Max Seq Problem

Question from before: Design an algorithm to return the largest sum of contiguous integers in an array of ints

Example: if the input is (-10, 2, 3, -2, 0, 5, -15), the largest sum is 8, which we get from (2, 3, -2, 0, 5).

Example

arr: -10 2 3 -2 0 5 -15
arrVal: 0 2 5 3 3 8 0
maxVal: 0 2 5 5 5 8 8

A New Algorithm

MaxSeq (int arr[], int n)

```
MaxSeq (int arr[], int n)
{
  int arrVal[] = new int[n];
  List arrMaxSubseq[] = new List[n];

  if (arr[0]>0)
  {
    arrVal[0] = arr[0];
    arrMaxSubseq[0] = {arr[0]};
  }
  else{
    arrVal[0] = 0;
    arrMaxSubseq[0] = { };
  }
  int maxVal = arrVal[0];
  List maxSubseq = arrMaxSubseq[0];
  for (int i=1;i<n;i++){
    int bestVal = arrVal[i-1] + arr[i];
    if (bestVal > 0){
      arrVal[i] = bestVal;
      arrMaxSubseq[i] = {arrMaxSubseq[i-1], arr[i]};
    }else{
      arrVal[i] = 0;
      arrMaxSubseq[i] = { };
    }
    if (arrVal[i] > maxVal){
      maxVal = arrVal[i];
      maxSubseq = arrMaxSubseq[i];
    }
  }
  return maxVal as the maximum value,
        and maxSubseq as the maximum subsequence
}
```
Loop Invariant

At the beginning of the i-th iteration of the for loop, the following is true:

- For all $0 \leq j < i$, $\text{arrVal}[j]$ gives the value of the maximum value subsequence with rightmost index $j$, and $\text{arrMaxSubseq}[j]$ gives a subsequence with rightmost index $j$ that has value $\text{arrVal}[j]$.
- The variable $\text{maxVal}$ gives the value of the maximum subsequence with rightmost index less than $i$, and $\text{maxSubseq}$ gives a subsequence with rightmost index less than $i$ that has value $\text{maxVal}$.

Loop1 Invariant

- **Initialization**: Before the 1-st iteration of the loop, $\text{arrVal}[0]$ gives the value of the maximum value subsequence with rightmost index 0, and $\text{arrMaxSubseq}[0]$ gives a subsequence with rightmost index at 0 that has value $\text{arrVal}[0]$. Further $\text{maxVal}$ gives the value of the maximum value subsequence with rightmost index less than 1, and $\text{maxSubseq}$ is a subsequence achieving this value.
- **Maintenance**: See next slide
- **Termination**: When $i = n$, the second part of the loop invariant says that $\text{maxVal}$ is the maximum of all possible subsequences in the array $\text{arr}$ (i.e. with rightmost index less than $n$), and that $\text{maxSubseq}$ gives a subsequence which achieves this value. These facts directly imply that the algorithm is correct.

Maintenance Sketch

We sketch only the first part of the maintenance proof.

- Since $\text{arrVal}[i-1]$ gives the best value of a subsequence with rightmost index $i - 1$ (by inductive hypothesis), the variable $\text{bestVal}$ gives the best value of a subsequence that includes $\text{arr}[i]$. If this value is greater than 0, then the value of the maximum value subsequence with rightmost index at $i$ is $\text{bestVal}$. Otherwise, the value of the maximum value subsequence with rightmost index at $i$ is 0.
- Thus $\text{arrVal}[i]$ is set correctly in the loop
- Must also show that $\text{arrMaxSubseq}[i]$, $\text{maxVal}$ and $\text{maxSubseq}$ are set correctly
- This is left as an exercise.

The Sorting Problem

- The Problem: we want to sort an array, $\mathcal{A}$, of integers in non-decreasing order
- E.g. if $\mathcal{A}$ is 3, 2, 2, 1, 5 at the start, we want it to be 1, 2, 2, 3, 5 at the end
- Sorting is a very common programming problem!
- Last time, we analyzed the Insertion-Sort Algorithm

Insertion Sort

```python
Insertion-Sort (A, int n)
for (j=1; j<n; j++){
    key = A[j];
    //Insert A[j] into the sorted sequence A[0,...,j-1],
    //in the location such that it is as large as all elems
    //to the left of it
    i = j-1
    while (i>=0 and A[i] > key){
        A[i+1] = A[i]
        i--
    }
    A[i+1] = key
}
```

Analysis

- Best case run time of Insertion Sort is $O(n)$ (if the array is already sorted)
- However, we proved before that the run time of Insertion Sort is $\Theta(n^2)$ in the worst case
- Q: Can we do better than this?
- A: Yes, we can use a recursive algorithm called Merge Sort
**Merge Sort**

High Level Idea:

- Split the array into two parts of the same size, $A_1$ and $A_2$
- Recursively sort $A_1$ and $A_2$
- Merge $A_1$ and $A_2$ together into one big sorted array

**Merge Sort Example**

```
arr 1 4 5 2 3 6
arrLeft 1 4 5
arrRight 2 3 6
arrRes 1 2 3 4 5 6
```

**Sketch of Correctness Proof**

- Assume the subroutine Merge does what it says (proof of this is given on page 30 of book)
- We can then prove by induction on the size of A, that Merge-Sort works
- Base case: if A.size() = 1, A is already in sorted order, so the algorithm returns the correct value.

**Merge**

```java
//PRE: arrLeft and arrRight are in sorted order
//POST: arrRes contains the elems of arrLeft and arrRight in sorted order
Merge(int arrLeft[], int arrRight[])
    int iLeft = 0, iRight = 0;
    int arrRes[] = new int[arrLeft.size()+arrRight.size()];
    for (int i=0;i<arrRes.size();i++){
        if (iRight == arrRight.size() || (iLeft<arrLeft.size() && arrLeft[iLeft]<=arrRight[iRight])){
            arrRes[i] = arrLeft[iLeft];
            iLeft++;
        }else{
            arrRes[i] = arrRight[iRight];
            iRight++;
        }
    }
    return arrRes;
}
```
Let’s analyze the worst case run time of Merge-Sort.
First need to analyze run time of the Merge subroutine.
Let $n = \text{arrLeft.size()} + \text{arrRight.size()}$.
Then the for loop has $n$ iterations, each taking $\Theta(1)$ time.
Thus, total time Merge takes is $\Theta(n)$.

Let $T(n)$ be the number of time steps Merge-Sort takes on an array of size $n$.
Total cost is 2 calls to Merge-Sort on arrays of size $\leq n/2$, one call to Merge which takes time $\Theta(n)$, plus $\Theta(n)$ time to split the array, plus $\Theta(1)$ time for assignments.
Thus, if $n = 1$, $T(n) = \Theta(1)$, and if $T(n) = 2 \times T(n/2) + \Theta(n)$.

The running time of an algorithm on a constant size input is always $\Theta(1)$.
Thus for convenience, we usually omit statements of the boundary conditions and just assume $T(n)$ is constant when $n$ is a constant.
Example: Instead of saying "If $n = 1$, $T(n) = \Theta(1)$, and if $T(n) = 2 \times T(n/2) + \Theta(n)$", we just say "$T(n) = 2 \times T(n/2) + \Theta(n)$".

"Oh how should I not lust after eternity and after the nuptial ring of rings, the ring of recurrence" - Friedrich Nietzsche, Thus Spoke Zarathustra

$T(n) = 2 \times T(n/2) + n$ is an example of a recurrence relation.
A Recurrence Relation is any equation for a function $T$, where $T$ appears on both the left and right sides of the equation.
We always want to “solve” these recurrence relation by getting an equation for $T$, where $T$ appears on just the left side of the equation.

We’ve found a function giving the run time of Merge-Sort.
But what does this mean: $f(n) = 2 \times T(n/2) + \Theta(n)$?
How can we write this in big-O or $\Theta$ notation?
How does this algorithm compare with the $O(n^2)$ run time of insertion sort?

Whenever we analyze the run time of a recursive algorithm, we will first get a recurrence relation.
To get the real run time, we need to solve the recurrence relation.
Substitution Method

- One way to solve recurrences is the substitution method aka “guess and check”
- What we do is make a good guess for the solution to $T(n)$, and then try to prove this is the solution by induction

Example

- Let’s guess that the solution to $T(n) = 2 \cdot T(n/2) + n$ is $T(n) = O(n \log n)$
- In other words, $T(n) \leq cn \log n$ for appropriate choice of constant $c$
- We can prove that $T(n) \leq cn \log n$ is true by plugging back into the recurrence

Proof

- We prove this by induction, Assume that $T(n/2) \leq cn/2 \log(n/2)$

\[
T(n) \leq 2T(n/2) + n \leq 2(cn/2 \log(n/2)) + n \leq cn \log(n/2) + n = cn(\log n - \log 2) + n = cn \log n - cn + n \leq cn \log n
\]

last step holds if $c \geq 1$

Recurrence Relations

- There are many ways to solve recurrence relations
- Next time, we’ll see some other methods.