Kaylin Brooks

March 19, 2025

CSC-3130-02

Assignment 5: Recurrences and Hash Tables

Problems:

1. (text) Recurrence [10 points] Solve the following recurrence relation using repeated substitution.

$$T(n) = 3T(n/4) + 4n$$

Then solve it by using the Master Method, showing in detail which rule applies

Note: Show your work. You will get 4 points if you identify the pattern, 3 points if you do the proof work necessary to show what it resolves to, and 3 points for solving it using the master theorem.

Solution:

The given recurrence relation is:

$$T(n) = 3T(n/4) + 4nT(n) = 3T(n/4) + 4n$$

Step 1: Expand the Recurrence

We apply repeated substitution:

1. **First expansion:**

$$T(n) = 3T(n/4) + 4nT(n) = 3T(n/4) + 4n$$

2. **Substituting $T(n/4)T(n/4)$:**

$$T(n/4) = 3T(n/16) + 4(n/4)T(n/4) = 3T(n/16) + 4(n/4)$$

Substituting this into the original equation:

$$\begin{aligned} T(n) = 3(3T(n/16) + 4(n/4)) + 4nT(n) &= 3(3T(n/16) + 4(n/4)) + 4n \\ &= 9T(n/16) + 3(4n/4) + 4n = 9T(n/16) + 3(4n/4) + 4n \\ &= 9T(n/16) + 3n + 4n = 9T(n/16) + 3n + 4n = 9T(n/16) + 7n \\ &= 9T(n/16) + 7n \end{aligned}$$

3. **Substituting $T(n/16)T(n/16)$:**

$$T(n/16) = 3T(n/64) + 4(n/16)T(n/16) = 3T(n/64) + 4(n/16)$$

Plugging this in:

$$\begin{aligned} T(n) &= 9(3T(n/64) + 4(n/16)) + 7nT(n) = 9(3T(n/64) + 4(n/16)) + 7n \\ &= 27T(n/64) + 9(4n/16) + 7n = 27T(n/64) + 9(4n/16) + 7n \\ &= 27T(n/64) + 9n/4 + 7n = 27T(n/64) + 9n/4 + 7n \\ &= 27T(n/64) + 9n/4 + 28n/4 = 27T(n/64) + 9n/4 + 28n/4 \\ &= 27T(n/64) + 37n/4 = 27T(n/64) + 37n/4 \end{aligned}$$

Step 2: Identify the Pattern

Observing the coefficients, we see:

$$T(n) = 3^k T(n/4^k) + \sum_{i=0}^{k-1} 3^i \cdot 4\left(n/4^i\right)$$

Rewriting the summation:

$$T(n) = 3^k T(n/4^k) + 4n \sum_{i=0}^{k-1} \left(\frac{3^i}{4^i}\right)$$

The sum inside is a geometric series:

$$\sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i$$

Using the formula for the sum of a geometric series:

$$S = \frac{1 - r^k}{1 - r}, \qquad where\ r = \frac{3}{4}$$

$$\sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i = \frac{1 - (3/4)^k}{1 - (3/4)}$$

$$= \frac{1 - (3/4)^k}{1/4} = 4(1 - (3/4)^k)$$

Thus, the recurrence simplifies to:

$$T(n) = 3^k T(n/4^k) + 16n(1 - (3/4)^k)$$

When kk is chosen such that $n/4k = 1n/4k = 1$, we solve for k:

$$k = log_4 n$$

Since $T(1)T(1)$ is a constant, say $T(1) = cT(1) = c$:

$$T(n) = 3^{log_4 n} T(1) + 16n(1 - (3/4)^{log_4 n})$$

Approximating the exponents:

$$T(n) = 3^{log_4 n} = n^{log_4 3}$$

$$(3/4)^{log_4 n} = n^{log_4 (3/4)}$$

Since $log_4(3/4)$ is negative, the term $n^{log_4(3/4)}$ approaches 0 for large n, giving:

$$T(n) = O(n^{log_4 3})$$

Solving Using the Master Theorem

The recurrence is of the form:

$$T(n) = aT(n/b) + f(n)$$

where:

- $a = 3$
- $b = 4$
- $f(n) = 4n$

Step 1: Compare $f(n)$ with $n^{log_b a}$

We compute:

$$log_4\ 3 \approx 0.792$$

Since $f(n) = O(n^1)$ and $1 > log_4\ 3$, we apply **Case 1 (Polynomially Larger $f(n)$)**:

Since $f\ (n) = O(n^1)$ dominates $n^{log_4 3}$, and it satisfies the regularity condition $af(n/b) \leq cf(n)$, we conclude:

$$T(n) = \Theta\big(f(n)\big) = \Theta(n)$$

Final Answer:

Using repeated substitution:

$$T(n) = O(n^{log_4\ 3})$$

Using the Master Theorem:

$$T(n) = \Theta(n)$$

2. (text) Master Theorem [20 points] Apply the Master method to solve each of the following recurrences, or state that the Master method does not apply. Justify your answers. Note that the Master method covers all the cases

a. $T(n) = 3T\left(\frac{n}{5}\right) + n^2$

b. $T(n) = 4T\left(\frac{n}{3}\right) + 7n$

c. $T(n) = 5T\left(\frac{n}{4}\right) + 10$

d. $T(n) = 9T\left(\frac{n}{3}\right) + n^4$

e. $T(n) = 6T\left(\frac{n}{8}\right) + n^3$

Solution:

Applying the Master Theorem to Each Recurrence

The **Master Theorem** applies to recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where:

- a is the number of subproblems,
- b is the factor by which the subproblem size decreases,
- $f(n)$ is the additional work done outside the recursive calls.

We compare $f(n) = O(n^d)$ with $n^{log_b a}$ to determine the time complexity.

Master Theorem Cases:

$$T(n) = \begin{cases} \Theta(n^{log_b a}) \text{ if } f(n) = O(n^c) \text{ where } c < log_b a \text{ (Case 1)} \\ \Theta(n^{log_b a} \log n) \text{ if } f(n) = \Theta(n^{log_b a}) \text{ (Case 2)} \\ \Theta(f(n)) \text{ if } f(n) = \Omega(n^c) \text{ where c} > log_b \text{a, and the regularity condition holds (Case3)} \end{cases}$$

Solving Each Recurrence

$(\boldsymbol{a})\ T(n) = 3T\left(\frac{n}{5}\right) + n^2$

Identify parameters:

- $a = 3,\ b = 5,\ f(n) = n^2$
- Compute exponent:

$$log_5 3 \approx 0.682$$

Compare f(n) with $n^{\log_b a}$:

- n^2 vs. $n^{0.682}$
- $Since\ 2 > 0.682\ (case\ 3\ applies),\ T(n) = \Theta(n^2)$

($\boldsymbol{b}$) $T(n) = 4T\left(\frac{n}{3}\right) + 7n$

Identify parameters:

- $a = 4,\ b = 3,\ f(n) = 7n$
- Compute exponent:

$$log_3 4 \approx 1.261$$

Compare f(n) with $n^{\log_b a}$:

- n^1 vs. $n^{1.261}$
- $Since\ 1 < 1.261\ (case\ 1\ applies),\ T(n) = \Theta(n^{1.261})$

($\boldsymbol{c}$) $T(n) = 5T\left(\frac{n}{4}\right) + 10$

Identify parameters:

- $a = 5,\ b = 4,\ f(n) = 20\ (which\ is\ O(1))$
- Compute exponent:

$$log_4 5 \approx 1.161$$

Compare f(n) with $n^{\log_b a}$:

- $O(1)$ vs. $n^{1.161}$
- $Since\ 0 < 1.261\ (case\ 1\ applies),\ T(n) = \Theta(n^{1.161})$

($\boldsymbol{d}$) $T(n) = 9T\left(\frac{n}{3}\right) + n^4$

Identify parameters:

- $a = 9,\ b = 3,\ f(n) = n^4$
- Compute exponent:

$$log_3 9 \approx 2$$

Compare f(n) with $n^{\log_b a}$:

- n^4 vs. n^2
- $Since\ 4 > 2\ (case\ 3\ applies),\ T(n) = \Theta(n^4)$

$(\boldsymbol{e})\ T(n) = 6T\left(\frac{n}{8}\right) + n^3$

Identify parameters:

- $a = 6,\ b = 8,\ f(n) = n^3$
- Compute exponent:

$$log_8 6 \approx 0.903$$

Compare f(n) with $n^{\log_b a}$:

- n^3 vs. $n^{0.903}$
- $Since\ 3 > 0.903\ (case\ 3\ applies),\ T(n) = \Theta(n^3)$

Final Answer:

Recurrence	Parameters	Comparison $f(n)$ vs. $n^{\log_b a}$	Case	Complexity
(a) $T(n) = 3T(n/5) + n^2$	$a = 3, b = 5, f(n) = n^2$	n^2 vs. $n^{0.682}$	Case 3	$\Theta(n^2)$
(b) $T(n) = 4T(n/3) + 7n$	$a = 4, b = 3, f(n) = 7n$	n^1 vs. $n^{1.261}$	Case 1	$\Theta(n^{1.261})$
(c) $T(n) = 5T(n/4) + 10$	$a = 5, b = 4, f(n) = O(1)$	$O(1)$ vs. $n^{1.161}$	Case 1	$\Theta(n^{1.161})$
(d) $T(n) = 9T(n/3) + n^4$	$a = 9, b = 3, f(n) = n^4$	n^4 vs. n^2	Case 3	$\Theta(n^4)$
(e) $T(n) = 6T(n/8) + n^3$	$a = 6, b = 8, f(n) = n^3$	n^3 vs. $n^{0.903}$	Case 3	$\Theta(n^3)$

3. (text) Radix Sort [10 points] Lexicographical ordering means order of the dictionaries to sequences of ordered symbols therefore (a < b < c < d < e < f < … < m < n < o < … < y < z). The same logic applies to Uppercase letters. Illustrate the operation of Radix-Sort on the following list of strings using lexicographic ordering.

CAP, COL, USD, SUN, JPY, VEE, ROW, JOB, COX, LOL, RAT, WOW, DOD, CAR, FIG, PIG, VIS, LOW, LOX, VEA, CAD, DOG, TSL

Solution:

Step 1: Radix Sort processes the strings **right to left**, meaning we start with the **last letter**, then the **second-to-last**, and finally the **first**.

Step 1: Sorting by the last letter

We group and sort based on the last letter:

Last Letter	Strings
A	CAP, CAD, CAR
B	JOB
D	USD, DOD, FIG, CAD, DOG
E	VEE, VEA
G	FIG, PIG, DOG
L	COL, LOL
N	SUN, TSL
O	ROW, LOX, COX
R	RAT, CAR
S	VIS
W	WOW, LOW
X	COX, LOX

Sorted order by last letter:
CAP, CAD, CAR, JOB, USD, DOD, DOG, FIG, VEE, VEA, COL, LOL, SUN, TSL, ROW, COX, LOX, RAT, VIS, WOW, LOW, PIG

Step 2: Sorting by the second letter

Now, we sort based on the **second-to-last letter** while maintaining the previous order.

Second Letter	Strings
A	CAD, CAP, CAR, RAT
B	JOB
D	USD, DOD, DOG
E	VEE, VEA
I	FIG, VIS, PIG
L	LOL, COL, LOX, LOW
O	ROW, COX, WOW
S	SUN, TSL
T	TSL
W	WOW

Sorted order by second letter:
CAD, CAP, CAR, RAT, JOB, USD, DOD, DOG, VEE, VEA, FIG, VIS, PIG, COL, LOL, LOX, LOW, ROW, COX, WOW, SUN, TSL

Step 3: Sorting by the first letter

Finally, we sort by the **first letter**, keeping the order from the previous step.

First Letter	Strings
C	CAD, CAP, CAR, COL, COX
D	DOD, DOG
F	FIG
J	JOB
L	LOL, LOX, LOW
P	PIG
R	RAT, ROW
S	SUN
T	TSL
U	USD
V	VEE, VEA, VIS
W	WOW

Final sorted order:
CAD, CAP, CAR, COL, COX, DOD, DOG, FIG, JOB, LOL, LOX, LOW, PIG, RAT, ROW, SUN, TSL, USD, VEA, VEE, VIS, WOW

4 (text) Double Hashing [15 points] Consider a hash table consisting of M = 13 slots, and suppose nonnegative integer key values are hashed into the table using the hash function h1() and that collisions are resolved by using double hashing with the secondary hash function Reverse(value), which reverses the digits of v and returns that value; for example, Reverse(3652) = 2563.

```
int h1 (int key) {
int x = (key + 19) * (key + 11);
x = x / 15;
x = x + key;
x = x % M; return x;
```

Add the following items [25, 14, 9, 7, 5, 3, 0, 21, 6, 33, 25, 42, 24, 107] to the HashTable in order.

For each key being inserted to the HashTable, show:
(1) The home slot (the initial hashed slot)
(2) The number of collisions and the probe sequence (if collisions occur)
(3) The final contents of the hash table

Hint: You will have to re-size and rehash once

Solution:

Key	Home Slot	Collisions	Probe Sequence
25	0	0	[]
14	4	0	[]
9	7	0	[]
7	12	0	[]
5	4	2	[8]
3	10	0	[]
0	0	2	[1]
21	2	0	[]
6	8	2	[3]
33	3	2	[6]
25	0	2	[1]
42	23	3	[3, 9]
24	20	2	[22]
107	6	0	[]

To solve this problem, I implemented **double hashing** with a **rehashing mechanism** when the load factor exceeded **70%**. I followed these steps:

1. **Initialized the hash table** with **M=13M=13 slots**.
2. **Defined the primary hash function h1(key)h1(key)**, as given in the problem:

   $$x=(key+19)*(key+11)$$

   Then, I performed integer division by 15, added the key, and took the modulus MM to find the **home slot**.

3. **Implemented the secondary hash function**, which reverses the digits of the hash value. This step was used to resolve collisions via **double hashing**.
4. **Inserted each key into the hash table**, checking:
   - If the **home slot** was empty, I placed the key there.
   - If a **collision occurred**, I used **double hashing** by computing a step size using the **Reverse function** and probing until an empty slot was found.
5. **Implemented rehashing**:
   - If the **load factor exceeded 70%**, I **doubled the table size** and **reinserted all elements** to maintain the hash table structure.
6. **Tracked and recorded**:
   - The **home slot** for each key.
   - The **number of collisions** that occurred.
   - The **probe sequence**, which records the alternative slots checked when a collision occurred.

Understanding Empty Probe Sequences

While recording the probe sequences, I noticed that some keys had **empty brackets (`[]`)** in their probe sequence. This happened because **no collisions occurred** for those keys. If a key was placed **directly into its home slot without conflicts**, then no additional slots were checked, resulting in an **empty probe sequence**.

On the other hand, if a collision did occur, I used the **reverse function** to determine the step size and then probed other slots. In these cases, the probe sequence contained the list of slots that were checked before finding an empty one.

For example:

- **Key** `25` was placed directly in **slot 0**, so its probe sequence is `[]` (no probing needed).
- **Key** `5` initially mapped to **slot 4**, but since it was occupied, I probed **slot 8**, making the probe sequence `[8]`.

After inserting all the given keys, I displayed the results, including **the final hash table, home slots, number of collisions, and probe sequences**.

7. (text) Algorithm Analysis [5 points] For each of the algorithms you wrote for problems 4-6, explain their time complexity and space complexity using Big-O notation. Explain how you arrived at your answer.

Solution:

Problem 4: Double Hashing with Rehashing

Time Complexity

- **Insertion:**
  - **Best case:** O(1) if no collisions occur (direct insertion).
  - **Worst case:** O(M) where MM is the table size (probes required due to collisions).
  - **Average case:** O(1) assuming a good hash function and low load factor.
- **Rehashing:**
  - When resizing occurs, **all elements are reinserted**, making it **O(N)** for N elements.
  - If rehashing happens logarithmically (e.g., doubling the table), the **amortized complexity remains O(1)**per insertion.

Space Complexity

- The hash table itself takes **O(M)** space.
- The mappings for hashing use **O(N)** additional space.
- **Total space complexity: O(M)**, where M is the table size.

Problem 5: Radix Sort for Strings

Time Complexity

- Radix Sort operates in **O(kN)** where:
  - NN is the number of words.
  - kk is the **maximum word length**.
  - Each pass sorts the words in **O(N)** using **counting sort**.
  - Since we perform kk passes, the final complexity is **O(kN)**.

Space Complexity

- Uses **buckets** for sorting (usually 256 for all ASCII characters).

- Stores input words separately.
- **Total space complexity: O(N+k)**.

Problem 6: Word Pattern Matching

Time Complexity

- Splitting the string takes **O(N)** where N is the length of `s`.
- Iterating over the pattern takes **O(P)** where P is the length of `pattern`.
- HashMap operations (insertion and lookup) are **O(1)** in the average case.
- **Total complexity: O(N+P)**.

Space Complexity

- Stores two HashMaps (pattern-to-word and word-to-pattern) → **O(P)**.
- Stores split words → **O(N)**.
- **Total space complexity: O(N+P)**.

Problem	Time Complexity	Space Complexity
Double Hashing	$O(1)$ (avg), $O(M)$ (worst)	$O(M)$
Radix Sort	$O(kN)$	$O(N+k)$
Word Pattern	$O(N+P)$	$O(N+P)$