Lecture 2: Analysis of Algorithms

Alfred Rossi, Ph.D.

Algorithms

Definition

An ***algorithm*** is a well-defined computational procedure that takes ***input*** and produces ***output***.

Algorithms

Note

Algorithms are ***required to halt*** by definition!

Code

```
def insertion_sort(xs):
    n = len(xs)

    for i in range(1, n):
        j = i
        while j > 0 and xs[j-1] > xs[j]:
            swap(xs, j, j-1)
            j -= 1
```

Figure. A Python-compatible pseudocode for **InsertionSort**.

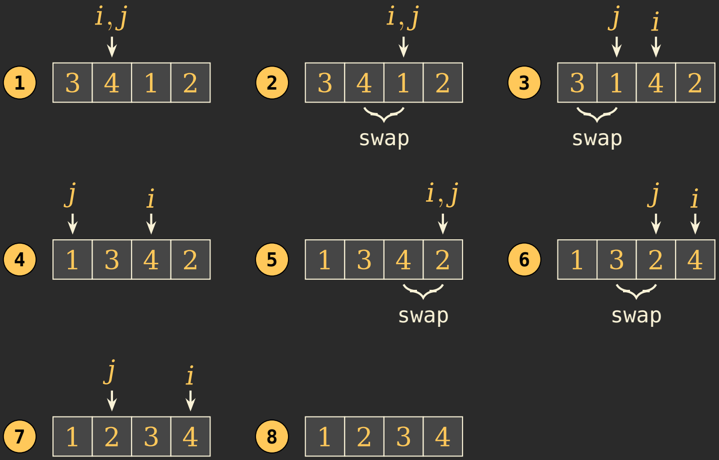

Figure. A worked example of **Insertion Sort** applied to the **Sorting** instance $\langle 3, 4, 1, 2 \rangle$.

The RAM and (word-) RAM Models

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.

[1]That's right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.

[1] That's right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.
- We have a constant number of constant time “atomic" operations (possibly taking different amounts of time)

[1] That’s right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.
- We have a constant number of constant time “atomic" operations (possibly taking different amounts of time)
  - Store (e.g. $M[p] \leftarrow r_1$, $M[p+1] \leftarrow M[p] + 1$)

[1] That’s right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.
- We have a constant number of constant time “atomic" operations (possibly taking different amounts of time)
  - Store (e.g. $M[p] \leftarrow r_1$, $M[p+1] \leftarrow M[p] + 1$)
  - Load (e.g. $r_1 \leftarrow M[p]$)

[1] That’s right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.
- We have a constant number of constant time “atomic" operations (possibly taking different amounts of time)
  - Store (e.g. $M[p] \leftarrow r_1$, $M[p+1] \leftarrow M[p]+1$)
  - Load (e.g. $r_1 \leftarrow M[p]$)
  - Arithmetic (e.g. $r_1 + r_2$, $r_1 \cdot r_2$, r_1 / r_2, $r_1 - r_2$, $r_1 \bmod r_2$)

[1]That’s right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.
- We have a constant number of constant time “atomic" operations (possibly taking different amounts of time)
  - Store (e.g. $M[p] \leftarrow r_1$, $M[p+1] \leftarrow M[p] + 1$)
  - Load (e.g. $r_1 \leftarrow M[p]$)
  - Arithmetic (e.g. $r_1 + r_2$, $r_1 \cdot r_2$, r_1 / r_2, $r_1 - r_2$, $r_1 \bmod r_2$)
  - Logic (e.g. $r_1 = r_2$, $r_1 \geq r_2$)

[1] That’s right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.
- We have a constant number of constant time “atomic" operations (possibly taking different amounts of time)
  - Store (e.g. $M[p] \leftarrow r_1$, $M[p+1] \leftarrow M[p] + 1$)
  - Load (e.g. $r_1 \leftarrow M[p]$)
  - Arithmetic (e.g. $r_1 + r_2$, $r_1 \cdot r_2$, r_1 / r_2, $r_1 - r_2$, $r_1 \bmod r_2$)
  - Logic (e.g. $r_1 = r_2$, $r_1 \geq r_2$)
  - Control Flow (e.g. CALL/RET, JMP)

[1] That’s right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- Memory is an unbounded[1] array, where each cell holds an integer. That is, $\forall p \in \mathbb{N}_0$, $M[p] \in \mathbb{Z}$.
- Likewise, there is a finite set of integer-valued registers $R = \{r_1, r_2, \ldots, r_k\}$. That is, $\forall r \in R, r \in \mathbb{Z}$.
- We have a constant number of constant time "atomic" operations (possibly taking different amounts of time)
  - Store (e.g. $M[p] \leftarrow r_1$, $M[p+1] \leftarrow M[p] + 1$)
  - Load (e.g. $r_1 \leftarrow M[p]$)
  - Arithmetic (e.g. $r_1 + r_2$, $r_1 \cdot r_2$, r_1 / r_2, $r_1 - r_2$, $r_1 \bmod r_2$)
  - Logic (e.g. $r_1 = r_2$, $r_1 \geq r_2$)
  - Control Flow (e.g. CALL/RET, JMP)
- Instructions are run sequentially

[1] That's right. Infinite memory, baby!

The Random-access machine (***RAM***) model

- What the book calls the RAM model is actually the word-RAM model.

The Random-access machine (***RAM***) model

- What the book calls the RAM model is actually the word-RAM model.
- Execution proceeds similarly to the ***RAM*** model (sequentially with constant-time operations).

The Random-access machine (***RAM***) model

- What the book calls the RAM model is actually the word-RAM model.
- Execution proceeds similarly to the ***RAM*** model (sequentially with constant-time operations).
- Machine registers and memory are comprised of w-bit words, which we can think of as representing unsigned integers $[0, \ldots, 2^w - 1]$, signed values $[-2^{w-1}, 2^{w-1} - 1]$, or any 2^w values of our liking.

The Random-access machine (***RAM***) model

- What the book calls the RAM model is actually the word-RAM model.
- Execution proceeds similarly to the ***RAM*** model (sequentially with constant-time operations).
- Machine registers and memory are comprised of w-bit words, which we can think of as representing unsigned integers $[0, \ldots, 2^w - 1]$, signed values $[-2^{w-1}, 2^{w-1} - 1]$, or any 2^w values of our liking.
- Finite memory. (Now limited to 2^w addressible locations.)

The Random-access machine (***RAM***) model

- What the book calls the RAM model is actually the word-RAM model.
- Execution proceeds similarly to the ***RAM*** model (sequentially with constant-time operations).
- Machine registers and memory are comprised of w-bit words, which we can think of as representing unsigned integers $[0, \ldots, 2^w - 1]$, signed values $[-2^{w-1}, 2^{w-1} - 1]$, or any 2^w values of our liking.
- Finite memory. (Now limited to 2^w addressible locations.)
- The book assumes that $w = c \log(n)$ for some $c \geq 1$, where n is the size of the input. This means available memory is $\Theta(n)$.

The Random-access machine (***RAM***) model

- What the book calls the RAM model is actually the word-RAM model.
- Execution proceeds similarly to the ***RAM*** model (sequentially with constant-time operations).
- Machine registers and memory are comprised of w-bit words, which we can think of as representing unsigned integers $[0, \ldots, 2^w - 1]$, signed values $[-2^{w-1}, 2^{w-1} - 1]$, or any 2^w values of our liking.
- Finite memory. (Now limited to 2^w addressible locations.)
- The book assumes that $w = c \log(n)$ for some $c \geq 1$, where n is the size of the input. This means available memory is $\Theta(n)$.
- Usually fine to analyze in the (non-word) RAM model.

Complexity in the (word-) RAM model

- ***Time-complexity*** is the number of executed instructions
- ***Space-complexity*** is the amount of memory used by the algorithm
- Usually will analyze in terms of a parameter, e.g. the input size

Problem 1

Code

```python
def f1(xs):
    n = len(xs)

    x = 0
    for i in range(n):
        x += xs[i]
        x -= 1

    return x
```

Problem 2

Code

```
def f2(xs):
    n = len(xs)

    x = 0
    for i in range(n):
        for j in range(n):
            x += i - j*i

    return x
```

Problem 3

Code

```
def f3(xs):
    n = len(xs)

    x = 0
    for i in range(3, n):
        for j in range(4n):
            x += i*i - j*j

    return x
```

Problem 4

Code

```python
def f4(xs):
    n = len(xs)

    x = 0
    for i in range(n):
        for j in range(i):
            x += i*i - j

    return x
```

Problem 5

Code

```python
def f5(xs):
    n = len(xs)

    x = 0
    for i in range(n):
        u = int(n**(1/2))
        for j in range(u):
            x += i*i - j

    return x
```

Problem 6

Code

```python
def f6(xs):
    n = len(xs)

    x = 0
    for i in range(n):
        u = int(i**(1/3))
        for j in range(u):
            x += i*i - j

    return x
```

Problem 7

Code

```python
def f7(xs):
    n = len(xs)

    x = 1
    i = 0
    while i < n:
        x *= xs[i]
        i += 1

    return x
```

Problem 8

Code

```python
def f8(xs):
    n = len(xs)

    x = 1
    i = 32
    while i < n:
        x *= xs[i]
        i += 1

    return x
```

Problem 9

Code

```
def f9(xs):
    n = len(xs)

    x = 1
    i = 0
    while i < n:
        x *= xs[i]
        i += 5

    return x
```

Problem 10

Code

```
def f10(xs):
    n = len(xs)

    x = 1
    i = 11
    while i < n:
        x *= xs[i]
        i *= 4

    return x
```

Best and Worst Case Analysis

But... wait.. can’t an algorithm be fast for one input but slow for another of the same size? ***YES!***

Let’s look at some running times for **InsertionSort** by size.

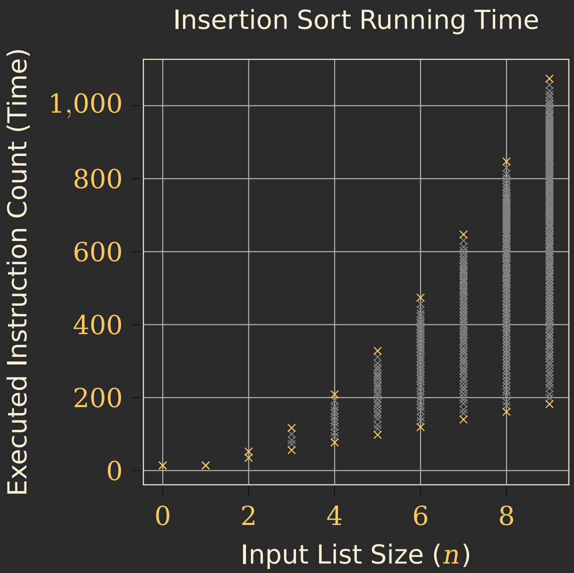

Figure. Number of executed Python bytecode instructions in sorting lists of length n for $0 \leq n \leq 9$ with **InsertionSort**.

Worst Case Analysis

In ***worst case analysis*** the goal is to find the best upper and lower bounds that we can on the tail of the ***upper*** envelope.

Code

```python
def insertion_sort(xs):
    n = len(xs)

    for i in range(1, n):
        j = i
        while j > 0 and xs[j-1] > xs[j]:
            swap(xs, j, j-1)
            j -= 1
```

Figure. A Python-compatible pseudocode for **InsertionSort**.

Note that the number of executed instruction is maximized when $xs[j-1] > xs[j]$ for all j. That is, if xs is ***reverse sorted***!

Best Case Analysis

In ***best case analysis*** the goal is to find the best upper and lower bounds that we can on the tail of the ***lower*** envelope.

Code

```
def insertion_sort(xs):
    n = len(xs)

    for i in range(1, n):
        j = i
        while j > 0 and xs[j-1] > xs[j]:
            swap(xs, j, j-1)
            j -= 1
```

Figure. A Python-compatible pseudocode for **InsertionSort**.

Observe that the number of executed instruction is ***minimized*** if $xs[j-1] \leq xs[j]$ for all j. That is, if xs is ***sorted***!

In other words, for any input of length n, a ***best case input*** for xs is when it is ***already sorted***, and a ***worst case input*** for xs is when it is ***reverse sorted***!