Algorithms

Short notes

Roudranil Das

June 18, 2022

Contents

1 Introduction **2**

2 Some short examples **2**
2.1 Fibonacci numbers . . . 2
2.2 GCD . . . 2

3 Computing runtimes **2**
3.1 Asymptotic notation . . . 2
3.2 Big-O notation . . . 3

4 Greedy Algorithms **3**
4.1 Covering points by segments . . . 4
4.2 Fractional Knapsack . . . 4

5 Divide and conquer **5**
5.1 Linear search . . . 5
5.2 Binary search . . . 6
5.3 Polynomial Multiplication . . . 6
5.4 Master Theorem . . . 7
5.5 Sorting problems: Selection and Merge . . . 8
5.5.1 Lower Bound for comparison based sorting . . . 9
5.5.2 Non comparison based sorting algorithms . . . 10
5.6 Quick Sort . . . 10
5.6.1 Random pivoting . . . 11
5.6.2 Equal elements? . . . 12

6 Dynamic Programming **13**
6.1 Money change . . . 13
6.2 String Alignment . . . 13
6.3 Discrete Knapsack . . . 15
6.3.1 With repetition . . . 15
6.3.2 Without repetition . . . 16
6.4 Parentheses placing . . . 16

7 Conclusion **18**

1 Introduction

Data structures and algorithms play a very important role.

Some tips learnt:

- Use stress testing.
- Stress testing is done by randomly generating vaild inputs, for large parameter values. Once error is obtained, start decreasing values to get a small case.

2 Some short examples

2.1 Fibonacci numbers

One of the most easiest ways to return the n^{th} Fibonacci number is recursive method. However this algorithm is extremely slow, mainly because while computing F_n we compute each F_{n-r} multiple times.

One way to get around this is to write down all Fib numbers, a list like method, as shown below:

```
F = []
F[0] = 0
F[1] = 1

for i from 2 to n:
    F[i] = F[i-1] + F[i-2]

return F[n]

```

The recursive one would need atleast F_n many iterations to compute F_n which would grow tremendously with valeus of n. The second one will not.

2.2 GCD

The naive way would be to run d from 1 to $a + b$, check if $d \mid a$ and $d \mid b$ and return the largest d. This is obviously slow. The faster way is to implement Euclidean algorithm recursively. The total number of steps required is $\approx Log(ab)$.

3 Computing runtimes

Runtime analysis of the Fibonacci code:

```
F = [] # memory management
F[0] = 0 # assignment
F[1] = 1 # assignment

for i from 2 to n: # increment, comparison, branch (condition)
    F[i] = F[i-1] + F[i-2] # lookup, assignment, addition of large integers

return F[n] # lookup, return from stack

```

So each line of code takes different amount of time. We want a new way to measure the runtime without knowing a large amount of metadata. For this we move on to asymptotic notation.

3.1 Asymptotic notation

All of these factors, usually although they change runtimes, they change it by a constant factor. So we measure runtime by ignoring these constants. We actually look at asymptotic runtime, where we measure how the runtime scales with the input size n. We see how it is proportional to some function of n. The order being:

$$\log(n) < \sqrt{n} < n < n\log(n) < n^2 < 2^n$$

3.2 Big-O notation

Definition: $f(n)$ is called *Big-O* of $g(n)$, i.e.,$f(n) = O(g(n))$ $(f \preceq g)$, if $\exists N \in \mathbb{N}$ and $c \in \mathbb{R}$ such that $\forall n \geq N, f(n) \leq c \cdot g(n)$.

f is bounded above by some multiple of g. Also base of logarithm doesnt matter as any base is essentially a constant multiple.

The disadvantage is that the constant multiple info is lost. So first improve the asymptotic situation. Then look into the smaller details. Also it may be the case that even if algo A is better in Big-O than algo B, for all practical purposes, algo B is better.

Some common rules:

1. Multiplicative contsants can be omitted. For example:
   $7n^3 = \mathrm{O}(n^3)$, $\frac{n^2}{3} = \mathrm{O}(n^2)$.
2. $n^a < n^b$ for $0 < a < b$:
   $n = \mathrm{O}(n^2)$, $\sqrt{n} = \mathrm{O}(n)$.
3. $n^a < b^n$ $(a > 0, b > 1)$:
   $n^5 = \mathrm{O}(\sqrt{2}^n)$, $n^{100} = \mathrm{O}(1.1^n)$
4. $(\log n)^a < n^b$ $(a, b > 0)$:
   $(\log n)^3 = \mathrm{O}(\sqrt{n})$, $n \log n = \mathrm{O}(n^2)$
5. If there is a sum of terms smaller terms in the sum can be omitted:
   $n^2 + n = \mathrm{O}(n^2)$, $2^n + n^9 = \mathrm{O}(2^n)$

Next we demonstrate Big-O in practice by analysing the runtime of the Fibonacci code:

Operation	Runtime
Create an array $F[0, 1 \ldots n]$	$\mathrm{O}(n)$
$F[0] \leftarrow 0$	$\mathrm{O}(1)$
$F[1] \leftarrow 1$	$\mathrm{O}(1)$
for i from 2 to n:	loop $\mathrm{O}(n)$ times
$F[i] \leftarrow F[i-1] + F[i-2]$	As integers are vvery large, $\mathrm{O}(n)$
return $F[n]$	$\mathrm{O}(1)$

Total: $\mathrm{O}(n) + \mathrm{O}(1) + \mathrm{O}(1) + \mathrm{O}(n) \cdot \mathrm{O}(n) + \mathrm{O}(1) = \mathrm{O}(n^2)$

There are some other notations too: Ω is used to give an approximate lower bound of the runtime, and Θ is used to give an approximately equal runtime.

Little-o notation o(): $f(n) = \mathrm{o}(g(n))$ or $f \prec g$ if $\frac{f(n)}{g(n)} \to 0$ as $n \to \infty$.

4 Greedy Algorithms

Greedy algorthms solve for local optimums in order to reach a global optimum. It involves breaking up a problem into a chunk of smaller problems, find a starting optimal solution, which will leave us with a smaller subproblem for which another optimal solution can be found by the same algorithm. Then we combine all the optimal solutions.

Main steps followed are:

- Make a first move
- Solve a problem of the same kind
- Then solve a subproblem of the same kind

A move is called a *safe move* if there is a optimal solution consistent with this move. Not all greedy moves or all first moves are safe.

4.1 Covering points by segments

Example: n children come to a party. Find the minimum number of groups to split the children into so that in any group the age of the children differ by at most 1.

Solution: A naive approach would be to consider all possible partitions of the set of all children and iterate over them to find the suitable partition, but it's complexity is $\Omega(2^n)$ which is not viable at all. So before anything we fomulate the problem mathematically. □

Covering points by segments

Input: A set of points $x_1, x_2, x_3, \ldots, x_n \in \mathbb{R}$
Output: The minimum number of segments of unit length needed to cover all the points.

Safe move: Cover the left most point with a segment whose left end is at that point. Then start with the first point left to the right of the segment, and continue with the same strategy.

Pseudocode: (assuming points are sorted in ascending)

PointsCoverSorted$(x_1, x_2, \ldots, x_n)$

$R \leftarrow \{\}, i \leftarrow 1$
while $i \leq n$:
    $[l, r] \leftarrow [x_i, x_i + 1]$
    $R \leftarrow R \cup \{[l, r]\}$
    $i \leftarrow i + 1$
    # Basically find the next point to the right of the current line segment
    while $i \leq n$ and $x_i \leq r$:
        $i \leftarrow i + 1$
return R

The time complexity is $O(n)$. In case points are not sorted, sort them by an algorithm of complexity $O(n \log(n))$. Thus final complexity will be $O(n \log(n))$.

4.2 Fractional Knapsack

Formulating in mathematical terms:

Fractional Knapsack

Input: Weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ of n items; capacity W.
Output: The maximum total value of fractions of items that fit into a bag of capacity W

We state the following lemma of the safe move:

Lemma

There exists an optimal solution that uses as much as possible of an item with the maximum value/unit of weight.

Greedy algorithm:

- while Knapsack is not full
- Choose item i with maximum $\frac{v_i}{w_i}$
- If item fits into the knapsack completely, then take all of it
- Otherwise take as much as to fll the knapsack entirely

- Return total value and amounts taken

The pseudocode is given as below:

Knapsack$(W, w_1, v_1, \ldots, w_n, v_n)$

$A \leftarrow [0, 0, \ldots 0], V \leftarrow 0$
repeat n times:
    if $W == 0$:
        return (V, A)
    select i with $w_i > 0$ and max $\frac{v_i}{w_i}$
    $a \leftarrow min(w_i, W)$
    $V \leftarrow V + a\frac{v_i}{w_i}$
    $w_i \leftarrow w_i - a, W \leftarrow W - a, A[i] \leftarrow A[i] - a$
return (V, A)

Complexit is O(n^2): why? Because selecting best item takes O(n) and loop is executed n times. **It is possible to optimise!! How?**

- Sort the array of $\frac{v}{w}$ in descending order
- In the above case we can skip the select statement
- after accounting for the sorting algorithm, we can sort in O($n \log n$)

5 Divide and conquer

Divide and conquer algorthms divide a problem into **non overlapping** subproblems of the **same type as of the OG problem**. We can recursively solve solutions.

One extremely simple example would be linear search in an unsorted array.

5.1 Linear search

Linear search

Input: An array with n elements and a key k
Output: An index i where $A[i] = k$. If there is no such i return NOT_FOUND.

Note: there might be multiple indices. Pseudocode:

LinearSearch$(A, low, high, key)$

if $high < low$:
    return `NOT_FOUND`
if $A[low] == key$:
    return low
return LinearSearch$(A, low + 1, high, key)$

The runtime analysis of such an algorithm can be made by a recursive relation like: $T(n) = T(n-1) + c$, c being some constant amount of work put in. Worst case would be $T(n) = n$(element not found) and best case would be $T(0) = c$(element in the first place). In linear search, we have mainly $\Theta(n)$ for linear search.

There is also an interative version for divide and conquer problems. One such is provided for linear search:

LinearSearch$(A, low, highkey)$

for i from low to $high$:
    if $A[i] == key$:
        return i
return `NOT_FOUND`

5.2 Binary search

Non trivial searching within a sorted array (ascending sort assumed).

Binary search

Input: A sorted array with $A[low, \ldots, high]$ $(\forall low \leq i < high : A[i] \leq A[i+1])$ and a key k
Output: An index i where $A[i] = k$. Otherwise the greates index i where $A[i] < k$. Otherwise $(k < A[low])$, the result is $low - 1$.

BinarySearch$(A, low, high, key)$

if $high < low$:
    return $low - 1$
$mid \leftarrow \lfloor \frac{high+low}{2} \rfloor$
if $A[mid] == key$:
    return mid
else if $key < A[mid]$:
    return BinarySearch$(A, low, mid - 1, key)$
else:
    return BinarySearch$(A, mid + 1, high, key)$

Runtime analysis of binary search: complexity is $\Theta(\log(n))$. We measure it by $T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + c$ in the worst case, and the best case is $T(0) = c$.

Similary we can devise an iterative implementation of binary search as well.

BinarySearch$(A, low, high, key)$

while $low \leq high$:
    $mid \leftarrow \lfloor \frac{high+low}{2} \rfloor$
    if $key == A[mid]$:
        return mid
    else if $key < A[mid]$:
        $high = mid - 1$
    else:
        $low = mid + 1$
return $low - 1$

5.3 Polynomial Multiplication

Divide and conquer algorthims can be used to implement fast algorithm for Multiplication of two polynomials. Multiplying polynomials are used in:

- Error correcting codes
- Large integer Multiplication

- generating functions
- Convolution in signal processing

Multiplying polynomials

Input: Two $n-1$ degree polynomials. (In case both have degree unequal, pad the smaller one with 0's with higher degree)
Output: Their product.

In this case we represent the polynomial $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ as the ordered tuple $(a_0, a_1, \ldots, a_n)$. So Basically we can represent any polynomial as an array of its coefficients. Then the naive algorithm would be two initialise a product array of all 0's, then run nested loops to evaluate each position there.

The naive divide and conquer algorithm is as follows: (requirement: if degree is $n-1$, then n should be a power of 2, pad otherwise). So Basically split $p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0$ into $p(x) = D_1(x)x^{\frac{n}{2}} + D_0(x)$ where,

$$D_1(x) = a_{n-1}x^{\frac{n}{2}-1} + a_{n-2}x^{\frac{n}{2}-2} + \cdots + a_{\frac{n}{2}}$$

$$D_0(x) = a_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + a_{\frac{n}{2}-2}x^{\frac{n}{2}-2} + \cdots + a_0$$

Do the same for the other polynomial too. Then their product will be $(D_1E_1)x^n + (D_1E_0 + D_0E_1)x^{\frac{n}{2}} + D_0E_0$ Recurrence time complexity: $T(n) = 4T\left(\frac{n}{2}\right) + kn$ Basically we are breaking each polynomial into two halves. Pseudocode:

Multi2(A, B, n, a_1, b_1)

$R = array[0, 1, 2 \ldots, 2n-2]$
if $n == 1$:
    $R[0] = A[a_1] \times B[b_1]$
    return R
$R[0, \ldots n-2] =$ `Multi2` $\left(A, B, \frac{n}{2}, a_1, b_1\right)$
$R[n, \ldots 2n-2] =$ `Multi2` $\left(A, B, \frac{n}{2}, a_1 + \frac{n}{2}, b_1 + \frac{n}{2}\right)$
$D_0E_1 =$ `Multi2` $\left(A, B, \frac{n}{2}, a_1, b_1\frac{n}{2}\right)$
$D_1E_0 =$ `Multi2` $\left(A, B, \frac{n}{2}, a_1 + \frac{n}{2}, b_1\right)$
$R[\frac{n}{2} \ldots n + \frac{n}{2} - 2] += D_1E_0 + D_0E_1$
return R

The time complexity of this is still $\Theta(n^2)$. The recurrence relation being: $T(n) = 4T\left(\frac{n}{2}\right) + \mathrm{O}(n)$. A faster divide and conquer algorithm will be provided below. This is known as the Karatsuba multiplication. A simplified example is provided:

Say we need to multiply the two polynomials: $A(x) = A_1(x)x + A_0$ and $B(x) = B_1(x)x + B_0$. By the previous algorithm, we would have written their multiplication as: $AB(x) = A_1B_1(x)x^2 + (A_1B_0(x) + A_0B_1(x))x + A_0B_0$. This requires **4** multiplications per call of the function. Rather than that, we would be doing $AB(x) = A_1B_1(x)x^2 + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)x + A_0B_0$. This we need to calculate only **3** multiplications per call, although we would have to perform two additions more. That is adjusted because its only a constant amount of work that is added. Thus this time complexity reduces to $\Theta(n^{1.58})$. The recurrence relation being: $T(n) = 3T\left(\frac{n}{2}\right) + \mathrm{O}(n)$.

5.4 Master Theorem

We need some sort of a theorem in order to quickly solve the recurrence relations which come out as a result of the divide and conquer algorithms. One such theorem is the master theorem, which states the following:

Master theorem

If $T(n) = aT\left(\lceil \frac{n}{b} \rceil\right) + \mathrm{O}(n^d)(a > 0, b > 1, d \geq 0)$, then:

$$T(n) = \begin{cases} \mathrm{O}(n^d) & \text{if } d > \log_b a \\ \mathrm{O}(n^d \log n) & \text{if } d = \log_b a \\ \mathrm{O}(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

5.5 Sorting problems: Selection and Merge

Note that we only consider sorting in ascending order.

First we look at a fairly inefficient but naive algorithm, the selection sort algorithm. This works on the following set of principles

- Find minimum and swap with the first element
- find minimum of the remaining element and swap with the second element
- continue

Pseudocode:

SelectionSort($A[1, \ldots, n]$)

```
for i from 1 to n:
        minIndex ← i
        for j from i + 1 to n:
                if A[j] < A[minIndex]:
                        minIndex ← j
        swap(A[i], A[minIndex])
```

The running time is obviously $\mathrm{O}(n^2)$. Actually number of execution should be $(n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$. Asymptotically however, it comes out to be $\Theta(n^2)$.

We propose an algorithm based on divide and conquer: *Merge Sort*. Principle behind this is as follows: Split an array into two equal parts, recursively sort them and then concatenate them, but with elements selected in order. Pseudocode is:

MergeSort($A[1, \ldots, n]$)

```
if n == 1:
        return A
m ← ⌊n/2⌋
B ← MergeSort(A[1, ..., m])
C ← MergeSort(A[m + 1, ..., n])
A' ← Merge(B, C)
return A'
```

The pseudocode for the merge function is given below:

Merge$(B[1,\ldots,p], C[1,\ldots,q])$

```
# Note that both arrays are sorted
D ← empty array of size p + q
while B and C are both non-empty:
        b ← the first element of B
        c ← the first element of C
        if b ≤ c:
                move b from B to the end of D
        else:
                move c from C to the end of D
move the rest of B and C to the end of D
return D
```

The running time of this algorthim is $\mathrm{O}(n\log(n))$. How so? Basically the merging time of the two arrays is obviously $\mathrm{O}(n)$. The two recursive calls are atmost $T\left(\frac{n}{2}\right)$. The recursively, the expression is

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \mathrm{O}(n)$$

We split the array exactly $\log_2 n$ times. At level i we do work $i \times c \times \frac{n}{i} = cn$ (c being the constant inside Big-O). Adding them we get $cn\log_2 n$. Adjusting the constants, the upper bound is found to be $\mathrm{O}(n\log(n))$.

5.5.1 Lower Bound for comparison based sorting

A **comparison based sorting algorithm** is one which sorts by comparing pairs of objects.
Example: merge and selection sort.

Lemma

Any comparison based algorithm performs $\Omega(n\log(n))$ comparisonsin the worst case to sort n objects.

Any comparison based sorting algorthim can be represented as a decision tree (which is a binary tree). The leaf nodes are some permutation of the original array. Hence, **the number of leaves l must be at least $n!$**. From this we can determine the tree depth. The worst running time will be the depth d of the tree (Basically the number of comparisons (or edges) required to reach the needed leaf). Now for a binary tree we have the following observation, $d \geq \log_2 l$ (or $2^d \geq l$). This is because maximum number of leaves in a tree of depth d is 2^d.

Lemma

$\log_2(n!) = \Omega(n\log(n))$

Proof

$$\begin{aligned}
\log_2(n!) &= \log_2(1\cdot 2\cdot 3\cdots n) \\
&= \log_2 1 + \log_2 2 + \ldots + \log_2 n \\
&\geq \log_2 \frac{n}{2} + \log_2\left(\frac{n}{2}+1\right) + \cdots \log_2 n \left(\text{taking the last } \frac{n}{2} \text{ terms}\right) \\
&\geq \frac{n}{2}\log_2\frac{n}{2} = \Omega(n\log(n))
\end{aligned}$$

5.5.2 Non comparison based sorting algorithms

Count Sort:
This works only for arrays of small postive integers. We basically count the number of occurances of each interger and then fill in a new array in the required order by filling in the intergers by the number of occurances. This sorts in linear time. Assuming that the elements of $A[1, 2, \ldots, n]$ are among the integers $1, \ldots, M$, we can provide the following pseudocode:

CountSort($A[1, 2, \ldots, n]$)

$Count[1, 2, \ldots, M] \leftarrow [0, 0, \ldots, 0]$
for i from 1 to n:
    $Count[A[i]] \leftarrow Count[A[i]] + 1$
$pos[1, 2, \ldots, M] \leftarrow [0, 0, \ldots, 0]$
$pos[1] \leftarrow 1$
The following loop basically stores the first position of i in the sorted array at $pos[i]$
for j from 2 to M:
    $pos[j] \leftarrow pos[j-1] + count[j-1]$
$B[1, 2. \ldots, n] \leftarrow [0, 0, \ldots, 0]$
for i from 1 to n:
    $B[pos[A[i]]] \leftarrow A[i]$
    $pos[A[i]] \leftarrow pos[A[i]] + 1$
first $A[i]$ is entered at the position of $A[i]$ and then the position is increased by 1 to show where the next occurance of the same value should be inserted.
return B

We provide the code too:

```
        def count_sort(A, M, n):
            count = np.zeros(shape=M, dtype=int)

            for i in range(n):
                count[A[i]] += 1
            pos = np.zeros(shape=M, dtype=int)
            pos[0] = 0

            for j in range(1, M):
                pos[j] = pos[j-1] + count[j-1]

            B = np.zeros(shape=n, dtype=int)
            for i in range(n):
                B[pos[A[i]]] = A[i]
                pos[A[i]] = pos[A[i]] + 1

            return B

```

The running time is $O(n + M)$. If $M = O(n)$, then, running time is $O(n)$

5.6 Quick Sort

This is also a sorting algorithm following the divide and conquer strategy, but this is one of the fastest and most efficient sorting algorithms. Its comparison based, and although running time on average is $O(n \log(n))$ its very efficient in practice.

In short, the algorthim picks a pivot. Then partitions the array such that all elements to the left of it is $\leq$ the pivot, and all elements to the right of it is $>$ the pivot. Obviously this means that the pivot is now in the correct position of where it should be in the sorted array. Then the two parts of the array (left and right) are sorted recursively.

How is the partitioning done? Well say the pivot is at index l and the element is $A[l]$. Then we move a counter i from $l+1$ to r (r being the index of the rightmost element allowed). We maintain two regions: $[l+1,\ldots,j]$ where all $A[k] \leq A[l]$ and $[j+1,\ldots,r]$ where all $A[k] > A[l]$. This is achieved by swapping elements if necessary.

The pseudocodes are as follows:

QuickSort(A, l, r)

```
if l ≥ r:
        return A
m ← Partition(A, l, r)
# m is the final position of the pivot A[l]
QuickSort(A, l, m − 1)
QuickSort(A, m + 1, r)
```

Partition(A, l, r)

```
x ← A[l] # Pivot
j ← l
for i from l + 1 to r:
        if A[i] ≤ x:
                j ← j + 1
                swap A[j] and A[i]
                # A[l + 1, ..., j] ≤ x, A[j + 1, ..., i] > x
swap A[l] and A[j]
return j
```

5.6.1 Random pivoting

Say sometimes we choose the minimum element always as the pivot. Then one of the partitions has all other elements always. In that case the time is $T(n) = n+T(n-1)$, ie, $T(n = n+(n-1)+(n-2)+\cdots+1 = \Theta(n^2))$ as seen before. (n is put because it is actually $O(n)$ to find the minimum element in the worst case). .

Say now we partition somehow that always into two parts of $n-5$ and 4. These are extremely **unbalanced partitions**. Then $T(n) = n + T(n-5) + T(4) \geq n + (n-5) + (n-10) + \cdots = \Theta(n^2)$ again.

Looking at **balanced paritions**, that is partitions are in roughly equal sizes, then we have: $T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n\log(n))$ as in Merge sort. Now we assume $T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + O(n)$ (unbalanced but still the time is logarithmic as previously. Only base of the logarithm is different).

In reality some partitions are balanced, some are unbalanced. The runtime depends on the balancedness of the paritions. So we need to randomly choose a pivot from the current subarray.

RandomizedQuickSort(A, l, r)

```
if l ≥ r:
        return A
k ← random number between l and r
swap A[l] and A[k]
m ← Partition(A, l, r)
# m is the final position of the pivot A[l]
RandomizedQuickSort(A, l, m − 1)
RandomizedQuickSort(A, m + 1, r)
```

Basically we can say that are always approximately or exactly $\frac{n}{2}$ elements in the sorted array in the middle, with $\frac{n}{4}$ elements each less and more than them. Thus when these elements are chosen as pivots we get a balanced partition. Thus the probability of getting a balanced parition is $\frac{1}{2}$.

We can state the following formally:

Theorem

Assuming that all the elements of $A[1, 2, \ldots, n]$ are pairwise different, then the average running time of the `RandomizedQuickSort(A)` is $O(n \log(n))$ and the worst case running time is $O(n^2)$.

5.6.2 Equal elements?

We make an observation first: Say we have an array A and A' is the sorted one. So given $i < j$ what is the probability that $A'[i]$ and $A'[j]$ are compared? Well if we choose any pivot which is greater or lesser than both or equal to any one of them then they are obviously compared. What happens when we choose one which is in between both? Then they are in different paritions and never compared.

So what if there are equal elements? That means for some $i < j$ all elements in $A'[i, \ldots, j]$ are equal. Thus choosing one in between them will not put them in different partitions. If all elements are equal then every paritions will be extremely unbalanced and the running time will be quadratic which is not something tha t we want.

Thus we make a change. Rather than returning a single point for the pivot m, we return two points (an interval) m_1, m_2 such that the following holds:

- for all $l \leq k \leq m_1 - 1, A[k] < x$
- for all $m_1 \leq k \leq m_2, A[k] = x$
- for all $m_2 + 1 \leq k \leq r, A[k] > x$

RandomizedQuickSort3(A, l, r)

```
if l ≥ r:
        return A
k ← random number between l and r
swap A[l] and A[k]
m1, m2 ← Partition3(A, l, r)
# m is the final position of the pivot A[l]
RandomizedQuickSort3(A, l, m1 − 1)
RandomizedQuickSort3(A, m2 + 1, r)
```

Partition3(A, l, r)

```
x ← A[l] # Pivot
low, high ← l
for i from l + 1 to r:
        if A[i] < x:
                low ← low + 1
                high ← high + 1
                temp ← A[i] ← A[high] ← A[low] ← temp
        else if A[i] == x:
                high ← high + 1
                swap A[high] and A[i]
swap A[l] and A[low]
return low, high
```

6 Dynamic Programming

6.1 Money change

We fill the array from left to right creating some sort of a lookup table.

DynamicMoneyChange($money, coins$)

$MinNumCoins[0, 1, 2, \ldots, money] \leftarrow [0, 0, \ldots, 0]$
for m from 1 to $money$:
    $MinNumCoins[m] \leftarrow m$
    for i from 1 to $len(coins)$:
        if $m \geq coins[i]$:
            $NumCoins \leftarrow MinNumCoins[m - coins[i]] + 1$
            if $NumCoins < MinNumCoins[m]$:
                $MinNumCoins[m] \leftarrow NumCoins$
return $MinNumCoins[money]$

6.2 String Alignment

Alignment Game

Remove all symbols from 2 given strings such that the numbe rof points is maximised. Points are awarded as follows:

- Remove first symbol from **both** strings
  - 1 point if symbols are same
  - 0 point if symbols are not same
- Remove first symbol from any of the strings
  - 0 points

Removal any symbol from any string is the same as pushing the other string one place ahead. Following is an example:

A	T	-	G	T	T	A	T	A
A	T	C	G	T	-	C	-	C

With the colours being: matches, insertions, deletions, mismatches. They are scored as follows: premium for every matches(+1), penalty for every mismatch (-μ) and indel(-σ).

$$\text{Alignment score} = \#matches - \mu \cdot \#mismatch - \sigma \cdot \#indel$$

Optimal alignment

INPUT: two strings, μ and σ
OUTPUT: an alignment of the string maximising the score.

Matches in the alignment string form a common subsequence.

Longest common subsequence

INPUT: two strings
OUTPUT: a longest common subsequence.

This is analogous to maximising score of alignment with $\mu = 0, \sigma = 0$.

Edit distance

INPUT: two strings
OUTPUT: the minimum number of elementary ops (insertions, deletions, substitution of symbols) to transform one string to another.

The minimum number of all such operations is called edit distance. One example is:

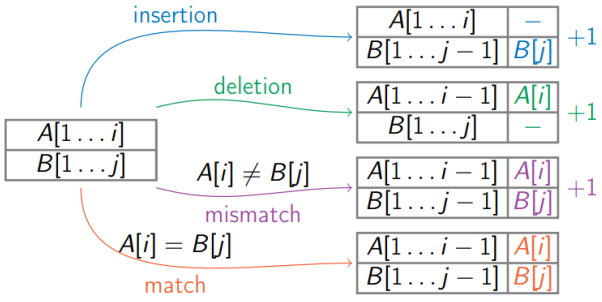
E	D	I	–	T	I	N	G	–
–	D	I	S	T	A	N	C	E

the total number of symbols in two strings=

$$\begin{array}{l} +2\cdot\#\text{matches} \\ +2\cdot\#\text{mismatches} \\ +1\cdot\#\text{insertions} \\ +1\cdot\#\text{deletions} \end{array} = \begin{array}{l} +2\cdot\#\text{matches} \\ -1\cdot\#\text{insertions} \\ -1\cdot\#\text{deletions} \end{array} \Bigg] \; 2\cdot\text{AlignmentScore}\ (\mu=0,\sigma=1/2) \; + \; \Bigg[\begin{array}{l} +2\cdot\#\text{mismatches} \\ +2\cdot\#\text{insertions} \\ +2\cdot\#\text{deletions} \end{array} \Bigg] \; 2\cdot\text{EditDistance}$$

That is number of symbols = c_1·edit distance + c_2·alignment score. Thus minimising edit distance $\Leftrightarrow$ maximising alignment score.

So how to implement dynamic programming? Basically notice that while working with two strings $A[1,\ldots,n]$ and $B[1,\ldots,m]$, say we want to know the optimal alignment of i-prefix of A(*i.e.*, $A[1,\ldots,i]$) and j-prefix of B(*i.e.*, $B[1,\ldots,j]$). Then the last column is one of insertion, deletion, mismatch or match. Removing that, all other previous columns, will be basically optimal arrangement of the previous part. Let $D(i,j)$ be the edit distance of $A[1,\ldots,i]$ and $B[1,\ldots,j]$. Then,



$$D(i,j) = \min \begin{cases} D(i,j-1)+1 & \\ D(i-1,j)+1 & \\ D(i-1,j-1)+1 & \text{if } A[i] \neq B[j] \\ D(i-1,j-1) & \text{if } A[i] = B[j] \end{cases}$$

DynamicEditDistance$(A[1,\ldots,n], B[1,\ldots,m])$

$D[i,0] \leftarrow i$ and $D[0,j] \leftarrow j$ for all i,j
for j from 1 to m:
    for i from 1 to n:
        $insertion \leftarrow D[i,j-1]+1$
        $deletion \leftarrow D[i-1,j]+1$
        $match \leftarrow D[i-1,j-1]$
        $mismatch \leftarrow D[i-1,j-1]+1$
        if $A[i] == B[j]$:
            $D[i,j] \leftarrow min(insertion, deletion, match)$
        else:
            $D[i,j] \leftarrow min(insertion, deletion, mismatch)$
return $D[n,m]$

Now to compute the optimal alignment:

DynamicOutputAlignment(i,j)

if $i == 0$ and $j == 0$:
    return
if $i > 0$ and $D[i,j] = D[i-1,j]+1$:
    `DynamicOutputAlignment`$(i-1,j)$
    print $\begin{array}{|c|}\hline A[i] \\ \hline \text{-} \\ \hline \end{array}$
else if $j > 0$ and $D[i,j] = D[i,j-1]+1$:
    `DynamicOutputAlignment`$(i,j-1)$
    print $\begin{array}{|c|}\hline \text{-} \\ \hline B[j] \\ \hline \end{array}$
else:
    `DynamicOutputAlignment`$(i-1,j-1)$
    print $\begin{array}{|c|}\hline A[i] \\ \hline B[j] \\ \hline \end{array}$

6.3 Discrete Knapsack

We have already done fractional knapsack. Discrete knapsack is basically where we either take whole of an item or we dont take an item. In fractional, we could have taken any amount of that item. (Think spices for fractional, but computers for discrete). There are two types:

1. Without repetition: There is only one copy of every item
2. With repetition: There is infinite number of copies of every item

This problem is not solvable with greedy algorithm. We solve it by dynamic programming.

6.3.1 With repetition

Let value(w) be the maximum value of a knapsack of max weight w. Then we state the following:

$$\text{value}(w) = \max_{i:w_i \leq w} \{\text{value}(w-w_i) + v_i\}$$

Then:

DynamicDKnapsackWR($W, n, w[1, \ldots, n], v[1, \ldots, n]$)

```
value[0] ← 0
for x from 1 to W:
        value[x] ← 0
        # computing the maximum as shown in the recurrent formula
        for i from 1 to n:
                if w[i] ≤ x:
                        val ← value[x − w[i]] + v[i]
                        if val > value[x]:
                                value[x] ← val
return value[W]
```

The running time is $O(nW)$. However even if it looks polynomial, it is not. Because running time is not proportional to W but to $\log(W)$. Thus the updated time is $O(n2^{\log W})$. So it is actually very slow for large W.

6.3.2 Without repetition

Say we have an optimal solution for a knapsack of max wt W. Then there are two cases:

1. The n^{th} item is present. If yes then if we remove it we have an optimal solution for a knapsack of max wt $W - w_n$ with items $i = 1, \ldots, n-1$
2. The n^{th} item is absent. Thus it is the optimal solution for a knapsack of max wt W with items $i = 1, \ldots, n-1$

Let $value(w, i)$ denote the maximum value of a knapsack of max weight w with items $k = 1, \ldots, i$. Then we state the following:

$$value(w, i) = \max\{value(w - w_1, i - 1), value(w, i - 1)\}$$

Thus we have the following:

DynamicDKnapsackWoR($W, n, w[1, \ldots, n], v[1, \ldots, n]$)

```
value[0, j] ← 0 for all j
value[w, 0] ← 0 for all w
for i from 1 to n:
        for x from 1 to W:
                value[x, i] ← value[x, i − 1]
                if w[i] ≤ x:
                        val ← value[x − w[i], i − 1] + v[i]
                        if val > value[x, i]:
                                value[x, i] ← val
return value[W, n]
```

6.4 Parentheses placing

Our next problem is basically maximising the value of an arithmetic expression by suitably placing parentheses in them. Illustrating with an example, say we want to maximise the value of the expression $1+2-3\times4-5$. We give the following combinations:

- $((((1 + 2) - 3) \times 4) - 5) = -5$ (obviously not maximum)
- $((1 + 2) - ((3 \times 4) - 5)) = -4$ (still not maximum)
- $((1 + 2) - (3 \times (4 - 5))) = 6$ (now its maximum)

A brute force algorithm for an expression with n arithmetic operations will take about $n!$ time which is very slow. So first we formulate the problem:

Placing parentheses

INPUT: A sequence of digits $d_1, d_2, \ldots, d_n$ and a sequence of operations $op_1, op_2, \ldots, op_{n-1} \in \{+, -, \times\}$
OUTPUT: An order of applying these operations that maximises the value of the arithmetic expression $d_1 op_1 d_2 op_2 \cdots d_{n-1} op_{n-1} d_n$

We also need the subproblems. So say op_k is the last operation to be performed. That we means we have two sub-expressions: $d_1 op_1 \cdots d_{k-1}$ and $d_{k+1} \cdots d_n$. Now we need to find the max and min values of both these sub-expressions. Why? Because if op_k is addition, then we basically need max values of both sub-expressions. If op_k is subtraction, we need max of the first one, and min of the second one. If it is multiplication, then we need maximum of the product of max and the product of min.

Let $E_{i,j}$ be the sub-expression $d_i op_i \cdots op_{j-1} d_j$, and let $M(i,j) = \max(E_{i,j})$ and $m(i,j) = \min(E_{i,j})$. Say $E_{i,i} = d_i$. Then we have the following recurrence relations:

$$M(i,j) = \max_{i \le k \le j-1} \begin{cases} M(i,k)\ op_k\ M(k+1,j) \\ M(i,k)\ op_k\ m(k+1,j) \\ m(i,k)\ op_k\ M(k+1,j) \\ m(i,k)\ op_k\ m(k+1,j) \end{cases}$$

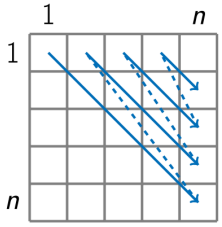
$$m(i,j) = \min_{i \le k \le j-1} \begin{cases} M(i,k)\ op_k\ M(k+1,j) \\ M(i,k)\ op_k\ m(k+1,j) \\ m(i,k)\ op_k\ M(k+1,j) \\ m(i,k)\ op_k\ m(k+1,j) \end{cases}$$

First we need to compute min and max for $E_{i,j}$, which we do by:

MinAndMax(i, j)

$min \leftarrow +\infty$
$max \leftarrow -\infty$
for k from i to $j-1$:
    $a \leftarrow M(i,k) \quad op_k \quad M(k+1,j)$
    $b \leftarrow M(i,k) \quad op_k \quad m(k+1,j)$
    $c \leftarrow m(i,k) \quad op_k \quad M(k+1,j)$
    $d \leftarrow m(i,k) \quad op_k \quad m(k+1,j)$
    $min \leftarrow \min(min, a, b, c, d)$
    $max \leftarrow \max(max, a, b, c, d)$
return (min, max)

When computing $M(i,j)$ all the values of $M(i,k), M(k+1,j), m(i,k), m(k+1,j)$ should be already computed. We will be solving subproblems in the increasing order of $(j-i)$. This is the order:



Finally:

Parentheses$(d_1 op_1 d_2 op_2 \cdots d_{n-1} op_{n-1} d_n)$

for i from 1 to n:
    $m(i,i) \leftarrow d_i, M(i,i) \leftarrow d_i$
for s from 1 to $n-1$:
    for i from 1 to $n-s$:
        $j \leftarrow i+s$
        $m(i,j), M(i,j) \leftarrow$ `MinAndMax`(i,j)
return $M(1,n)$

7 Conclusion

Sayonara.