(1) Recurrence sequences

(a) $T(n) = 3T(\sqrt{n}) + \log_2 n$

Let's substitute $n = 2^m$ so we get rid of the logarithmic term:

$$T(2^m) = 3T(2^{m/2}) + m$$

Now let's define $S(m) = T(2^m)$, so hence;

$$S(m) = 3S(m/2) + m$$

Proof for the above definition:

We will show that m solves for $S \leftrightarrow 2^m$ solves for T

(1) Assume m solves for S:

Then $S(m) - 3S(m/2) - m = 0$

and since $T(2^m) = S(m)$, we also have:

$T(2^m) - 3T(2^{m/2}) - m = 0$; Then 2^m solves T

(2) Assume 2^m solves T:

Then $T(2^m) - 3T(2^{m/2}) - m = 0$

and since $S(m) = T(2^m)$, we also have:

$S(m) - 3T(m/2) - m = 0$; Then m solves S

This shows that the recurrence relations S & T are equivalents in terms of arguments.

$S(m) = 3S(m/2) + m$ can now easily be solved using the Master Theorem!

$c = \log_2 3$ & setting $k = 1$

Since $k < c$, by the Master Theorem;

$$S(m) = \Theta(m^{\log_2 3})$$

Now substituting $m = \log_2 n$;

$$T(2^{\log_2 n}) = \Theta((\log_2 n)^{\log_2 3})$$

So therefore: $T(n) = \Theta((\log_2 n)^{\log_2 3})$

(b)

$$T(n) = T(n-1) + T(n/2) + n$$

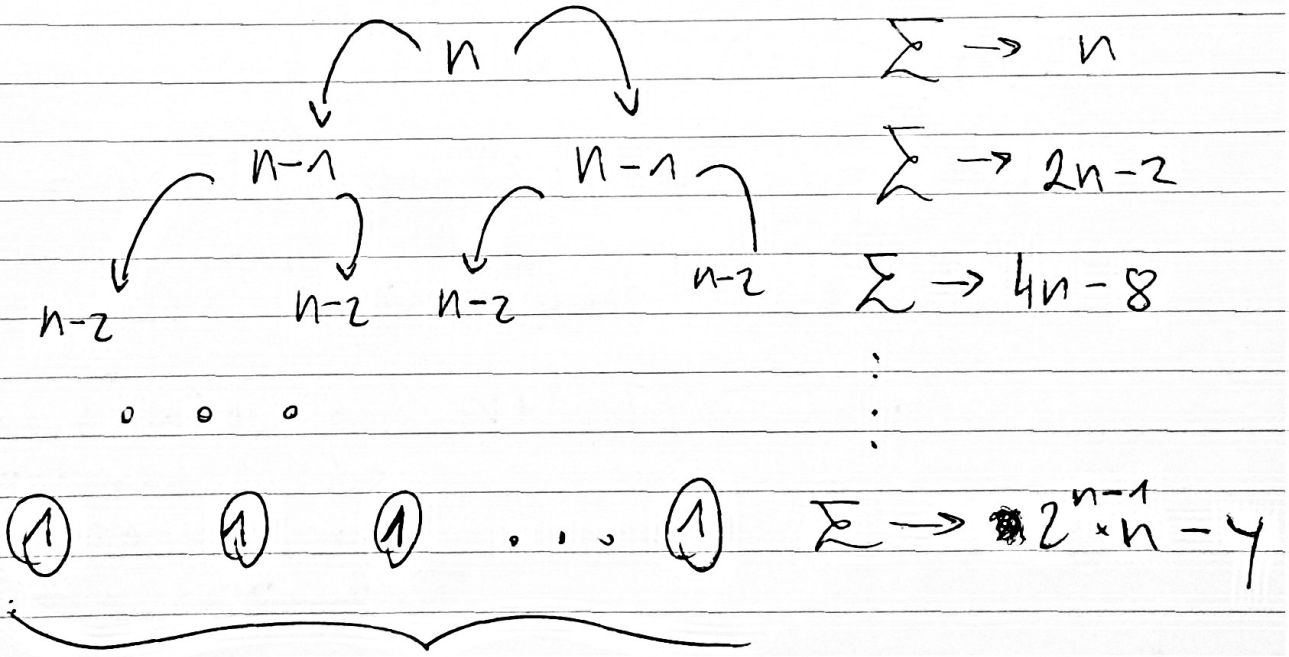
Let's define $T'(n)$ such that:

$$T'(n) = T(n-1) + T(n-1) + n =$$
$$= 2T(n-1) + n$$

So hence we know that:

$$T'(n) \geq T(n)$$

If we analyse algorithm $T'(n)$



So we know for the principles of the binary tree that there 2^{n-1} nodes at level $n-1$

Since we have $n-1$ levels and if I generalise that at each level there is exactly $2^{n-1}n$ being done, then:

$$T'(n) = \Theta(2^{n-1}n^2)$$

Since we know that $T'(n) \geq T(n)$ I also claim that $T(n) = O(n^2 2^{n-1})$. I will prove that $T(n) \leq cn^2 2^{n-1}$ for some constants $c > 0$

$$T(n) = T(n-1) + T(n/2) + n$$

$$T(n-1) \leq c(n-1)^2 2^{n-2}$$

$$T(n/2) \leq c \times (n/2)^2 2^{n/2-1} = cn^2 2^{n/2-3}$$

Now by using substitution method:

$$T(n) \leq c(n-1)^2 2^{n-2} + cn^2 2^{n/2-3} + n =$$

$$= cn^2 2^{n-2} - c2n2^{n-2} + c2^{n-2} + cn^2 2^{n/2-3} + n =$$

$$= cn^2 2^{n-2} - cn2^{n-1} + c2^{n-2} + cn^2 2^{n/2-3} + n \leq$$

$cn2^{n-1} \geq c2^{n-2} + n$ so therefore we can get rid of these three terms since they are not significant as $n \to \infty$

$$\leq cn^2 2^{n-2} + cn^2 2^{n/2-3} \leq$$

$cn^2 2^{n/2-3} \to cn^2 2^{n-2}$

$$\leq 2cn^2 2^{n-2} = \underline{cn^2 2^{n-1}}$$

This holds for every $n \geq 0$ with the following base cases:

$n = 0$; $T(0) = 0$

$n = 1$; $T(1) = 0$

(6) Strassen's Algorithm:

$$W(n) = 7W(n/2) + O(n^2) \Rightarrow O(n^{\log_2 7})$$

My Asymptotically faster Algorithm:

$$M(w) = aT(n/4) + O(n^2) \Rightarrow O(n^x)$$

$$x < \log_2 7$$

$$\downarrow$$

$$\log_b a < \log_2 7 \quad (b=4)$$

$$\log_4 a < \log_2 7$$

$$a < 4^{\log_2 7} = 49$$

$$\underline{a < 49}$$

This algorithm needs less than 49 subproblems in order to be asymptotically faster than strassen's algorithm.

(2) DFT

(a) Compute the DFT of the vector $(0,1,2,5)$

$DFT(4,[0,1,2,5])$

$\rightarrow A_{even} = [0,2]$

$\rightarrow A_{odd} = [1,5]$

$DFT(2,[0,2])$
$\rightarrow A_{even} = [0]$
$\rightarrow A_{odd} = [2]$

$DFT(2,[1,5])$
$\rightarrow A_{even} = [1]$
$\rightarrow A_{odd} = [5]$

$DFT(1,[0])$ — Base case $\rightarrow [0]$

$DFT(1,[2])$ — Base case $\rightarrow [2]$

$DFT(1,[1])$ — Base case $\rightarrow [1]$

$DFT(1,[5])$ — Base case $\rightarrow [5]$

$DFT(2,[0,2]) = [2,-2]$

$DFT(2,[1,5]) = [6,-4]$

$DFT(4,[0,1,2,5]) = [8, -2-4i, -4, -2+4i]$

(b)

Polynomial $A(x)$ of degree n:

$$A(x) = q(x)(x - x_0) + r$$

$$A(x) - r = q(x)(x - x_0)$$

Let's introduce algorithm ~~Coefficient-At~~ Value-At $((x_0), [a_0, \ldots, a_{n-1}])$ that can generate value of the polynomial at x_0

Algorithm Value-At$((x_0), [a_0, \ldots, a_{n-1}])$

```
1)    power = 1
2)    value = 0
3)    for i=0 to n:
4)        value = value + a[i] * power
5)        power += 1
6)    return value
```

Analysis of Value-At$((x_0), [a_0, \ldots, a_{n-1}])$ algorithm:

Lines 1) and 2) get executed in constant time. Line 3) gets executed $n+1$ times. The loop gets executed n times, each execution repeats lines 4) and 5), which are constant work. Lastly, line 6) gets executed in constant time by returning the value. Therefore:

$$T(n) = \Theta(n+1) + \Theta(n) = \Theta(n)$$

We can easily compute coefficients of $A(x)-r$ by subtracting r from coefficients.

Since $r = \text{Value-At}((x_0), [a_0 \dots a_{n-1}])$ we can perform the following subtraction:

$$[a_0, a_1, \dots, a_n]_x - [r, 0, 0 \dots, 0] = [a_0 - r, a_1 \dots a_n],$$

which can be implemented in constant time. Now we have $A(x) - r = Q(x)(x - x_0)$ and $Q(x) = \frac{A(x)-r}{x-x_0}$

Now by performing long-division of $A(x) - r$ by $(x - x_0)$ we get the following recurrence for coefficients $Q(x)$:

$$c_{n-2} = a_{n-1+0} = a_{n-1}$$

$$c_{n-2-i} = a_{n-2-i+1} + x_0 c_{n-2+1-i} =$$

$$= a_{n-1-i} + x_0 c_{n-1-i}$$

where c represents the coefficient of factor x^{n-i} of polynomial $Q(x)$

This can easily be converted into an algorithm:

1) $Q = C[c_0 \dots c_{n-2}]$
2) $c[n-2] = a[n-1]$
3) for $i = 1$ up to $n-2$:
4) $\quad c[n-2-i] = a[n-1-i] + x_0 c[n-1-i]$

This loop gets executed $n-2$ times, conducting 5 constant work, hence $\Theta(n)$

This proves that the coefficients of $Q(x)$ (array C) can be computed in linear time.

(C)

6. Given list of zeros of polynomial, compute a polynomial in $O(n \lg^2 n)$

Let's define an algorithm Compute-Poly, which in-takes a list of zeros and outputs, coefficients list of the polynomial.

Compute-Poly $([z_0, z_1, \ldots, z_{n-1}])$

1) if len $[z_0, z_1, \ldots, z_{n-1}] = 1$
2) return $[-z_0, 1]$

3) $z_{first} = [z_0 \ldots z_{\lceil \frac{n-1}{2} \rceil}]$
4) $z_{second} = [z_{\lceil \frac{n-1}{2} \rceil} \ldots z_{n-1}]$

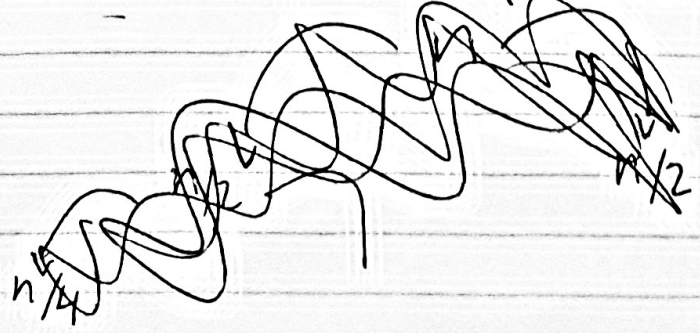
5) a = Compute-Poly (z_{first})
6) b = Compute-Poly (z_{second})

7) return DFT-Multiplication $((a, b), n+1)$

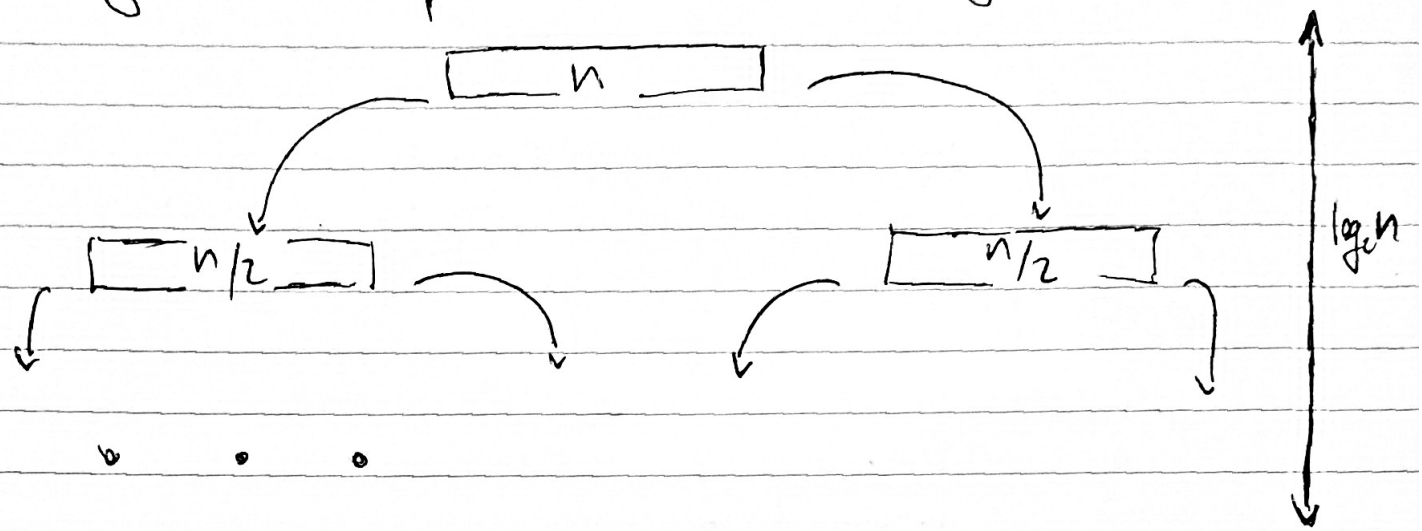
DFT-Multiplication was defined during lectures and runs in $O(n \lg n)$. It multiplies polynomials and computes a polynomial bound by $n+1$.

(Discrete Fourier Transform)

~~since we compute Compute-Poly~~



Using the tree presentation of this algorithm:



We can see that there are $\log_2 n$ levels and at each level the algorithm performs $n \log_2 n$ of work. Hence, in total Compute-Poly $= O(n \lg^2 n)$

So hence algorithm Compute-Poly is indeed $O(n \lg^2 n)$

Now let's prove that Compute-Poly indeed computes polynomial $P(x)$ of degree-bound $n+1$.

Base case: Let $[z_0]$ be a single root.
Compute-Poly $([z_0])$ returns $[-z_0, 1]$, which coresponds to $P(x) = -z_0 + x$.
$P(z_0) = z_0 - z_0 = 0$ ✓
$\deg(P(x)) = 1 < n+1 = n+1 = 2$

So hence it holds true for the base case.

(a)

So we know that two sets A and B are in range of 0 to 10n.
We wish to compute the Cartesian sum of A and B, defined by:

$$C := \{\{x+y \mid x \in A \text{ and } y \in B\}$$

where integers of C are in range of 0 to 20n

If we translate this problem into polynomial multiplication we can apply the DFT algorithm and solve efficiently.
Let's suppose set A represents powers of the polynomial (as well as set B) where the coefficients (either a 0 or a 1) tells us if the power (= element) is present in the set.

Once we perform the multiplication of polynomials we can easily analyse the resulting polynomial. The non-zero coefficients of the terms are the frequency of elements in C, and the power of non-zero-coefficient terms are the elements in C.