Exercises (Chapter 2)

Xinyu Xiao

June 23, 2020

2.3-4

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1...n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1...n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

Algorithm 1 recursive insertion sort

```
function RECURSIVE_INSERTION_SORT(A)
    n = A.length
    if n > 1 then
        recursive_insertion_sort(A[1..n − 1])
        insert(A[1..n − 1], A[n])
    end if
end function
```

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ T(n-1) + \Theta(n) & if\ n > 1 \end{cases}$$

$$Let\ \Theta(1) = c \Rightarrow T(n) = \frac{c}{2}(n^2 + n)$$

2.3-5

Refering back to the searching problem(see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against $\mathcal{V}$ and eliminate half of the sequence from further consideration. The *binary search* algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(lgn)$.

Algorithm 2 binary search $\rightarrow \Theta(lgn)$

Require: sorted array A(incremental), the start and end index p,q search value $\mathcal{V}$

Ensure: the index i such that $A[i] = \mathcal{V}$ or the special value "NIL" if $\mathcal{V}$ does not appear in A

```
function BinarySearch(A, p, q, v)
    if A[p] > v Or A[q] < v then
        return NIL
    end if
    mid = ⌊(p+q)/2⌋
    if A[mid] < v then
        return BinarySearch(A, mid + 1, q, v )
    else if A[mid] > v then
        return BinarySearch(A, p, mid − 1, v )
    else
        return mid
    end if
end function
```

2.3-6

Observe that the **while** loop of lines $5-7$ of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j-1]$. Can we use a binary search(see Exercise $2.3-5$) instead to improve the overall worst-case running time of insertion sort to $\Theta(nlgn)$?

No, we can't. Because we need to shift the whole subarray $A[k..j = 1]$, such that $A[k-1] < key < A[k]$ right one index. The running time was still $\Theta(n^2)$

2.3-7 ★

Describe a $\Theta(nlgn)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(nlgn)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to **coarsen** the leaves of the recursion by using insertion sort

Require: A set S of n integers and another integer x
Ensure: whether or not there exist two elements in S whose sum is exactly x.

```
function FindElements(S, n, x)
    MergeSort(S, 0, S.length )
    for i = 0 To S.length do
        v = x − S[i]
        Get the S_extra
        BinarySearch(S_extra, 0, S_extra.length, v)
    end for
end function
```

Require: A set S of n integers and another integer x
Ensure: whether or not there exist two elements in S whose sum is exactly x.

```
function FindElements(S, n, x)
    MergeSort(S, 0, S.length )
    i = 0
    j = n − 1
    while i < j do
        if S[i] + S[j] = x then
            return true
        end if
        if S[i] + S[j] < x then
            i = i + 1
        end if
        if S[i] + S[j] > x then
            j = j − 1
        end if
    end while
    return false
end function
```

within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which $\frac{n}{k}$ sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

1. Show that insertion sort can sort the $\frac{n}{k}$ sublists, each of length k, in $\Theta(nk)$ worst-case tim.

   Insertion sort applied on the sublist of length k runs in $\Theta(k^2)$ worst-case time. We have $\frac{n}{k}$ sublists, so the running time will be $\Theta(\frac{n}{k} * k^2 = \Theta(nk)$.

2. Show how to merge the sublists in $\Theta(n \log(\frac{n}{k}))$ worst-case time.

   each layer will take $\Theta(n)$ worst-case time, we have $\log(\frac{n}{k})$ layers, so the merge will take $\Theta(n \log(\frac{n}{k}))$ worst-case time.

   Suppose we have coarseness k. This means we can just start using the usual merging procedure, except starting it at the level in which each array has size at most k. This means that the depth of the merge tree is $lg(n) - lg(k) = lg(n/k)$. Each level of merging is still time cn, so putting it together, the merging takes time $\Theta(nlg(n/k))$.

3. Given that the modified algorithm runs in $\Theta(nk+n \log(\frac{n}{k}))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

   $k < \log(n)$

   Viewing k as a function of n, as long as $k(n) \in O(lg(n))$, it has the same asymptotics. In particular, for any constant choice of k, the asymptotics are the same.

4. How should we choose k in practice?

2-2 Correctness of bubblesort

2-3 Correctness of Horner's rule

1. In term of Θ-notation, what is the running time of this code fragment for Horner's rule?

   $\Theta(n)$

2. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

```
function EvaluatePolynomial(A, x )
    n = A.length − 1
    y = 0
    for k = 0 to n do
        temp = 1
        j = k
        while  j > 0  do
            temp = x ∗ temp
            j = j − 1
        end while
        y = A[k] ∗ temp
    end for
    return y
end function
```

The running time of this algorithm is $\Theta(n^2)$. Compare with Horner's rule of runnint time $\Theta(n)$, it is not so good.

3. Consider the following loop invariant:

   At the start of each iteration of the **for** loop of line $2-3$,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

   Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0} O^n a_k x^k$