

Recitation 7

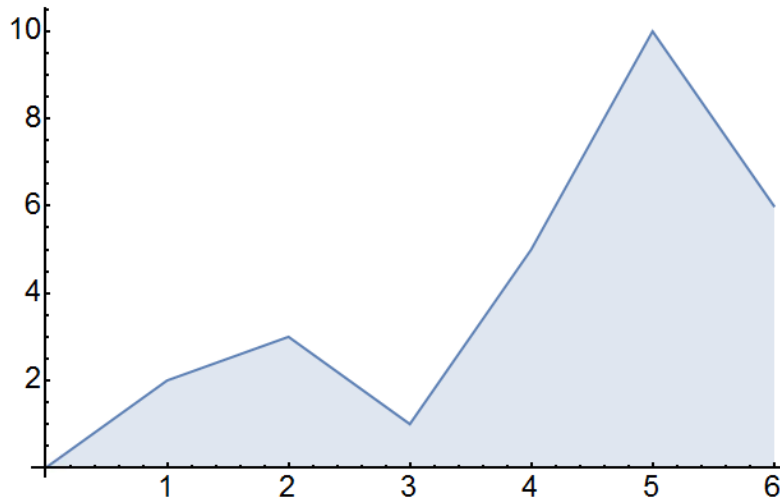
Augmented and Ordered Tables

7.1 Announcements

- *RangeLab* will be released on **Thursday**.

7.2 Stock Market

Suppose you're working as a stock market analyst. You want to be able to quickly determine the largest increase in stock value during a specific time interval. For example, if the stock values of some company from time 0 to time 6 are $\langle 0, 2, 3, 1, 5, 10, 6 \rangle$, then the maximum increase in the time interval $[2, 4]$ is $5 - 1 = 4$, while the maximum increase within the interval $[0, 6]$ is $10 - 0 = 10$.



Task 7.1. Implement the function

```
val maxIncrease : int Seq.t → (int * int) → int
```

where $(\text{maxIncrease } S \ (t_1, t_2))$ returns the maximum increase of the stock values S within the time interval $[t_1, t_2]$. Assume that $0 \leq t_1 < t_2 < |S|$.

Your implementation must be **staged** such that it requires linear work upon application of the first argument, and logarithmic work upon application of the second. For example, in the following, line 1 should require $O(|S|)$ work while lines 2 and 3 should each require $O(\log |S|)$ work.

```
1 val queryInterval = maxIncrease S
2 val _ = queryInterval (t1, t2)
3 val _ = queryInterval (t3, t4)
```

First, suppose that we ignore the staging requirements. Instead, let's just try to implement $(\text{maxIncrease } S \ (t_1, t_2))$ in $O(|S|)$ work. We can do so with a strengthened divide-and-conquer.

Imagine a `Seq.reduce` over $S[t_1, \dots, t_2]$ whose combining function returns (a) the minimum stock value, (b) the maximum stock value, and (c) the maximum increase. The combining function then just needs to separately consider the cases where the maximum increase comes from the left subresult, the right subresult, or “straddles the middle.” The code is as follows.

Algorithm 7.2. *Implementing `maxIncrease` by avoiding the preprocessing step.*

```

1 fun maxOf3 (x, y, z) = Int.max (x, Int.max (y, z))
2
3 fun combine ((min1, max1, inc1), (min2, max2, inc2)) =
4   (Int.min (min1, min2),
5    Int.max (max1, max2),
6    maxOf3 (inc1, max2 - min1, inc2))
7
8 fun maxIncrease S (t1, t2) =
9   let val S' = Seq.map (fn v  $\Rightarrow$  (v, v, 0)) S[t1, ..., t2]
10    val (_, _, x) = Seq.reduce combine ( $\infty$ ,  $-\infty$ ,  $-\infty$ ) S'
11  in x
12  end
```

Now all that is left is to “dynamize” the reduce with augmented binary search trees. Specifically, the *keys* of our BST will be indices (time-steps) of the input, the *values* will be the singletons $(v, v, 0)$ for each $v \in S$, and the *reduced values* will be the triples containing (a) the min value, (b) the max value, and (c) the maximum increase.

Note that we can build such a tree in linear time because the input is presorted by key. (The keys are just indices!) We can query the tree by requesting the reduced value of the chunk of the tree which lies between t_1 and t_2 .

In the 15-210 library, all of this can be accomplished with the `MkTreapAugTable` functor, which takes structures `Key` and `Val` as input (fixing the key and value types of the table) and produces an implementation of tables as augmented treaps. The resulting structure ascribes to `AUG_ORDTABLE`, which has ordered table functions such as `split`, `join`, and `getRange`, as well as `reduceVal` which extracts reduced values.

Note that the key type of our table is `int`. The 15-210 library contains a structure `IntElt` which defines the functions necessary to use integers as keys, such as comparison, hashing, etc. We'll have to build the `Val` structure ourselves. It must ascribe to `MONOID`.¹

¹The term “monoid” comes from the field of abstract algebra. Monoids are just sets along with a binary associative operation and an identity element. For example, $(\mathbb{Z}, +, 0)$ is a monoid, since $+$ is associative, and the integer 0 is the additive identity.

Algorithm 7.3. *Implementing `maxIncrease` with separate preprocessing and query steps. Pay close attention to lines 21 through 34 to see how to correctly stage a function in SML.*

```

1 fun maxOf3 (x, y, z) = Int.max (x, Int.max (y, z))
2
3 structure MyVal =
4 struct
5   type t = int * int * int
6
7   fun f ((min1, max1, inc1), (min2, max2, inc2)) =
8     (Int.min (min1, min2),
9      Int.max (max1, max2),
10      maxOf3 (inc1, max2 - min1, inc2))
11
12   val I = ( $\infty$ ,  $-\infty$ ,  $-\infty$ )
13
14   val toString = Int.toString
15 end
16
17 structure AugTable =
18   MktreapAugTable (structure Key = IntElt
19                     structure Val = MyVal)
20
21 fun maxIncrease S =
22   let
23     fun singleton (i,v) = AugTable.singleton (i,(v,v,0))
24     val S' = Seq.mapIdx singleton S
25     val T = Seq.reduce AugTable.join (AugTable.empty ()) S'
26
27     fun query (t1,t2) =
28       let val T' = AugTable.getRange T (t1,t2)
29         val (_, _, x) = AugTable.reduceVal T'
30       in x
31       end
32   in
33     query
34   end

```

As for cost bounds, notice that line 24 is clearly linear. Line 25 is more subtle; if you write a recurrence, you'll see that it has the form $W(n) = 2W(n/2) + O(\log n)$. We've solved this recurrence before – it's linear!

Finally, line 28 requires logarithmic work. `getRange` is implemented as two splits: one for the lower key, and one for the higher key. To make it inclusive, we have to follow up each split with an insertion.