Divide&Conquer: MergeSort

Algorithmic Thinking
Luay Nakhleh
Department of Computer Science
Rice University
Spring 2014

Divide-and-Conquer Algorithms

- **Divide-and-conquer** algorithms work according to the following general plan:

  1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.

  2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).

  3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

Sorting

- Given a list L of n elements, we want to sort them in ascending order.
- The most basic brute-force algorithm would go through every permutation of the n elements, and return one that is sorted in ascending order.
- In the worst case, this algorithm takes $O(n \cdot n!)$ time, since there are $O(n!)$ permutations, and for each one, it takes $O(n)$ to check if it is sorted in ascending order.
- Can we do better? Of course; much better!

MergeSort: Verbal Description

- MergeSort divides the list L[0..n-1] into two halves L[0..$\lfloor n/2 \rfloor$-1] and L[$\lfloor n/2 \rfloor$..n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
- Question: How do we sort each of the two halves?
  - Answer: by dividing each into two halves, sorting them, and then merging.
- Question: When do we stop dividing the list?
  - Answer: When we reach a list of size 1 (since we know how to sort it).

MergeSort: Pseudo-Code
The Divide Phase

MergeSort: Pseudo-Code The Divide Phase

MergeSort
Input: List L of ``orderable” elements
Modifies: List L is sorted in-place in ascending order
Output: None

If n>1
  copy L[0..$\lfloor n/2 \rfloor$-1] to A[0..$\lfloor n/2 \rfloor$-1];
  copy L[$\lfloor n/2 \rfloor$..n-1] to B[0.. $\lfloor n/2 \rfloor$];
  MergeSort(A[0..$\lfloor n/2 \rfloor$-1]);
  MergeSort(B[0..$\lfloor n/2 \rfloor$-1]);
  Merge(A,B,L);

MergeSort: Pseudo-Code
The Combine Phase

MergeSort: Pseudo-Code The Combine Phase

Merge
Input: Two sorted lists A[0..p-1] and B[0..q-1], and list L
Modifies: List L contains the elements of A and B sorted in ascending order
Output: None

```
i←0; j←0; k←0;
While i<p and j<q
    If A[i]≤B[j]
        L[k]←A[i];
        i←i+1;
    Else
        L[k]←B[j];
        j←j+1;
    k←k+1;
If i=p
    copy B[j..q-1] to L[k..p+q-1]
Else
    copy A[i..p-1] to L[k..p+q-1]
```

What Is The Running Time of MergeSort?

- For simplicity, let us assume n is a power of 2 (that is, $n=2^m$ for some m).

- Let $C(n)$ be the number of steps **MergeSort** takes on a list L that has n elements.

- Then, we have the <u>recurrence</u>

$$C(n) = 2C(n/2)+C_{merge}(n) \text{ for } n>1, \text{ and } C(1)=1.$$

- Notice that $C_{merge}(n)=O(n)$.

- Question: What function $g(n)$ gives us $C(n) = O(g(n))$?

Sequences and Recurrence Relations

- A (numerical) <u>sequence</u> is an ordered list of numbers.
- Examples: 1,1,2,3,5,8,13,21,...
- A sequence can also be viewed as a function $x(n)$: its argument n indicates a position of a number in the list, while the function's value $x(n)$ stands for that number itself.
- $x(n)$ is called the <u>generic term</u> of the sequence.

Sequences and Recurrence Relations

- There are two principal ways to define a sequence:
  - by an explicit formula expressing its generic term as a function of n; e.g., $x(n) = 2n$ for $n \geq 0$,
  - by an an equation relating its generic term to one or more other terms of the sequences, combined with one or more explicit values for the first term(s); e.g.,

$$x(n) = x(n-1) + n \quad \text{for } n > 0,$$
$$x(0) = 0.$$

Sequences and Recurrence Relations

- There are two principal ways to define a sequence:
  - by an explicit formula expressing its generic term as a function of n; e.g., $x(n) = 2n$ for $n \geq 0$,
  - by an an equation relating its generic term to one or more other terms of the sequences, combined with one or more explicit values for the first term(s); e.g.,

$$x(n) = x(n-1) + n \quad \text{for } n > 0,$$
$$x(0) = 0.$$

recurrence

Sequences and Recurrence Relations

- There are two principal ways to define a sequence:
  - by an explicit formula expressing its generic term as a function of n; e.g., $x(n) = 2n$ for $n \geq 0$,
  - by an an equation relating its generic term to one or more other terms of the sequences, combined with one or more explicit values for the first term(s); e.g.,

$$x(n) = x(n-1) + n \quad \text{for } n > 0,$$ recurrence
$$x(0) = 0.$$ initial condition

Sequences and Recurrence Relations

- To solve a given recurrence subject to a given initial condition means to find an explicit formula for the generic term of the sequence that satisfies both the recurrence and the initial condition or to prove that such a sequence does not exist.

Common Recurrence Types in Algorithm Analysis

- There are a few recurrence types that arise in the analysis of algorithms with remarkable regularity.
- This happens because they reflect one of the fundamental design techniques.
- Of particular interest to us is a recurrence that arises when analyzing the running time of divide&conquer algorithms.

Common Recurrence Types in Algorithm Analysis

- Assuming all smaller instances (resulting from the divide phase) have the same size n/b, with a of them being actually solved, we get the following recurrence valid for $n=b^k$, $k=1,2,\ldots$:

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

  where $a \geq 1$, $b \geq 2$, and $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and combining their solutions.

The Master Theorem

- Let T(n) be an eventually nondecreasing function that satisfies the recurrence

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \quad \text{for } n = b^k, \quad k = 1, 2, \ldots \\
T(1) &= c,
\end{aligned}
$$

  where $a \geq 1$, $b \geq 2$, $c > 0$.

- If $f(n) = O(n^d)$ where $d \geq 0$, then

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

The Master Theorem: The MergeSort Example

- We've seen that a recurrence for the running time of MergeSort is: $C(n) = 2C(n/2)+C_{merge}(n)$ for $n>1$, and $C(1)=1$.

The Master Theorem: The MergeSort Example

- We've seen that a recurrence for the running time of MergeSort is: $C(n) = 2C(n/2)+C_{merge}(n)$ for $n>1$, and $C(1)=1$.
- Recall the recurrence in the Master Theorem:

$$T(n) = aT(n/b) + f(n) \quad \text{for } n = b^k, \quad k = 1, 2, \ldots$$
$$T(1) = c,$$

The Master Theorem: The MergeSort Example

- We've seen that a recurrence for the running time of MergeSort is: $C(n) = 2C(n/2)+C_{merge}(n)$ for $n>1$, and $C(1)=1$.
- Recall the recurrence in the Master Theorem:

$$T(n) = aT(n/b) + f(n) \quad \text{for } n = b^k, \quad k = 1, 2, \ldots$$
$$T(1) = c,$$

- Thus, we have: $a=2$, $b=2$, $c=1$, and $f(n)=O(n^1)$, i.e., $d=1$.

The Master Theorem: The MergeSort Example

- We've seen that a recurrence for the running time of MergeSort is: $C(n) = 2C(n/2)+C_{merge}(n)$ for $n>1$, and $C(1)=1$.
- Recall the recurrence in the Master Theorem:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \quad \text{for } n = b^k, \quad k = 1, 2, \ldots \\ T(1) &= c, \end{aligned}$$

- Thus, we have: $a=2$, $b=2$, $c=1$, and $f(n)=O(n^1)$, i.e., $d=1$.
- Since we have $a=b^d$, it follows from the Master Theorem that

$$T(n) = O(n^1 \log n) = O(n \log n)$$