Homework 1

By Eric Schles

Assignment

Exercises:

- 2.3-3

- 2.3-5 (page 39)
- 3.1-1,
- 3.1-2,
- 3.1-4 (page 52-53)
- 4.3-1

- 4.3-6 (page 87)
- 4.4-1

- 4.4-4 (page 92-93)
- 4.5-1 (page 96)

Problems:

- 2-1 (page 39-40)
- 3-2 (page 61)
- 4-1 (page 107)

Chapter 2

Exercises

2.3-3

Problem statement:

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

T(n) = 2 if n = 2

T(n) = 2T(n/2) + n, if n = 2^k, for k > 1

is T(n) = n lg n.

Solution:

For this proof, we will make use of mathematical induction.

We begin by showing the base case. In this case, we start with n=2 and show T(n) = 2.

We begin by making use of the definition:

T(2) = 2 lg 2 by assumed base of lg is 2, we have:

lg 2 = 1 therefore,

T(2) = 2 * 1 = 2,

Thus the base case is equivalent, we have proved the base case.

Next we assume the ith case, namely, $n = 2^i$:

$$T(2^i) = 2^i\ lg\ 2^i$$

Therefore,

$$T(n) = 2^i * i$$

By, lg of base 2.

Now we prove that the ith + 1 case follows the pattern, assuming the ith case is true, namely \$ n = 2^{i+1} \$:

$$T(2^{i+1}) = 2T(2^{k+1}/2) + 2^{k+1}$$

(division by 2)

$$= 2T(2^k) + 2^{k+1}$$

(by ith step true)

$$= 2 * 2^k * k + 2^{k+1}$$

$$= 2^{k+1}(k+1)$$

(by definition of lg base 2)

$$= 2^{k+1}\ lg\ 2^{k+1}$$

Q.E.D.

Therefore by the inductive hypothesis $T(n) = n\ lg\ n$.

2.3-5

Problem Statement:

Referring 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 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 psuedocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\theta(lg\ n)$.

Solution:

For this problem I did both versions. Below is the recursive version:

```
from typing import List

def binary_search(arr: List, value: int) -> int:
    if len(arr) == 1 and arr[0] != value:
        return -1
    if arr[0] == value:
        return 1
    mid_point = len(arr)//2
    if value < arr[mid_point]:
         return binary_search(arr[:mid_point], value)
     else:
         return binary_search(arr[mid_point:], value)
```

And here is the iterative:

```
from typing import List

def iterative_binary_search(arr: List, value: int) -> int:
    low = 0
    high = len(arr)-1
    while low < high:
        mid_point = (high + low)//2
        if arr[mid_point] < value:
            low = mid_point + 1
         elif arr[mid_point] > value:
             high = mid_point - 1
         elif arr[mid_point] == value:
             return mid_point
     return -1
```

Aside:

For both solutions, please note that I am using the `typing` library, but this just gives type annotations. All the code is 'psuedocode', in that I'm not making extensive use of the Python language. I'm hoping this is okay. But if you would prefer I didn't include type annotations in the future, please let me know.

Showing the worst-case running time:

The above algorithm has a worst-case running time of $\theta(n\ lg\ n)$. To see this we need to consider each time the function is called. For this I will make use of the convention used in the chapter with respect to insertion sort.

```
from typing import List

def binary_search(arr: List, value: int) -> int:
    if len(arr) == 1 and arr[0] != value:             # c1 * 1
        return -1                                     # c2 * 1
    if arr[0] == value:                               # c3 * 1
        return 1                                      # c4 * 1
    mid_point = len(arr)//2                           # c5 * 1
    if value < arr[mid_point]:                        # c6 * 1
         return binary_search(arr[:mid_point], value) # c7 * 1
     else:                                            # c8 * 1
         return binary_search(arr[mid_point:], value) # c9 * 1
```

Therefore the equation for a single call to binary search is:

c1 + c2 + c3 + c4 + c5 + c6 + c7 + c8 + c9

As you can see, none of these terms contain an `n` therefore, an individual call is $O(1)$. However, we need to specify the full equation for $T(n)$ still. From that we see:

$T(n) = O(1)$ if n = 1

$T(n) = T(\frac{n}{2}) + O(1)$ if n >= 2

Thus $T(n) = lg\ n$.

This follows from the master method discussed in chapter 4.

Problems

2-1 Insertion sort on small arrays in merge sort

Problem Statement:

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 within merge sort when subproblems become sufficiently small. Consdier 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.

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

b. Show how to merge the sublists in $\theta(nlg(\frac{n}{k}))$ worst-case time.

c. Given that the modified algorithm runs in $\theta(nk + nlg(\frac{n}{k}))$ 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?

d. How should we choose k in practice?

Solution:

First, I provide the psuedocode for the algorithm, even though this is not used in any of the specific questions, I feel it is a useful touchstone for thinking about the questions.

```
from typing import List

def merge_sort(k: int, arr: List) -> List:
    if len(arr) <= k:
        return insertion_sort(arr)
    else:
        mid = len(arr)//2
        left = merge_sort(k, arr[:mid])
        right = merge_sort(k, arr[mid:])
        return merge(left, right)

def merge(left, right):
    left_index = 0
    right_index = 0
    new_arr = []
    while left_index < len(left) and right_index < len(right):
        if left[left_index] <= right[right_index]:
            new_arr.append(left[left_index])
            left_index += 1
        else:
            new_arr.append(right[right_index])
            right_index += 1
    while left_index < len(left):
        new_arr.append(left[left_index])
        left_index += 1
    while right_index < len(right):
        new_arr.append(right[right_index])
        right_index += 1
    return new_arr

def insertion_sort(arr: List) -> List:
    for j in range(1, len(arr)):
        key = arr[j]
        i = j - 1
        while i >= 0 and arr[i] > key:
            arr[i + 1] = arr[i]
            i -= 1
        arr[i + 1] = key
    return arr
```

a. We start by stating the worst-case running time for insertion sort:

$$\theta(n^2)$$

.

Then we recognize that we are sorting $\frac{n}{k}$ sublists, each of length k. Therefore we have running time:

$$\theta(k^2 * \frac{n}{k})$$

Which is equal to:

$$\theta(nk)$$

b. We begin by observing that we have $\frac{n}{k}$ sorted sublists. To do the merging of these sublists, we sort two sublists at a time.

Therefore we can think of the problem as follows:

There are $\frac{n}{k} = j$ and with each recursive call, we sort 2 of them. Also note, we will need to iterative over k elements to merge all the elements at a given call. Therefore we can think of the total run time of the merging as:

$$T(j) = 2T(\frac{j}{2}) + O(k)$$

The general form of the solution for the above worst-case running time function is: $T(n) = n\ lg\ n$.

And in particular, we have $\theta(n\ lg\ \frac{n}{k})$ since we have to iterate $\frac{n}{k}$ times. Notice, we did not specify a size for k, which is why this is a $\theta(n\ lg\ \frac{n}{k})$ algorithm.

c. We can find the value of k by setting the worst-case running time of the modified algorithm equal to the worst-case running time of merge sort:

$$\theta(nk + n\ lg\ \frac{n}{k}) = \theta(n\ lg\ n)$$

So we will drop the θ for ease of notation, just to make the mathematics easier to work with, but it hasn't gone away. Anyway, so we have:

$$nk + n\ lg\ \frac{n}{k} = n\ lg\ n$$

$$nk = n\ lg\ n - n\ lg\ \frac{n}{k}$$

$$nk = n\ lg\ \frac{n}{\frac{n}{k}}$$

$$nk = n\ lg\ n * \frac{k}{n}$$
$$k = lg\ k$$

So $k = \theta(lg\ n)$.

We can verify this by starting with the modified version and plugging in for k with $\theta(lg\ n)$.

$$\theta(nk + n\ lg(\frac{n}{k}))$$
$$\theta(n\ lg\ n + n\ lg\ n - n\ lg(lg\ n))$$
$$\theta(2n\ lg\ n - n\ lg(lg\ n))$$
$$\theta(n\ lg\ n)$$

.

Q.E.D.

d. This will depend on how disordered the lists are, the programming language, the hardware involved and other factors that are typically not accounted for in worst-case analysis. Since some of these factors maybe hard to account for, I would probably run experiments on random data on a typical machine at my company. Assuming this is any modern shop, we are likely using AWS EC2 instances, and the specs will likely be standardized for the problem domain. After accounting for computational consistency, then I would make sure to run all experiments using the language that will be used in production. Finally, when generating my randomly sampled elements, I would try to ensure that the data was similarly distributed. Assuming that we have sample production data, I would make use of that to learn a distribution and fit parameters. And then during the data generation process, I would randomly sample with the same distribution and fit parameters. Then I would experimentally verify the values of k for which the modified algorithm performed better consistently over traditional merge sort. As far as number of samples, I'm assuming sorting is a fairly typical operation, so I would test different numbers of samples, based on typical usage. A simple K-means clustering algorithm should give me the relative central tendencies for sample sizes. Finally, I would set the number of experiments to $10e^8$ to ensure each set of experiments had low likelihood of low sample bias. I would record the optimal k's and store them in some state file, perhaps json? And then call them along with the algorithm based on the array size and the nearest optimized k.

Chapter 3

Exercises

3.1-1

Let f(n) and g(n) be asymptotically nonnegative functions. Using the basic definition of θ-notation, prove that

$$max(f(n),\ g(n)) = \theta(f(n) + g(n))$$

Solution:

Recall the definition of $\theta(g(n))$:

{ f(n): $\exists c_1, c_2, n_0$ s.t. $0 <= c_1 g(n) <= f(n) <= c_2 g(n)\ \forall n >= n_0, c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N}$ }

Therefore we simply need to find suitable values such that:

$$c_1(f(n) + g(n)) <= max(f(n),\ g(n)) <= c_2(f(n) + g(n))$$

LHS:

First we will find c_1:

We begin by obsering that:

$$a + b <= 2max(a, b)$$
$$\frac{a+b}{2} <= max(a, b)$$

Thus the arithmetic average is bounded by the max of a and b.

Since there is nothing special about f(n) + g(n), we can make use of this to conclude that:

$$\frac{1}{2}(f(n) + g(n)) <= max(f(n), g(n))$$

RHS:

For finding c_2 we simply need to observe that the max function will only pick the bigger of the two numbers, so, as long as 0 < f(n), 0 < g(n) any 1 < c_2 will yield:

$$max(f(n), g(n)) <= c_2(f(n) + g(n))$$

For simplicity, let's choose $c_2 = 2$.

Thus in conclusion we have:

$$\frac{1}{2}(f(n) + g(n)) <= max(f(n), g(n)) <= 2(f(n) + g(n))$$

So

$$max(f(n), g(n)) = \theta(f(n) + g(n))$$

.

Q.E.D.

3.1-2

Show that for any real constants a and b, where b > 0,

$$(n + a)^b = \theta(n^b)$$

Solution:

We begin by using the equation form of $(n + a)^b$.

By the bionomial theorem we have:

$$(n + a)^b = \sum_{k=0}^{b} \binom{n}{k} b^{b-k} a^k$$

from the above functional form, we can take $\binom{n}{k} = c_k$ - what we are saying here is that since $\binom{n}{k}$ is simply a constant, we can generate a list of indexed constants. Thus we can rewrite the above equation as:

$$\sum_{k=0}^{b} \binom{n}{k} n^{b-k} a^k = \sum_{k=0}^{b} c_k n^{b-k} a^k$$

Now we recall the definition of $\theta(g(n))$:

{ f(n): $\exists c_1, c_2, n_0$ s.t. $0 <= c_1 g(n) <= f(n) <= c_2 g(n)$ $\forall n >= n_0, c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N}$ }

So we simply need to find c_1, c_2 and n_0.

To get the lower bound we can simply take:

$$c_1 n^b <= \sum_{k=0}^{b} c_k n^{b-k} a^k <= \sum_{k=0}^{b} c_k n^b$$

This is true for $n_0 = 1$.

Thus,

$$(n+a)^b = \theta(n^b)$$

3.1-4

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

Solution:

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

Yes. Since $2^{n+1} = 2 * 2^n$ we have:

$2 * 2^n <= 3 * 2^n$

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

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

No. Since $2^{2n} = (2^n)^2$. Therefore, this function is not bounded above by any $c * 2^n$. Eventually the exponentiation would exceed any constant, c.

Problems

3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below whether A is O, o, Ω, ω, or θ of B. Assume that k >= 1, $\epsilon > 0$ and c > 1 are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	o	Ω	ω	θ
$lg^k\ n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{sin\ n}$	no	no	no	no	no
2^n	$2^{\frac{n}{2}}$	no	no	yes	yes	yes
$n^{lg\ c}$	$c^{lg\ n}$	yes	no	yes	no	yes
$lg(n!)$	$lg(n^n)$	yes	no	yes	no	yes

Chapter 4

Exercises

4.3-1

Show that the solution $T(n) = T(n-1) + n$ is $O(n^2)$.

Solution:

Let's guess $T(n) <= cn^2$,

$T(n) <= c(n-1)^2 + n = cn^2 - 2cn + c + n = cn^2 + n(1-2c) + c <= cn^2$,

Where the last step holds for $c > \frac{1}{2}$.

4.3-6

Show that the solution to $T(n) = 2T(\lfloor \frac{n}{2} \rfloor + 17)) + n$ is O($nlgn$).

Solution:

We guess that $T(n) <= c(n-a)\ lg\ (n-a)$:

$$T(n) <= 2c(\lfloor \frac{n}{2} \rfloor + 17 - a)\ lg(\lfloor \frac{n}{2} \rfloor + 17 - a) + n$$

$$\begin{aligned}
&<= 2c(\frac{n}{2} + 17 - a)\ lg(\frac{n}{2} + 17 - a) + n \\
&<= c(n + 34 - 2a)\ lg(\frac{n}{2} + 17 - a) + n \\
&= c(n + 34 - 2a)\ lg(\frac{n + 34 - 2a}{2}) + n \\
&= c(n + 34 - 2a)\ (lg(n + 34 - 2a) - lg(2)) + n \\
&= c(n + 34 - 2a)\ (lg(n + 34 - 2a) - 1) + n \\
&= c(n + 34 - 2a)\ lg(n + 34 - 2a) - c(n + 34 - 2a) + n \\
&<= c(n + 34 - 2a)\ lg(n + 34 - 2a) \\
&<= c(n - a)\ lg(n - a)
\end{aligned}$$

4.4-1

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 3T(\lfloor \frac{n}{2} \rfloor) + n$. Use the substitution method to verify your answer.

Solution:

Drawing recursion trees in LaTex is super hard so I'll just describe the trees in my solution below.

At the first level we have cn, this is because of the n term in the recurrence. Then we have three leaves in the second level. Each with $c\frac{n}{2}$ running time. At the ith level of the tree, the size of the subproblem is $\frac{n}{2^i}$ and there are 3^{i} many nodes. In total the tree has $lg\ n + 1$ levels and n^{lg3} leaves.

Based on this, total cost over all nodes at depth i should be $(\frac{3}{2})^i * i$.

Thus we have:

$$T(n) = n + \frac{3}{2}n + (\frac{3}{2})^2 n + ... + (\frac{3}{2})^{lg\ n-1} + \theta(n^{lg\ 3})$$

$$= \sum_{i=0}^{lg\ n-1} (\frac{3}{2})^i n + \theta(n^{lg\ 3})$$

$$= \frac{(\frac{3}{2})^{lg\ n} - 1}{\frac{3}{2} - 1} * n + \theta(n^{lg\ 3})$$

by (pg 1179).

calculation of the demonominator:

$$\frac{3}{2} - 1 = \frac{1}{2}$$

and

$$\frac{1}{\frac{1}{2}} = 2$$

Therefore:

$$= 2((\frac{3}{2})^{lg\ n} - 1)n + \theta(n^{lg\ 3})$$

$$= 2(n^{lg\ \frac{3}{2}} - 1)n + \theta(n^{lg\ 3})$$

by ($a^{log_b c} = c^{log_b a}$).

$$= 2(n^{lg\ 3 - lg\ 2} - 1)n + \theta(n^{lg\ 3})$$

$$= 2(n^{lg\ 3-1} - 1)n + \theta(n^{lg\ 3})$$

$$= 2(n^{lg\ 3-1+1} - n) + \theta(n^{lg\ 3})$$

$$= 2(n^{lg\ 3-1+1} - n) + \theta(n^{lg\ 3})$$

$$= 2(n^{lg\ 3} - n) + \theta(n^{lg\ 3})$$

$$= 2n^{lg\ 3} - 2n + \theta(n^{lg\ 3})$$

$$<= cn^{lg\ 3} + \theta(n^{lg\ 3})$$

$$<= O(n^{lg\ 3})$$

Now that we have our general form, let's make our guess for the substitution method:

$$T(n) <= cn^{lg3} - dn$$

$$T(n) = 3T(\lfloor\frac{n}{2}\rfloor) + n$$

$$<= 3 * (c(\frac{n}{2})^{lg\ 3} - d(\frac{n}{2})) + n$$

$$= 3 * (c(\frac{n}{2})^{lg\ 3} - d(\frac{n}{2})) + n$$

$$= 3\frac{1}{2^{lg\ 3}}cn^{lg\ 3} - d(\frac{3n}{2}) + n$$

Dealing with $\frac{1}{2^{lg\ 3}}$:

$$2^{lg\ 3} = 3^{lg\ 2} = 3^1$$

So we get:

$$= \frac{3}{3}cn^{lg\ 3} - d(\frac{3n}{2}) + n$$

$$= cn^{lg\ 3} - d(\frac{3n}{2}) + n$$

$$= cn^{lg\ 3} + (1 - \frac{3d}{2})n$$
$$<= O(n^{lg\ 3})$$

As long as d >= 3.

4.4-4

Use a recursion tree to determine a good asymptotic upper bound on the recurrence T(n) = 2T(n - 1) + 1. Use the substitution method to verify your answer.

Solution:

Drawing recursion trees in LaTex is super hard so I'll just describe the trees in my solution below.

At the first level we have c, this is because of the 1 term in the recurrence. Then we have two leaves in the second level. Each with $c(n-1)$ running time. At the ith level of the tree, the size of the subproblem is $c(n-i)$ and there are 2ˆi many nodes. In total the tree has n levels and 2^n leaves.

Based on this, in total there should be $(2)^i$ many nodes, at depth i.

Thus we have:

$$T(n) = 2^n + 2^{n-1} + ... + 1$$

$$= \sum_{i=0}^{n-1} 2^i$$

$$= \frac{2^n - 1}{2 - 1}$$

by (pg 1179).

$$= 2^n - 1$$

$$= \theta(2^n)$$

Now that we have our general form, let's make our guess for the substitution method:

$$T(n) <= c2^n + n$$

$$= 2 * c2^{n-1} + (n-1) + 1$$

$$= c2^n + n$$

$$T(n) = O(2^n)$$

4.5-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(\frac{n}{4}) + 1$

Solution:

a = 2, b = 4, f(n) = 1

First we determine which case we are in:

$n^{log_4 2} > f(n) = 1$, and is obviously more than n^{ϵ} bigger.

So we are in case 1:

$$T(n) = \theta(n^{log_4\ 2})$$

b. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

Solution:

a = 2, b = 4, $f(n) = \sqrt{n}$

First we determine which case we are in:

$n^{log_4 2} = n^{\frac{1}{2}} == f(n) = n^{\frac{1}{2}}$, and is obviously equivalent.

So we are in case 2:

$$T(n) = \theta(n^{log_4\ 2}\ lg\ n)$$

c. $T(n) = 2T(\frac{n}{4}) + n$

Solution:

a = 2, b = 4, f(n) = n

First we determine which case we are in:

$n^{log_4 2} = n^{\frac{1}{2}} < f(n) = n^1$, and is obviously n^{ϵ} smaller where $\epsilon = \frac{1}{2}$.

So we are in case 3:

$$T(n) = \theta(n)$$

d. $T(n) = 2T(\frac{n}{4}) + n^2$

Solution:

a = 2, b = 4, $f(n) = n^2$

First we determine which case we are in:

$n^{log_4 2} = n^{\frac{1}{2}} < f(n) = n^2$, and is obviously n^{ϵ} smaller where $\epsilon = \frac{3}{2}$.

So we are in case 3:

$$T(n) = \theta(n^2)$$

Problems

4-1

Given asymptotic upper and lower bounds for T(n) in each of the following recurrences. Assume that T(n) is a constant for n >= 2. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(\frac{n}{2}) + n^4$

Solution:

a = 2, b = 2, $f(n) = n^4$

First we determine which case we are in:

$n^{log_2 2} = n^1 < f(n) = n^4$, and is obviously n^ϵ smaller where $\epsilon = 3$.

So we are in case 3:

$$T(n) = \theta(n^4)$$

b. $T(n) = T(7n/10) + n$

Solution:

a = 1, b = $\frac{10}{7}$, $f(n) = n$

First we determine which case we are in:

$n^{log_{\frac{10}{7}} 1} = n^0 < f(n) = n$, and is obviously n^ϵ smaller where $\epsilon = 1$.

So we are in case 3:

$$T(n) = \theta(n)$$

c. $T(n) = 16T(\frac{n}{4}) + n^2$

Solution:

a = 16, b = 4, $f(n) = n^2$

First we determine which case we are in:

$n^{log_4 16} = n^2 == f(n) = n^2$, and is obviously the same size.

So we are in case 2:

$$T(n) = \theta(n^2\ lg\ n)$$

d. $T(n) = 7T(\frac{n}{3}) + n^2$

Solution:

a = 7, b = 3, $f(n) = n^2$

First we determine which case we are in:

$n^{log_3 7} \approx n^{1.77124} < f(n) = n^2$, and is smaller by n^ϵ for $\epsilon = 0.23$.

So we are in case 3:

$$T(n) = \theta(n^2)$$

e. $T(n) = 7T(\frac{n}{2}) + n^2$

Solution:

a = 7, b = 2, $f(n) = n^2$

First we determine which case we are in:

$n^{log_2 7} \approx n^{2.80735} > f(n) = n^2$, and is bigger by n^ϵ for $\epsilon \approx 0.8$.

So we are in case 1:

$$T(n) = \theta(n^{log_2 7})$$

f. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

Solution:

a = 2, b = 4, $f(n) = n^{\frac{1}{2}}$

First we determine which case we are in:

$n^{log_4 2} = n^{\frac{1}{2}} == f(n) = n^{\frac{1}{2}}$, and is equal.

So we are in case 2:

$$T(n) = \theta(n^{\frac{1}{2}}\ lg\ n)$$

g. $T(n) = T(n-2) + n^2$

$\theta(n^3)$