CS302 – Analysis and Design of Algorithms

Algorithm Analysis

Content
Recurrences
The Master Theorem
Exercises

Recurrences

- A recurrence function is a function that calls itself in a sequence on other smaller argument.

- Most of the time they are found with divide-and-conquer algorithms.

- Recursive functions have two or more cases:
  - **Recursive case**– invoking the function itself on smaller inputs.
  - **Base case** –it’s the last invocation of the function.

Recurrences

- A recurrence $T(n)$ is algorithmic if, for every sufficiently large threshold constant $n_0 > 0$, the following two properties hold:

1. For all $n < n_0$, we have $T(n) = \Theta(1)$.
2. For all $n \geq n_0$, a recursive invocation.

Recurrences

Four methods to solve recurrences:

1. **Substitution method**: guess the form of a bound and then prove your guess correct and solve for constants.
   - Roust method but requires you to make a good guess and to prove it.

2. **Recursion tree**: models the recurrence as a tree whose nodes represent the costs incurred at various levels of the recursion.

Recurrences

Four methods to solve recurrences:

3. **Master method**: solve recurrences of the form

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

- Where $a > 0$ and $b > 1$ are constants and $f(n)$ is a driving function.
- It means: the algorithm divides a problem of size n into a subproblems, each of size $\mathrm{n}/b < n$
- $f(n)$ is the divide and combine time.

Recurrences

Four methods to solve recurrences:

4. Akra-Bazzi method: a general method for solving divide-and-conquer recurrences.

- Involves calculus.
- Used to solve more complicated recurrences.

Content
Recurrences
The Master Theorem
Exercises

The Master Method

- Three cases to solve $T(n) = aT(n/b) + f(n)$

1. If there exist a constant $\epsilon > 0$ | $f(n) = O\left(n^{\log_b a - \epsilon}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If there exist a constant $\mathrm{k} \geq 0$ | $f(n) = \Theta\left(n^{\log_b a} \lg^k n\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$
3. If there exist a constant $\epsilon > 0$ | $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ and $f(n)$ satisfies the *regularity condition* $a(f(n/b)) \leq cf(n)$ for some constant $c < 1$, then $T(n) = \Theta\left(f(n)\right)$

The Master Method

- The $f(n)$ is called the driving function.
- The $n^{\log_b a}$ is called the watershed function.

The Master Method

- Case 1 applies when $n^{\log_b a}$ grows **<u>polynomially</u>** faster than $f(n)$.
  - i.e., $n^{\log_b a}$ is greater than $f(n)$ by n^{ϵ}

- Case 2 applies when the watershed and driving functions grow at nearly the same asymptotic rate.
  - We assume $f(n)$ grows faster than $n^{\log_b a}$ by <u>a factor of $\Theta(\lg^k n)$</u>

- Case 3 applies when $f(n)$ grows **<u>polynomially</u>** faster than $n^{\log_b a}$ by at least a factor of n^{ϵ}

Content
Recurrences
The Master Theorem
Exercises

Exercises

- Solve $T(n) = T(n/2) + n$ using Master Theorem

Exercises

- Solve $T(n) = T(n/2) + n$ using Master Theorem

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

1. Check $n^{\log_b a}$ → $n^{\log_2 1} = n^0 = 1$
2. Compare $f(n)$ with $\mathrm{n}^{\log_b a}$: $f(n) = n$ and $n^{\log_b a}$ → $f(n)$ grows faster than $n^{\log_b a}$ → case 3 applies → check regularity condition.
3. Regularity condition: check that $af\left(\frac{n}{b}\right) \leq cf(n)$ for $c < 1$ → $\frac{n}{2} \leq c\, n$ → we can set $c = 1/2$ → $\frac{n}{2} \leq \frac{n}{2}$
4. $\therefore T(n) = \Theta(n)$

Exercises

- Solve $T(n) = 2T(n/2) + n^2$ using Master Theorem

Exercises

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

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = n^2$ is polynomially greater than n

Case 3 applies → check regularity condition.

Check $a(f(n/b)) \leq cf(n)$ → $\left[2\left(\left(\frac{n}{2}\right)^2\right) = 2\frac{n^2}{4} = \frac{n^2}{2}\right] \leq cn^2$

The regularity condition holds for $c = 1/2$

$\therefore T(n) = f(n) = \Theta(n^2)$

Exercises

- Solve $T(n) = 9T(n/3) + n$ using Master Theorem

Exercises

$a = 9, b = 3, f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = n^2$

$\because\ f(n) = n = n^{2-\epsilon} \quad \rightarrow \quad n^2 > n$

$\therefore$ case 1 applies

$$\therefore T(n) = \Theta(n^2)$$

Exercises

- Solve $T(n) = T(2n/3) + 1$ using Master Theorem

Exercises

$a = 1, b = 3/2, f(n) = 1$

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

$\because n^{\log_b a} = f(n) = 1$

$\therefore$ case 2 applies.

$\because f(n) = 1 = \Theta\left(n^{\log_b a} \lg^{\mathrm{k}} n\right)$, and we have $n^{\log_b a} = 1$

$\therefore f(n) = 1 = \Theta(1 \times \lg^0 n)$, i.e., $k = 0$

$\therefore T(n) = \Theta\left(n^{\log_b a} \lg^{\mathrm{k}+1} n\right) = \Theta(\lg n)$

Exercises

- Solve $T(n) = 3T(n/4) + n \lg n$ using Master Theorem

Exercises

$a = 3, b = 4, f(n) = n \lg n$

$n^{\log_b a} = n^{\log_4 3} \approx n^{0.79}$

$\because f(n) = n \lg n = \Omega\left(n^{\log_4 3+\epsilon}\right)$, where $\epsilon \approx 0.2$

$\therefore$ case 3 applies

Check the regularity condition: $a\, f(n/b) \leq cf(n)$ for sufficiently large n and $c < 1$

$\therefore \left[a\, f\left(\frac{n}{b}\right) = 3 \times \frac{n}{4} \times \lg \frac{n}{4}\right] \leq \left[\frac{3}{4} \times n \times \lg n = c\, n \lg n\right]$ for $c = \frac{3}{4}$.

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

Exercises

- Solve $T(n) = 2T(n/2) + n \lg n$ using Master Theorem

Exercises

$a = 2, b = 2, f(n) = n \lg n$

$n^{\log_b a} = n^{\log_2 2} = n$

$\because\ f(n) = n \lg n$, which is <u>not polynomially</u> greater than n, but greater than by the factor $\lg n$

$\therefore$ case 2 applies.

$\because\ f(n) = n \lg n = n^{\log_b a} \lg^k n = n^1 \lg^1 n = n \lg n$

$T(n) = \Theta\left(n^{\log_\mathrm{b} a} \lg^{\mathrm{k}+1}\mathrm{n}\right) = \Theta(n \lg^2 n)$

Exercises

- Solve $T(n) = 2T(n/2) + \Theta(n)$ using Master Theorem

Exercises

$a = 2, b = 2, f(n) = \Theta(n)$

$n^{\log_b a} = n^{\log_2 2} = n$

$\because f(n) = \Theta(n) = n = f(n)$

$\therefore$ case 2 applies

$\because f(n) = n = n^{\log_b a} \lg^k n = n^{\log_2 2} \lg^0 n = n$

$\therefore T(n) = n^{\log_b a} \lg^{k+1} n = n \lg n$

Exercises

- Solve $T(n) = 8T(n/2) + \Theta(1)$ using Master Theorem

Exercises

$a = 8, \mathrm{b} = 2, \mathrm{f(n)} = \Theta(1)$

$n^{\log_b a} = n^{\log_2 8} = n^3$

$\because$ n^3 is polynomially greater than $\Theta(1)$, where $\epsilon = 3$

$\therefore$ case 1 applies

$\therefore$ $T(n) = \Theta(n^3)$

Exercises

- Solve $T(n) = 7T(n/2) + \Theta(n^2)$ using Master Theorem

Exercises

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

$n^{\log_b a} = n^{\log_2 7} = n^{2.8}$

$\because$ $n^{2.8}$ is polynomially greater than $\Theta(n^2)$, where $\epsilon = 0.8$

$\therefore$ case 1 applies

$\therefore$ $T(n) = \Theta\left(n^{\lg 7}\right)$

Exercises

- Solve $T(n) = 2T(n/4) + 1$ using Master Theorem

Exercises

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

$n^{\log_b a} = n^{\log_4 2} = n^{\frac{1}{2}} = \sqrt{n}$

$\because \sqrt{n}$ is polynomially greater than 1, where $\epsilon = 0.5$

$\therefore$ case 1 applies

$\therefore T(n) = \sqrt{n}$

Exercises

- Solve $T(n) = 2T(n/4) + \sqrt{n}\lg^2 n$ using Master Theorem

Exercises

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

$n^{\log_b a} = n^{\log_4 2} = n^{\frac{1}{2}} = \sqrt{n}$

$\because f(n) = \sqrt{n} \lg^2 n$ which is larger than $\sqrt{n}$ by the factor $\lg^2 n$

$\therefore$ case 2 applies

$T(n) = \Theta(\sqrt{n} \lg^3 n)$

Exercises

- Solve $T(n) = 2T(n/4) + n$ using Master Theorem

Exercises

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

$n^{\log_b a} = n^{\log_4 2} = n^{\frac{1}{2}} = \sqrt{n}$

$\because f(n) = n$ is polynomially greater than $\sqrt{n}$, where $\epsilon = 0.5$.

$\therefore$ case 3 applies

Check regularity condition: $a\, f(n/b) \leq cf(n) \rightarrow \frac{2n}{4} \leq cn$ for $c = \frac{1}{2}$

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

Exercises

- Give a recursive definition of $L(w)$, the length of the string w?

Exercises

Let's start with an example:

$$L(abcde) = 1 + L(abcd) = 2 + L(abc) = 3 + L(ab) =$$
$$4 + L(a) = 5 + L(empty\ string) = 5 + 0 = 5$$

$$L(\phi) = 0$$
$$L(wx) = L(w) + 1$$

Exercises

- Solve the recurrence relation using iterative method: $T(n) = 7T(n/2) + n^2$ where $(T(1) = 1)$

Exercises

First expansion: $T(n) = 7T\left(\frac{n}{2}\right) + n^2$

Second expansion: substitute for $T(n/2)$:

$$T\left(\frac{n}{2}\right) = 7T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2 = 7T\,(n/4) + \frac{n^2}{4}$$

Then,

$$T(n) = 7\left(7T\left(\frac{n}{4}\right) + \frac{n^2}{4}\right) + n^2 = 49T\left(\frac{n}{4}\right) + 7 \cdot \frac{n^2}{4} + n^2$$

$$= 49T\left(\frac{n}{4}\right) + \frac{11n^2}{4}$$

Exercises

- Third expansion: substitute for $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right) = 7T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2 = 7T\left(\frac{n}{8}\right) + \frac{n^2}{16}$
- Thus,

$$T(n) = 49\left(7T\left(\frac{n}{8}\right) + \frac{n^2}{16}\right) + \frac{11n^2}{4} = 343T\left(\frac{n}{8}\right) + \frac{49n^2}{16} + \frac{11n^2}{4}$$

- Convert $11n^2/4$ to sixteenths: $\frac{11n^2}{4} = \frac{44n^2}{16}$
- Then,

$$T(n) = 343T\left(\frac{n}{8}\right) + \left(\frac{49n^2}{16} + \frac{44n^2}{16}\right) = 343T\left(\frac{n}{8}\right) + \frac{93n^2}{16}$$

Exercises

- General pattern: after k expansion, we have $$T(n) = 7^k T\left(\frac{n}{2^k}\right) + P_k(n)$$
- The base case is when $n = 1$ ($T(1) = 1$), there is no more expansions. Thus,
$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$$
- Then,
$$T(n) = 7^{\log_2 n} T(1) + P_k(n) = n^{\log_2 7} + P_k(n)$$
- Since $P_k(n) = O(n^2)$
$$T(n) = n^{\log_2 7} + O(n^2)$$
$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$$

Exercises

- Describe $O(n \log n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. (For example: if S={3, 5, 4, 2, 6, 7, 9, 12, 18} and when input x=5 then the output (2, 3))

Exercises

- Describe $O(n \log n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. (For example: if S={3, 5, 4, 2, 6, 7, 9, 12, 18} and when input x=5 then the output (2, 3))

Tip:

Every time we see $\lg n$, we should think of divide-and-conquer algorithms. It inherently means how many times n can be divided by 2, i.e. repeated division of n elements in two groups.

Exercises

So, basically, it's a search problem. We search for two elements, $a \in S$ and $b \in S$, such that $a + b = x$.

We can use binary search algorithm, which is $\Theta(\lg n)$. But it requires a sorted array.

Thus, we can use merge sort to sort the array. It takes $\Theta(n \lg n)$.

$\because T(n) = \Theta(n \lg n) + \Theta(\lg n)$

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

Exercises

Steps:

1. Read the list S and the number x
2. Sort S using merge sort.
3. For every element in S, compute $b = x - S[i]$
4. Binary search for the element b in S.
5. If an element is found, then return $(S[i], b)$
6. Otherwise, repeat steps 3 and 4.

Exercises

```
SUM-SEARCH (S, x)
MERGE-SORT(S, 1, S.length)
for i = 1 to S.length
    index = BINARY-SEARCH(S, x − S[i])
    if index ≠ NIL and index ≠ i
        return true
return false
```

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

Exercises

This problem can be solved in another way which still uses a $\Theta(n \lg n)$ sorting algorithm but instead of using binary search, it uses a two-way search, i.e., simultaneous search from both end of the array, to check if two elements sums up to expected sum, x.

Exercises

```
SUM-SEARCH (S, x)
MERGE-SORT(S, 1, S.length)
left = 1
right = S.length
while (left < right)
    if S[left] + S[right] == x
        return true
    else if S[left + S[right] < x
        left = left + 1
    else
        right = right - 1
return false
```

Exercises

- Give a recursive function that calculates Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, …? Write an algorithm using your recursive relation.

Exercises

1. The sequence starts with 0 and 1.
2. The next number is computed by summing the preceding two numbers.
3. We continue until reach n.

For example, to compute Fibonacci(6):

1: 0, 1

2: 0, 1, 1

3: 0, 1, 1, 2

4: 0, 1, 1, 2, 3

5: 0, 1, 1, 2, 3, 5

6: 0, 1, 1, 2, 3, 5, 8

Exercises

So, in general, to compute Fib(n), we compute Fib(n-1) + Fib(n-2) until reaching the base case 0 and 1.

The recurrence relation is

$$fib(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ fib\,(n-1) + fib(n-2) & if\ n > 1 \end{cases}$$

Exercises

Algorithm 33: Recursive Fibonacci

Data: Integer n
Result: Fibonacci number $F(n)$
Function `Fibonacci(`n`)`**:**

```
    if n = 0 then
        return 0 ;
    end
    if n = 1 then
        return 1 ;
    end
    return Fibonacci(n − 1) + Fibonacci(n − 2) ;
```

Exercises

Time complexity (not part of the question):

$T(n) = T(n-1) + T(n-2) + c$
$= 2T(n-1) + c$ //from the approximation T(n-1) ~ T(n-2)
$= 2 * (2T(n-2) + c) + c$
$= 4T(n-2) + 3c$
$= 8T(n-3) + 7c$
$= 2^k * T(n - k) + (2^k - 1) * c$

Since $n - k = 0$ is a base case, then when we reach it, we get $k = n$

$T(n) = 2^n * T(0) + (2^n - 1) * c = O(2^n)$

Exercises

- Give a recursive function $f(n)$ that represents a^n where a is a non-zero real number and n is a nonnegative integer? Write an algorithm to compute a^n using your recursive relation.

Exercises

To compute a^n recursively, we can compute $a \times a^{n-1}$ until we reach $n = 1$.

The base case is $n - 1$, thus $a^0 = 1$.

The recursive definition is

$$f(n) = \begin{cases} 1 & if\ n = 0 \\ a \times f(n-1) & if\ n > 1 \end{cases}$$

Exercises

Algorithm 34: Power

```
Data: Non-zero real number a, non-negative integer n
Result: Compute a^n
Function Power(a, n):
    if n == 0 then
        return 1 ;
    end
    else
        return a× Power(a, n − 1) ;
    end
```

The time complexity of this recursive algorithm is $O(n)$ because it makes n recursive calls

Exercises

- Professor Caesar wants to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into $n/4 \times n/4$ submatrices, and the divide and combine steps together will take $\Theta(n^2)$ time. Suppose that the professor's algorithm creates a recursive subproblems of size $n/4$. What is the largest integer value of a for which his algorithm could possibly run asymptotically faster than Strassen's?

- Strassen's algorithm: $T(n) = 7T(n/2) + \Theta(n^2) = \Theta\left(n^{\lg 7}\right) = n^{2.81}$

Exercises

Strassen's algorithm: $T(n) = 7T(n/2) + \Theta(n^2) = \Theta\left(n^{\log_{\mathrm{b}} a}\right) = \Theta\left(n^{\lg 7}\right) = n^{2.81}$

Caesar's algorithm: $T(n) = a\, T(n/4) + \Theta(n^2)$

The professor's algorithm should be faster than Strassen's. Thus,

$$n^{\log_b a} = n^{\log_4 a} < n^{\lg 7}$$

Notice that the base of the log in Strassen's algorithm is 2, while it is 4 in Caesar's algorithm. Hence, we can assume that $a = 7\times 7 = 49$. Therefore,

Caesar's algorithm will be $n^{\log_b 49} = n^{2.81}$. Thus, the largest value for a to be faster than Strassen's algorithm should be $a = 48$.

Exercises

- Solve $T(n) = 2T(n/2) + n/\lg n$ using Master Theorem.

Exercises

MT doesn't apply here.

Given $a = 2$, $b = 2$, and $f(n) = n/\lg n$

$n^{\log_2 2} = n$, none of the three cases apply.

The first case cannot be used because, although n is asymptotically greater than $f(n) = n/\lg n$, n is not polynomially greater than $n/\lg n$.

The second case cannot be used since $f(n) = n/\lg n = \Theta(n \lg^k n)$ where $k = -1$, but k must be nonnegative for case 2 to apply.

Case 3 doesn't apply because it handles the case where $f(n)$ is larger than $n^{\log_b a}$

TASK

- Solve $T(n) = 2T(n/4) + \sqrt{n}$ using Master Theorem
- Solve $T(n) = 2T(n/4) + n^2$ using Master Theorem