Inf 2B: Asymptotic notation and Algorithms

Lecture 2B of ADS thread

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh

Reminder of Asymptotic Notation

Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. We say that:

- *f is* $O(g)$ if there is some $n_0 \in \mathbb{N}$ and some $c > 0 \in \mathbb{R}$ such that for all $n \geq n_0$ we have

$$0 \leq f(n) \leq c\, g(n).$$

- *f is* $\Omega(g)$ if there is an $n_0 \in \mathbb{N}$ and $c > 0$ in $\mathbb{R}$ such that for all $n \geq n_0$ we have

$$f(n) \geq c\, g(n) \geq 0.$$

- ***f*** *is* $\Theta(g)$, or *f has the same asymptotic growth rate* as g, if f is $O(g)$ and $\Omega(g)$.

Worst-case (and best-case) running-time

We almost always work with Worst-case running time in Inf2B:

Definition

The *(worst-case) running time* of an algorithm A is the function $T_{\mathrm{A}} : \mathbb{N} \to \mathbb{N}$ where $T_{\mathrm{A}}(n)$ is the maximum number of computation steps performed by A on an input of size n.

Definition

The *(best-case) running time* of an algorithm A is the function $B_{\mathrm{A}} : \mathbb{N} \to \mathbb{N}$ where $B_{\mathrm{A}}(n)$ is the minimum number of computation steps performed by A on an input of size n.

We only use Best-case for explanatory purposes.

Asymptotic notation for Running-time

How do we apply O, Ω, Θ to analyse the running-time of an algorithm A?

Possible approach:

- We analyse A to obtain the worst-case running time function $T_{\mathrm{A}}(n)$.
- We then go on to derive upper and lower bounds on (the growth rate of) $T_{\mathrm{A}}(n)$, in terms of $O(\,\cdot\,)$, $\Omega(\,\cdot\,)$.

In fact we use asymptotic notation with the analysis, much simpler (no need to give names to constants, takes care of low level detail that isn't part of the big picture).

- We aim to have matching $O(\,\cdot\,)$, $\Omega(\,\cdot\,)$ bounds hence have a $\Theta(\,\cdot\,)$ bound.
- Not always possible, even for apparently simple algorithms.

Example

```
algA(A,r,s)
if $r < s$ then
    for $i \leftarrow r$ to $s$ do
        for $j \leftarrow i$ to $s$ do
            $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$
            algB($A, i, m-1$)
            algB($A, m, j$)
    $m \leftarrow \lfloor \frac{r+s}{2} \rfloor$
    algA($A, r, m-1$)
    algA($A, m, s$)
```

```
algB(A,r,s)
if $A[r] < A[s]$ then
    swap $A[r]$ with $A[s]$
if $r < s - r$ then
    algA($A, r, s-r$)
```

linSearch

Input: Integer array A, integer k being searched.
Output: The least index i such that $A[i] = k$.

Algorithm linSearch(A, k)

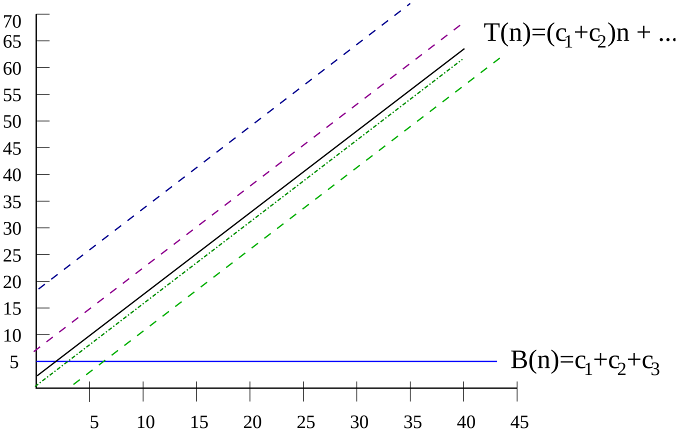
```
for $i \leftarrow 0$ to $A.length - 1$ do
    if $A[i] = k$ then
        return $i$
return $-1$
```

(Lecture Note 1) Worst-case running time $T_{\text{linSearch}}(n)$ satisfies

$$\begin{aligned}(c_1 + c_2)n + \min\{c_3, c_1 + c_4\} &\leq T_{\text{linSearch}}(n)\\ &\leq (c_1 + c_2)n + \max\{c_3, c_1 + c_4\}.\end{aligned}$$

Best-case running time satisfies $B_{\text{linSearch}}(n) = c_1 + c_2 + c_3$.

Picture of $T_{\text{linSearch}}(n)$, $B_{\text{linSearch}}(n)$



$T_{\text{linSearch}}(n) = O(n)$

Proof.

From Lecture Note 1 we have

$$T_{\text{linSearch}}(n) \le (c_1 + c_2) \cdot n + \max\left\{c_3, (c_1 + c_4)\right\}.$$

Take $n_0 = \max\{c_3, (c_1 + c_4)\}$, $c = c_1 + c_2 + 1$.
Then for every $n \ge n_0$, we have

$$\begin{aligned} T_{\text{linSearch}}(n) &\le (c_1 + c_2)n + n_0 \\ &\le (c_1 + c_2 + 1)n = cn. \end{aligned}$$

Hence $T_{\text{linSearch}}(n) = O(n)$. $\square$

$T_{\text{linSearch}}(n) = \Omega(n)$

We know $T_{\text{linSearch}}(n) = O(n)$.
Also true: $T_{\text{linSearch}}(n) = O(n\lg(n))$, $T_{\text{linSearch}}(n) = O(n^2)$.

Is $T_{linSearch}(n) = O(n)$ the best we can do?

YES, because …

$T_{\text{linSearch}}(n) = \Omega(n)$.

Proof.

$T_{\text{linSearch}}(n) \geq (c_1 + c_2)n$, because all c_i are positive.
Take $n_0 = 1$ and $c = c_1 + c_2$ in defn of Ω. □

$$T_{\text{linSearch}}(n) = \Theta(n).$$

Misconceptions/Myths about O and Ω

MISCONCEPTION 1

> If we can show $T_{\mathrm{A}}(n) = O(f(n))$ for some function $f : \mathbb{N} \to \mathbb{R}$, then the running time of A on inputs of size n is bounded by $f(n)$ for sufficiently large n.

FALSE: Only guaranteed an upper bound of $cf(n)$, for some constant $c > 0$.

Example: Consider linSearch. We could have shown $T_{\mathrm{linSearch}} = O(\frac{1}{2}(c_1 + c_2)n)$ (or $O(\alpha n)$, for any constant $\alpha > 0$) *exactly* as we showed $T_{\mathrm{linSearch}}(n) = O(n)$ **but . . .**
the worst-case for linSearch is greater than $\frac{1}{2}(c_1 + c_2)n$.

Misconceptions/Myths about O and Ω

MISCONCEPTION 2

> Because $T_{\mathsf{A}}(n) = O(f(n))$ implies a $c\,f(n)$ upper bound on the running-time of A for *all* inputs of size n, then $T_{\mathsf{A}}(n) = \Omega(g(n))$ implies a similar lower bound on the running-time of A for *all* inputs of size n.

FALSE: If $T_{\mathsf{A}}(n) = \Omega(g(n))$ for some $g : \mathbb{N} \to \mathbb{R}$, then there is some constant $c' > 0$ such that $T_{\mathsf{A}}(n) \geq c'\,g(n)$ for all sufficiently large n.
But A can be much faster than $T_{\mathsf{A}}(n)$ on other inputs of length n that are not worst-case! No lower bound on *general* inputs of size n. linSearch graph is an example.

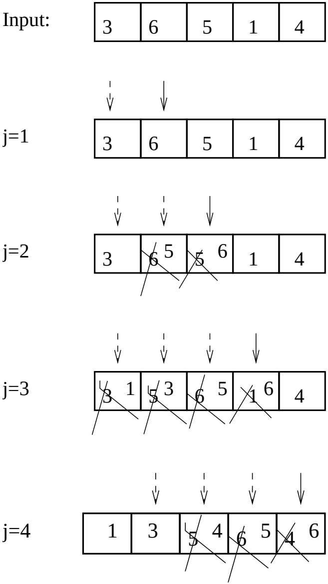
Insertion Sort

Input: An integer array A
Output: Array A sorted in non-decreasing order

Algorithm insertionSort(A)

```
for $j \leftarrow 1$ to $A.length - 1$ do
    $a \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $i \geq 0$ and $A[i] > a$ do
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow a$
```

Example: Insertion Sort



Big-O for $T_{\text{insertionSort}}(n)$

```
Algorithm insertionSort($A$)
1. for $j \leftarrow 1$ to $A.length - 1$ do
2.         $a \leftarrow A[j]$
3.         $i \leftarrow j - 1$
4.         while $i \geq 0$ and $A[i] > a$ do
5.                 $A[i+1] \leftarrow A[i]$
6.                 $i \leftarrow i - 1$
7.         $A[i+1] \leftarrow a$
```

Line 1	$O(1)$ time, *executed* $A.length - 1 = n - 1$ times.
Lines 2,3,7	$O(1)$ time each, *executed* $n - 1$ times.
Lines 4,5,6	$O(1)$-time, executed together as **for**-loop. No. of executions depends on **for**-test, j. For fixed j, **for**-loop at 4. takes *at most* j iterations.

```
Algorithm insertionSort($A$)
1.  for $j \leftarrow 1$ to $A.length - 1$ do
2.      $a \leftarrow A[j]$
3.      $i \leftarrow j - 1$
4.      while $i \geq 0$ and $A[i] > a$ do
5.          $A[i+1] \leftarrow A[i]$
6.          $i \leftarrow i - 1$
7.      $A[i+1] \leftarrow a$
```

For a fixed j, lines 2-7 take at most

$$\begin{aligned} O(1)+&O(1)+O(1)+O(j)+O(j)+O(j)+O(1) \\ &= O(1)+O(j) \\ &= O(1)+O(n) \\ &= O(n). \end{aligned}$$

There are $n-1$ different j-values. Hence

$$T_{\text{insertionSort}}(n) = (n-1)O(n) = O(n)O(n) = O(n^2).$$

$T_{\text{insertionSort}}(n) = \Omega(n^2)$

Harder than $O(n^2)$ bound.
Focus on a **BAD** instance of size n:
Take input instance $\langle n, n-1, n-2, \ldots, 2, 1\rangle$.

- For every $j = 1 \ldots, n-1$, insertionSort uses j executions of line 5 to insert $A[j]$.

Then

$$\begin{aligned} T_{\text{insertionSort}}(n) &\geq \sum_{j=1}^{n-1} c\, j \\ &= c \sum_{j=1}^{n-1} j \;=\; c\frac{n(n-1)}{2}. \end{aligned}$$

So $T_{\text{insertionSort}}(n) = \Omega(n^2)$ and $T_{\text{insertionSort}}(n) = \Theta(n^2)$.

“Typical” asymptotic running times

- $\Theta(\lg n)$ (logarithmic),
- $\Theta(n)$ (linear),
- $\Theta\big(n \lg n\big)$ (n-log-n),
- $\Theta(n^2)$ (quadratic),
- $\Theta(n^3)$ (cubic),
- $\Theta(2^n)$ (exponential).

Further Reading

- Lecture notes 2 from last week.
- If you have Goodrich & Tamassia [GT]:
  All of the chapter on “Analysis Tools” (especially the “Seven functions” and “Analysis of Algorithms” sections).
- If you have [CLRS]:
  Read chapter 3 on “Growth of Functions.”