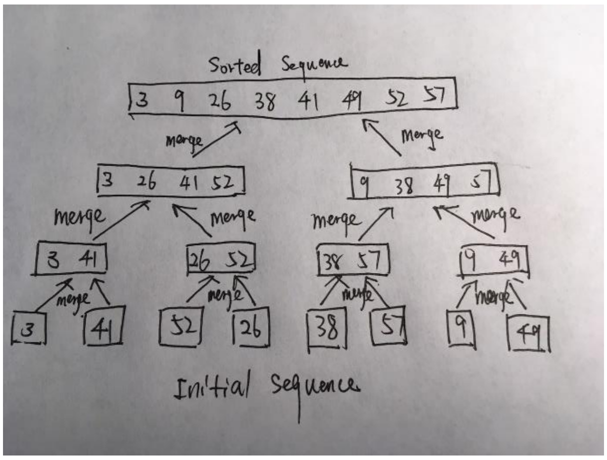
Due by Thursday, January 25th at 10pm, submit via git.

- Exercise 2.3-1
- Exercise 2.3-2
- Exercise 2.3-4
- Exercise 3.1-3
- Exercise 3.1-4
- Exercise 3.1-6
- Problem 2-1

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array A = {3, 41, 52, 26, 38, 57, 9, 49}

Solution:



2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

Solution:

Merge(A, p, q, r)

Input: two sorted array of integers

Output: a new array in sorted order

1. n 1 = p – q + 1

2. n 2 = r – q

3. let L [1……n1] and R [1……n2] be new arrays

4. for i = 1 to n1

5. L[i] = A[p + i – 1]

6. for j = 1 to n2

7. R[j] = A[q + i]

8. i = 1

9. j = 1

10. k = p

11. while i <= n1 and j <= n2

12. if L[i] <= R[j]

13. A[k] = L[i]

```
            i = i + 1
        else A[ k ] = R[ j ]
            j = j + 1
        k = k + 1
while i <= n1
        A[ k ] = L[ i ]
        i = i + 1
      k = k + 1
while j <= n2
        A[ k ] = R[ j ]
        j = j + 1
      k = k + 1
```

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].

Solution:

```
Insertion-Sort( A, n)
If n <= 1
        return
else Insertion-Sort(A, n – 1)
last = A[ n – 1]
j = n – 2
while j > 0 and A[ j ] >= last
        A[j + 1] = A[ j ]
        j = j – 1
A[ j + 1] = last
```

2-1 Insertion sort on small arrays in merge sort

1). The Show that insertion sort can sort the n=k sublists, each of length k, in θ(nk) worst-case time.

Solution:

Insertion sort can sort each length k part in θ(k^2) worst-time, so for total n / k parts, the time will be θ(n / k * (k^2)) = θ(nk).

2). Show how to merge the sub-lists in θ(nlg(n/k)) worst-case time.

Solution:

Since now we have totally n/k sub-lists to merge, which means our recursion tree's height is lg(n / k).

And for each level we still got n items to merge, which will take n times of comparison in the worst case.

so the total time is θ(nlg(n/k)) in this case.

3). Given that the modified algorithm runs in θ(nk +nlg(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?

Solution:

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

4). How should we choose k in practice?

Solution:

In particular, a constant choice of k is optimal. In practice we could find the best choice of this k by just trying and timing for various values for sufficiently large n.

3.1-3

Explain why the statement, “The running time of algorithm A is at least $O(n^2)$ is meaningless.

Solution:

It is true because big O notation is just an upper bound, and there are lots of functions that have growth rate less than n^2, We could say that a linear or constant has big $O(n^2)$, but it is of no use for us to evaluate our functions.

3.1-4

Is $2^{(n + 1)} = O(2^n)$? Is $2^{(2n)} = O(2^n)$?

Solution:

For $2^{(n + 1)}$, it equals to $2 * 2^n$ for all n that is greater than 0. So it is $O(2^n)$.

While for $2^{(2n)}$, if it were, there will exist a n_0 that for all $n > n_0$, $2^{(2n)} < c * 2^n$, for which c is a constant.

We do some calculation and will find that $2^n < c$, which is impossible, so $2^{(2n)}$ is not $O(2^n)$.

3.1-6

Prove that the running time of an algorithm is $\theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

Solution:

By Theorem 3.1, if the running time is $\Theta(g(n))$. the running time is $O(g(n))$, which implies that for any input of size $n \geq n_0$ the running time is bounded above by $c_1*g(n)$ for some c_1 . This includes the running time on the worst-case input.

Theorem 3.1 also implies the running time is $\Omega(g(n))$, which implies that for any input of size $n \geq n_0$ the running time is bounded below by $c_2*g(n)$ for some c_2 . This includes the running time of the best-case input.

On the other hand, the running time of any input is bounded above by the worst-case running time and bounded below by the best-case running time. If the worst-case and best-case running times are $O(g(n))$ and $\Omega(g(n))$ respectively, then the running time of any input of size n must be $O(g(n))$ and $\Omega(g(n))$. Theorem 3.1 implies that the running time is $\Theta(g(n))$.