ICCS313 - Algorithm Analysis, T1 2019/20

Lecture 3: Recurrences & Divide and Conquer

29 September 2019

Lecturer: Chaya Hiruncharoenvate *Scribe: -*

```
- Quiz 1 is next Saturday
- Quiz first, then continue with Week 4 materials
- You can have 1 two-sided A4 cheat sheet
```

1 Recurrences

One approach to solve a problem is to write an algorithm containing a recursive call to itself. The running time of such algorithm can be described by a recurrence.

1.1 Definition

A *recurrence* is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, the running time $T(n)$ of the merge sort algorithm can be described by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Note here that we neglect certain technical details when we state and solve recurrences. One example here is the assumption of integer arguments to functions. Normally, $T(n)$ is only defined when n is an integer, because for most algorithms, the size of the input is always an integer. In the case of merge sort, the running time is actually

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{if } n > 1 \end{cases}$$

When we state and solve recurrences, we often omit floors and ceilings and later determine whether or not they matter. They usually don't, but it is important to know when they do.

With that said, we will look at 2 methods for solving recurrences: iteration method and master method.

1.2 Iteration method

The idea behind the iteration method is to expand (iterate) the recurrence and express it as a summation of terms dependent only on n and the initial condition.

Example: Consider the recurrence

$$T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$$

We can iterate it as follows:

$$
\begin{aligned}
T(n) &= n + 3T(\lfloor \frac{n}{4} \rfloor) \\
&= n + 3\left[\lfloor \frac{n}{4} \rfloor + 3T(\lfloor \frac{n}{16} \rfloor)\right] \\
&= n + 3\left[\lfloor \frac{n}{4} \rfloor + 3\left\{\lfloor \frac{n}{16} \rfloor + 3T(\lfloor \frac{n}{64} \rfloor)\right\}\right] \\
&= n + 3\lfloor \frac{n}{4} \rfloor + 9\lfloor \frac{n}{16} \rfloor + 27T(\lfloor \frac{n}{64} \rfloor)
\end{aligned}
$$

How far must we go? Until we discover that the summation contains a decreasing geometric series:

$$
\begin{aligned}
T(n) &\leq n + \frac{3}{4}n + \frac{9}{16}n + \frac{27}{64}n + \ldots \\
&= n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i \\
&= 3n \in O(n)
\end{aligned}
$$

1.3 Master method

Theorem 1.1 (Master Theorem)**.** *If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$ and $d \geq 0$, then*

$$
T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}
$$

Proof. To prove this, lets start by assuming for the sake of convenience that n is a power of b. This will make things easier for us and it won't really effect the final bound in any important way – after all, n is at most a multiplicative factor of b away from some power of b. Doing so allows us to ignore the rounding effect in $\lceil n/b \rceil$.

Next, notice that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. This is the height of the recursion tree.

Its branching factor is a, so the k-th level of the tree is made up of a^k subproblem, each of size n/b^k.

[DRAW THE RECURSION TREE]

So the total work at level k is given by:

$$
a^k \times O((\frac{n}{b^k})^d) = O(n^d) \times (\frac{a}{b^d})^k
$$

As k goes from 0 to $\log_b n$, the overall work done can be writen as:

$$
\sum_{k=0}^{\log_b n} O(n^d) \times (\frac{a}{b^d})^k
$$

When consider following cases:

- When $a < b^d$, we have a decreasing geometric series. The sum can be bounded by its first term: $O(n^d)$.

- When $a > b^d$, we have a increasing geometric series. The sum can be bounded by its last term: $O(n^{\log_b a})$.

$$n^d(\frac{a}{b^d})^{\log_b n} = n^d(\frac{a^{\log_b n}}{(b^{\log_b n})^d}) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{log_b a}$$

- When $a = b^d$, all $O(\log n)$ terms are equal to $O(n^d)$.

□

Example: Power
Assume that $*$ is $O(1)$. Consider the following pseudocode.

```
1 long power(long x, long n) {
2   if (n == 0)
3     return 1;
4   else
5     return x * power(x, n-1);
6 }
```

Question: What is the running time?

$$T(n) = \begin{cases} T(n-1) + c_1 & \text{if n > 0} \\ c_2 & \text{otherwise} \end{cases}$$

Take-away: Master theorem helps only when the recurrence relation matches with the pattern.
Now consider the alternative version:

```
1 long power(long x, long n) {
2   if (n==0) return 1;
3   if (n==1) return x;
4   if ((n % 2) == 0)
5     return power(x*x, n/2);
6   else
7     return power(x*x, n/2) * x;
8 }
```

Question: What is the running time?

$$T(n) = \begin{cases} T(n/2) + c_1 & \text{if n > 1} \\ c_2 & \text{if n = 1} \\ c_3 & \text{if n = 0} \end{cases}$$

2 Divide and Conquer

The divide-and-conquer strategy solves a problem by:

1. Divide instance of problem into two or more smaller instances
2. Recursively solving these instances
3. Obtain solution to original (larger) instance by combining these solutions

[DRAW A PICTURE OF A RECURSION TREE]
Normally the real work of a divide conquer algorithm happens at three different places:

1. when we partition the problem into smaller subproblems.
2. when the problems are small enough that we can solve them easily.
3. when we combine answers together.

3 Mergesort

Let's take a look at an iconic divide-and-conquer algorithm, Mergesort. Mergesort is an algorithm for sorting a list of numbers that works by spilting the list into two halves, recursively sort each half, and then merge the two sorted sublists.

Here's a pseudocode of Mergesort.

```
function mergesort(a[0..n-1]):
1    if n > 1:
2      b = a[0..(n/2)-1]
3      c = a[(n/2)..n-1]
4      mergesort(b)
5      mergesort(c)
6      merge(b,c,a)

function merge(b[0..p-1],c[0..q-1],a[0..p+q-1]):
1    // merge two sorted list: b,c into a
2    // idea: compare smaller elemment of b and c and stick it to a
3    // when one of b or c runs out, copy the rest to a
4    // details to be filled by students
```

Question: What is the time complexity of `merge()`?

Example: Perform a mergesort of $6, 4, 1, 10, 5, 2, 4, 8$ on the board.

3.1 Analysis of Mergesort

The correctness of this algorithm is self-evident, as long as a correct merge subroutine is specified. As for the running time, let $T(n)$ be the running time of mergesort on an array of n elements.

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

This recurrence relation solves to $T(n) = O(n \log n)$. We will take a look at how to solve this in a minute.

3.2 Solving recurrence relations using Master Theorem

Using Master Theorem to solve for the running time of our merge sort algorithm, we have $a = 2$, $b = 2$, and $d = 1$, or

$$T(n) = 2 * T(n/2) + O(n^1)$$

Question: Does the time complexity using Master Theorem match with what we calculated earlier?

4 Counting Inversion

Given a list of integers L of size n. We say that two elements $L[i]$ and $L[j]$ form an inversion if $L[i] > L[j]$ and $i < j$. So if array is already sorted then inversion count is 0 and, on the other hand, if array is sorted in reverse order then the inversion count is at its maximum.

Our goal here is to design an algorithm to count the number of inversions given an array.

Idea 1: Naive solution Consider all possible pairs and just count. What is the running time for this one? $O(n^2)$

Can we do better? It turns out that we can!

Idea 2: Divide-and-Conquer We can solve this problem using D&Q strategy.

- Divide: separate list into 2 halves A and B
- Conquer: count inversions in each list
- Combine step: count inversion(a,b) with a $\in$ A and b $\in$ B. Return sum of the counts

The combine step is tricky, because if done wrong, you will end up having:

$$T(n) = 2T(\frac{n}{2}) + O(n^2) \in O(n^2)$$

Let's take a closer look at the combine step. Suppose A and B are sorted.

$$A = 3, 7, 10, 18$$
$$B = 2, 11, 20, 23$$

We can efficiently count the number of inversions while merging A and B. During the merge, when we consider a_i and b_j, there are two possibilites:

1. If $a_i < b_j$ then a_i is not inverted with any element left in B
2. If $a_i > b_j$ then b_j is inverted with every element left in A

So if we sum up these number of inversions throughout the merging, we will have the total number of inversion between A and B.

Here's the pseudocode for the algorithm.

```
function Sort-and-Count(L)
1    If L has one element:
2        return (0,L)
3    else:
4        // Split L into A,B
5        (r_a, A) <-- Sort-and-Count(A)
6        (r_b, B) <-- Sort-and-Count(B)
7        (r_ab, L') <-- Merge-and-Count(A,B)
8        return (r_a + r_b + r_ab, L')
```

Question: What is the running time of `Sort-and-Count`?

5 Closest Pair of Points

Given n points in a 2-D plane. Fine a pair of points (u, v) with the smallest Euclidean distance between them.

[DRAW points on a plane and show the closest pair]

Definition 5.1. The Euclidean distance between point u and point v in 2D is given by

$$||u - v|| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Idea 1: Naive solution Consider all possible pairs.

```
for u in P
    for v in P\{u}
    x = EuclideanDistance(u,v)
    A.add(x,u,v)
return min(A)
```

Question: What is the running time?
As you may have guessed, we want to do better. Let's try using our beloved D&Q strategy.

1. Divide: Let's sort points by their x-coordiate. Draw a line L so that $n/2$ points are on each side.
2. Conquer: find the closest pair in each side recursively.
3. Combine: find the closest pair with one point on each side. Return the minimum of 3 candidates.

Similar to the previous problem, if the combine is done inefficiently, the whole algorithm will be $O(n^2)$.

So, now the question becomes, "how do we efficiently find the closest pair with one point in each side?"

Suppose we found that the distance between the closest pair left of line L and right of line L is l and r respectively. Let $\delta = \min(l, r)$.

Claim 5.2. *We only need to consider point within δ of line L.*

Proof. Left as class discussion □

Claim 5.3. *If we sort the points in this 2δ-band by their y-coordinates, the closest pair, if exists, will appear within 11 positions in the sorted list.*

Proof. Let s_i be the point in the 2δ-band, with the i-th smallest y-coordinate. No two points lie in the same $\delta/2$-by-$\delta/2$ box because that would contradict the fact that δ is the smallest distance on either side. Moreover, two points that are two rows apart have distance $\geq 2(\delta/2)$. So there are maximum of 11 positions to check. □

Note: The fact remains true if we replace 12 by 7.
Here's the pseudocode for `Closest-Pair(p_1,p_2,..,p_n)`

- Compute separation line L such that half the points are on each side. $\rightarrow O(n \log n)$
- $l \leftarrow$ Closest Pair in the left half $\rightarrow O(\frac{n}{2})$
- $r \leftarrow$ Closest Pair in the right half $\rightarrow O(\frac{n}{2})$
- $\delta \leftarrow \min(l, r)$
- Delete all points further than δ from L $\rightarrow O(n)$
- Sort points in y-order $\rightarrow O(n \log n)$
- Scan points in y-order and compute distance between each point and next 11 neighbors, and update δ accordingly $\rightarrow O(11n) = O(n)$
- return δ

So, the total running time is:

$$T(n) = 2T(\lceil n/2 \rceil) + O(n \log n)$$

which is $O(n(\log n)^2)$

This looks good but we can improve it to $O(n \log n)$ by *not* sort the points in the strip from scratch each time. In each recursive call, return TWO lists, points sorted in X and point sorted in Y. You can then sort by merging the two pre-sort lists.

The running time becomes:

$$T(n) = 2T(\lceil n/2 \rceil) + O(n)$$

which is $O(n \log n)$