CS 2500: Algorithms

Lecture 11: Divide-and-Conquer: Merge Sort

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

September 24, 2024

General Method

- The divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$, yielding k subproblems.
- These subproblems must be solved, and a method must be found to combine subsolutions into a solution of the whole.
- If the subproblems are still relatively large, the divide-and-conquer strategy can possibly be reapplied.
- Often, the subproblems resulting from a divide-and-conquer design are of the *same* type as the original subproblem.
- For those cases, the reapplication of divide-and-conquer is naturally expressed by a recursive algorithm.
- Now, smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced.

Control Abstraction

- We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look.
- Control abstraction: A procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.

Control Abstraction

Algorithm DAndC(P)

if Small(P) **then**
    return S(P)
else
    Divide P into smaller instances $P_1, P_2, \ldots, P_k$, where $k \geq 1$
    Apply DAndC to each of these subproblems
    return Combine(DAndC(P_1), DAndC(P_2), ..., DAndC(P_k))
end if

Control Abstraction

- Algorithm DAndC is initially invoked as DAndC(P) where P is the problem to be solved.
- The function `Small(P)` is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting.
  - If so, function S is invoked. Otherwise, the problem P is divided into smaller subproblems.
- These subproblems $P_1, P_2, \ldots, P_k$ are solved by recursive applications of the divide-and-conquer method.
- The function `Combine` is a function that determines the solution to P using the solutions to the k subproblems.

Control Abstraction

The computing time of DAndC is described by the recurrence relation:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \ldots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

where

- $T(n)$ is the time for DAndC on any input of size n.
- $g(n)$ is the time to compute the answer directly for small inputs.
- $f(n)$ is the time for dividing P and combining the solutions to subproblems.

Recurrence Equation for Divide-and-Conquer Algorithms

The complexity of many divide-and-conquer algorithms is given by recurrences of the form:

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(\frac{n}{b}) + f(n) & n > 1 \end{cases}$$

- a and b are known constants.
- Assumptions:
  - $T(1)$ is known.
  - n is a power of b (i.e., $n = b^k$).

Introduction

- Merge Sort is an example of the divide-and-conquer approach.
- The algorithm has a worst-case time complexity of $O(n \log n)$.
- We assume that the elements are to be sorted in non-decreasing order.

Merge Sort

Splitting and Merging

- Given a sequence of n elements $a[1], \ldots, a[n]$, the sequence is split into two sets:

$$a[1], \ldots, a[\lfloor n/2 \rfloor] \quad \text{and} \quad a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$$

- Each set is individually sorted, and then merged to form a single sorted sequence of n elements.

Merge Sort

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

Merge Sort: Tree Calls

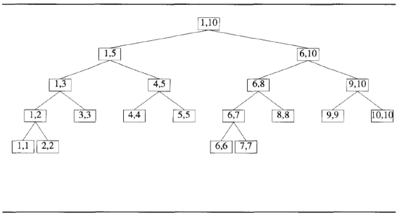

Figure: `MergeSort(1,10)`

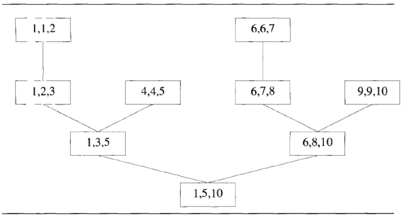

Figure: `Merge`

Merge Sort: Time Complexity Analysis

We start with the recurrence relation:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + n, & n > 1 \\ 1, & n = 1 \end{cases}$$

Now, iterating the recurrence:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \\
&= 2^2 T\left(\frac{n}{4}\right) + n + n \\
&= 2^2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + n + n \\
&= 2^3 T\left(\frac{n}{8}\right) + n + n + n \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kn
\end{aligned}$$

Merge Sort: Time Complexity Analysis

- The stopping condition occurs when $n = 1$. Then $T(n) = 1$.
- Putting $\frac{n}{2^k} = 1$, we get $k = \log_2 n$ and $n = 2^k$.
- From this we get:

$$\begin{aligned} T(n) &= 2^k T\left(\frac{n}{2^k}\right) + kn \\ &= n \cdot T(1) + n \cdot \log_2 n \\ &= n + n \log_2 n \\ &= O(n \log_2 n) \end{aligned}$$

Merge Sort: Space Complexity Analysis

Extra space:

- Merge Sort requires additional space of $2n$.
- This space is needed because Merge Sort does not merge the sorted subsets in place.
- But despite this additional space, the algorithm must still copy the result from $b[low : high]$ to $a[low : high]$ on each call of `Merge`.

Solution: Associate a new field of information with each key. This field is used to:

- Link the keys and any associated information together in a sorted list.
- Change the link values, without moving records.
- Use less space, as only links are updated, not the entire records.

Merging with Link Array

Initial array: `a[] = [50, 20, 40, 10, 30]`
Links: `link[] = [0, 1, 2, 3, 4]`

Merge Step	Compared	Resulting Link Array	Explanation
Initial `a[]` values	N/A	`[0, 1, 2, 3, 4]`	No sorting yet. `link[]` refers to the original array.
1st merge (left side)	Compare `a[1] = 20` and `a[0] = 50`	`[1, 0, 2, 3, 4]`	Since 20 < 50, update `link[]` to point to `a[1]` first.
2nd merge (left side)	Compare `a[2] = 40` with `a[1] = 20` and `a[0] = 50`	`[1, 2, 0, 3, 4]`	20 < 40 < 50, so `link[]` is updated to reflect the sorted order: `a[1]`, `a[2]`, `a[0]`.
3rd merge (right side)	Compare `a[3] = 10` and `a[4] = 30`	`[1, 2, 0, 3, 4]`	Since 10 < 30, the link array reflects this order.
Final merge	Compare `a[3] = 10` with `a[1] = 20`, then merge the rest	`[3, 1, 4, 2, 0]`	10 < 20 < 30 < 40 < 50, so the final sorted order is reflected in `link[]`.

Merging with Link Array

Initial Array and Link Array

- We start with the following array of elements:

$$a[] = [50, 20, 40, 10, 30]$$

- The corresponding initial link array simply refers to the indices of `a[]`:

$$link[] = [0, 1, 2, 3, 4]$$

- Each entry in the `link[]` array points to an element in `a[]`. For example, `link[0]` points to `a[0] = 50`.

Merging with Link Array

Recursive Splitting in Merge Sort. Merge Sort first recursively splits the array `a[]` into smaller subarrays:

- Left subarray: `a[0:2] = [50, 20, 40]`
- Right subarray: `a[3:4] = [10, 30]`

These subarrays are further divided:

- Left subarray: Split into `a[0] = 50`, `a[1] = 20`, `a[2] = 40`.
- Right subarray: Split into `a[3] = 10` and `a[4] = 30`.

Merging with Link Array

We begin merging individual elements back together, updating the link array to reflect the correct sorted order.

1. Merge `a[1]` and `a[0]`:
   - Compare `a[1] = 20` and `a[0] = 50`.
   - Since 20 ¡ 50, update the `link[]` array:

$$link[] = [1, 0, 2, 3, 4]$$

   - Now, the link array points to the elements in the correct order for the first two elements.

Merging with Link Array

2. Merge `a[1:2]` (sorted as `[20, 50]`) with `a[2]`:
   - Compare `a[2] = 40` with `a[1] = 20` and then with `a[0] = 50`.
   - The element 40 is smaller than 50 but larger than 20, so the link array is updated:

     $$link[] = [1, 2, 0, 3, 4]$$

   - Now, the left subarray is fully sorted using the link array.

Merging with Link Array

3. Merge the right subarray `a[3:4]`:
   - Compare `a[3] = 10` with `a[4] = 30`.
   - Since 10 ¡ 30, update the link array:

$$link[] = [1, 2, 0, 3, 4]$$

   - The right subarray is now correctly ordered as `a[3] = 10` and `a[4] = 30`, which are accessed using the link array.

Merging with Link Array

4. Final Merge of Left and Right Subarrays.
   - Merge the left sorted subarray (`link[0] = 1`, `link[1] = 2`, `link[2] = 0`) and the right sorted subarray (`link[3] = 3`, `link[4] = 4`).
   - Compare `a[3] = 10` with `a[1] = 20`.
   - Since 10 ¡ 20, update the first entry in `link[]`:

$$link[] = [3, 1, 2, 0, 4]$$

   - Continue merging and comparing until the link array is updated to reflect the fully sorted order:

$$link[] = [3, 1, 4, 2, 0]$$

Merging with Link Array

The **final sorted order** is accessed using the link array:

$$\text{Sorted order from } link[] = [10, 20, 30, 40, 50]$$

The link array now points to the elements in the correct sorted order:

- $a[link[0]] = a[3] = 10$
- $a[link[1]] = a[1] = 20$
- $a[link[2]] = a[4] = 30$
- $a[link[3]] = a[2] = 40$
- $a[link[4]] = a[0] = 50$

Thus, the final sorted order is: `10, 20, 30, 40, 50`.

Merge Sort: Space Complexity Analysis

Another issue: Stack space needed due to recursion.

- Each recursive call splits the input into two approximately equal-sized subsets.
- The maximum depth of the recursion is proportional to $\log n$.
- The need for stack space stems from the top-down nature of the algorithm.