CS 4310 Algorithms: II. Divide and Conquer Methods and Recurrence Relations

E. de Doncker[1]

[1]Dept. of Computer Science, WMU, U.S.A.

January 2024

Outline

1. Divide and conquer methods
   - Binary search
   - Merge sort
   - Quicksort
   - Strassen's matrix multiplication

Divide and conquer

A divide and conquer method applies the following steps [1, 2]:

Divide and conquer

— Divide the problem instance into smaller instances. This is usually done recursively until instances are obtained that are small enough to be solved directly.
— Conquer by solving the smallest instances, and combining the solutions (if needed) while going back up the recursion levels, until the original instance is solved.

In this section we will cover several algorithms adhering to this paradigm, obtain and solve recurrence relations for their time complexity. The corresponding chapters in the text books are Ch. 2 in [2] and Ch. 3 in [1].

Binary search algorithm

Example: https://www.youtube.com/watch?v=kazvOikPQrk
Recursive algorithm [2]: called as $BinarySearch(1, n)$ on top level.

```
BinarySearch(low, high) { // Determine if key x is in array A (range low to high)
// where A is sorted in nondecreasing order. If x is found,
// return its index (location) in A, else return 0. A and x are global.

    if ($low > high$) return 0;
    else {
        $mid = (low + high)/2$;   // Divide into two subproblems
        // Solve the appropriate subproblem
        if ($x == A[mid]$) return $mid$;
        else if ($x < A[mid]$) return $BinarySearch(low, mid - 1)$;
        else return $BinarySearch(mid + 1, high)$;
    }
}
```

Note: Since A and x do not change, passing them in the recursive calls would be inefficient; they can be made global.

Binary search analysis

We denote the worst case time for an array of length n by $W(n)$.
$W(n)$ represents the number of comparisons with x in the course of the algorithm. One way to incur the worst case is when $x >$ all elements in the given array. We count one comparison followed by the recursive call on each level of the recursion. Then, if n is a power of 2,

$$W(n) = \begin{cases} W(\frac{n}{2}) + 1 & n > 1 \quad \text{(top level of recursion)} \\ 1 & n = 1 \end{cases}$$

To solve the recurrence relation by expansion (or substitution), we replace $W(\frac{n}{2})$ on the right by $W(\frac{n}{4}) + 1$ and keep replacing:

$$\begin{aligned} W(n) &= W(\frac{n}{2}) + 1 \\ &= W(\frac{n}{4}) + 2 = W(\frac{n}{8}) + 3 \\ &\dots \\ &= W(1) + \log_2 n = (\lg n) + 1 \end{aligned}$$

In general, $W(n) = \lfloor \lg n \rfloor + 1$, so $W(n) = \Theta(\lg n)$.

Binary search analysis

Note: We can check if the recurrence on the previous slide was solved correctly (for n a power of 2). Using $W(n) = (\lg n) + 1$, is $W(1) = 1$ (it is), and does $W(n) = W(\frac{n}{2}) + 1$ hold for $n \geq 2$?
It does: starting from the right hand side,
$W(\frac{n}{2}) + 1 = (\lg \frac{n}{2}) + 2 = (\lg n) - 1 + 2 = (\lg n) + 1 = W(n)$.

More formally, for n a power of 2, prove that $W(n) = (\lg n) + 1$ by induction on n.
Basis: For $n = 1$, $W(n) = (\lg 1) + 1 = 1$.
Induction hypothesis: For $n = k$, assume $W(k) = (\lg k) + 1$.

Induction step: For $n = 2k$, $W(2k) = W(k) + 1 = (\lg k) + 1 + 1 = (\lg 2k) + 1$

(shows that the property holds for $n = 2k$).

Merge sort algorithm

Examples:
https://www.geeksforgeeks.org/merge-sort/
https://www.youtube.com/watch?v=ZRPoEKHXTJg

(Recursive) Algorithm (cf., *mergesort2()* in [2]; for *Merge()* see text.)

```
MergeSort(low, high) { // Sort array A in nondecreasing order (range low to high).
                       // A is global.
    if (low < high) {
        mid = (low + high)/2;    // Divide into two subproblems
        // Solve the two subproblems
        MergeSort(low, mid);    // Sort subarray in range low to mid
        MergeSort(mid + 1, high);    // Sort subarray in range (mid + 1) to high
        // Combine the (sub)solutions
        Merge(low, mid, high);    // Merge sorted subarrays
    }
}
```

Merge sort analysis

The time is expressed based on the number of comparisons occurring in *Merge()*. For simplicity we assume that the array length n is a power of 2, and show that $W(n) = \Theta(n \log n)$ (see [2]).
On the top level of the recursion, the time to sort the subarrays (each of length $\frac{n}{2}$) is $2W(\frac{n}{2})$. The worst case time for the merge of subarrays of length ℓ and m is $\ell + m - 1 = n - 1$ for total length $\ell + m = n$. This leads to the recurrence

$$W(n) = \begin{cases} 2W(\frac{n}{2}) + n - 1 & n > 1 \\ 0 & n = 1 \end{cases}$$

Solving by expansion gives:

$$\begin{aligned} W(n) &= 2(2W(\frac{n}{4}) + \frac{n}{2} - 1) + n - 1 = 4W(\frac{n}{4}) + 2n - 3 \\ &= 4(2W(\frac{n}{8}) + \frac{n}{4} - 1) + 2n - 3 = 8W(\frac{n}{8}) + 3n - 7 \\ &\dots \\ &= n\,W(1) + (\lg n)\,n - (n-1) = n\,(\lg n) - (n-1) = \Theta(n \log n) \end{aligned}$$

Merge sort analysis

A proof that $T(n) = \mathcal{O}(n \log n)$ (where $T(n)$ denotes time as a function of n in general, and n is a power of 2) can be written for *MergeSort()* as follows: From the pseudocode we have that $T(n) \leq 2T(\frac{n}{2}) + cn$,
where the last term is for the merge (of order $\mathcal{O}(n)$). Expansion yields:

$$\begin{aligned} T(n) &\leq 2(2T(\frac{n}{4}) + c\frac{n}{2}) + cn \;=\; 4T(\frac{n}{4}) + 2cn \\ &\leq 4(2T(\frac{n}{8}) + c\frac{n}{4}) + 2cn \;=\; 8T(\frac{n}{8}) + 3cn \\ &\quad \ldots \\ &\leq nT(1) + (\lg n)cn = \mathcal{O}(n \log n) \end{aligned}$$

Quicksort algorithm

(Recursive) Algorithm (cf., [2, 1])

```
QuickSort(low, high) { // Sort array A in nondecreasing order (range low to high).
                       // A is global.
   if (low < high) {
      // Divide the problem into two subproblems
      Partition(low, high, pivotindex);    // Partition array so that
                       // all elements < A[pivotindex] are to its left
                       // and all elements > A[pivotindex] are to its right
      // Solve the two subproblems
      QuickSort(low, pivotindex − 1);   // Sort subarray in range low to (pivotindex − 1)
      QuickSort(pivotindex + 1, high);  // Sort subarray in range (pivotindex + 1) to high
      // No need to combine solutions
   }
}
```

QuickSort algorithm

Partition$(low, high, pivotindex)$:
– Sets the *pivotitem* element. Options are:
(i) first element, $pivotitem = A[low]$
(ii) last element, $pivotitem = A[high]$
(iii) median of three elements $(A[low], A[mid], A[high])$
– For a couple different versions see [1, 2].
– Partition in [2] moves through the array while comparing the elements with the *pivotitem* element; when an element $< pivotitem$ is found, it is exchanged with an element $> pivotitem$.
– Partition in [1] moves through the array (index i) from left to right to find an element $> pivotitem$, and from right to left (index j) to find an element $< pivotitem$; when found these are exchanged; this proceeds until i and j cross.

– A demonstration is given at https://www.youtube.com/watch?v=tIYMCYooo3c using the last element ($A[high]$) as the pivotitem, and moving left to right to find elements $> pivotitem$ and exchanging with elements $< pivotitem$.

QuickSort worst case analysis

Let us express $T(n)$ as the number of comparisons with the *pivotitem* in *Partition()*.
When the array is already sorted, with $pivotitem = A[low]$, the call
Partition$(low, high, pivotindex)$ leaves the left subarray empty. So only the *pivotitem* is split off, on the top level and subsequent recursion levels.
Partition [2] uses $n-1$ comparisons, thus *QuickSort()* time in case of an already sorted array is $T(n) = \begin{cases} T(n-1)+n-1 & n>0 \\ 0 & n=0 \end{cases}$
By expansion, replace $T(n-1)$ on the right by $T(n-2)+n-2$, and so on. Thus

$$\begin{aligned} T(n) &= T(n-1)+(n-1) = T(n-2)+(n-2)+(n-1) \\ &= T(n-3)+(n-3)+(n-2)+(n-1) = \ldots \\ &= T(n-n)+(n-n)+(n-(n-1))+(n-(n-2))+\ldots+(n-1) \\ &= 1+2+\ldots+(n-1) = \frac{n(n-1)}{2} \end{aligned}$$

Therefore the worst case time $W(n) \geq \frac{n(n-1)}{2}$. It is shown in [2] that also $\leq$ holds, thus $W(n) = \frac{n(n-1)}{2} = \Theta(n^2)$.

QuickSort average case analysis

It is assumed that each array index p has equal probability $(= \frac{1}{n})$ to be the location of the *pivotitem* after partition. By partitioning the array of length n, the left subarray is then of length $p - 1$ and the right subarray of length $n - p$. The average time $Av(n)$ of quicksort is an average over all possible values of locations p $(= 1, \dots, n)$. Then on the top level of the recursion:

$$Av(n) = \frac{1}{n} \sum_{p=1}^{n} (Av(p-1) + Av(n-p)) + n - 1,$$

where $n - 1$ is the time in *Partition()*, and the sum divided by n is the time for the two recursive calls averaged over the possible pivotindex values p.

It is shown in [1, 2] that $Av(n) = \Theta(n \log n)$.

Randomized QuickSort

The quadratic worst case behavior of *QuickSort( )* cannot be avoided by the median of three strategy for picking the pivotitem. Randomized QuickSort takes the *pivotitem* at random from $A[low], \dots, A[high]$. Algorithm *RQuicksort( )* [1] runs in expected (average) time $E(n) = \mathcal{O}(n \log n)$, where the expectation is over the space of possible outcomes of the randomizer (rather than the space of possible inputs).
The expected time $E(n)$ of *RQuicksort( )* for any input of n elements satisfies a recurrence of the same form as that for the average time of *QuickSort( )*.

Algorithm RQuickSort (cf., [1])

```
RQuickSort(low, high) { // Sort array A in nondecreasing order (range low to high).
                        // A is global.
    if (low < high) {
        if ((high − low) > 5) Interchange(low+random() % (high − low + 1), low);
        // Random array element was interchanged at pivotindex (low) position
        Partition(low, high, pivotindex);   // Partition array so that
                                  // all elements < A[pivotindex] are to its left
                                  // and all elements > A[pivotindex] are to its right
        RQuickSort(low, pivotindex − 1);   // Sort subarray in range low to (pivotindex − 1)
        RQuickSort(pivotindex + 1, high);  // Sort subarray in range (pivotindex + 1) to high
    }
}
```

Divide and conquer matrix multiplication

For the multiplication of two $n \times n$ matrices, $C = A \times B$, the element $c_{i\,j} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Since each element involves n multiplications, and there are n^2 elements to compute, there are n^3 multiplications in total, and the conventional matrix multiplcation algorithm takes $\Theta(n^3)$ time.
If n is a power of 2, a divide and conquer method can be given where the matrices A, B and C are each subdivided into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$ (see [1]). The product is then given by

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
$$= \begin{pmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{pmatrix}$$

In view of the 8 matrix multiplications of size $\frac{n}{2} \times \frac{n}{2}$, the total number of multiplications in the resulting matrix satisfies the recurrence $T(n) = \begin{cases} 8T(\frac{n}{2}) & n > 1 \\ 1 & n = 1 \end{cases}$ which is solved by $T(n) = n^3$. So there is no gain compared to the direct matrix multiplication.

Divide and conquer matrix multiplication

In view of the 8 matrix multiplications of size $\frac{n}{2} \times \frac{n}{2}$ matrices, and 4 matrix additions to combine these matrix products, a program implementing this method will take time satisfying the following recurrence, where k and c are constants [1]:

$$T(n) = \begin{cases} k & n \leq 2 \\ \underbrace{8T(\frac{n}{2})}_{\text{multiplications}} + \underbrace{cn^2}_{\text{additions}} & n > 2 \end{cases}$$

We will learn a method later to solve this type of recurrence easily. The solution is $T(n) = \mathcal{O}(n^3)$.

Strassen's matrix multiplication

Strassen's matrix multplication uses a more complicated combination of the eight submatrices $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}$ to express the product matrix C. There are 7 matrix multiplications of size $\frac{n}{2} \times \frac{n}{2}$ (i.e., one less than for the previous divide and conquer method) and a number of matrix additions. The total number of multiplications satisfies the recurrence $T(n) = \begin{cases} 7T(\frac{n}{2}) & n > 1 \\ 1 & n = 1 \end{cases}$

which is solved by $T(n) = n^{\lg 7} \approx n^{2.81}$, and is thus of lower order than the direct matrix multiplication.
A program implementing this method will take time satisfying the following recurrence, where k and c are constants [1]:

$$T(n) = \begin{cases} k & n \leq 2 \\ \underbrace{7T(\frac{n}{2})}_{\text{multiplications}} + \underbrace{cn^2}_{\text{additions}} & n > 2 \end{cases}$$

which is solved by $T(n) = \mathcal{O}(n^{\lg 7})$.

BIBLIOGRAPHY

S. Sahni E. Horowitz and S. Rajasekeran.
Computer Algorithms/C++.
Computer Science Press, 2nd edition, 1998.
ISBN 0-7167-8315-0.

R. E. Neapolitan.
Foundations of Algorithms.
Jones and Bartlett, 5th edition, 2015.
ISBN 978-1-284-04919-0.