node (the result of comparing them to the element value is false). For example, here is a sorted binary tree containing 6 elements using < as the comparison function:



The top node has element value 7, and its left subtree is a tree containing the tree elements whose values are less than 7. The null subtrees are not shown. For example, the left subtree of the element whose value is 12 is null. Note that although there are 6 elements in the tree, we can reach any element from the top by following at most 2 branches. By contrast, with a list of 6 elements, we would need 5 `cdr` operations to reach the last element.

The `first-half`, `second-half`, and `append` operations that had running times in $\Theta(n)$ for our list representation can be implemented with running times in $O(1)$ using the tree representation. For example, `first-half` can be implemented using `tree-left`, and `second-half` can be implemented using `tree-right`. To implement `append` requires making a new tree, which is also a constant time procedure.

Here is a definition of `insert-one-tree` that inserts an element in a sorted binary tree.

```
(define (insert-one-tree cf el tree)
  (if (null? tree)
      (make-tree null el null)
      (if (cf el (tree-element tree))
          (make-tree
           (insert-one-tree cf el
                            (tree-left tree))
           (tree-element tree)
           (tree-right tree))
          (make-tree
```

```
                (tree-left tree)
                (tree-element tree)
                (insert-one-tree cf el
                                 (tree-right tree))))))
```

When the input tree is null, the new element is the top element of a new tree whose left and right subtrees are null. Otherwise, it compares the element to insert to the element at the top node of the tree. If the comparison evaluates to a true value, the new element belongs in the left subtree. The result is a tree where the left tree is the result of inserting this element in the old left subtree, and the element and right subtree are the same as they were in the original tree. For the alternate case, the element is inserted in the right subtree, and the left subtree is unchanged.

Unlike `insert-one` for lists, the `insert-one-tree` procedure involves no applications of non-constant time procedures, except for the recursive application. Assuming the tree is well balanced (that is, the left and right subtrees contain the same number of elements), each recursive application halves the size of the input tree. Hence, the running time of `insert-one-tree` to insert an element in a well balanced tree is in $\Theta(\log n)$.

To use our `insert-one-tree` procedure to perform sorting we need to extract a list of the elements in the tree in the correct order. The leftmost element in the tree should be the first element in the list. Starting from the top node, all elements in its left subtree should appear before the top element, and all the elements in its right subtree should follow it. The `extract-elements` procedure does this:

```
(define (extract-elements tree)
  (if (null? tree) null
      (append
       (extract-elements (tree-left tree))
       (cons (tree-element tree)
             (extract-elements
              (tree-right tree))))))
```

The total number of applications of `extract-elements` is between n (the number of elements in the tree) and $3n$ since there can be up to two null trees for each leaf element. For each application, we need to evaluate an `append` application, where the first parameter is the elements extracted from the left subtree.

The end result of all the `append` applications is the output list, containing the n elements in the input tree. Hence, the total size of all the appended lists is at most n, and the running time for all the `append` applications is in $\Theta(n)$. Since this is the *total* time for all the `append` applications, not the time for *each* application of `extract-elements`, the total running time for `extract-elements` is the time for the recursive applications, in $\Theta(n)$, plus the time for the append applications, in $\Theta(n)$, which is in $\Theta(n)$.

Finally, we use `insert-one-tree` and `extract-elements` to define the `insert-sort-tree` procedure that takes a list and a comparison function, and evaluates to a sorted binary tree containing the elements in the list. To produce the sorted list, we apply `extract-elements` to the result of `insert-sort-helper`.

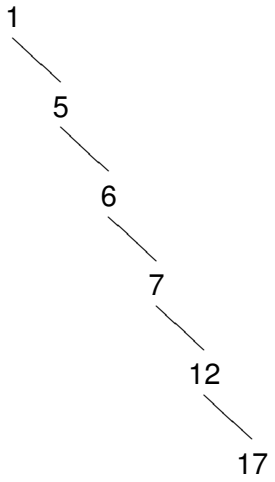
```
(define (insert-sort-tree lst cf)
  (define (insert-sort-helper lst cf)
    (if (null? lst) null
        (insert-one-tree
         cf (car lst)
         (insert-sort-helper (cdr lst) cf))))
  (extract-elements
   (insert-sort-helper lst cf)))
```

Assuming well-balanced trees as above (we will revisit this assumption later), the expected running time of `insert-sort-tree` is in $\Theta(n \log n)$ where n is the size of the input list. There are n applications of `insert-sort-helper` since each application uses `cdr` to reduce the size of the input list by one. Each application involves an application of `insert-one-tree` (as well as only constant-time procedures), so the expected running time of each application is in $\Theta(\log n)$. Hence, the total running time for `insert-sort-helper` is in $\Theta(n \log n)$ since there are n applications of `insert-one-tree` which has expected running time in $\Theta(\log n)$. In addition to the application of `insert-sort-helper`, there is also an evaluation of an application of `extract-elements`. As analyzed above, this has running time in $\Theta(n)$, which means it does not impact the overall running time growth. Thus, the expected total running time for an application of `insert-sort-tree` to a list containing n elements is in $\Theta(n \log n)$.

Now, we return to the well-balanced trees assumption. Our analysis depended on the left and right halves of the tree passed to of `insert-one-tree` having

approximately the same number of elements. If the input list is in random order, this assumption is valid: each element we insert has equal probability of going in the left or right half, so the halves will maintain approximately the same number of elements all the way down the tree. But, if the input list is not in random order this may not be the case.

For example, suppose the input list is already in sorted order. Then, each element that is inserted will be the rightmost node in the tree when it is inserted. This produces an unbalanced tree like the one shown below.



This tree contains the same six elements as the earlier example, but because it is not well-balanced the number of branches that must be traversed to reach the deepest element is 5 instead of 2. Similarly, if the input list is in reverse sorted order, we will have an unbalanced tree where only the left branches are used.

In these pathological situations, the tree becomes effectively a list. The number of recursive applications of `insert-one-tree` needed to insert a new element will not be in $\Theta(\log n)$, but rather will be in $\Theta(n)$. Hence, the worst case running time for `insert-sort-tree` is in $\Theta(n^2)$ since the worst case time for `insert-one-tree` is in $\Theta(n)$ and there are $\Theta(n)$ applications of `insert-one-tree`.

8.1.5 Quicksort

Although building and extracting elements from trees allows us to sort with expected time in $\Theta(n \log n)$, the constant time required to build all those trees and extract the elements from the final tree is high. In fact, we can use the same approach to sort without needing to build trees. Instead, we keep the two sides of the tree as separate lists, and sort them recursively.

The `quicksort` procedure uses `filter` (from Example 5.3.3) to divide the input list into sublists containing elements below and above the comparison element, and then recursively applies `quicksort` to sort those sublists.

```
(define (quicksort lst cf)
  (if (null? lst) lst
      (append
       (quicksort
        (filter
         (lambda (el) (cf el (car lst)))
         (cdr lst))
        cf)
       (list (car lst))
       (quicksort
        (filter
         (lambda (el) (not (cf el (car lst))))
         (cdr lst))
        cf))))
```

This is the *quicksort* algorithm that was invented by Sir C. A. R. (Tony) Hoare in 1962. Quicksort is probably the most widely used sorting algorithm.[5]

As with `insert-sort-tree`, the expected running time for a randomly arranged list is in $\Theta(n \log n)$ and the worst case running time is in $\Theta(n^2)$. In the expected cases, each recursive call halves the size of the input list (since if the list is randomly arranged we expect about half of the list elements are below the value

[5] As we will see in Chapter **??**, there is no general sorting procedure that has expected running time better than $\Theta(n \log n)$, so there is no algorithm that is asymptotically faster than quicksort. There are, however, sorting procedures that may have advantages such as how they use memory which may provide better absolute performance in some situations.

of the first element), so there are approximately $\log n$ expected recursive calls. Each call involves an application of `filter`, which has running time in $\Theta(m)$ where m is the length of the input list. At each call depth, the total length of the inputs to all the calls to `filter` is n since the original list is subdivided into 2^d sublists, which together include all of the elements in the original list. Hence, the total running time is in $\Theta(n \log n)$ in the expected cases where the input list is randomly arranged. As with `insert-sort-tree`, the worst case running time is still in $\Theta(n^2)$.

Exercise 8.6. Estimate the time it would take to sort a list of one million elements using `quicksort`. $\diamondsuit$

Exercise 8.7. Both the `quicksort` and `insert-sort-tree` procedures have expected running times in $\Theta(n \log n)$. How do their actual running times compare? $\diamondsuit$

Exercise 8.8. Is there a best case input for `quicksort`? Describe it and analyze the asymptotic running time for `quicksort` on best case inputs. $\diamondsuit$

8.2 Genome Alignment

A genome is a sequence of nucleotides, represented using the letters `a` (Adenine), `g` (Guanine), `c` (Cytosine), and `t` (Thymine). As a species evolves through the generations, mutations will change the sequence of nucleotides in its genome. Mutations may replace one nucleotide with a different nucleotide. They may also insert additional nucleotides in the sequence, or remove them. In order to understand how two organisms are related, a biologist wants to compare their genomes to identify similarities and differences. Because of the insertions and deletions, however, it is difficult to compare the genomes directly. Instead, we want to *align* them in a way that best reveals their similarities.

The *sequence alignment problem* takes as input two or more sequences, and produces as output an arrangement of those sequences that highlights their similarities and differences. This is done by introducing gaps (typically denoted using dashes) in the sequences so the similar segments are aligned. Sequence alignment is an important component of many genome analyses. For example, it is used to de-

termine which sequences are likely to have come from a common ancestor and to construct phylogenetic trees that explain the most likely evolutionary relationships among a group of genes, species, or organisms.

To identify a good sequence alignment, we need a way of measuring how well an arrangement is aligned. This is done using a *goodness* metric, that takes as input the two aligned sequences and calculates a score that indicates how well aligned they are. We will use a simple goodness metric where the goodness can be computed by summing the score for each position in the resulting alignment using a scoring function. The score function takes two inputs, representing the nucleotide or gap at corresponding positions in the aligned genomes:

$$Score(a,b) = \begin{cases} c & : \quad a = b \\ 0 & : \quad a \neq b \text{ (and neither } a \text{ or } b \text{ is a gap)} \\ -g & : \quad a \text{ or } b \text{ is a gap} \end{cases}$$

The values of c and g are constants chosen to reflect the relative likelihood of a point mutation and insertion or deletion. For these examples, we will use values of $c = 10$ and $g = 2$.

For example, suppose the input genomes are:

U = `catcatggaa`
V = `catagcatgg`

To compute the goodness score, we sum the scores at each position and subtract the gap penalties. If we align the genomes with no gaps, we have (the vertical lines connect matching nucleotides in the alignment):

```
catcatggaa
|||
catagcatgg
```

and the goodness score is $3c = 30$. This is not the best possible alignment of the sequences. We can do better by inserting gaps:

```
cat--catggaa
|||   |||||
catagcatgg--
```

This alignment has goodness score $8c - 4g = 72$.

The `find-best-alignment` procedure takes as input two genome sequences and produces an alignment of the genomes that has the maximum possible goodness score (for simplicity, the values of `c` and `g` are defined, rather than passed as parameters). The output is a list consisting of the two genomes with gaps inserted to produce the best possible alignment.

To represent a genome we will use a list of characters. Characters are primitives. A character is denoted by `#\`*letter*. For example, the letter `a` is `#\a`. We define the `gap` character for representing a gap in an aligned genome:

```
(define gap #\-)
```

Ambitious students should attempt to define `find-best-alignment` themselves before reading further.

First, we define a procedure to compute the goodness score of a given alignment. This procedure will take an alignment as its input, and produce a number as its output that is the goodness score of the input alignment. The input is a pair of two aligned genomes (that is, genomes of the same length with gaps inserted). To compute the goodness score, we need to compare the corresponding elements of each genome. Each genome is represented by a list of characters (each of which either represents a nucleotide or a gap).

We define a scoring procedure that takes two inputs that are either a nucleotide or a gap, and evaluates to the score for that position. The `score` procedure implements the $Score$ function defined above:

```
(define (score a b)
  (if (eq? a b) 10
      (if (or (eq? a gap)
              (eq? b gap))
          -2
          0)))
```

The running time of the `score` procedure is constant.

The `goodness` procedure needs to compute the sum of applying `score` to every pair of aligned nucleotides or gaps. To do this we need a version of the `map`

procedure that operates simultaneously on two lists[6]. The `map2` procedure does this:

```
(define (map2 proc list1 list2)
  (if (null? list1)
      null
      (cons (proc (car list1) (car list2))
            (map2 proc (cdr list1) (cdr list2)))))
```

The input lists must be the same length. The running time of `map2` is in $\Theta(n)$ where n is the number of elements in each input list. Then, we can define `goodness` using `map2` by applying `score` to each position in the aligned genomes, and summing the result (using the `sumlist` procedure defined in Section 5.3.1):

```
(define (goodness align)
  (sumlist (map2 score (car align) (cdr align))))
```

The running time of `goodness` is in $\Theta(n)$ where n is the number of elements in each input list (that is, each component of the alignment). It involves an application of `map2` to the input lists, this has running time in $\Theta(n)$ where n is the number of elements in each input list. Then, it applies `sumlist` to the resulting list of length n. The running time of `sumlist` is linear in the length of the input. Hence, the total running time is in $\Theta(n) + \Theta(n)$ which is equivalent to $\Theta(n)$.

A brute force technique for finding the best alignment is to try all possible alignments, and keep the one with the best goodness score. We can do this by recursively trying all three possibilities at each position:

- No gap
- Insert a gap for the first sequence
- Insert a gap for the second sequence

[6]In fact, the built-in `map` procedure already does this. It can take any number of input lists, and applies the procedure to one element from each list.

(The fourth possibility of inserting a gap in both sequences makes no sense, since we could always improve the goodness score by removing that double gap.)

To pick the best alignment, we use the `find-best` procedure (from Section 8.1.1) that takes a procedure and a list as inputs, and outputs the element in the list that produces true when compared with every other element in the list. As we analyzed in Section 8.1.1, its running time is in $\Theta(n)$ where n is the length of the input list.

Finally, we define `find-best-alignment` using `find-best`:

```
(define (pad-with-gaps lst n)
  (if (= n 0)
      lst
      (append (pad-with-gaps lst (- n 1))
              (list gap))))

(define (prepend-alignment c1 c2 p)
  (cons (cons c1 (car p))
        (cons c2 (cdr p))))

(define (find-best-alignment u v)
  (if (or (null? u) (null? v))
      (cons (pad-with-gaps u (length v))
            (pad-with-gaps v (length u)))
      (find-best
       (lambda (a1 a2)
          (> (goodness a1) (goodness a2)))
       (list
        (prepend-alignment
         (car u) (car v)
         (find-best-alignment (cdr u) (cdr v)))
        (prepend-alignment
         gap (car v)
         (find-best-alignment u (cdr v)))
        (prepend-alignment
         (car u) gap
         (find-best-alignment (cdr u) v))))))
```

The `find-best-alignment` procedure takes two inputs, each of which rep-

resents a genome sequence. The base case is when either of the lists is empty. The result is an alignment where there is a gap aligned with each element in the other list. The `pad-with-gaps` procedure produces the necessary padding. It takes two inputs, a list and a number `n`, and appends `n` gaps to the end of the input list. There will be n recursive applications, where n is the value of the input parameter `n`.

Each application involves an application of `append`, which has running time in $\Theta(m)$ where m is the length of the first input list. The length of the input list is initially the length of the `lst` parameter to `pad-with-gaps`, but it grows by 1 every time a gap is added, up to the length of `lst` + n. Hence, the running time of `pad-with-gaps` is in $\Theta(n(m+n))$ where m is the length of the input list, and n is the value of the second parameter. Since the maximum values of n and m are the lengths of the input lists to `find-best-alignment`, the running time of `find-best-alignment` is in $\Theta(uv)$ where u and v are the respective lengths of the input parameters to `find-best-alignment`. Since this is only done once for the entire evaluation, it will not be a significant term in our final result.

In the recursive case, we use `find-best` with a comparison procedure that uses the `goodness` procedure to compare the goodness scores of two alignments. This will evaluate to the best alignment from the list of possible alignments passed in as the second parameter. This list includes the three options mentioned earlier:

- No gap — align the first elements, and find the best alignment of the rest of the elements.
- Gap in the first genome — align a gap in the first genome with the first element in the rest of the genome, and find the best alignment of the first genome with the rest of the elements in the second genome.
- Gap in the second genome — align the first element of the first genome with a gap, and find the best alignment of the rest of the first genome with the elements in the second genome.

The `prepend-alignment` procedure creates the full alignments, but putting the first elements (or gaps) back at the front of each aligned genome.

The `find-best-alignment` procedure makes three recursive applications, one corresponding to each option. The total size of the input is the sum of the

sizes of `u` and `v`, so increasing the size by one approximately triples the number of recursive calls. This means the number of recursive applications scales as $\Theta(k^w)$ where w is the total length of the two input lists and k is some constant between 2 and 3.[7]

There's no sense in being precise when you don't even know what you're talking about.
John von Neumann

Each application involves an evaluation of `find-best` which has running time in $\Theta(n)$ where n is the length of the input list. Since the input list is always a list of three elements, this is a constant amount of time. There is an invalid assumption however: that the procedure passed to `find-best` does not have constant running time. It involves two applications of `goodness`, on the candidate genomes. The running time of `goodness` is in $\Theta(n)$ where n is the total length of the input lists. In this case, the total length is length of both alignments. The maximum length of an alignment is $u + v$, since we never insert gaps at the same position in both genomes. Hence, the total running time for each application is in $\Theta(w)$ where w is the total length of the two input lists.

Thus, the total running time for `find-best-alignment` is in $\Theta(wk^w)$ where w is the total length of the two input lists, and k is some constant between 2 and 3. This is exponential in the size of the input.

Exercise 8.9. ($\star\star$) Define a `find-best-alignment` whose running time is polynomial in the size of the input. (Hint: compare the `find-best-alignment` procedure with the `fibo` procedure and try and do a similar transformation to what we did to make `fast-fibo`.) $\diamondsuit$

[7] Determining the actual value of k is outside the score of this book, but it is not very important for understanding the running time growth of `find-best-alignment`. Once we know it is exponential, that is enough to understand how it scales as the input scales.