Divide-and-Conquer

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

September 8, 2024

Divide-and-Conquer Algorithms

Divide-and-Conquer Algorithms

- **Divide:** Break the problem (instance) into subproblems of sizes that are fractions of the original problem size.
- **Conquer:** Recursively solve each of the subproblems. If the subproblems are “small enough” (base case), then solve the subproblem in a straightforward manner.
- **Combine:** Put the solutions for each of the subproblems together to obtain a solution for the original problem.

A Simple Example

- Example: Use binary search to find an element *k* in a sorted array.

A Simple Example

- Example: Use binary search to find an element k in a sorted array.
- **Divide:** Compare the middle element with k.
- **Conquer:** If the middle element is not k, we can recursively solve the subproblem where we check half of the current array.
- **Combine:** There is only 1 subproblem in binary search, and so the combine step is trivial.

Merge Sort

Merge Sort

- The stable algorithm of divide-and-conquer method.
- The sorting problem: Given an array of n numbers, sort the numbers in the array in non-decreasing order.
- A divide and conquer sorting algorithm:
  - **Divide:** Break the problem into 2 subproblems of size $n/2$.
  - **Conquer:** Recursively sort each of the 2 subarrays.
  - **Combine:** Combine the two sorted subarrays with $n/2$ elements into one sorted array of n elements in $O(n)$

Merge Sort Pseudo-code

Merge sort for an array of numbers, A[p:q]:

```
int[] merge\_sort(A[p:q]){
        n = q - p + 1; //n is array size
        /* based case */
        if(n == 1)
           return A; // no need to sort A with one element
        /* divide */
        A1 = A[p:p+n/2]; // take the ceiling of n/2 here
        A2 = A[p+(n/2)+1:q];
        /* conquer */
        A1 = merge_sort(A1);
        A2 = merge_sort(A2);
        /* merge A1 and A2 into A */
        l = m = 1;
        for(i = p; i <= q; i++){
           if(A1[l] <= A2[m]){
               A[i] = A1[l]; l++;
           }
           else{
               A[i] = A2[m]; m++;
           }
        }
}
```

Merge Sort Illustration

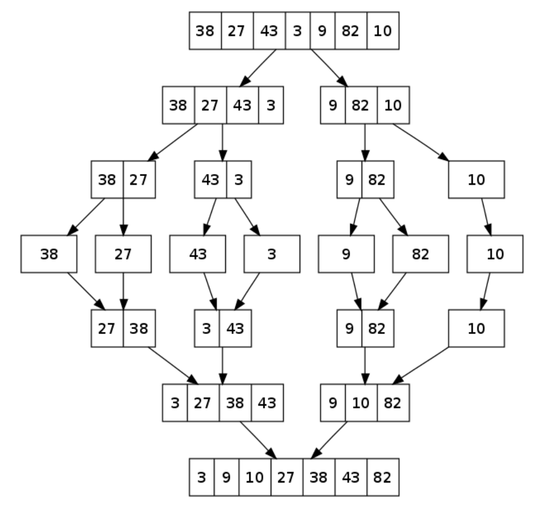

Image copied from Wikipedia at this link.

Merge Sort Time Complexity

- For an array A of size n, let the complexity function be $T(n)$
- Divide: partition the array take constant time, so the time complexity is $O(1)$
- Conquer: the time complexity of solving two sub arrays is $2T(\frac{n}{2})$
- Merge: the time complexity of merging n items is $O(n)$
- Put every thing together, we have $T(n) = 2T(\frac{n}{2}) + O(n)$.
- We have solved this equation before: $T(n) = \Theta(n \lg n)$

Bubble Sort

- As a comparison, let's consider bubble sort for an array of numbers, A[p:q]:

```
int[] compare_sort(A[p:q]){
  for(i = 0; i < p-q; i++){
    for(j = p; j < q-i; j++){
      if(A[j] > A[j+1])
        swap(A[j], A[j+1]);
    }
  }
}
```

- There are two loops. The first loop has n iterations; the second loop has $n-1, n-2, n-3, \ldots, 1$ loop. Recall our analysis in Asymptotic analysis, the time complexity of bubble sort is $O(n^2)$.

The Benefit of Divide-and-Conquer in Sorting

- Merge sort runs in $\Theta(n \lg n)$ time.
- $\Theta(n \lg n)$ grows more slowly than the $\Theta(n^2)$ running time of bubble sort as n approaches infinity.
- Therefore, merge sort asymptotically beats bubble sort in the worse case.
- In practice, merge sort performs better than bubble sort on arrays with 30 or more numbers.

Time Analysis for Divide-and-Conquer

Guess and Induction

- Divide-and-Conquer involves recursion, which is usually hard to analyze.
- Expanding the time complexity recursive function (or drawing a recursion tree) models the running time of a recursive algorithm, but in some cases can be unreliable and/or difficult to analyze.
- It is good for generating *guesses* of what the running time could be.
- In such cases, we may need to verify our guess is correct via a proof by induction.

Induction Based Proof for Merge Sort

- Since there are $\log n$ steps of recursion, and each step needs at most n time, we can guess the run time of merge sort is $O(n \lg n)$.
- Now let's prove $T(n) = O(n \lg n)$ with induction.
  - When $n = 2$, the run-time of merge sort is a constant, d. And $d \leqslant c \cdot 2 \cdot \lg 2$, for any $c \geqslant d/2$. That is $T(2) = O(2 \cdot \lg 2)$. (Note that we start from $n = 2$ instead of $n = 1$, b/c the guess won't hold for $n = 1$.)
  - Assume $T(k) = O(k \cdot \lg k)$, for any $k < n$. We want to prove $T(n) = O(n \lg n)$. Based on $T(n)$'s definition and induction assumption, we have

$$
\begin{aligned}
T(n) &= 2 \cdot T(\frac{n}{2}) + d \cdot n \leqslant 2 \cdot (c \cdot \frac{n}{2} \lg \frac{n}{2}) + d \cdot n \\
&= c \cdot n \cdot (\lg n - \lg 2) + d \cdot n = c \cdot n \cdot \lg n - c \cdot n + d \cdot n \\
&= c \cdot n \cdot \lg n - (d - c) \cdot n \leqslant c \cdot n \cdot \lg n (when\ c > d) \\
&= O(n \lg n)
\end{aligned} \tag{1}
$$

Induction Exercises

- For merge sort, can you prove $T(n) = \Omega(n \lg n)$.
- Divide and conquer to compute a^n.

Induction Exercises

- For merge sort, can you prove $T(n) = \Omega(n \lg n)$.
- Divide and conquer to compute a^n.
  - Algorithm: compute $a^{n/2}$ recursively, then compute a^n as $a^n = a^{n/2} \cdot a^{n/2}$.

Induction Exercises

- For merge sort, can you prove $T(n) = \Omega(n \lg n)$.
- Divide and conquer to compute a^n.
  - Algorithm: compute $a^{n/2}$ recursively, then compute a^n as $a^n = a^{n/2} \cdot a^{n/2}$.
  - Recursive run time: $T(n) = T(n/2) + O(1)$

Induction Exercises

- For merge sort, can you prove $T(n) = \Omega(n \lg n)$.
- Divide and conquer to compute a^n.
  - Algorithm: compute $a^{n/2}$ recursively, then compute a^n as $a^n = a^{n/2} \cdot a^{n/2}$.
  - Recursive run time: $T(n) = T(n/2) + O(1)$
  - Guess: $T(n) = O(\lg n)$, because each are $\lg n$ recursive steps, and each step takes constant time.

Induction Exercises

- For merge sort, can you prove $T(n) = \Omega(n \lg n)$.
- Divide and conquer to compute a^n.
  - Algorithm: compute $a^{n/2}$ recursively, then compute a^n as $a^n = a^{n/2} \cdot a^{n/2}$.
  - Recursive run time: $T(n) = T(n/2) + O(1)$
  - Guess: $T(n) = O(\lg n)$, because each are $\lg n$ recursive steps, and each step takes constant time.
  - Induction proof:
    - When $n = 2$, the run-time is a constant, d. And $d \leqslant c \cdot \lg 2$, for any $c \geqslant d$. That is $T(2) = O(\lg 2)$.
    - Assume $T(k) = O(\lg k)$, for any $k < n$. We want to prove $T(n) = O(\lg n)$. Based on $T(n)$'s definition and induction assumption, we have

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + d \leqslant c \cdot \lg \frac{n}{2} + d \\
&= c \cdot (\lg n - \lg 2) + d = c \cdot \lg n + (d - c) \\
&\leqslant c \cdot \lg n (when\ c > d) \\
&= O(\lg n)
\end{aligned} \tag{2}
$$

Matrix Multiplication

Naive Matrix Multiplication Algorithm

- Multiply tow matrices: $C = A \times B$.

$$\begin{bmatrix} c_{11} & c_{12} & \dots \\ \vdots & \ddots & \\ c_{n1} & & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots \\ \vdots & \ddots & \\ a_{n1} & & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \dots \\ \vdots & \ddots & \\ b_{n1} & & b_{nn} \end{bmatrix}$$

- A naive algorithm:

```
for i = 1; i ≤ n; i = i + 1 do
    for j = 1; j ≤ n; j = j + 1 do
        c_ij = 0
        for k = 1; k ≤ n; k = k + 1 do
            c_ij = c_ij + a_ik b_kj
        end for
    end for
end for
```

Naive Matrix Multiplication Algorithm Run Time

- This naive algorithm has three loops, each loop has n iterations.
- Therefore, the total run time is $O(n^3)$.

Naive Divide-and-Conquer Algorithm

- Break each matrix into four sub matrices.

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

- Each sub matrix' size is $\frac{n}{2} \times \frac{n}{2}$.
- To compute C:
  - r = a·e + b ·g
  - s = a·f + b ·h
  - t = c·e + d ·g
  - u = c·f + d ·h

Naive Divide-and-Conquer Algorithm Run time

- The naive Divide-and-Conquer algorithm needs 8 sub-array multiplication and one $n \times n$ array summation.
- Therefore, the run time is $T(n) = 8 \cdot T(\frac{n}{2}) + \Theta(n^2)$.
- If we solve this recursive function, we have $T(n) = O(n^{\lg 8}) = O(n^3)$.
- This is no reduction in time!!!

Naive Divide-and-Conquer Algorithm Run time cont.

- The problem is that, as long as we go with the standard matrix multiplication definition, there must be n^3 multiplications.
  - Because there are n^2 elements in C, and each elements need n multiplications.
- As a comparison, in merge sort, we actually reduce the number of comparisons, as compared to bubble sort.
  - The key to good algorithm design, is to eliminate all unnecessary operations.
  - Unfortunately, for matrix multiplication, you need very good understanding of linear algebra to remove the unnecessary operations.

Strassen's ALgorithm

- Recursively compute the following matrices (note only 7 multiplications):
  - $P_1 = a \cdot (f - h)$
  - $P_2 = (a + b) \cdot h$
  - $P_3 = (c + d) \cdot e$
  - $P_4 = d \cdot (g - e)$
  - $P_5 = (a + d) \cdot (e + h)$
  - $P_6 = (b - d) \cdot (g + h)$
  - $P_7 = (a - c) \cdot (e + f)$
- We can then compute C as so:
  - $r = P_5 + P_4 - P_2 + P_6$
  - $s = P_1 + P_2$
  - $t = P_3 + P_4$
  - $u = P_5 + P_1 - P_3 - P_7$

Strassen's Algorithm Run time

- The naive Divide-and-Conquer algorithm needs **7** sub-array multiplication and quite a few sub array summations/subtractions.
- Therefore, the run time is $T(n) = 7 \cdot T(\frac{n}{2}) + \Theta(n^2)$.
- If we solve this recursive function, we have $T(n) = O(n^{\lg 7}) = O(n^{2.80})$.
- Strassen's algorithm is slightly better than the native algorithm when $n \geqslant 30$. But it needs more space.
- Essentially, Strassen found a way to group some multiplications, and replace multiplications with summations and subtractions.
- The best we can do today is $O(n^{2.37x})$ (Stothers, Andrew 2010;Davie, A.M. and Stothers, A.J. 2011; Williams, Virginia 2011; Francois Le Gall, 2014).