CS3230 Chapter 3 - Divide and Conquer

Based on lectures by Chang Ee-Chien
Notes taken by Andrew Tan
AY18/19 Semester 1

These notes are not endorsed by the lecturers, and I have modified them (often significantly) after lectures. They are nowhere near accurate representations of what was actually lectured, and in particular, all errors are almost surely mine.

1 Divide and Conquer

"Divide and conquer" denotes a general approach in algorithmic design:

Divide the problem into a number of sub-problems

Conquer the sub-problems by solving recursively. if the sub-problems become trivial enough, solve them directly.

Combine the solutions to the sub-problems into the solution fro the original problem.

This chapter will explore several divide-and-conquer algorithms common in computer science.

2 Merge sort

Divide: the n-element sequence to be sorted into two subsequences of $(n/2)$ elements each.

Conquer: Sort the two subsequences recursively.

Combine: Merge the two sorted subsequences to produce the sorted output.

```
MergeSort(A):
   if n == 1 then return
   Copy the first ⌊n/2⌋ elements of A to array B
   Copy the remaining ⌈n/2⌉ elements of A to array C

   MergeSort(B)
   MergeSort(C)

   Merge(A, B, C)
```

```
Merge(A, B, C):
   B[n + 1] ← ∞
   C[m + 1] ← ∞
   i ← 1; j ← 1

   for k ← 1 to n + m
      if B[i] < C[j]:
         A[k] ← B[i]
         i ← i + 1
      else:
         A[k] ← C[j]
         j ← j + 1
```

2.1 Analysis of merge sort

Let $T(n)$ be the number of comparisons in the worst case.

- The divide step takes no comparison
- The two recursive calls of MergeSort take $T(r)$ and $T(n-r)$ comparisons respectively.
- The compine step consists of two parts: Merge on r elements, and merge on $(n-r)$ elements. In total, it takes at most $(r+(n-r))$ comparisons.

Therefore,

$$T(n) = T(r) + T(n-r) + n$$

Assuming n is a power of 2, thus $r = n/2$. So we have the following recurrence equation:

$$T(n) = 2T(n/2) + n \; T(1) = 0$$

Solving the recurrence equation, we obtain:

$$T(n) \in \Theta(n \log(n))$$

Hence, the number of comparisons taken by MergeSort is in $\Theta(n \log(n))$

3 Quick sort

Divide: Picks a "pivot" and rearranges the array so that

- all elements to the left of the pivot are smaller or equal to the pivot.
- all elements to the right of the pivot are larger than the pivot

Conquer: Recursively quicksort the left and right subarrays

Combine: Return the sorted array.

```
QuickSort(A, p, r):
   if p < r:
      j ← partition(A, p, r)
      QuickSort(A, p, j − 1)
      QuickSort(A, j + 1, r)
```

We can easily partition the array by using an algorithm that scans from left to right:

```
Partition(A, p, r):
   x ← A[r]; i ← p − 1
   for j ← p to r − 1:
      if A[j] ≤ x:
         i ← i + 1
         swap (A[i], A[j])
   swap (A[i + 1], A[r])
   return (i + 1)
```

3.1 Analysis of quick sort

For the partition function:

- The number of comparisons taken by Partition is $(r-p)$
- The number of swaps is $\leq (r-p+1)$
- The running time is in $\Theta(r-p)$

In the best or worst case, the running time is in $\Theta(r-p)$. Since the input size is $(r-p+1)$, partition is a linear time algorithm.

3.1.1 Worst case analysis

Consider an inpput instance where the elements are already sorted. Let $T_1(n)$ be the time taken where the input is sorted.
For such input, after partition, the size of one subproblem is $(n-1)$ and the other is 0.
Hence,

$$T_1(n) = T_1(n-1) + T_1(0) + n$$

Solving the recurrence equation, we obtain $T_0(n) \in \Theta(n^2)$.

As every element can only be the pivot at most once, there are at most n pivots, and for every pivot, at most n comparisons and n swaps are required. Hence the worst case running time is indeed in $\Theta(n^2)$.

3.1.2 Best case analysis

Let $T_2(n)$ be the best case,

$$T_2(n) = \min_{0<r\leq n} \{T_2(n-r) + T_2(r-1)\} + n$$

The minimum occurs at the halfway mark, i.e. $r = n/2$. The proof for this is left as an exercise for the reader.
By simplifying the equation further by ignoring the rounding of $n/2$ and the difference in size of the two halves (assuming the size of the array is not a power of 2), we obtain

$$T_2(n) = 2T_2(n/2) + n$$

And solving the recurrence equation, we have

$$T_2(n) \in \Theta(n \log(n))$$

3.1.3 Average case analysis

We assume that all permutations of the input are equally likely.

Let $T_3(n)$ be the average number of comparisons.

$$\begin{aligned}
T_3(n) &= (1/n)(T_3(0) + T_3(n-1) + (n-1)) \\
&\quad +(1/n)(T_3(1) + T_3(n-2) + (n-1)) \\
&\quad +(1/n)(T_3(2) + T_3(n-3) + (n-1)) \\
&\quad \dots \\
&\quad +(1/n)(T_3(n-1) + T_3(0) + (n-1)) \\
&\quad = (1/n)\{\sum_{i=0}^{n-1} T_3(i) + T_3(n-i) + (n-1)\}
\end{aligned}$$

$T_3(1) = 0$
$T_3(0) = 0$

From this recurrence equation, we will be able to solve it to obtain

$$T_3(n) \in \Theta(n \log(n))$$

The average case analysis assumes that the probability that the pivot is the i^{th} smallest is $(1/n)$, regardless of the value of i.
Further proof of the solution to the recurrence equation is described in the appendix.

3.2 Randomized quicksort

A **randomized algorithm** has access to an additional stream of random numbers, and the output is computed based on both the input and the stream of random numbers.
So on the same input, the output could be different.

For randomized quicksort with an input array A, the algorithm randomly and uniformly picks

an r from $\{1, 2, ..., n\}$, partitions A with $A[r]$ as the pivot, and recusively sorts both halfs of A.

For a deterministic algorithm, its average running time is the average over the input distribution. If the algorithm performs poorly on a particular input, its running time on that input is always poor.
Now consider a randomized algorithm. Its *expected* running time is the "average" over the choice of random numbers used by the randomized algorithm. For a same input, the running time is probabilistic. Hence, the expected running time for certain probabilistic algorithms would be the same for all input instances of the same size.

3.2.1 Expected running time

Let $T(n)$ be the random variable for the running time on an input of size n.
Let $X_k = 1$ if the pivot is the k^{th} smallest, and $X_k = 0$ otherwise.
Since the pivot is randomly chosen, $Pr(X_k = 1) = 1/n$ for ny k.

Note that,
$T(n) = T(0) + T(n-1) + n \qquad$ if $X_1 = 1$
$T(n) = T(1) + T(n-2) + n \qquad$ if $X_2 = 1$
...
So,
$E(T(n)) = Pr(X_1 = 1)\{E(T(0)) + E(T(n-1)) + n\}$
$\quad + Pr(X_2 = 1)\{E(T(1)) + E(T(n-2)) + n\}$
$\quad + \ldots$

We use the linearity of expectation and the fact that the X_i's are independent from other random variables to derive the above statements.
Solving the above, we obtain

$$E(T(n)) \in \Theta(n \log(n))$$

Hence, the expected running time taken by randomized quicksort is in $\Theta(n \log(n))$.

4 Matric multiplication

Given two n by n matrices A and B, we would like to find $C = AB$.

A straightforward algorithm that multiples 2 matrices A and B would simply be:

```
Multiply(A, B, C):
  for i ← 1 to n:
    for j ← 1 to n:
      temp ← 0;
      for k ← 1 to n:
        temp ← temp + A[i,k] * B[k,j]
      C[i,j] ← temp
```

The number of multiplications is in n^3, and the number of additions is in n^3.

We can use a divide and conquer algorithm to reduce the number of additions:
Divide each A, B, and C into 4 sub-matrices

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

Thus we obtain:

$r = ae + bf$

$s = ag + bh$

$t = ce + df$

$u = cg + dh$

And the above takes 8 matrix multiplications and 4 matrix additions.
After obtaining the following recurrence relation:

$$T(n) = 8T(n/2)$$
$$T(2) = 8$$

We obtain $T(n) = n^3$

There are other methods that manage to reduce the number of matrix multiplications at each dividing step, and the best *lower bound* known is in $\Omega(n^2)$, that is, all methods must take $\Omega(n^2)$ multiplications.

A Linearity of expectation

For any random variables X_1, X_2, and constants α, β,

$$E(\alpha X_1 + \beta X_2) = \alpha E(X_1) + \beta E(X_2)$$

The above always holds even when X_1 and X_2 are not independent.

B Proof of the average running time of quicksort

Because of the symmetry $T_3(i) = T_3(n-i)$ for any i,
$T_3(n) = (1/n) \sum_{i=0}^{n-1} \{T_3(i) + T_3(n-i) + (n-1)\}$
$(2/n) \sum_{i=0}^{n-1} \{T_3(i) + (n-1)/2\}$

Now,
$nT_3(n) - (n-1)T_3(n-1)$
$= 2 \sum_{i=0}^{n-1} \{T_3(i) + (n-1)/2\} - 2 \sum_{i=0}^{n-2} \{T_3(i) + (n-2)/2\}$
$= 2\{T_3(n-1) + (n-1)/2 + (n-1)/2\}$

$T_3(n)/(n+1) = T_3(n-1)/n + 2(n-1)/(n(n+1))$
$T_3(n)/(n+1) = T_3(n-1)/n + 2/(n+1) - 2/(n^2+n)$

Let $G(n) = T_3(n)/(n+1)$,
Thus, we have
$G(n) = G(n-1) + 2/(n+1) - 2/(n^2+n)$
$G(0) = G(1) = 0$

$G(n) = G(1) + \sum_{k=2}^{k} (2/(k+1) - 2/(k^2+k))$
As $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} \approx \ln(k)$,

$$G(n) \approx 2\ln(n)$$

Thus,

$$T_3(n) \in \Theta(n \log(n))$$