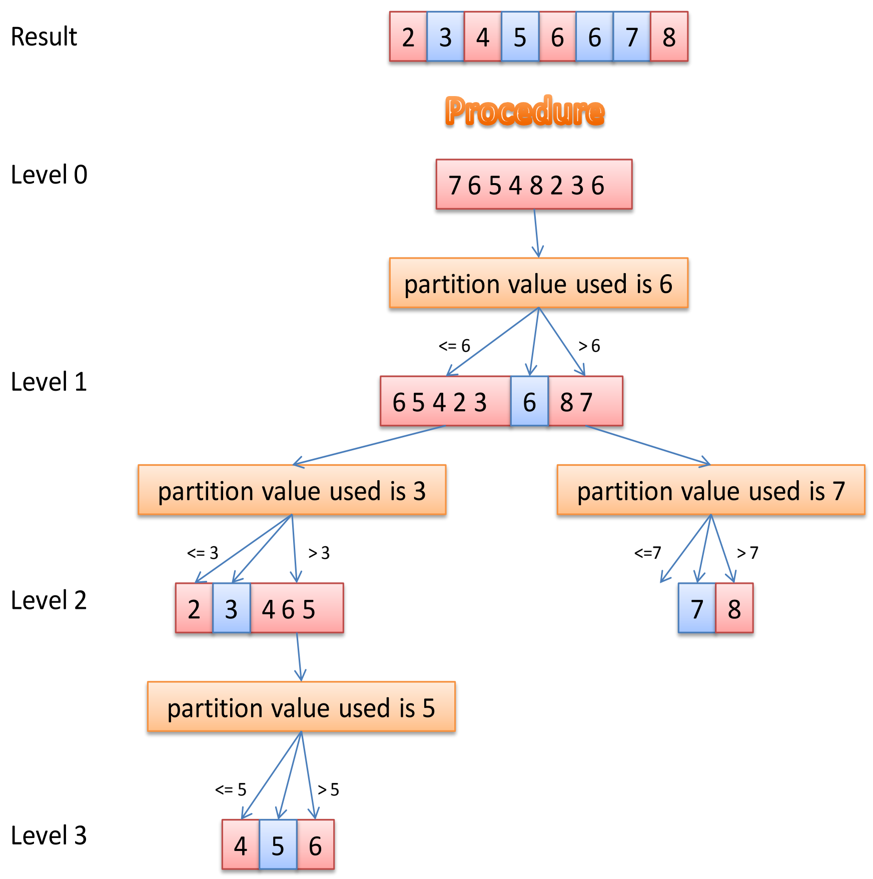
CS2223 ALGORITHMS D term 2008

Dept. of Computer Science, WPI

Homework 5 - Solutions by Yaobin Tang

Problem 1

(1) **Trace of the Quicksort and Partition Algorithms:**



Step by Step

```
Start of Sorting A[1...8]: 7 6 5 4 8 2 3 6

Start of Partition A[1...8]:

    Partition Value = 6

        i=0,j=1,A[j]=7, A[j] > 6, i unchanged, j increased by 1.

        Array A[1...8] After Operation: 7 6 5 4 8 2 3 6

        i=0,j=2,A[j]=6, A[j] <= 6, i increased by 1, i=1,j=2. Swap A[1] with A[2].

        Array A[1...8] After Operation: 6 7 5 4 8 2 3 6

        i=1,j=3,A[j]=5, A[j] <= 6, i increased by 1, i=2,j=3. Swap A[2] with A[3].
        Array A[1...8] After Operation: 6 5 7 4 8 2 3 6

        i=2,j=4,A[j]=4, A[j] <= 6, i increased by 1, i=3,j=4. Swap A[3] with A[4].

        Array A[1...8] After Operation: 6 5 4 7 8 2 3 6

        i=3,j=5,A[j]=8, A[j] > 6, i unchanged, j increased by 1.

        Array A[1...8] After Operation: 6 5 4 7 8 2 3 6

        i=3,j=6,A[j]=2, A[j] <= 6, i increased by 1, i=4,j=6. Swap A[4] with A[6].

        Array A[1...8] After Operation: 6 5 4 2 8 7 3 6

        i=4,j=7,A[j]=3, A[j] <= 6, i increased by 1, i=5,j=7. Swap A[5] with A[7].

        Array A[1...8] After Operation: 6 5 4 2 3 7 8 6

    Put the partition value into right place: Swap A[6] with A[8].

    The partiton value is place at A[6]

    Array A[1...8] After Operation: 6 5 4 2 3 6 8 7

End of Partition of A[1...8].

After partition, A[1...8] broken into A[1...5], A[6],A[7...8]

Start of Sorting A[1...5]: 6 5 4 2 3

Start of Partition A[1...5]:

    Partition Value = 3

        i=0,j=1,A[j]=6, A[j] > 3, i unchanged, j increased by 1.

        Array A[1...5] After Operation: 6 5 4 2 3

        i=0,j=2,A[j]=5, A[j] > 3, i unchanged, j increased by 1.

        Array A[1...5] After Operation: 6 5 4 2 3

        i=0,j=3,A[j]=4, A[j] > 3, i unchanged, j increased by 1.

        Array A[1...5] After Operation: 6 5 4 2 3

        i=0,j=4,A[j]=2, A[j] <= 3, i increased by 1, i=1,j=4. Swap A[1] with A[4].

        Array A[1...5] After Operation: 2 5 4 6 3

    Put the partition value into right place: Swap A[2] with A[5].

    The partiton value is place at A[2]

    Array A[1...5] After Operation: 2 3 4 6 5

End of Partition of A[1...5].

After partition, A[1...5] broken into A[1...1], A[2],A[3...5]

Start of Sorting A[1...1]: 2

    Array A[1...1] After Operation: 2

End of Sorting A[1...1].

Start of Sorting A[3...5]: 4 6 5
Start of Partition A[3...5]:

    Partition Value = 5

        i=2,j=3,A[j]=4, A[j] <= 5, i increased by 1, i=3,j=3. Swap A[3] with A[3].

        Array A[3...5] After Operation: 4 6 5

        i=3,j=4,A[j]=6, A[j] > 5, i unchanged, j increased by 1.

        Array A[3...5] After Operation: 4 6 5

    Put the partition value into right place: Swap A[4] with A[5].

    The partiton value is place at A[4]

    Array A[3...5] After Operation: 4 5 6

End of Partition of A[3...5].

After partition, A[3...5] broken into A[3...3], A[4],A[5...5]

Start of Sorting A[3...3]: 4

Array A[3...3] After Operation: 4

End of Sorting A[3...3].

Start of Sorting A[5...5]: 6

Array A[5...5] After Operation: 6

End of Sorting A[5...5].

Array A[3...5] After Operation: 4 5 6

End of Sorting A[3...5].

Array A[1...5] After Operation: 2 3 4 5 6

End of Sorting A[1...5].

Start of Sorting A[7...8]: 8 7

Start of Partition A[7...8]:

    Partition Value = 7

        i=6,j=7,A[j]=8, A[j] > 7, i unchanged, j increased by 1.

        Array A[7...8] After Operation: 8 7

    Put the partition value into right place: Swap A[7] with A[8].

    The partiton value is place at A[7]

    Array A[7...8] After Operation: 7 8

End of Partition of A[7...8].

After partition, A[7...8] broken into A[7...6], A[7],A[8...8]

Start of Sorting A[7...6]:

Array A[7...6] After Operation:

End of Sorting A[7...6].
```

```
Start of Sorting A[8...8]: 8

Array A[8...8] After Operation: 8

End of Sorting A[8...8].

Array A[7...8] After Operation: 7 8

End of Sorting A[7...8].

Array A[1...8] After Operation: 2 3 4 5 6 6 7 8

End of Sorting A[1...8].
```

(2) **Quicksort Correctness Proof:**

Preliminary:

Before proving that the quicksort algorithm works correctly, let us first look at the partition algorithm. We want to prove that the partition algorithm partitions the original array (let us call the original array A) into 3 parts: a single value d which is used to partition the original array, a sub-array which contains all the elements (except d itself) that are less or equal than d (let us call this sub-array LTE), a sub-array which contains all the elements that are larger than d (let us call this sub-array GT).

The idea of the partition algorithm is to maintain the following invariance before and during the process of the for-loop: A[p..i] contains elements from A that are less than or equal to the pivot value (A[r]), A[i+1..j] contains elements from A that are larger than the pivot value.

Before the for-loop: i=p-1, j undefined, so A[p..i] is empty, and A[i+1,j] has no meaning (since j has not been defined yet). The invariance trivially holds.

During the for-loop: Let us **assume** right after the kth iteration (with j = m), the invariance holds, and now we enter the beginning of the (k+1)-th iteration (with j=m+1). Because invariance is maintained right after the k-th iteration (**our assumption**), we know A[p..i] contains elements from A that are less than or equal to the pivot value, and A[i+1..m] contains elements from A that are larger than the pivot value. However, since in the (k+1)-th iteration, j is incremented from m to m+1, the invariance "A[i+1..j] contains elements from A that are larger than the pivot value" may not be true. That is because A[j] = A[m+1] may be less than or equal to the pivot value. If A[m+1] is really less than or equal to the pivot value, in order to maintain the invariance, we need to swap A[m+1] with some value that is sure to be larger than the pivot value. Because we know from the assumption, A[i+1] is larger than the pivot value, we can swap A[m+1] with A[i+1]. Now A[p..i+1] contains elements from A that are less than or equal to the pivot value, and A[i+2..j] contains elements from A that are larger than the pivot value. To maintain the form of the invariance, we need to update i to i+1. Then after the (k+1)-th iteration, the invariance is still maintained.

After the for-loop: We know A[p..i] contains elements from A that are less than or equal to the pivot value (A[r]), A[i+1..r-1] contains elements from A that are larger than the pivot value. If we swap A[i+1] with A[r], we know A[p..i] is the sub-array LTE, A[i+1] is the pivot value, and A[i+2..r] is the sub-array GT.

After looking at the partition algorithm, let us prove the quicksort algorithm works correctly.

Proof by Induction.

Let us induct on the size of the array to sort.

If the size is 1, it is trivial that the algorithm works correctly.

Let us assume the quicksort algorithm works correctly for any size <= K.

We need to prove that the quicksort algorithm still works correctly when size is K+1.

After the partition algorithm, the original array is partitioned into LTE, d, GT.

The size of LTE, and the size of GT are <= K, so we can use the quicksort algorithm to correctly sort each of them separately (our assumption).

Since any element in LTE is less than or equal to d, and any element in GT is larger than d, the concatenation of sorted LTE, d, sorted GT is the correct sorting of the original array.

(3) Worst-case partitioning for quicksort: Runtime Analysis of the Partitioning Algorithm

Let n=p-r+1

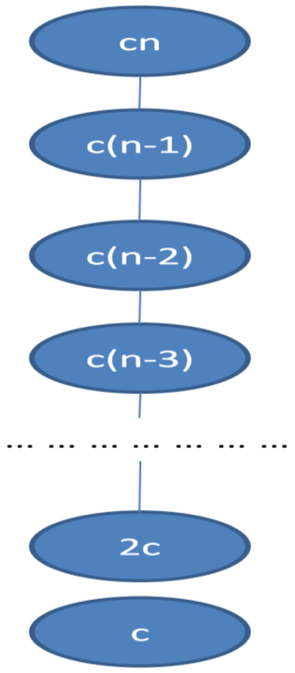
```
partition(A,p,r) {
  x := A[r]                  Θ(1)
  i := p-1    Θ(1)
  For j := p to r-1 do {     Θ(1) x n
    If A[j] ⩽ x then {       Θ(1) x n
      i := i+1               Θ(1) x n
      exchange A[i] with A[j]  Θ(1) x n
    }
  }
  exchange A[i+1] with A[r]  Θ(1)
  return i+1                 Θ(1)
}
```

So the total time complexity is Θ(n).

(4) **Worst-case partitioning for quicksort: Runtime Analysis**

T(n) = T(n-1) + Θ(n). Hence, T(n) ≤ T(n-1) + cn, for some constant c > 0

Recursion-tree method



Level 0: # of nodes 1, runtime per node cn, total runtime of the level c*n

Level 1: # of nodes 1, runtime per node c(n-1), total runtime of the level c*(n-1)

…

Level k: # of nodes 1, runtime per node c(n-k), total runtime of the level c*(n-k)

…

The total runtime is the sum of runtimes over all the levels:

T(n) $\le cn + c(n-1) + c(n-2) + \cdots + 2c + c = c \cdot \frac{(n+1)n}{2} = O(n^2)$

Substitution Method

Let $\Theta(\mathrm{n}) = \mathrm{f}(\mathrm{n}) \le \mathrm{dn}$, d > 0

Basis: (This boundary condition does not appear in the problem statement, added here for completeness. It is sort of arbitrary.) T(1)=e, e>0.

Guess: $T(n) \le cn^2 + cn, c \ge \max\left(\frac{d}{2}, \frac{e}{2}\right)$.

Induction Hypothesis: $T(n-1) \le c(n-1)^2 + c(n-1)$

Induction:

T(n) = T(n-1) + Θ(n) ≤ $c(n-1)^2 + c(n-1) + \Theta(\mathrm{n}) = cn^2 - cn + \Theta(\mathrm{n}) \le cn^2 - cn + dn$

Since $c \ge \frac{d}{2}$, T(n) ≤ $cn^2 - cn + dn \le cn^2 + cn$

hence $cn^2 + cn = O(n^2)$, and so T(n)=O(n^2).

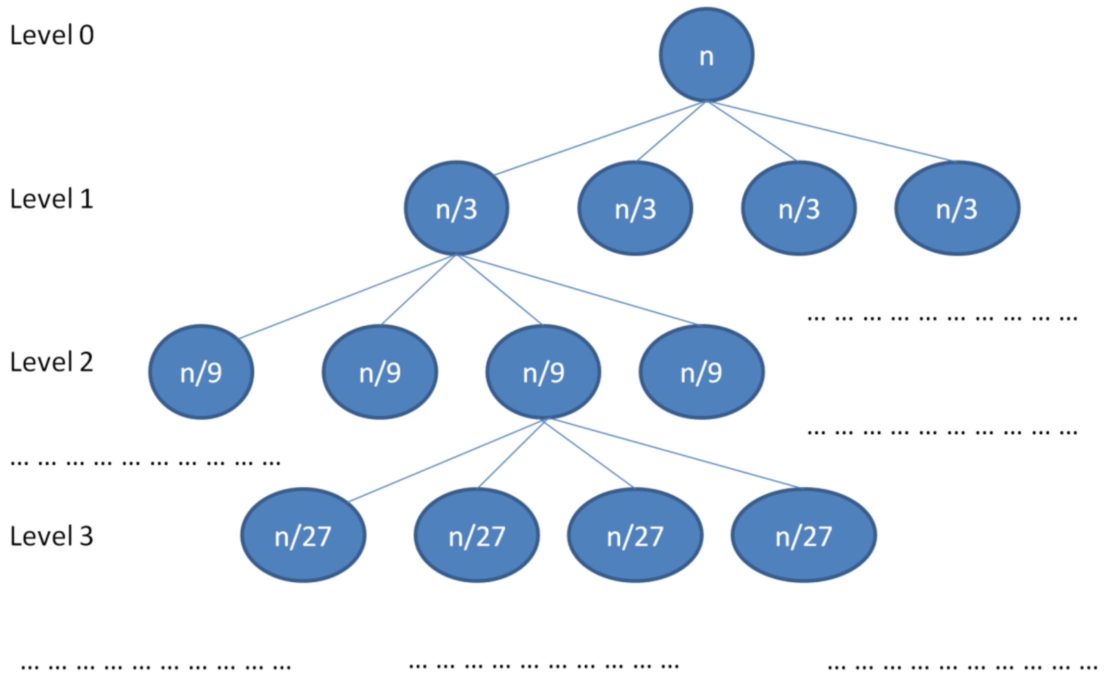
(5) Best-case partitioning for quicksort: Runtime Analysis

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(\mathrm{n})$$

This is the same recurrence that describes the runtime of the Mergesort algorithm analyzed in class and in the textbook. See Section 5.1 of the textbook, p210. Hence, $T(n) = \Theta(\mathrm{n}\log_2 \mathrm{n})$.

Problem 2

(1) Recursion-tree method



Level 0: # of nodes 1, runtime per node n, total runtime of the level 1*n

Level 1: # of nodes 4^1, runtime per node $\frac{n}{3}$, total runtime of the level $4^1 * \frac{n}{3}$

Level 2: # of nodes 4^2, runtime per node $\frac{n}{3^2}$, total runtime of the level $4^2 * \frac{n}{3^2}$

……

Level k: # of nodes 4^k, runtime per node $\frac{n}{3^k}$, total runtime of the level $4^k * \frac{n}{3^k}$

……

Let $\frac{n}{3^k}$ = 1, we have k=$\log_3 n$. So we know $\log_3 n$ is the maximum level.

Level $\log_3 n$: # of nodes $4^{\log_3 n}$, runtime per node 1, total runtime of the level $4^{\log_3 n} * 1$

Sum up runtime of each level, we get

$T(n) = \sum_{i=0}^{\log_3 n} 4^i \frac{n}{3^i} = n \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i$

According to the formula of geometric series, $\sum_{k=0}^{n} r^k = \frac{r^{n+1}-1}{r-1}$,

So

$$\sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i = \frac{\left[\frac{4}{3} \cdot \left(\frac{4}{3}\right)^{\log_3 n}\right] - 1}{\frac{4}{3} - 1} = 4 \cdot \left(\frac{4}{3}\right)^{\log_3 n} - 3 = 4 \cdot \frac{4^{\log_3 n}}{3^{\log_3 n}} - 3 = 4 \cdot \frac{4^{\log_3 n}}{n} - 3$$

Above, we used the fact that $3^{\log_3 n} = n$. So

$T(n) = n \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i = n\left(\frac{4}{n} \cdot 4^{\log_3 n} - 3\right) = 4 \cdot 4^{\log_3 n} - 3n = 4 \cdot 4^{\log_3 n} - 3n = 4 \cdot n^{\log_3 4} - 3n$

Note that $4^{\log_3 n} = n^{\log_3 4}$, or more generally $a^{\log_b n} = n^{\log_b a}$.

Note that $\log_3 4 > 1$, therefore $n^{\log_3 4} > n$. So according to the definition of Big O, we have $T(n) = O(n^{\log_3 4})$.

(2) Substitution Method

Basis: (This boundary condition does not appear in the problem statement, added here for completeness. It is sort of arbitrary.) T(1)=1

Guess: $T(n) \le cn^{\log_3 4} + dn$.

Induction Hypothesis: $T(m) \le c\, m^{\log_3 4} + dm$, for all m < n.

Induction: Since n/3 < n:

$$T(n) = 4T\left(\frac{n}{3}\right) + n \le 4\left(c(\frac{n}{3})^{\log_3 4} + \frac{dn}{3}\right) + n$$

$$= 4\left(c\frac{n^{\log_3 4}}{4} + \frac{dn}{3}\right) + n = cn^{\log_3 4} + \frac{4}{3}dn + n$$

In order to make $T(n) \le cn^{\log_3 4} + dn$, we try to make $cn^{\log_3 4} + \frac{4}{3}dn + n \ = \ cn^{\log_3 4} + dn$.

$$\leftrightarrow \frac{4}{3}dn + n = \ dn \leftrightarrow d = -3$$

Revising our guess to be: $T(n) \le cn^{\log_3 4} - 3n$

and our induction hypothesis to be: $T(m) \le c\, m^{\log_3 4} - 3m$, for all m < n

we have proven that $T(n) \le cn^{\log_3 4} - 3n$. Since $\log_3 4 > 1$, then $n^{\log_3 4} > n$ and then we have $T(n) = O(n^{\log_3 4})$.

<u>Problem 3</u>

According to the rule of matrix multiplication, $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a^2 + bc & b(a+d) \\ c(a+d) & cb + d^2 \end{bmatrix}$

So we only need five multiplications: a*a, b*c, b*(a+d), c*(a+d), d*d.

Note than since a, b, c, d are numbers (and not submatrices) and commutatively of number multiplication holds, then bc=cb; a*b + b*d = b*(a+d); and c*a + d*c = c*(a+d).