Divide and Conquer

Contents:

The Master Theorem

We will cover this first since it is useful for solving many of the recurrence relations that will appear below. The theorem is used to solve recurrences of the form:

$T_n = aT\frac{n}{b} + f(n)$

where $a \geq 1$ and $b > 1$, and $f(n)$ is an asymptocially positive function. There are three cases for the theorem:

1. $n^{log_b(a)}$ is asymptotically greater than $f(n)$. Then $T_n = \Theta(n^{log_b(a)})$.
2. $f(n)$ is asymptotically greater than $n^{log_b(a)}$. Then $T_n = \Theta(f(n))$.
3. $n^{log_b(a)}$ is asymptotically equal to $f(n)$, i.e. $f(n) = \Theta(n^{log_b(a)})$. Then $T_n = \Theta(f(n) \cdot log_b(n)) = \Theta(n^{log_b(a)} \cdot log_b(n))$.

It is important to note that in the first two cases, one function must be *polynomially* greater than the other, i.e. $n^{log_b(a)-\epsilon} > f(n)$ for $\epsilon > 0$ and vice versa. There is also an additional condition for case 2, where it must also be true that $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$. This is known as the *regularity* condition, and it is satisfied by most polynomially-bounded functions (but it should still be checked).

The Chip Problem

Problem: *A factory produces chips that are either good (G) or bad (B). To evaluate their state, the chips can be paired in a testing device (oracle) where the two chips evaluate each other, i.e. each chip gives its opinion on whether the other is good or bad. Good chips tell the truth, while bad chips are unreliable. Given N chips, determine their state (G/B), with the premise that the set of good chips G is larger than the*

set of bad chips B: $|G| > |B|$. Complexity is measured in terms of the number of uses of the oracle.

Solution:

In this problem, it is important to note that the algorithm only works if it is always strictly true for any set of chips that $|G| > |B|$. The algorithm to find one good chip then works as follows:

- Pair up all of the chips and get them to evaluate each other. There are 3 possible outcomes:
  - GG: This means that either both chips are good, or both chips are bad.
  - BG/GB and BB: This means that at least one of the chips is bad.
- Throw out all pairs that are not GG. Then, throw out one chip from each of the GG pairs.
  - We must verify that the $|G| > |B|$ property is preserved after this operation. By throwing out all pairs that are not GG, we are throwing out either one or two bad chips with each pair, and so the number of bad chips thrown out will be greater than or equal to the number of good chips thrown out.
  - Now there are only GG pairs left, and we know that we still have $|G| > |B|$. This means there are more Good-Good than Bad-Bad pairs, so eliminating one of them from each pair will still guarantee that $|G| > |B|$.
  - We know that there must be at least one GG pair at the start, since $|G| > |B|$.
- Repeat these steps recursively until there is only one chip remaining, at which point we will know it is good since $|G| > |B|$ is preserved throughout.

These steps work for the case when the number of chips is even. When the number of chips is odd, we require a slight modification, explained below:

- Pick a chip at random and have the other $N - 1$ chips evaluate it. Since there are more good chips than bad, we can take the majority vote to determine if the chip we have chosen is good or bad.
- If the chip is good, then we can stop. Otherwise, throw the chip out and proceed with the even case algorithm.

Once we have found a good chip, we can test all of the other chips with $n - 1$ evaluations to find the entire set of good chips.

Complexity:

Even case: $T_n \leq \frac{n}{2} + T_{\frac{n}{2}}$

$T_{\frac{n}{2}}$ is the time required to compute the recursive call of size $\frac{n}{2}$, and the $\frac{n}{2}$ term is the time required to evaluate all pairs at each step of the algorithm.

Odd case: $T_n \leq n - 1 + \frac{n-1}{2} + T_{\frac{n-1}{2}}$

Similar to above, except for the extra $n - 1$ term which denotes the time required to evaluate the first chip picked at random.

We can solve the recurrence using substitution, done below:

$T_n \leq \frac{n}{2} + T_{\frac{n}{2}}$

$T_n \leq \frac{n}{2} + \frac{n}{4} + T_{\frac{n}{4}}$

$T_n \leq \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + T_{\frac{n}{4}}$

$T_n \leq \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 1$

$T_n \leq \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 1$

$T_n \leq n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots) + 1 = n + 1$

Therefore $T_n = O(n)$. The sum of fractions in the last step can be evaluated as an infinite geometric series, where $r = \frac{1}{2}$ and the sum is then $\frac{a_0}{1-r} = \frac{\frac{1}{2}}{1-\frac{1}{2}} = 1$.

Karatsuba Multiplication

Problem: *Multiply two numbers x and y together in faster than $\Theta(n^2)$ time, i.e. faster than the traditional grade school algorithm.*

Solution:

Let's assume we have two numbers x and y which both have the same number of digits. This algorithm can be used for numbers in base 10 as well as binary numbers. We will describe the algorithm for numbers in base 10 first.

- First take, both numbers and split their digits in half to get four new numbers a, b, c, d, where a contains the left half digits of x and b contains the right half, and vice versa for c and d.
- Then, we can express the original multiplication as follows:

  $x \cdot y = (10^{\frac{n}{2}} \cdot a + b)(10^{\frac{n}{2}} \cdot c + d)$

  $= 10^n ac + 10^{\frac{n}{2}} ad + 10^{\frac{n}{2}} bc + bd$

  $= 10^n ac + 10^{\frac{n}{2}} (ad + bc) + bd$

We can see above that there are four multiplications that need to be computed to get the final result (ac, ad, bc and bd). This gives 4 recursive calls, but also a runtime of $\Theta(n^2)$, which is the same as the grade-school algorithm. However, the key insight of this algorithm is to use a trick to reduce the number of recursive calls to 3. This is explained below:

- Recursively compute ac and bd, as before.
- Instead of computing ad and bc, compute $(a + b)(c + d)$. This can be expanded out to $(a + b)(c + d) = ac + ad + bc + bd$.
- The key trick is hidden in this formula: since we have already computed ac and bd, we can subtract these values from the equation above and we're left with just $ad + bc$, which is exactly the same as the middle term in $10^n ac + 10^{\frac{n}{2}} (ad + bc) + bd$!
- This means we can do the same computation in only 3 recursive calls.

Complexity:

The algorithm makes 3 recursive calls on subproblems of size $\frac{n}{2}$, and there are also some shifts and additions done at each step as well. If we assume a cost of 1 for each shift and addition, then the time complexity can be expressed as:

$T_n \leq 2n + 10n + 3T_{\frac{n}{2}}$

With the $2n$ representing the time taken to shift numbers by 10^n and $10^{\frac{n}{2}}$, and the $10n$ representing the time required to add all of the numbers up.

Overall, this gives a time complexity of $T_n = \Theta(n^{log_2 3})$, which can be shown with the Master Theorem.

Mergesort

This algorithm sorts a list/array in time $\Theta(n\log n)$. The algorithm works as follows:

- Split the array in half recursively until the partitions are of size 1 (base case).
- Merge the partitions such that they are sorted when merged together.
- Do this recursively until the entire list is merged back together.

Pseudocode:

```
func mergesort(list) {
    if (size(list) <= 1) return list
    else {
        leftList = mergesort(list[0..middle])
        rightList = mergesort(list[middle+1..size(list)-1])

        merge(leftList, rightList)
    }
}
```

Complexity:

At each step of the recursion, the list (of size n) is split into two halves of (roughly) size $\frac{n}{2}$.
Additionally, each merge step requires $n-1$ comparisons, so the overall time complexity is:

$T_n \leq n - 1 + T_{\frac{n}{2}} + T_{\frac{n}{2}}$

$T_n \leq n - 1 + 2T_{\frac{n}{2}}$

We can solve this recurrence with the master theorem. Here we have $a = 2$, $b = 2$, and $f(n) = n - 1$. Then we get:

$n^{log_2 2} = n$

Which is asymptotically equal to $n - 1$. Therefore the overall complexity is $\Theta(f(n) \cdot \log_b n) = \Theta(n\log_2 n)$, as expected.

Strassen Multiplication

Problem: *Given two $n \times n$ matrices, multiply them together, i.e. compute $A \times B = C$.*

Solution:

This operation can be done in a straightforward way that takes $\Theta(n^3)$ time by using the method learned in MATH 133. However, we can utilize a divide-and-conquer approach if we split A and B up into four quadrants, where each quadrant has size $\frac{n}{2} \times \frac{n}{2}$. Then, these quadrants can be used to compute the four quadrants of C in 8 recursive calls. Unfortunately, this also takes $\Theta(n^3)$ time, so it is no better than the traditional approach.

Using a carefully selected set of operations, it is possible to compute C using only 7 recursive calls. The operations are:

$$M_1 = (A_1 + A_4)(B_1 + B_4)$$

$$M_2 = (A_3 + A_4)B_1$$

$$M_3 = A_1(B_2 - B_4)$$

$$M_4 = A_4(B_3 - B_1)$$

$$M_5 = (A_1 + A_2)B_4$$

$$M_6 = (A_3 - A_1)(B_1 + B_2)$$

$$M_7 = (A_2 - A_4)(B_3 + B_4)$$

Assuming C_1 to C_4 represent the four quadrants of C, where C_1 is the top left quadrant, we can calculate C with the following equations:

$$C_1 = M_1 + M_4 - M_5 + M_7$$

$$C_2 = M_3 + M_5$$

$$C_3 = M_2 + M_4$$

$$C_4 = M_1 - M_2 + M_3 + M_6$$

Complexity:

The 7 recursive calls give a runtime of:

$$T_n \leq n^2 + 7T_{\frac{n}{2}}$$

Which evaluates to $\Theta(n^{log_2 7})$ by the master theorem.

Fast Exponentiation

The simple and naive way to calculate n^m is to simply multiply n by itself $m-1$ times, giving a complexity of $\Theta(m-1)$. However, the same result can be calculated faster using repeated squaring/the Russian Peasant algorithm. The steps to apply it are described below:

- Keep track of a variable $base$ and a variable $result$. Initialize $base$ to the base of the number being computed and initialize $result$ to 1. The final answer will be contained in $result$.
- At every step of the algorithm, square $base$. If the exponent is odd, then also multiply $result$ by the value of $base$ before it was squared. Then, halve the exponent (take the floor), and continue.
- Repeat until the exponent becomes 1, at which point $result$ will contain the final answer.

This can be done more visually by using two columns, with one containing $base$ and the other containing $result$. For example, to calculate 2^{13}:

Number	Base	Result/Accumulator
Start	2	1
2^{13}	$2^2 = 4$	$1 \cdot 2 = 2$
2^6	$4^2 = 16$	2
2^3	$16^2 = 256$	$2 \cdot 16 = 32$
2^1	$256^2 = 65,536$	$32 \cdot 256 = 8,192$

Where 8,192 is the final answer.

An alternate way of doing fast exponentiation is to follow the same process, but to look at the bits of the exponent instead. Going from left to right, a 1 bit signifies that the result should be squared and also multiplied by the base of the number, while a 0 bit signifies that the result should only be squared.

Psuedocode:

```
func fastExp(exponent, base, result) {
    if (exponent <= 1) return result
    else {
        if (exponent mod 2 = 1) { // exponent is odd
            return fastExp((exponent-1)/2, base*base, result*base)
        } else { // exponent is even
            return fastExp(exponent/2, base*base, result)
        }
    }
}
```

Complexity:

Since a number N only needs $logN$ bits to represent it, it is clear from the bit version of the algorithm that only $logN$ operations need to be done, hence $T_n = \Theta(\log n)$.

Euclid's Algorithm for GCD

Problem: *Given two numbers a and b, compute their greatest common denominator efficiently.*

This is a neat algorithm that most people will probably also see/have seen in MATH 240, but it was also mentioned in lectures for 251.

Solution:

Let $gcd(a, b)$ be a function that computes the GCD of two integers a and b. The algorithm for computing the GCD is as follows:

- Place the larger of the two numbers in the left slot of the function. For simplicity, assume that $a > b$ for the steps below.
- Compute $\frac{a}{b}$, and find the remainder r. If $r = 0$, then the GCD is b and the algorithm halts. If $r > 0$, then we call $gcd(b, r)$ and repeat until the remainder becomes zero.

Pseudocode:

```
func gcd(a, b) {
    remainder = a mod b

    if (remainder = 0) return b
    else {
        return gcd(b, remainder)
    }
}
```

Complexity:

The complexity is $T_n \leq 2log_2 n$ for $gcd(n, b)$, since n get halved every 2 calls. We also know that a number can only get halved $log_2 n$ times, and hence the algorithm must terminate with that many recursive calls in the worst case.

Linear Time Selection

Problem: *Given an unsorted set of elements A, return the k^{th} smallest element from the set.*

Solution:

The following is an algorithm created by Blum, Floyd, Pratt, Rivest and Tarjan that solves the problem above in guaranteed linear time. The steps of the algorithm can be summarized as follows:

- Group all of the elements in A into groups of 5. One group will have 5 or less elements if $|A|$ is not cleanly divisible by 5.

- In each group of 5, find the median element. This can always be done with 6 comparisons, i.e. in constant time.
- Group all of the medians into a new set M, and repeat the 2 steps above recursively. Once the size of the set becomes smaller than or equal to 5, then the median can be found directly. Ultimately the "median of medians" is returned by this recursive process.
- Once the median of medians is found (call it m), then it can be used as a pivot element. Each element in A is placed into one of two sets: L if the element is smaller than m, and R if the element is greater than m. If $|L| = k - 1$, then we can return m as the k^{th} smallest element. Otherwise, the algorithm will be called recursively on either L or R.
- Continue doing this until m is the desired element, or until the size of the set becomes ≤ 5.

Pseudocode:

```
func select(k, A) {
    if (size(A) <= 5) {
        sort(A) // In 6 comparisons
        return A[k] // Return the kth element of A
    } else {
        // Group all elements into groups of 5 and find the median of each
        M = group(A)

        // Find the median of medians
        // |M|/2 corresponds to the middle element of M
        m = select(size(M)/2, M)

        // Add the elements of A to either L or R
        for i = 0 to size(A)-1 do {
            if (A[i] < m) L.add(A[i])
            if (A[i] > m) R.add(A[i])
        }

        if (size(L) = k-1) {
            return m
        } else if (size(L) > k-1) {
            // The kth smallest element is somewhere in L
            return select(k, L)
        } else {
            // The kth smallest element is somewhere in R
            return select(k-size(L)-1, R)
        }
    }
}
```

Complexity:

The time complexity is $\Theta(n)$. The recurrence relation (when the algorithm is slightly optimized) is:

$T_n \leq \frac{8}{5}n + T_{\frac{n}{5}} + T_{\frac{7n}{10}}$

Quickselect

This algorithm, developed by Tony Hoare, provides an alternate method of finding the k^{th} smallest element. It has good average-case performance but its worst case performance is not great.

Quickselect works similar to Quicksort in that it selects a pivot and partitions the list elements into two (not necessarily equal-sized) halves, where one half contains all elements smaller than the pivot and the other contains all elements greater than the pivot. After partitioning, the pivot will be in its final sorted position, and so we know which half the k^{th} smallest is in. The algorithm then recurses on this half only.

Pseudocode:

```
func quickselect(list, k) {
    if (size(list) = 1) return list[0]
    else {
        // Select a pivot index, typically done at random
        pivotIndex = ...

        // Assume partition returns the pivotIndex's sorted location
        pivotIndex = partition(list, pivotIndex)

        if (pivotIndex = k) {
            return list[pivotIndex]
        } else if (pivotIndex < k) {
            quickselect(list[pivotIndex+1..size(list)-1], k)
        } else {
            quickselect(list[0..pivotIndex-1], k)
        }
    }
}
```

Complexity:

In the best and average case runtimes, Quickselect runs in $\Theta(n)$ time. However, in the worst case it runs in $\Theta(n^2)$ time. The algorithm's performance depends almost entirely on how the pivot is selected, and if this is done poorly (e.g. always selecting the first element in an already sorted list) the problem size will only decrease by 1 with each iteration, giving a runtime of $\Theta(n^2)$.

Binary Search

Problem: *Given a sorted list and a target value k, return the index of k in the sorted list or -1 if it is not in the list.*

This algorithm can be done fairly simply with a ternary oracle, but it can also be done with a binary oracle. We will assume the typical implementation where we have a ternary oracle that compares two elements x and y and tells us if $x < y$, $x = y$ or $x > y$.

Solution:

- Starting with the entire list, pick the middle element. If it is equal to k, then halt. If not, then go search the left half of the array if the middle element is greater than k and the right half if it is less than k.
- Repeat this until k is found or until a subarray of size 1 not equal to k is reached, at which point -1 should be returned.

Pseudocode:

```
// Recursive implementation
func binarySearch(list, k) {
    if (size(list) < 1) {
        return -1
    } else if (list[middle] = k) {
        return middle
    } else if (list[middle] > k) {
        return binarySearch(list[0..middle-1], k)
    } else {
        return binarySearch(list[middle+1..size(list)-1], k)
    }
}

// Iterative implementation
func binarySearch(list, k) {
    low = 0
    high = size(list)-1
    mid = (low+high)/2

    while (low <= high) {
        if (list[mid] = k) {
            return mid
        } else if (list[mid] < k) {
            // Go right
            low = mid+1
            mid = (low+high)/2
        } else {
            // Go left
            high = mid
            mid = (low+high)/2
        }
    }
    return -1
```

```
}
```

Complexity:

At each step of the algorithm, we make a call to a problem of size $\frac{n}{2}$ and also make use of the oracle, which takes constant time. Therefore:

$T_n \leq 1 + T_{\frac{n}{2}}$

We can determine the overall complexity using the master theorem. We have $a = 1, b = 2$ and $f(n) = 1$, which gives us:

$n^{log_2 1} = n^0 = 1$

Which is directly equal to $f(n)$. Hence the overall complexity is $\Theta(f(n) \cdot log_b n) = \Theta(log_2 n)$.