Chapter 2 Getting Started

Yue Chen

June 5, 2023

Having $n = 2^k$, and $T(n) = nlg(n)$, we want to show that:

$$T(2n) = 2n \cdot lg(2n)$$

$$\begin{aligned} T(2n) &= 2 * T(n) + 2n \\ &= 2n \cdot lg(n) + 2n \\ &= 2n \cdot (lg(n) + 1) \\ &= 2n \cdot (lg(n) + lg(2)) \\ &= 2n \cdot lg(2n) \end{aligned}$$

Algorithm 1 Binary Search for 2.3-5

```
$l \leftarrow 0$
$r \leftarrow A.length - 1$
while $l < r$ do
    $i \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    if $A[i]$ =target then
        return i
    else if target $\geq A[i]$ then
        $r \leftarrow i - 1$
    else
        $l \leftarrow i + 1$
    end if
end while
return NIL
```

Exercise 2.3-7

For me, this is a pretty intuitive way of thinking this question. It is too simple that I didn't believe it in the beginning and denied it after a few seconds of imagining the procedure in my head. However, I must admit that proving the correctness of this algorithm is non-trivial, at least for me.

I was taught by the solution written by Andrew Lohr in `https://sites.math.rutgers.edu/~ajl213/CLRS/Ch2.pdf`. Andrew really helped me a lot! Thanks!

Here I will just try to write my own understanding of the second part of this solution, where we try to do the two-way search.

In order to prevent too much copy-paste, let me put my own implementation of the algorithm written in Andrew's solution first.

```
bool find_sum(std::vector<int> &vec, const int S) {
  size_t l = 0;
  size_t r = vec.size() - 1;
  ms_sorter(vec, 0, r);   // merge sort. It does change vec here.
  while (l < r) {
    auto cur_sum = vec[l] + vec[r];
    if (cur_sum == S)
      return true;
    if (cur_sum < S)
      ++l;
    else
      --r;
  }
  return false;
}
```

The essence of this algorithm is the second part — the two-way search inside the while-loop. The most important thing to notice is that, as Andrew mentioned, if $A[l] + A[r] < S$, we know that $\forall k$ such that $l \leq k \leq r, A[l] + A[k] < S$, and this works the other way.

So, we know that the immediate next value/combination we should try is $A[l+1] + A[r]$, assuming that the sequence is already sorted. This new combination gives us the next smallest sum we can try to approach the target S gradually. And it works for the other side as well.

Let me stop my innocent explanation here as Andrew has provided a rigorous proof of the correctness of this algorithm already.

Problems 2-2

(a) We should also need to prove that all the elements in A still exist in the new sequence.

(b)

Initialization: Before the for-loop begins, we may have the smallest element of the sublist $A[i..j]$ to be somewhere between i and j. Let's use k to denote the index of the position of the index of the "smallest" element. So we must have $i \leq k \leq j$ since $j = A.length$ initially.

Maintenance and Termination: Now, we can see how it maintains the loop invariant merely by checking the situation when $j = k$ as other situations won't affect the result. When $j = k$, we have $A[j] = A[k]$ to be the smallest in the sublist $A[i..j]$. So it will swap the two elements $A[j]$ and $A[j-1]$. Now we must have the smallest element to be in the sublist $A[i..j-1]$ in the current loop, which also gives us the property that the "smallest" one stays in $A[i..j']$ in the next loop, where $j' = j - 1$.

For the next and future loops, we will continue swapping the smallest element forward, resulting that the "smallest" one ends up to be in the position i.

(c)

Initialization: Prior to the first iteration, we have $i = 1$ and the sublist $A[1..i-1]$ contains $i - 1 = 0$ elements so it is sorted.

Maintenance: Now, for every iteration, we would first have $i - 1$ sorted elements in the sublist $A[1..i-1]$.Then, since the inner for-loop gives us a termination that the smallest element in the sublist $A[i..A.length]$ will be swapped to the position i, we know that the i^{th} smallest element is in the position i. Then we must have $A[1] \leq ... \leq A[i]$.

Termination: We would have n sorted elements in the list $A[1..n]$

(d) Let $n = A.length$. For the inner loop, since we must go over all the iterations, it takes $n - i$ compares and swap (if required). Noting that swap takes constant time. And since we need to go through the outer loop from $i = 1$ to $i = n - 1$, the total cost is $\sum_{i=1}^{n-1}(n-i)$, which belongs to $\Theta(n^2)$. And since that all the iterations must be went over, the best case is also $\Theta(n^2)$.

Problems 2-3

(a) Assuming that multiplication and addition takes constant time. Then for every iteration, we would have $\Theta(1)$, resulting $T(n) = \Theta(n)$

(b)

Algorithm 2 naive polynomial-evaluation algorithm for 2-3

```
$p \leftarrow 0$
**for** $k \leftarrow 0, n$ **do**
$m \leftarrow 1$
    **for** $i \leftarrow 1, k$ **do**                                  ▷ Calculate $x^k$
        $m \leftarrow m \cdot x$
    **end for**
$p \leftarrow p + a_k \cdot m$
**end for**
```

Now that we need to calculate x^k in every iteration, which requires $c_2 k$ calculations. And since we iterate from $k = 0$ to $k = n$, in total we have $\Theta(n^2)$ complexity.

(c)

Initialization: Before the loop begins, we have y = 0.

Maintenance: $\forall 0 \leq i \leq n$, we can expand the expression of y for each iteration as follows:

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

$$= a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1} \tag{2}$$

$$= a_i + \sum_{k=1}^{n-i} a_{k+i} x^k \tag{3}$$

$$= \sum_{k=0}^{n-i} a_{k+i} x^k \tag{4}$$

In step 4, we recognize that a_i is actually the term in the form $a_{k+i} x^k$ where k = 0.

Termination: And since in every iteration, y is re-calculated without using previous y, we end up having y to be the value calculated by the last iteration, where $i = 0$. It ends with the equation we want when $i = 0$.

Problem 2-4

(b)
The array that has $A[1] > .. > A[n]$ has the most inversions. It has $\sum_{i=1}^{n-1} i$ pairs of inversion.

(c)
During the insertion sort procedure, we have a part where we would try to run a while-loop and insert the key in to the sorted part. Within that while-loop, we are actually operating/copying all pairs of inversion for that particular key. And inversion requires $i < j$ and $A[i] > A[j]$, it means that we only need to check the elements that has smaller index than the key. Summing the number of the operation we have made in the insertion sort for every key, we obtain the exact same number of total inversion.

(d) My thoughts when designing this algorithm:
When we have two sublists, any element on the "right" side being copied to the merged list gives us that it has index greater than all the elements in the left sublist, and it has value smaller than all the elements in the left as well. Therefore, we may calculate the number of total inversions by summing up the length of the left sublist whenever the right-side element gets copied.

Algorithm 3 Counting Inversion

```
    MERGE(A, p, q, r)
 1: $n_1 := q - p + 1$
 2: $n_2 := r - q$
 3: let $L[1..n_1]$ and $R[1..n_2]$ be new arrays
 4: for $i := 1$ to $n_1$ do
 5:     $L[i] := A[p + i - 1]$
 6: end for
 7: for $j := 1$ to $n_2$ do
 8:     $R[i] := A[q + j]$
 9: end for
10: $i := 1$
11: $j := 1$
12: $k := p$
13: $c := 0$                                  ▷ inversion count for this merge
14: while $i < n_1 + 1$ and $j < n_2 + 1$ do
15:     if $L[i] \leq R[j]$ then
16:         $A[k] := L[i]$
17:         $i := i + 1$
18:     else
19:         $A[k] := R[j]$
20:         $j := j + 1$
21:         $c := c + (n_1 - i + 1)$
22:     end if
23: end while
24: return c
```

```
    MERGESORT(A, p, r)
 1: $c \leftarrow 0$
 2: if $p < r$ then
 3:     $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
 4:     $c \leftarrow c+$ MERGESORT(A, p, q)
 5:     $c \leftarrow c+$ MERGESORT(A, q+1, r)
 6:     $c \leftarrow c+$ MERGE(A, p, q, r)
 7:     return c
 8: end if
```

```
**return** $0$

```