COMPSCI 330: Design and Analysis of Algorithms **Jan 15, 2019**

Lecture 2: Divide and Conquer

Lecturer: Rong Ge *Scribe: Haoming Li*

2.1 Overview

There are three basic algorithm design techniques – divide-and-conquer, dynamic programming and greedy algorithms. With divide-and-conquer, we break the problem into several unrelated sub-problems, then combine the solutions to solve the original problem. A general framework of divide-and-conquer algorithms is as follows:

Algorithm 1 Divide-and-Conquer

```
if the instance is small (base case) then
  Solve the base case.
end if
Partition the problem into smaller sub-problems.
Call Divide-and-Conquer recursively to solve the sub-problems
Call a Merge procedure to combine the results of the sub-problems.
```

Below are example applications of divide-and-conquer algorithms.

2.2 Merge Sort

We will first apply divide and conquer to the *sorting* problem: given an array $a[1...n]$ of n numbers, sort the numbers in ascending order.

Base Case: When an array contains only one number, it is considered sorted.

Partitioning: Break $a[1...n]$ into two arrays $b[], c[]$, where $b[]$ contains the first half of the entries ($a[1...n/2]$) and $c[]$ contains the second half of the entries ($a[n/2+1...n]$).

Recursive Calls: Recursively call the algorithm to sort each half of the array.

Merging: After the recursive call, the sub-problems $b[]$ and $c[]$ are both sorted. We now merge the two sorted arrays into a single sorted array. This can be done efficiently by traversing each array once.

We have now finished designing the MergeSort algorithm. Below is the pseudo-code.

Algorithm 2 MergeSort(a[])

```
if length(a) < 2 then {Base case}
  return a[]
end if
Partition a[] evenly to two arrays b[], c[]. {Divide}
b[] = MergeSort(b[])
c[] = MergeSort(c[]) {Recursive Calls}
return Merge(b, c) {Merge}
```

Algorithm 3 Merge(b[], c[])

```
Allocate an empty array a[]
i = 1, j = 1
while (i ≤ length(b) or (j ≤ length(c)) do
  if b[i]< c[j] then
    Append b[i] to a[], i = i + 1
  else
    Append c[j] to a[], j = j + 1
  end if
end while
return a[]
```

Note that in the Merge pseudo-code we didn't handle the case when one of the list is already empty and the other list is not ($i > length(b)$ or $j > length(c)$). In that case the remaining numbers should just be added to the end of the array.

2.3 Running Time of Merge Sort

Next we will analyze the running time of MergeSort. Notice that the work of the algorithm can be put into two categories: 1. the recursive cost, which is the time it takes to sort $b[]$ and $c[]$ recursively; 2. the merge cost, which includes the cost of partitioning the array, and merging the result.

The merge cost is easy to analyze, as the merge algorithm makes one pass on both b and c. We can say the running time of the merge algorithm is bounded by $A \cdot n$ where A is a constant (the constant depends on the loop, maintaining i, j and cost of comparison).

For the recursive cost, we need to handle it by a recurrence relation. Let $T(n)$ be the running time for the MergeSort algorithm on n numbers, then the recursive cost is just $T(n/2)$ (in this course we will not talk about rounding errors, it is safe to assume $n/2$ is also an integer). Therefore, the running time of the algorithm can be bounded by

$$T(n) = 2T(n/2) + An.$$

We call this formula a *recurrence relation* for the running time of MergeSort algorithm. To analyze the running time, we need to solve the recurrence relation to get $T(n) = O(f(n))$ where $f(n)$ is one of the familiar functions like n^2. There are several ways to do this.

2.3.1 Guess and Prove

The first method we will talk about is Guess and Prove. This method might look a bit mysterious (especially the "guess" step). However this is a way to prove the running time rigorously and can often get the tightest bound.

We first guess the running time of the algorithm. For MergeSort we will guess that $T(n) \le An\log_2 n$. This guess can be obtained by evaluating the recurrence relation for small n, or by some other intuitive algorithm such as the Recursion Tree method (discussed later).

We will prove $T(n) \leq An\log_2 n$ by induction. For the recurrence relation of algorithms, we are usually free to assume whatever base cases as they will not change the running time of the algorithm by more than a constant factor. In this case we assume $T(1) = 0$.

Proof: We first check the base case: $T(1) = 0 \leq A \cdot 1 \cdot \log_2 1$.

Next, we perform induction. Assume $T(x) \leq Ax\log_2 x$ is true for all $x < n$, we are going to prove $T(n) \leq An\log_2 n$.

To do that, we use the recurrence relation:

$$\begin{aligned}
T(n) &= 2T(n/2) + An && \text{Recurrence relation} \\
&\leq 2 \cdot A(n/2)\log_2(n/2) + An && \text{Induction Hypothesis on } n/2 \\
&= An(\log_2 n - 1) + An && \text{Simplification: } \log_2(n/2) = \log_2 n - 1. \\
&= An\log_2 n
\end{aligned}$$

This finishes the induction. Therefore we know $T(n) \leq An\log_2 n$ is true for all n. The running time of the algorithm is $O(n\log n)$. ■

2.3.2 Recursion Tree

Recursion tree is a more intuitive way for analyzing the running time of divide and conquer algorithms. To use the recursion tree method, we draw a tree that includes all the recursive calls made by the algorithm (see Figure 2.1). The nodes in the tree are divided into different layers corresponding to different depth of the recursive call. The top layer of the recursion tree corresponds to the single call to the problem of size n, the bottom layer of the recursion tree corresponds to the base cases.

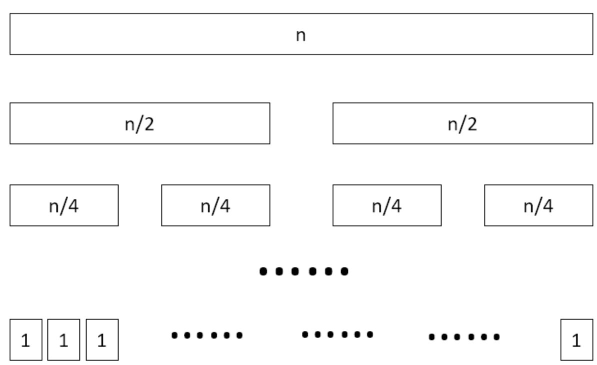

Figure 2.1: Recursion Tree for MergeSort

To compute the running time, we use the following recursion tree lemma:

Lemma 2.1 *The running time of the algorithm is equal to the sum of the merge costs for all the nodes in the decision tree. In particular, if all nodes in the same layer are of the same size, then we have*

$$T(n) = \sum_{i=1}^{depth} m_i \times n_i.$$

Here depth is the number of layers in the tree, m_i is the merge cost for each node at layer i, and n_i is the number of nodes at layer i.

Proof: We can prove this by induction. The induction hypothesis is that, for each node, its total cost is equal to the sum of merge costs for all the nodes in the sub-tree rooted at this node. This is obviously true for the leaves, because they correspond to the base cases (for base cases we define the merge cost to just be the cost of the base case).

Induction step: Suppose this is true for all the children of a node u. By the recurrence relation we know the time it takes to solve u is equal to the merge cost at u, plus the total cost for all children of u. By induction hypothesis, the cost of a child v of u is equal to the sum of merge costs for all the nodes in the sub-tree rooted at v. For the sub-tree rooted at u, a node is either u itself, or in the sub-tree rooted at one of the children of u. Therefore the total running time for u is also equal to the sum of merge costs for all nodes in the sub-tree rooted at u.

A more intuitive way of seeing this for the Example in Figure 2.1 is

$$\begin{aligned}
T(n) &= 2T(n/2) + An \\
&= 4T(n/4) + 2 \cdot A \cdot (n/2) + An \\
&= 8T(n/8) + 4 \cdot A \cdot (n/4) + 2 \cdot A \cdot (n/2) + An \quad \text{the 3 additional terms are merge costs at layer 3, 2, 1 respectively} \\
&= \sum_{i=1}^{\log_2 n} 2^{i-1} \cdot A \cdot (n/2^{i-1}) \\
&= \sum_{i=1}^{\log_2 n} An \\
&= An \log_2 n.
\end{aligned}$$

In particular, for this case n_i (number of nodes at layer i) is 2^{i-1}, m_i (merge cost for each node at layer i) is $n/2^{i-1}$, and number of layers is $\log_2 n$. ■

2.4 Counting Inversions

The next example we look at is the problem of counting inversions. Given an array $a[1..n]$, we say a pair $(i, j)(i, j \in \{1, 2, ..., n\})$ is an inversion if $i < j$ but $a[i] > a[j]$. The goal is to count how many inversions are there for the array $a[]$.

Intuitively the number of inversions is a way of measuring how far the array $a[]$ is from sorted in ascending order. If a is sorted then the number of inversions is 0. If the number of inversions is large, then we can think of a as far from being sorted.

First Attempt As a first attempt, we will try to follow the idea of MergeSort. First, we split the array into two halves, and count the inversions in each of them. Then we try to merge the result.

As an example, consider $a[] = \{6, 2, 4, 1, 5, 3, 7, 8\}$, which is an array with 9 inversions. After splitting, the number of inversions in the two halves are 5 (for $\{6, 2, 4, 1\}$) and 1 (for $\{5, 3, 7, 8\}$). In order to count the number of inversions for the entire array, we need to include the inversions that are entirely in one of the two halves, and we also need to include the number of inversions *between* the two halves. In this case the number of inversions between two halves is 3.

However, counting the number of inversions between the two halves is not very easy. Naïvely we can do this by enumerating all the pairs, but that will take $\Theta(n^2)$ time, which is no better than the brute force algorithm.

Improved Algorithm To improve the algorithm, the key observation is that if we not only know the number of inversions of the two parts, but we also know they are sorted, then counting the number of inversions between the two parts is going to be simple.

When we are merging two sorted arrays into one, suppose $b[i] < c[j]$ and we are putting element $b[i]$ to the sorted array. In this case, we know $b[i]$ must be larger than $c[1], c[2], ..., c[j-1]$ (because those went into the sorted array before $b[i]$), but $b[i]$ is smaller than $c[j]$. Therefore the total number of inversions related to $b[i]$ is equal to $j-1$. See the following pseudo-code.

Algorithm 4 CountingInversion(a[])

```
if length(a) < 2 then {Base case}
  return a[], 0
end if
Partition a[] evenly to two arrays b[], c[]. {Divide}
b[], count_b = CountingInversion(b[])
c[], count_c = CountingInversion(c[]) {Recursive Calls}
a[], count_bc = MergeCount(b, c) {Merge}
return a[], count_b+count_c+count_bc
```

Algorithm 5 MergeCount(b[], c[])

```
Allocate an empty array a[]
i = 1, j = 1, count = 0
while (i ≤ length(b) or (j ≤ length(c)) do
  if b[i]< c[j] then
    Append b[i] to a[], i = i + 1
    count = count + (j − 1)
  else
    Append c[j] to a[], j = j + 1
  end if
  return a[], count
end while
```

Running Time The recurrence relation of CountingInversion is exactly the same as the recurrence relation of MergeSort. Therefore they also have the same running time $\Theta(n \log_2 n)$.