Problem Set 5

Ekkapot Charoenwanit
Course: Efficient Algorithms

October 16, 2020

Problem 5.1. Binary Insertion Sort

1) In the vanilla version of insertion sort we discussed in the lecture, at any iteration i, we have to traverse the entire subarray $A[1...i-1]$ in the worst case to look for the correct position to place the key $A[i]$ into. How do you apply binary search to improve this? Also analyze the running time in the worst case of your binary insertion sort.

Solution: Observe that the subarray $A[1...i-1]$ is already sorted prior to the start of iteration i[1], where $key = A[i]$. Since $A[1...i-1]$ is sorted, binary search can be used to efficiently search for the correct position for the key. Although binary search requires $\mathcal{O}(\log i)$ comparisons in the worst case, once the correct position is found, we still have to shift at most $i-1 \in \mathcal{O}(i)$ elements to the right[2]. Therefore, at iteration i, the time complexity is still dominated by this shift operation, which costs $i-1$ copying operations. Hence, the total number of copying operations is $\sum_{i=2}^{n}(i-1) = \frac{n^2}{2} - \frac{n}{2} \in \mathcal{O}(n^2)$. Therefore, the worst-case running time of binary insertion sort remains asymptotically the same as that of vanilla insertion sort.

2) In your binary insertion sort algorithm you proposed in 1), how many comparisons do you need in the worst case and how many swaps do you need in the worst case?

Solution:
The number of key comparisons at each iteration i is at most $\lfloor \log_2(i-1) \rfloor + 1$. Thus, the total number of comparisons is at most $\sum_{i=2}^{n}(\lfloor \log_2(i-1) \rfloor + 1) \in \mathcal{O}(n \log n)$.
The number of key swaps (copying operations) is at most $\frac{n^2}{2} - \frac{n}{2} \in \mathcal{O}(n^2)$ as demonstrated in 1).
This shows that the number of swaps asymptotically subsumes the number of comparisons. $\square$.

Problem 5.2. Quick Sort

1) What is the running time of quick sort when all the array elements have the same value.

Solution: Based on the partitioning routine in Algorithm 1, when all elements have the same value, the partitioning routine returns r. This leads to the situation where the partitioning is maximally unbalanced: only the call $QuickSort(A, p, r-1)$ to sort the resulting left subarray $A[p...r-1]$ survives the recursion, whereas the other call $QuickSort(A, r+1, r)$ to sort the resulting right subarray $A[r+1...r]$ terminates almost immediately because its size is 0 (See lines 4 and 5 of Algorithm 2). Thus, the running time is $T(n) = T(n-1) + T(0) + \Theta(n)$. Solving this recurrence, we have $T(n) = \Theta(n^2)$. $\square$

[1] Insertion sort starts with $i = 2$.
[2] In the worst case, the correct position for the key is $A[1]$.

Algorithm 1 implements Partition

```
1: procedure PARTITION(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for j = p → r − 1 do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             A[i] ⟺ A[j]
8:     A[i + 1] ⟺ A[r]
9:     return i + 1
```

Algorithm 2 implements Quick Sort

```
1: procedure QUICK-SORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
```

2) Modify the algorithm of quick sort presented in the lecture so that it sorts an array in non-increasing order.

Solution: Change the comparison test in line 5 of Algorithm 1 from $\leq$ to $\geq$.

Note: To formally show that the modified algorithm works as required, we need to show this using loop invariant analysis as we did in the lecture.

3) Under what situations does the running time of quick sort end up in the worst case ?

Solution: The algorithm results in the worst-case running time when the partitioning is maximally unbalanced at all levels of recursion: we demonstrated in 2) one special case where if all elements have the same value, the running time becomes $\Theta(n^2)$.

Generally, there are two ways to arrive at maximally unbalanced partitioning at every level of recursion as follows:

The left subarray has $n - 1$ elements ,wheres the right subarray has 0 elements. The recurrence of the running time is then $T(n) = T(n - 1) + T(0) + \Theta(n)$.

The right subarray has 0 elements ,wheres the right subarray has $n - 1$ elements. The recurrence of the running time is then $T(n) = T(0) + T(n - 1) + \Theta(n)$.

In both cases, the running time is $\Theta(n^2)$. $\square$

Problem 5.3. Strassen's Algorithm

1) How do you apply Strassen's Algorithm to multiply two $n \times n$ matrices where n is not an exact power of two?

Solution: Suppose $2^k < n < 2^{k+1}$ for some integer k. We can pad the original matrices A and B with $2^{k+1} - n$ rows and columns of zeroes as shown below.

$$\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & 0 \end{bmatrix}$$

2) What is the running time $T(n)$ of the modified Strassen's algorithm in asymptotic notation?

Solution: Suppose $2^k < n < 2^{k+1} = m$. Thus, $m = 2^{k+1} < 2n < 2^{k+2} = 2m$
Consider $m < 2n$.
Hence, we have

$$m^{\log_2 7} < (2n)^{\log_2 7}$$

$$m^{\log_2 7} < 2^{\log_2 7} n^{\log_2 7}$$

$$m^{\log_2 7} < 7n^{\log_2 7} \tag{1}$$

Consider $m > n$.
Hence, we have

$$m^{\log_2 7} > n^{\log_2 7} \tag{2}$$

Combining Eq.1 and Eq.2,

$$n^{\log_2 7} < m^{\log_2 7} < 7n^{\log_2 7}$$

$$n^{\log_2 7} \leq m^{\log_2 7} \leq 7n^{\log_2 7} \quad [< \text{implies} \leq]$$

Thus, the running time of the modified algorithm is $T(n) = \Theta(n^{\log_2 n})$ where $\frac{m}{2} < n < m$ and m is an exact power of two. □

Problem 5.4. Recurrences and Master Theorem

1) Write a recurrence for the number of scalar additions for Strassen's Algorithm using asymptotic notation (not in the exact form).

Solution: Let $C(n)$ be the number of scalar additions for multiplying two $n \times n$ matrices using Strassen's algorithm. In Strassen's algorithm, the original problem is broken down into 7 smaller subproblems of size $\frac{n}{2}$, each of which yields $C(\frac{n}{2})$ scalar additions. To combine these subproblems, we need 18 **matrix-level** additions, each of which amounts to $(\frac{n}{2})^2$ scalar additions. Thus, the total number of scalar additions can be formulated as a recurrence relation as follows.

$$A(n) = \begin{cases} 7A(\frac{n}{2}) + 18(\frac{n}{2})^2, & \text{if } n > 1. \\ 0 & \text{if } n = 1. \end{cases}$$

Asymptotically, we have

$$A(n) = 7A(\frac{n}{2}) + \Theta(n^2)$$

2) Solve the recurrence using the recursion tree method and confirm your answer with Master theorem.

Solution: Assume that n is an exact power of two. In other words, $n = 2^k$ for some integer k.

$$A(2^k) = 7A(2^{k-1}) + 4^k c \text{ for some } c \in R^+$$

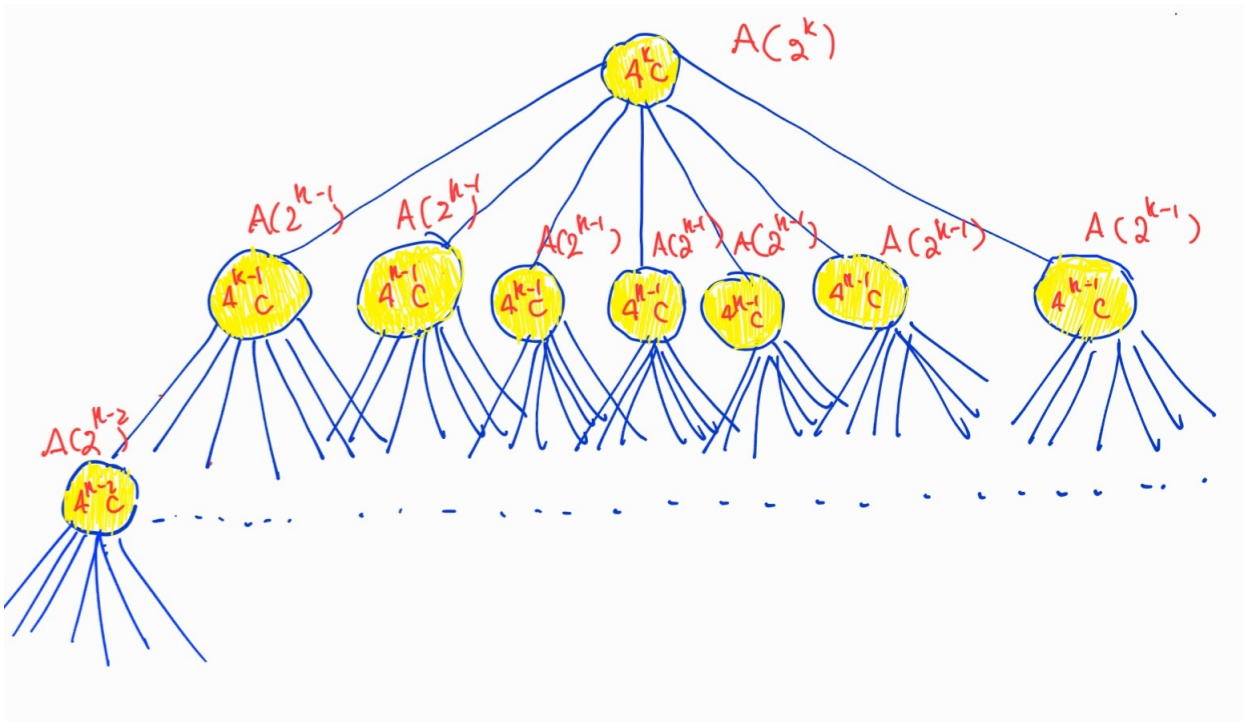

Figure 1: Calculating the number of scalar additions performed in Strassen's algorithm

At level i, the work done at each node is $4^{k-i}c$ and there are 7^i such nodes. Thus, the work done at level i is $7^i 4^{k-i}c$. Recursion bottoms out when $2^{k-i} = 1$, that is, when $i = k$. Since $A(1) = 0$, the total work done at level k is 0. Therefore, the total work can be computed by summing up all the work done from level $i = 0$ to level $i = k - 1$ as follows:

$$A(2^k) = 4^k c + 7 \cdot 4^{k-1}c + 7^2 \cdot 4^{k-2}c + ... + 7^{k-1} \cdot 4c$$

$$A(2^k) = 7^0 4^k c + 7^1 4^{k-1}c + 7^2 4^{k-2}c + ... + 7^{k-1} 4^1 c$$

$$A(2^k) = 4^k c \sum_{i=0}^{k-1} (\frac{7}{4})^i \quad \text{[Factoring } 4^k \text{ out]}$$

$$A(2^k) = 4^k c \{ \frac{(\frac{7}{4})^0 ((\frac{7}{4})^k - 1)}{\frac{7}{4} - 1} \} \quad \text{[Geometric Sum]}$$

$$A(2^k) = \frac{4}{3} c (7^k - 4^k)$$

Since $n = 2^k$, we have $k = \log_2 n$.
Thus,

$$A(n) = \frac{4}{3} c (7^{\log_2 n} - 4^{\log_2 n})$$

$$A(n) = \frac{4}{3} c (n^{\log_2 7} - n^{\log_2 4}) \quad [x^{\log_a y} = y^{\log_a x}]$$

$$A(n) = \frac{4}{3} c (n^{\log_2 7} - n^2)$$

Therefore, $A(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Since $A(n) = 7A(\frac{n}{2}) + \Theta(n^2)$, we have $a = 7$, $b = 2$ and $d = 2$. Since $a > b^d$, $A(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$ by Master Theorem. $\square$