Lecture 27: (PROVISIONAL) Insertion Sort, Selection Sort, and Merge Sort

10:00 AM, Nov 8, 2017

Contents

1 Program Structure and Data Structure .......................................................... 1

2 Sorting .................................................................................................................. 2

3 Insertion Sort ...................................................................................................... 2
   3.1 Analysis of Insertion Sort ............................................................................... 3

4 Selection Sort .................................................................................................... 3

5 Mergesort ............................................................................................................ 4
   5.1 Divide and Conquer ....................................................................................... 4
   5.2 Implementation of mergesort ......................................................................... 4
   5.3 Implementation of merge ............................................................................. 6
   5.4 Analysis ........................................................................................................... 8

6 Summary ............................................................................................................. 9

Objectives

By the end of this lecture, you will know:

- how to follow the Racket rules of evaluation (review)
- how program structure reflects data structure
- how to use recursive diagrams v2.0
- the insertion sort algorithm and runtime
- the selection sort algorithm and runtime
- the merge sort algorithm and runtime
1 Program Structure and Data Structure

All semester I’ve been harping on the idea that the structure of your data must match the structure of your program.

For example, whenever we’ve written a procedure that recurs on lists, we’ve always asked “What should our procedure do with the empty list?” and “What should our procedure do with a cons list?”. We did this because we knew that a list is only either an empty or a cons, and therefore our procedure had to be able to handle these two cases.

How does this apply to Rackette?

To answer that question, let’s start by taking a look at Rackettecita from Homework 9. Recall that Rackettecita was a small subset of Rackette, in which we only considered ints and procedure-application expressions of arithmetic operators (like + and *) to be valid programs. For example, (+ 3 5) is a valid Rackettecita program, but (+ 3) would not be.

In BNF form, Rackettecita looks like:

```
program := int | proc-app
```

where an int was something like 4, 5, 6, etc. and a proc-app looked something like “(<proc-name> program program),” where <proc-name> could be +, -, *, or /.

This BNF is basically data about the Rackettecita programming language because it tells you how many forms a Rackettecite program can take.

So, if we were to write a program for parsing a Rackettecita program, how would this data come into play?

Well, since our procedure would parse a concrete_program and produce a syntactically valid Rackettecita program, we need to define an output type which represents a Rackettecita program. So, we’ll define a type program which fully captures each different form of a valid Rackettecita program.

For example,

```
type program = Num of int | ProcApp of (...)
```

This exact same concept will apply to Rackette.

It’s important to note that if our BNF had been slightly more complicated (i.e. had contained more than one line), every item on the left side of the “:=” in the BNF would had to have had a type statement (like the one above) in our theoretical parsing program.

**Question:** How many types will our Rackette program have?

**Answer:** How many lines are there in the BNF for Rackette? There would definitely have to be at least as many types.

**Question:** What about evaluation? How does this fit with our whole idea of the structure of the program matching the data?

**Answer:** The rules of processing are our description of the meaning of the language, as opposed to the syntax, so they are the data which informs the structure of eval. There are maybe 8-10 rules
for evaluating Rackette programs, so your eval procedure is probably going to have 8-10 match cases, where the code for each match case comes from the written description of the rule.

2 Sorting

Today, we introduce a fundamental problem in computer science: sorting. We all know what sorting is. We encounter this task on a regular basis. To make it easier to find what or who we’re looking for, we alphabetize books by author or sort them by Dewey decimal number. We alphabetize names on sports rosters or callback lists. And applications that display information from databases—such as songs or recipes—allow us to view the information organized in ascending or descending sorted order based on a given field—such as artist, album, or title; or name, cuisine, date posted, or rating.

Sorting is a fundamental problem in computer science because it is applicable to a wide range of problems. In the face of today’s burgeoning information networks, sorting algorithms help us to create order out of chaos. Given the ubiquity of sorting, it’s important to invent sorting algorithms that are as time (and space) efficient as possible.

3 Insertion Sort

How could we sort the list [8; 3; 4; 2] in increasing order?

There are many possible choices for this. The first one that comes to mind to me is, naturally, the recursive solution:

- original input: [3; 8; 4; 2]
  - recursive input: [8; 4; 2]
  - recursive output: [2; 4; 8]
- original output: [2; 3; 4; 8]

Why is my recursive input the tail of the list? Well, we’re most familiar with structural recursion, so I decided to use that. Notice that my recursive output is simply the sorted list, and my original output is the head of the original input inserted into the recursive output. What’s important to note is that it was inserted into the right place.

Can we write a helper that inserts an element into a sorted list while still keeping the list sorted? Well, that sounds like a recursive procedure:

- If the element is less than the head of the list, then cons the element onto the list
- If it’s not, then you insert it into the tail and cons the first element onto the result.

Assuming we’ve written our insert procedure, our sort procedure is easy. We simply recursively call sort on the tail of the list, and then insert the head of the list onto the sorted tail.
This process is actually a well-established sorting algorithm in computer science, called insertion sort.

### 3.1 Analysis of Insertion Sort

Let $U(n) = \text{max time to run insertion sort on any list of } n \text{ distinct integers}$. If the list is \textit{empty}, we just return the \texttt{empty} list, so $U(0) = A$. If the list is not \textit{empty}, then we have to insert the first item of the list into the result of sorting the rest of the list. Since the rest of the list has size $n - 1$, sorting the rest of the list will take time $U(n - 1)$. Let $C(n)$ be the cost of inserting $hd$ into a sorted list of $n$ items. So, $U(n) \leq B + C(n) + U(n - 1)$. Therefore, insertion sort runs in $O(n^2)$.

### 4 Selection Sort

We need a faster way to sort. Let’s try another method, and determine if it is any faster. How about this idea: “find the smallest item in the list, pull it out, sort the rest of the list, and then place in the smallest item in the front”? This eliminates the need to traverse the list at the end of each step.

Let’s go through how we would do selection sort. Our implementation will depend on a helper than returns a tuple of the smallest item and a sorted list.

- In our main procedure, if the list is empty, return that empty list.
- If not, call our helper.
- Within the helper:
  - If there is a one element list, return the tuple of that item, and the empty list.
  - If there is a head and a tail, find the minimum of the tail:
    * If the head is smaller than the minimum of the tail, return the tuple of the head and the rest of the list.
    * Otherwise, return the minimum of the tail, and the rest of the list ($hd :: rest2$).

This is still a quadratic-time procedure, though, as it takes linear time to find the smallest element in the list.

### 5 Mergesort

#### 5.1 Divide and Conquer

To improve upon insertion and selection sort, we introduce a new algorithmic technique: \textit{divide-and-conquer}\footnote{This name actually derives from the Latin \textit{divide et impera}, meaning divide and rule.}. This approach is named for the Roman military strategy of breaking up the enemy forces into small factions, and then proceeding to dominate those factions one by one.

In writing procedures that operate on lists, we have generally followed our template for list recursion, performing some operation on the \texttt{head} of the list and recurring on the \texttt{tail} of the list. This
recursive decomposition of the list naturally follows the structure of our data: a list is either empty or the cons of an element onto a list.

Now, a list of length \( n - 1 \) is certainly smaller than a list of length \( n \), as a list of length \( n - 2 \) is smaller than a list of length \( n - 1 \). However, it’s not much smaller. In order to write a quicker procedure, we want to decompose our list in such a way that it gets smaller quickly. How can we do that? Well, let’s try breaking it in half.

How can this help us? A quick look at the below figure suggests the answer. By dividing the list in half rather than recurring on rest, we reach the base case much more quickly. But how much? That is, how many times can we divide a list of length \( n \) in two (before reaching the base case)?

Another way to phrase this question is: to what power we must raise 2 to arrive at \( n \)? Now the answer should be clear: \( \log_2 n \). (If you want, refer to the below figure for confirmation.) The speed-up prophesied by a divide-and-conquer approach is the first of three fundamental insight(s) in our development of mergesort.

![Figure 1: A figure illustrating the difference in the depth of recurring using the rest of a list and recurring by breaking the list in half. In each case, the elements in black indicate the item being handled currently, while the elements in gray indicate the elements of the list which will be handled in the recursive call(s). The structural recursion strategy (left), which we are used to, has depth \( \Theta(n \mapsto n) \); the divide-and-conquer recursive strategy (right) has depth \( \Theta(n \mapsto n \log n) \).](image)

### 5.2 Implementation of mergesort

What distinguishes our various sorting algorithms are their recursive aspects, so we immediately turn our attention to this puzzle. To guide the development of our recursive strategy, we pose, yet again, the two basic questions. By now, you must know them by heart:

- how do we derive the recursive input(s) from the original input?
- how do we derive the original output from the recursive output(s)?

We took two approaches last time: The first was to use the tail of the input as the recursive input; we sorted that, and then had to place the “head” into the tail in the appropriate spot, a process called “insertion.” The alternative was to extract smallest (or largest, but smallest turns out to work slightly more simply), which we then need to cons onto something ... and that “something”
has to be the remainder of the sorted list. That process was called “selection sort,” and it’s one of the first examples we’ve seen of non-structural recursion – we didn’t use the template for recursion there, in which we recur on the “tail” of the list, which is a piece that comes from the very structure of the list itself.

That freedom – to recur on something other than the tail – opens up another possibility: maybe the recursive input could be very different from the input.

With that in mind, there was a third possibility, which was to divide the input list in half, sort each half, and then “merge” the two results. This is called “mergesort.”

We now construct our recursive diagram with an original input from our test cases, along with the corresponding recursive inputs. For convenience, we assume that when dividing a list of odd length, the first is shorter.

```
original input: [4 ; 2 ; 1 ; 3 ; 6 ; 9 ; 4 ; 2 ; 5 ; 8 ; 7]
recursive input: [4 ; 2 ; 1 ; 3 ; 6] recursive input: [9 ; 4 ; 2 ; 5 ; 8 ; 7]
recursive output: ? recursive output: ?
original output: [1 ; 2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]
```

We then fill in the corresponding recursive outputs and original output according to the specification of mergesort (or any sorting procedure): that the output contains the elements of the input in ascending order.

```
original input: [4 ; 2 ; 1 ; 3 ; 6 ; 9 ; 4 ; 2 ; 5 ; 8 ; 7]
recursive input: [4 ; 2 ; 1 ; 3 ; 6] recursive input: [9 ; 4 ; 2 ; 5 ; 8 ; 7]
recursive output: [1 ; 2 ; 3 ; 4 ; 6] recursive output: [2 ; 4 ; 5 ; 7 ; 8 ; 9]
original output: [1 ; 2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]
```

Finally, to code our procedure we need only determine how to transform the recursive outputs into the original output. At a high level, and as the name of the algorithm implies, we merge them. We will write a helper procedure, merge. This will leave our mergesort code clear and concise.

Note here that we are dividing the original list in half, so the two resulting lists will necessarily be smaller than the original. . . or will they?

What would happen if we were to attempt to divide a one-element list into two? One of the resulting lists would be empty. That’s good: the empty list is smaller than a one-element list. But the other would be the original one-element list itself. This observation suggests that we should use as our base case a one-element list. Otherwise, our procedure would not terminate.

On the other hand, we should also be able to sort a zero-element list (the result is the same zero-element list), so this new recursion actually has an extra base case.

```
original input: [4 ; 2 ; 1 ; 3 ; 6 ; 9 ; 4 ; 2 ; 5 ; 8 ; 7]
recursive input: [4 ; 2 ; 1 ; 3 ; 6] recursive input: [9 ; 4 ; 2 ; 5 ; 8 ; 7]
recursive output: [1 ; 2 ; 3 ; 4 ; 6] recursive output: [2 ; 4 ; 5 ; 7 ; 8 ; 9]
original output: [1 ; 2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]
```

Let’s take a moment here to absorb what we’ve just learned about non-structural recursion. Notice that finding the base case is not as simple as only checking if the list is empty. Instead, we have to be more active in figuring out what the base case is because the most “obvious answer” is often not the right answer.

In addition, in non-structural recursion we may have more than one base case, as we will see in mergesort right now!

Long story short: Think hard and be on the lookout for base cases!
This discussion gives rise to an implementation of `merge`, which produces the empty list when it consumes the empty list, the same list as the input for one-element lists, and otherwise splits the list in two, recursively calls `merge_sort`, and then merges them together.

### 5.3 Implementation of `merge`

We now turn our attention to the `merge` procedure. As always, the design recipe will lead us to our implementation.

"Examples of the Data"

| 15 = [1; 2; 3; 4; 6] |
| 16 = [2; 4; 5; 7; 8; 9] |

(* Input: two lists of integers, sl1 and sl2, sorted in ascending order  
Output: a list of integers containing all the elements of alod1 and alod2, sorted in ascending order *)

```plaintext
let rec merge (sl1 : int list) (sl2 : int list) : int list = ...
```

"Test Cases for merge"

| check_expect (merge [] []) [] |
| check_expect (merge 15 []) [1; 2; 3; 4; 6] |
| check_expect (merge [] 16) [2; 4; 5; 7; 8; 9] |
| check_expect (merge 15 16) [1; 2; 2; 3; 4; 4; 5; 6; 7; 8; 9] |

We use as our template for `merge`, which recurs on two lists, a combination of two of the standard templates for structural recursion on one list:

```plaintext
let rec merge (sl1 : int list) (sl2 : int list) : int list = 
  match sl1, sl2 with 
  | [], _ -> ... 
  | _, [] -> ... 
  | hd1 :: tl1, hd2 :: tl2 -> ...
```

First, we consider the base cases. If one list is empty, we just return the other list. So, if `sl1` is the empty list, we return `sl2`. If `sl2` is the empty list, we return `sl1`.

```plaintext
let rec merge (sl1 : int list) (sl2 : int list) : int list = 
  match sl1, sl2 with 
  | [], _ -> sl2 
  | _, [] -> sl1 
  | hd1 :: tl1, hd2 :: tl2 -> ...
```

Next, we consider the recursive step. We construct a recursive diagram to aid us, considering as input the lists from our test case above.

- original input: [1; 2; 3; 4; 6] [2; 4; 5; 7; 8; 9]
  - recursive input: ?
  - recursive output: ?
original output: [1 ; 2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]

Since there are two inputs but only one output, it is easier to determine how to derive the original output from the recursive output. As is usual when we construct a recursive diagram in this direction, our operation of choice is \texttt{cons}. If the original output is the result of a \texttt{cons}, then the recursive output is that same list \textit{sans} the first element.

original input: [1 ; 2 ; 3 ; 4 ; 6] [2 ; 4 ; 5 ; 7 ; 8 ; 9]
  - recursive input: ?
  - recursive output: [2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]

original output: [1 ; 2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]

From here, we can infer the recursive inputs. They should be two lists, which, when merged, yield [2; 2; 3; 4; 4; 5; 6; 7; 8; 9]. Note that since 1 is not part of the recursive output, it cannot be part of either recursive input. So, we have recurred on the \texttt{rest} of the first list. All elements of the second list appear in the recursive output, so we recur on all of that list.

original input: [1 ; 2 ; 3 ; 4 ; 6] [2 ; 4 ; 5 ; 7 ; 8 ; 9]
  - recursive input: [2 ; 3 ; 4 ; 6] [2 ; 4 ; 5 ; 7 ; 8 ; 9]
  - recursive output: [2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]

original output: [1 ; 2 ; 2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]

Why should we recur on the \texttt{rest} of the first list and the entirety of the second? As many of you have already undoubtedly noticed, we \texttt{cons} the 1 to produce our final, sorted list because 1 is the smallest element in both lists. Since both of the input lists are sorted in ascending order, finding the smallest number among them reduces to choosing the smaller \texttt{first} element. We recur on the \texttt{rest} of the list with such an element, and the entirety of the other list. Consider as another example the following recursive diagram:

original input: [3 ; 4 ; 6] [2 ; 4 ; 5 ; 7 ; 8 ; 9]
  - recursive input: [3 ; 4 ; 6] [4 ; 5 ; 7 ; 8 ; 9]
  - recursive output: [3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]

original output: [2 ; 3 ; 4 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]

In this case, because the second list contains the smaller element, we recur on the entirety of the first list and the \texttt{rest} of the second.

We are now ready to fill in our recursive step.

```
let rec merge (sl1 : int list) (sl2 : int list) : int list =
  match sl1, sl2 with
  | [], _ -> sl2
```

8
Now, let’s analyze this procedure!

5.4 Analysis

Let $M(n)$ be the worst runtime for `merge_sort` on an $n$-item list.

$$M(n) \leq 2M\left(\frac{n}{2}\right) + Q(n)$$

(1)

$$M(n) \leq 2M\left(\frac{n}{2}\right) + An + B$$

(2)

We can then do plug and chug:

$$M(0) \leq C$$

(3)

$$M(1) \leq C$$

(4)

$$M(2) \leq 2M(1) + 2A + B = 2C + 2A + B$$

(5)

$$M(4) \leq 2M(2) + 4A + B = 2(2C + 2A + B) + 4A + B$$

(6)

$$M(4) \leq 4C + 4A + B$$

(7)

There’s a pattern. For powers of two, we have $M(2^k) \leq 2^k(C + kA) + B$. Using well-ordering, we can prove that `merge_sort` is in $O(n \rightarrow n\log n)$.

6 Summary

Ideas

- As always, the structure of a program should reflect the structure of its data! For parsing programs, this data is the BNF of the language, and for evaluating programs, its the rules of evaluation.

- Insertion sort is one strategy for sorting a list of $n$ items. It involves placing each item from the original list into the result of recursively sorting the rest of the list. Insertion sort runs in $O(n^2)$.

- Selection sort is another strategy for sorting a list of $n$ items. It involves find the smallest item in the list, pulling it out, sorting the rest of the list, and then placing the smallest item at the front. Selection sort runs in $O(n^2)$.

- A third way to sort lists is using merge sort. It involves splitting the input list into two parts, sorting those parts, and then merging them back together. Merge sort runs in $O(n \rightarrow n\log n)$.
Skills

- The words `val`, `sig`, `struct`, `end` are keywords in OCaml - if you try to use them as identifiers OCaml will complain.
- How to approach non-structural recursion, like in mergesort

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)