CS512 LECTURE NOTES - LECTURE 3

1 Summations

There are several summations that we will need to use a lot when analyzing algorithms:

1. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

   proof:

S=	1	+	2	+	3	+	...	+	n
S=	n	+	n-1	+	n-2	+	...	+	1
2S=	(n+1)	+	(n+1)	+	(n+1)	+	...	+	(n+1)

   So we have that $2S = n(n+1)$. Solving for S we get:

$$S = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

2. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ for all $n \geq 0$

   proof by induction on n

   BASIS: If $n = 0$ then the left hand side of what we want to prove is equal to 1, and the right hand side is $\frac{1(1+1)(2+1)}{6} = 1$

   INDUCTIVE HYPOTHESIS.
   Assume that

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

   is true for a fixed positive integer k

INDUCTIVE STEP. We need to show that for $k+1$ the property is true, this is

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+2)(2k+3)}{6} = \frac{(k+1)(2k^2+7k+6}{6}$$

Starting from the left hand side:

$$\sum_{i=1}^{k+1} i^2 = \sum_{i=1}^{k} i^2 + (k+1)^2$$

$$= \frac{k(k+1)(2k+1)}{6} + (k+1)^2 = (k+1)\left(\frac{k(2k+1)}{6} + k + 1\right)$$

$$= (k+1)\left(\frac{k(2k+1)+6k+6}{6}\right)$$

$$= \frac{2k^2+7k+6}{6}$$

Therefore the property is true for all $n \geq 1$

3. $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

4. $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$

proof.

$S =$	a^0	$+$	a^1	$+$	a^2	$+$	a^3	$+\ldots+$	a^n		
$aS =$			a^1	$+$	a^2	$+$	a^3	$+\ldots+$	a^n	$+$	a^{n+1}
$aS - S =$	-1									$+$	a^{n+1}

solving for S we have:

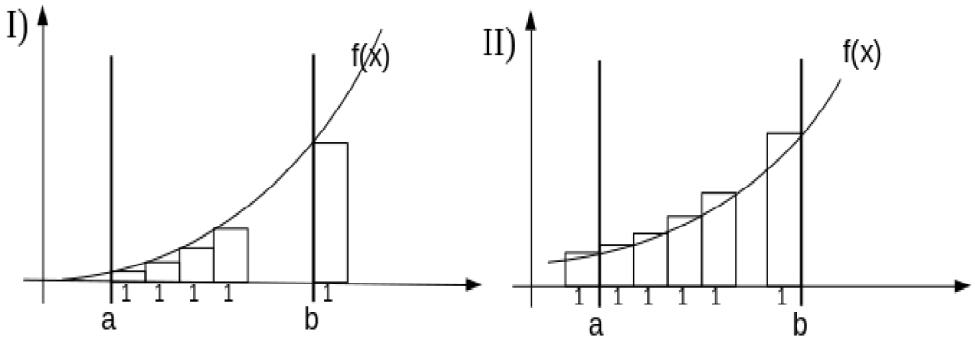
$$S = \sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

2 Using Integrals

Integrals can be used to find bounds for summations. Notice that in the following figure.

Since each rectangle has a width equal to 1 and height given by f(x), we can use the area under f(x) as a bound for the summation in the following way:;

INCREASING FUNCTIONS:



Case I):

$$\sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)dx$$

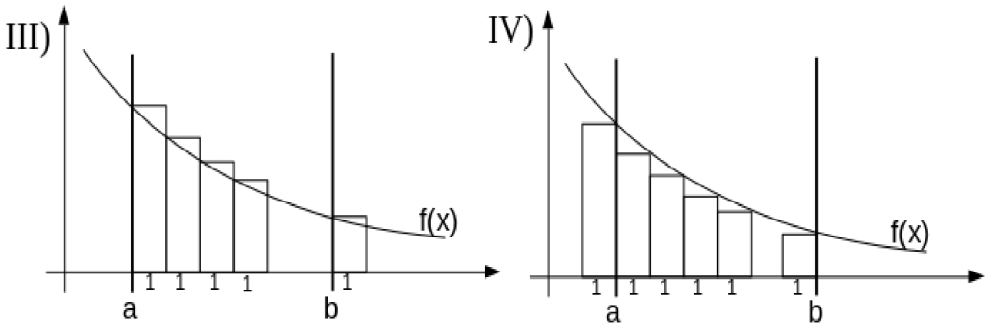
Case II):

$$\int_{a-1}^{b} f(x)dx \leq \sum_{i=a}^{b} f(i)$$

Therefore, in the case of increasing functions we have that

$$\int_{a-1}^{b} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)dx$$

DECREASING FUNCTIONS:



Case III):

$$\int_a^{b+1} f(x)dx \leq \sum_{i=a}^{b} f(i)$$

Case IV):

$$\sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)dx$$

Therefore, in the case of decreasing functions we have that

$$\int_a^{b+1} f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)dx$$

Examle

Find an upper bound for $\sum_{i=1}^{n} \frac{1}{i}$ (harmonic series).

Since $\frac{1}{i}$ is decreasing we would have to evaluate the integral from 0 to n, since the integral is $\ln x$, we cannot compute it when $x = 0$, so we will start the summation from 2:

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i} \leq 1 + \int_1^n \frac{dx}{x} = 1 + \ln x \,|_1^n = 1 + \ln n$$

So we have that $\sum_{i=1}^{n} \frac{1}{i} \in O(\ln n)$

Design techniques

There are several techniques that help us to devise new algorithms. We will talk about them in this course including:

- Divide and conquer
- Dynamic programming
- Greedy algorithms

3 Recurrence Relations

3.1 Divide and Conquer

The divide and conquer strategy is used to find algorithms to problems that can be broken into smaller pieces:

1. Divide the problem into smaller pieces
2. Recursively solve each one of the pieces
3. Combine the solutions

Examples of divide and conquer algorithms include merge sort and quick sort. We will be using divide and conquer strategy in this course.

3.2 Binary number multiplication

Known algorithm

Suppose that we want to multiply two n-bit numbers, for the sake of simplicity we will assume that n is a power of 2, i.e. $n = 2^k$ for some integer k.

Let us recall the usual algorithm that is used to multiply two numbers:

```
          x x x x x x x x x x
          y y y y y y y y y y
         --------------------
       r r r r r r r r r r
     r r r r r r r r r r
   r r r r r r r r r r
  .
  .
  .
----------------------------
s s s s s s s s s s s s s s s s
```

Notice that the number of rows labeled "r" is equal to n, and each row has $n + 1$ elements. Therefore, just to compute them, requires $O(n^2)$ time.

Lower bound

We know that every algorithm that multiplies two n digit binary numbers, must at least write the answer. The answer is given by the row labeled "s" and can have at most $2n + 1$ digits. Therefore we can say that

ANY ALGORITHM THAT MULTIPLIES TWO n-bit BINARY NUMBERS MUST DO AT LEAST $\Omega(n)$ OPERATIONS.

Known algorithm	$O(n^2)$	UPPER BOUND
Every algorithm must do	$\Omega(n)$	LOWER BOUND

Notice that there is a **GAP** between the lower and the upper bound, which means that there is room for improvement, we can try to either:

- Find a better algorithm, which will decrease the upper bound
- Find a larger lower bound

Upper bound

In this case we will use the **Divide and Conquer** strategy to try to find a better algorithm.

A better algorithm.

Suppose that x and y are the number that we want to multiply. We are going to divide each one in half in the following way:

$$\begin{aligned} x &= 2^{n/2}x_1 + x_2 \\ y &= 2^{n/2}y_1 + y_2 \end{aligned}$$

Example: is like dividing $x = 3545$ in $x_1 = 35$ and $x_2 = 45$, since $x = 10^2 x_1 + x_2$ (example in base 10 for simplicity).

So we have that:

$$\begin{aligned} xy &= (2^{n/2}x1 + x2)(2^{n/2}y_1 + y2) \\ &= 2^n x_1 y_1 + 2^{n/2}(x_1 y_2 + x_2 y_1) + x_2 y_2 \end{aligned}$$

Notice that

$$(x_1 + x_2)(y_1 + y_2) = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2$$

From which we get:

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

So we can use the following process to compute xy.

1. $p_1 = x_1y_1$
2. $p_2 = x_2y_2$
3. $p_3 = (x_1 + x_2)(y_1 + y_2) - p_1 - p_2$
4. $xy = 2^n p_1 + 2^{n/2} p_3 + p_2$

Counting Operations

Notice that all multiplications times powers of 2 can be done by using "binary shift" operations, which will take $\Theta(n)$ time. All additions can be done in $\Theta(n)$ time too. The only "expensive" operation that we have is the multiplication operation.

Multiplication of two n-bit numbers will be handled recursively using the process described above. Notice that the total number of multiplications needed to compute xy is equal to 3 multiplications of numbers of size $n/2$. Therefore, total running time of our algorithm is given by the following recurrence relation:

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

Solving a recurrence relation by iteration

We will use iteration to solve the recurrence relation $T(n) = 3T(\frac{n}{2}) + O(n)$, assuming a base case $T(1) = 1$.

$$\begin{aligned}
T(n) &= 3T(\frac{n}{2}) + cn \\
\text{Iteration 1:} \quad &= 3(3T(\frac{n}{2} + c\frac{n}{2}) + cn \\
&= 3^2T(\frac{n}{2}) + \frac{3}{2}cn + cn \\
\text{Iteration 2:} \quad &= 3^2(3T(\frac{n}{2^2}) + c\frac{n}{2^2}) + \frac{3}{2}cn + cn \\
&= 3^3T(\frac{n}{2^2}) + (\frac{3}{2})^2cn + \frac{3}{2}cn + cn \\
\vdots \quad &\vdots \quad \vdots \\
T(n) &= 3^kT(\frac{n}{2^k}) + cn\sum_{i=0}^{k-1}(\frac{3}{2})^i
\end{aligned}$$

Since we said that n is a power of 2, we can iterate until

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad n = 2^k$$

So we have that

$$T(n) = 3^{\lg n}T(1) + cn\sum_{i=0}^{\lg n-1}(\frac{3}{2})^i$$

We know that

$$\sum_{i=0}^{r} a^i = \frac{a^{r+1}-1}{a-1}$$

Therefore

$$\begin{aligned}\sum_{i=0}^{\lg n-1}(\frac{3}{2})^i &= \frac{(\frac{3}{2})^{\lg n}-1}{\frac{3}{2}-1}\\ &= 2(\frac{3}{2})^{\lg n}-2\\ &= 2(\frac{3^{\lg n}}{n})-2\end{aligned}$$

Finally we compute $T(n)$

$$\begin{aligned}T(n) &= 3^{\lg n}+cn(2(\frac{3^{\lg n}}{n})-2)\\ &= n^{\log_3 2}+2cn^{\log_3 2}-2cn\\ &\in O(n^{\log_3 2})\end{aligned}$$

$T(n) \in O(n^{1.59})$ is better than $O(n^2)$. So we were able to improve the bit-multiplication algorithm by using the **Divide and Conquer Technique**

4 Master Theorem

The Master Theorem is used to solve recurrence relations of the form:

$$T(n) = aT(\frac{n}{b}) + f(n), \quad T(1) \in \Theta(1)$$

These recurrence relations usually appear when analyzing algorithms designed using the divide and conquer strategy. Since the time it takes for the algorithm to run on input size n is equal to a times the time it takes to recursively solve each piece of size n/b plus the time it takes to split and combine, given by $f(n)$.

The Master Theorem is helpful since we do not have to do all the iterative work and algebraic simplification that we did in our previous example, we just need to apply the appropriate case of the theorem. In order to use the Master Theorem we need to compare $n^{\log_b a}$ vs $f(n)$.