

Analysis of 2D Maxima Algorithm

(Class 4)

MAXIMA (int n, Point P[1 ... n])

1 For i ← 1 to n n times

2 Do *maximal* ← true

3 For j ← 1 to n n times

4 Do

5 If (i ≠ j) and (P[i].x ≤ P[j].x) and (P[i].y ≤ P[j].y) 4 memory accesses

6 Then *maximal* ← false

7 Break

8 If (*maximal* = true)

9 Then output P[i].x, P[i].y 2 memory accesses

- We want to calculate worst case scenario.
- We have pair of nested summations, one for i-loop and the other for the j-loop.

$$T(n) = \sum_{i=1}^n \left(2 + \sum_{j=1}^n 4 \right)$$

- We counted the total number of times the memory is accessed in both inner and outer loops.

Worst Case Running Time of 2D Maxima Algorithm

$$T(n) = \sum_{i=1}^n \left(2 + \sum_{j=1}^n 4 \right)$$

$\sum_{j=1}^n 4 = 4n$, and so

$$T(n) = \sum_{i=1}^n (2 + 4n)$$

$$T(n) = n (2 + 4n) = \mathbf{4n^2 + 2n}$$

- So, we get a polynomial after calculating the summations.

- For small values of n , any algorithm is fast enough.
- What happens when n gets large?
- Running time does become an issue.
- When n is large, n^2 term will be much larger than the n term and will dominate the running time.

- Here $T(n)$ is the function that gives the running time of the algorithm.
- We can relate the value of the $T(n)$ with the seconds or minutes.
- For example, one memory access takes 1 millisecond.
- And we provided the input data of 100 cars to the algorithm ($n = 100$).

- The execution time will be:

$$T_{worst}(100) = 4(100)^2 + 2(100)$$

$$T_{worst}(100) = 40200 \text{ ms}$$

$$T_{worst}(100) = \mathbf{40.2 \text{ seconds}}$$

- Note that if we ignore the $2n$ term in the equation,
- Then in the answer, we will see the difference of only 200 milliseconds.
- Such a small value is negligible as compared to the $4n^2$ (i.e., 40000 milliseconds).
- That is why we said that the $4n^2$ term dominates.

- We will say that the worst-case running time is $O(n^2)$.
- This is called the asymptotic growth rate of the function.
- We will discuss this O-notation (Big-O) more formally later.

- Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
- Big O is a member of a family of notations invented by Paul Bachmann and others.
- The letter O was chosen by Bachmann to stand for *Ordnung*, meaning the order of approximation.

Example 1

- Associate a “cost” with each statement.
- Find the “total cost” by finding the total number of times each statement is executed.
- $Total\ Cost = c + c + c + \dots + c = c \times N$

Algorithm 1	Cost
array[0] \leftarrow 0	c
array[1] \leftarrow 0	c
array[2] \leftarrow 0	c
...	...
array[N-1] \leftarrow 0	c

Example 2

- *Total Cost* = $(N + 1) \times c_2 + N \times c_1 = (c_1 + c_2) \times N + c_2$

Algorithm 2	Cost
for i \leftarrow 0 to n	c_2
array[i] \leftarrow 0	c_1

Example 3

- $Total\ Cost = c_1 + c_2 \times (N + 1) + c_2 \times N \times (N + 1) + c_3 \times N^2$

Algorithm 3	Cost
sum \leftarrow 0	c_1
for i \leftarrow 0 to n	c_2
for j \leftarrow 0 to n	c_2
sum \leftarrow sum + array[i][j]	c_3

Summations

- The analysis involved computing a summation.
- Summation should be familiar but let us review a bit here.
- We will use summations as a tool to solve the algorithm.

- Given a finite sequence of values $a_1, a_2, a_3, \dots, a_n$.
- Their sum $a_1 + a_2 + a_3 + \dots + a_n$ is expressed in summation notation as:

$$\sum_{i=1}^n a_i$$

- If $n = 0$, then the sum is additive identity, 0.

Some Facts about Summation

$$\sum_{i=1}^n ca_i = c \sum_{i=1}^n a_i$$

and

$$\sum_{i=1}^n (a_i + b_i) = \sum_{i=1}^n a_i + \sum_{i=1}^n b_i$$

Some Important Summations

- Arithmetic Series
- Quadratic Series
- Geometric Series
- Harmonic Series

Arithmetic Series

$$\begin{aligned}\sum_{i=1}^n i &= 1 + 2 + 3 \dots + n \\ &= \frac{n(n+1)}{2} \\ &= O(n^2)\end{aligned}$$

- Here, from the polynomial $\frac{n^2}{2} + \frac{n}{2}$, $\frac{n^2}{2}$ is dominating term.
- So, we further took only n^2 as dominating term.

Quadratic Series

$$\begin{aligned}\sum_{i=1}^n i^2 &= 1 + 4 + 9 \dots + n^2 \\ &= \frac{2n^3 + 3n^2 + n}{6} \\ &= O(n^3)\end{aligned}$$

- Here, from the polynomial $\frac{2n^3}{6} + \frac{3n^2}{6} + \frac{n}{6}$, $\frac{2n^3}{6}$ is dominating term.
- So, we further took only n^3 as dominating term.

Geometric Series

$$\sum_{i=1}^n x^i = 1 + x + x^2 \dots + x^n$$
$$= \frac{x^{n+1} - 1}{x - 1}$$

- If $0 < x < 1$ then this is $O(1)$, and
- If $x > 1$, then this is $O(x^n)$.

Harmonic Series

- For $n \geq 0$

$$\begin{aligned} H_n &= \sum_{i=1}^n \frac{1}{i} \\ &= 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(n) \\ &\approx O(\ln(n)) \end{aligned}$$

- Its dominating term is Natural log (base e) of n .

A More Complex Example

```
1 for i ← 1 to n
2 do
3     for j ← 1 to 2i
4         do k ← j ...
5             while (k ≥ 0)
6                 do k ← k - 1 ...
```

- Why we say it is complex example?
- Due to more nested loops and their conditions.
- So, how to perform analysis on this type of nesting loops, etc.

- How do we analyze the running time of an algorithm that has complex nested loop?
- The answer is we write out the loops as summations and then solve the summations.
- To convert loops into summations, we work from inside-out.
- Starting from inner-most loop.

Inner-Most Loop

```
1 for i ← 1 to n
2 do
3     for j ← 1 to 2i
4     do k=j ...
5         while (k ≥ 0) ←
6 do k = k - 1 ...
```

- Consider the inner most while loop.
- It is executed for $k = j, j - 1, j - 2, \dots, 0$
- We don't consider what is happening in while loop.
- And we assumed time spent inside the while loop is constant.

- Let $I()$ be the time spent in the **while** loop, then:

$$I(j) = \sum_{k=0}^j 1 = j + 1$$

$$I(j) = j + 1$$

- Because it is the constant time inside the while loop.
- That is why, we write that constant time in the form of summation.

Middle Loop

- Now we consider the middle **for** loop.

```
1 for i ← 1 to n
2 do
3     for j ← 1 to 2i ←
4     do k=j ...
5         while (k ≥ 0)
6         do k = k - 1 ...
```

- It's running time is determined by i .
- Let $M()$ be the time spent in the **for** loop:

$$M(i) = \sum_{j=1}^{2i} I(j)$$

- This mathematical expression will be solved as follows:

$$M(i) = \sum_{j=1}^{2i} I(j) = \sum_{j=1}^{2i} (j + 1)$$

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


$$M(i) = \frac{2i (2i + 1)}{2} + 2i$$

$$\mathbf{M(i) = 2i^2 + 3i}$$

- This is the running time of inner 2 loops.

Outer-Most Loop

- Finally outer most *for* loop.

```
1 for i ← 1 to n ←   
2 do  
3     for j ← 1 to 2i  
4     do k=j ...  
5         while (k ≥ 0)  
6         do k = k - 1 ...
```

- Let $T()$ be running time of the entire algorithm:

$$T(n) = \sum_{i=1}^n M(i)$$

$$T(n) = \sum_{i=1}^n M(i) = \sum_{i=1}^n (2i^2 + 3i)$$

$$T(n) = \sum_{i=1}^n 2i^2 + \sum_{i=1}^n 3i$$

$$M(i) = 2 \frac{2n^3 + 3n^2 + n}{6} + 3 \frac{n(n+1)}{2}$$

$$M(i) = \frac{4n^3 + 15n^2 + 11n}{6}$$

$$\mathbf{M(i) = O(n^3)}$$

The dominating term from the expression is n^3 .

- In this way, we can say that the exponent value of the n term increases as the order of loop nesting increases.
- But this is not a rule.
- e.g., 1 loop means running time is n
- 2 order nesting mean n^2
- 3 order nesting mean n^3
- And so on.

2D Maxima Revisited

- Can we improve the performance of the 2D maxima algorithm?
- So that we reduce the running time from n^2 to some low value.
- It is a huge contribution in the research to improve the performance of an existing algorithm.

- But it is very difficult and complex procedure and need massive research.
- So how to improve the performance?
- We cannot improve the performance by just changing the programming techniques and methodologies.
- But we have to make fundamental changes in the algorithm mathematically.
- In the next class we try to improve the performance of the 2D maxima.