CSE 431/531: Analysis of Algorithms (Summer 2023) Recursion and Master Theorem

Chen Xu

June 6, 2023

Recall from the last lecture

- I hope you all went through the proof the O-notation.
- You will see O-notation everywhere from now.

The most frequently asked questions

- Does your algorithm work?
- What is the running time of your algorithm?
- How much space does your algorithm need?

Today we introduce some methods to solve the asymptotic running time of an algorithm. (We can solve the space asymptotics analogously)

A trivial algorithm

Algorithms with no branches are trivial. It implies that every line will be executed exactly once.

Compute $3x + 2y + 1$

Input: x, y of 32-bit integers
Output: The value of $3x + 2y + 1$

```
1: COMPUTE(x, y)
2:     z ← 3x
3:     w ← 2y
4:     z ← z + w
5:     z ← z + 1
6:     return z
```

The instance of the above algorithm has a fixed input size of 32×2 bits.
So $T(n) \in O(1)$

General idea

- For any algorithm on some certain input of size n, suppose S is the collection of steps that are executed. Then the running time of these steps $T_S(n)$ is:

$$T_S(n) = \sum_{s \in S} T_s(n)$$

- Simply put, $T_S(n)$ is the sum of the running time of every step.
- In the previous example,
  $T_2(n) = T_3(n) = T_4(n) = T_5(n) = T_6(n) \in O(1)$, then
  $T(n) = T_{\{2,3,4,5,6\}}(n) \in O(1)$. In algorithm analysis, we usually ignore these parts with the constant $O(1)$ running time.

- If your algorithm has a part that is repeatedly executed, let's define that part of steps as U.
- Carefully examine the structure of the algorithm, $T_U(n)$ can sometimes be expressed in terms of other set of steps U', such that $T_U(n) = T_{U'}(n')$. This is where we get the recursion.
- Let's take a look at a simple example.

Sum of the array

Compute sum of an array

Input: An array of 32-bit integers A
Output: The sum of the array

```
1: ArraySum(A, n)
2:     sum ← 0
3:     for i = 1 to n do
4:         sum ← sum + A[i]
5:     return sum
```

In this algorithm, line 4 will be repeated. We know step 4,6,8,10,... all execute line 4. Each time the value of i increases. The new *sum* is computed based on the old sum. So we can infer that
$T_{\{4,6,8,10,\ldots\}}(n) = T_{\{6,8,10,12,\ldots\}}(n) + O(1) = T_{\{4,6,8,10,\ldots\}}(n-1) + O(1)$
Ignoring the other constant overhead, we have
$T(n) = T(n-1) + O(1) \in O(n)$

Merge two sorted arrays

Merge two sorted arrays

Input: Sorted Arrays L, R

Output: Merged sorted array B

```
1: MERGE(L, R)
2:     B ← empty list
3:     while L ≠ empty and R ≠ empty do
4:         if first(L) ≤ first(R) then
5:             append first(L) to B and remove first from L
6:         else
7:             append first(R) to B and remove first from R
8:     return B concatenated with the remaining of L and R
```

Merge two sorted arrays

Let us look at line 4 to 7:

if first(L) $\leq$ first(R) **then**
    append first(L) to B and remove first from L
else
    append first(R) to B and remove first from R

- From the code, we know that the smaller one will be removed and appended to B.
- The total number of this comparison will be executed $min(|L|, |R|)$ number of times.
- The remaining part of L or R will then be appended to the end of B so the total number of calls of append is $|L| + |R|$ times.
- Thus the running time is $O(|L| + |R|)$.

The Merge Sort Algorithm

Let us utilize the **MERGE(L,R)** function to solve the sorting problem.

MergeSort

```
1: MergeSort(A)
2:     if length(A) < 2 then
3:         return A
4:     else
5:         m ← length(A)/2
6:         L ← MergeSort(A[1, ..., m])
7:         R ← MergeSort(A[m + 1, ..., n])
8:         return Merge(L, R)
```

What is the recursion of merge-sort?

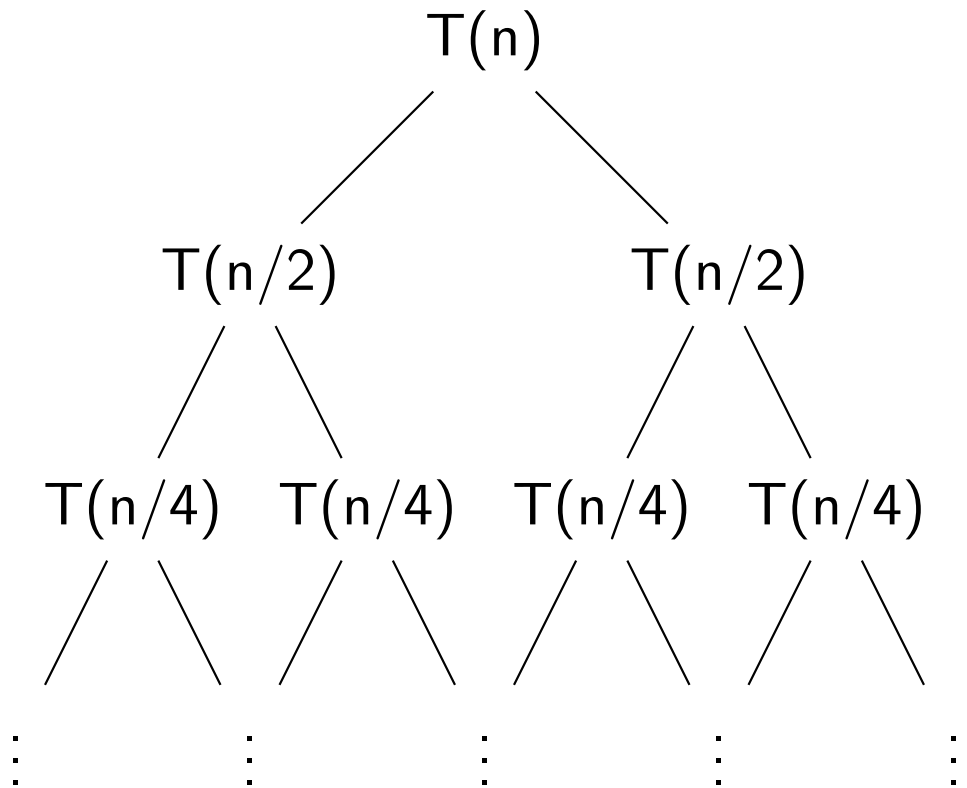
$L \leftarrow \text{MergeSort}(A[1, \ldots, m])$
$R \leftarrow \text{MergeSort}(A[m+1, \ldots, n])$

Line 6 and line 7 contains two calls to the **mergesort** function itself, with the array cut into two halves.

return MERGE(L, R)

Line 8 calls the **merge** function once. The running time for this function is $O(|L| + |R|) \leq O(n)$.
Ignoring all the constant overhead, the running time has the form of the recursion $T(n) = 2T(n/2) + O(n)$

Recursion call visualized



- The depth of the tree is $O(\log n)$ since the number of times of taking a half until 1 from n is $log_2 n$.
- To help us solve such kind of recursion, we have a good tool called master theorem.

Master Theorem

Definition

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

Applications of the Master Theorem

Example 1

$T(n) = 2T(n/2) + \Theta(1)$
$a = 2, b = 2, f(n) = \Theta(1) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$.
$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

Example 2

$T(n) = 4T(n/4) + \Theta(n)$
$a = 4, b = 4, f(n) = \Theta(n) = \Theta(n^{\log_b a})$.
$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.

Example 3

$T(n) = 2T(n/2) + \Theta(n^2)$
$a = 2, b = 2, f(n) = \Theta(n^2) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$. And
$af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n.
$T(n) = \Theta(f(n)) = \Theta(n^2)$.

Example to check case 3 condition

Example 4

$T(n) = aT(n/b) + f(n), f(n) = 2^n$

- We would like to verify if the condition $a2^{n/b} \leq c2^n$ holds for some constant $c < 1$ and sufficiently large n.
- Take $c = 0.1, n_0 = \frac{-5-\log a}{1/b-1}$, for all $n \geq n_0$ we have

$$\begin{aligned} a2^{n/b} - c2^n &= 2^n(2^{\log a + n/b - n} - 0.1) \\ &\leq 2^n(2^{\log a + n_0/b - n_0} - 0.1) \\ &= 2^n(2^{-5} - 0.1) \\ &< 0 \end{aligned}$$

- Therefore the condition holds, we apply case 3, $T(n) = \Theta(2^n)$

The running time of the mergesort

Let us apply the master theorem to resolve the recursion we had.

$$T(n) = 2T(n/2) + O(n)$$

- $a = b = 2$, applying case 2 we have $T(n) = O(n \log n)$
- "Usually" this $O(n \log n)$ upper bound is optimal for a lot of algorithms.

Detecting Duplicates

Detect Duplicates

Input: array A

Output: True/False if A has duplicates

```
1: ContainsDuplicates(A)
2:     for i = 1 to length(A) do
3:         for j = i + 1 to length(A) do
4:             if A[i] = A[j] then
5:                 return True
6:     return False
```

- "Usually" the depth d of the for loop determines the degree of the running time $O(n^d)$.
- In this algorithm, we have a depth 2 for-loop.
- The worst-case time complexity is $O(n^2)$. Can you do it in $O(n)$?

Standard Matrix Multiplication

Matrix Multiplication

Input: $A, B \in \mathbb{R}^{n,n}$

Output: $C = A \times B$

```
1: MatrixMultiply(A, B, n)
2:     C = [][]
3:     for i = 1 to n do
4:         for j = 1 to n do
5:             for k = 1 to n do
6:                 C[i][j] = C[i][j] + A[i][k] · B[k][j]
7:     return C
```

- It has the depth 3 for-loop. The worst-case time complexity is $O(n^3)$.
- Again, this algorithm is not optimal, the current best time is $O(n^{2.37188})$

Problem: Subset Sum

Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to the given sum.

Subset Sum Problem

Input: S, n, sum

Output: True/False if there exists a subset of S that sums up to sum.

```
1: IsSubsetSum(S, n, sum)
2:     if sum = 0 then
3:         return True
4:     if n = 0 and sum ≠ 0 then
5:         return False
6:     if S[n − 1] > sum then
7:         return IsSubsetSum(S, n − 1, sum)
8:     return IsSubsetSum(S, n − 1, sum) or
   IsSubsetSum(S, n − 1, sum − S[n − 1])
```

- The recursion function for this problem is $T(n) = 2T(n-1) + O(1)$. Solving this we get $T(n) = O(2^n)$.
- Exercise: Prove that if the input is sorted we still have running time $O(2^n)$.

Traveling Salesman Problem (TSP)

Brute-Force TSP(non-recursive)

Input: Graph $G = (V, E, W)$

Output: A sequence of vertices (Path) such that it has the minimum sum of weight.

```
1: BruteForceTSP(V, E, W)
2:     Generate all permutations of V as V'
3:     minPath = INF
4:     for each permutation V' do
5:         currWeight = 0
6:         for i = 0 to |V'| − 2 do
7:             currWeight = currWeight + W(E(V'[i], V'[i + 1]))
8:         minPath = min(minPath, currWeight)
9:     return minPath
```

- The running time mainly depends on how many permutations we have in line 4, and it is a very scary $O(n!)$.
- The running time for the above algorithm is $O(n!)$
- Exercise: Can you write the recursive version of the above algorithm. Prove that it still has the complexity of $O(n!)$.

Recursion that cannot be easily analyzed

We have to mention that there are algorithms where we don't know if it always terminates although the program looks extremely simple.
Here is one example:

Collatz Conjecture

Input: a positive integer n
Output: True

```
1: COLLATZ(n)
2:     if n = 1 then return True
3:     if n mod 2 = 0 then
4:         COLLATZ(n/2)
5:     else
6:         COLLATZ(3n + 1)
```