LEK2: Analysing Algorithms: Insertion sort, RAM, Notations, Complexity.

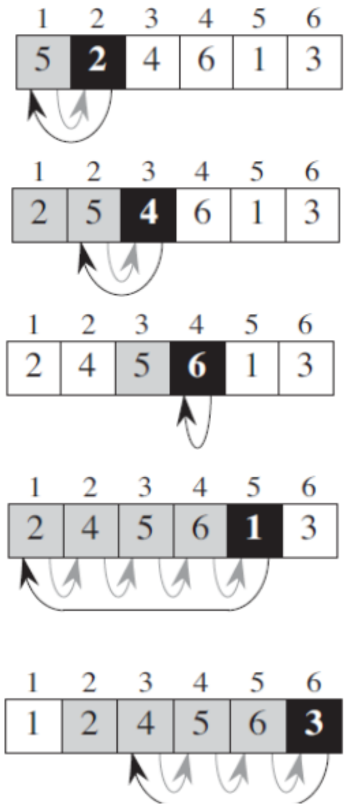
Insertion Sort

Strategy:

- Start with an “empty hand,” say left hand.
- Insert a card in the correct position of left hand, where the numbers are already sorted.
- Continue until all cards are inserted/sorted.

> *INPUT*: A[1..n] – an array of integers, n>0
> *OUTPUT*: a permutation of A such that A[1]≤ A[2] ≤ **…** ≤ A[n]
>
> ```
> for j = 2 to n
>     key = A[j]
>     // Insert A[j] into the sorted A[1..j-1].
>     i=j-1
>     while i>0 and A[i]>key
>          A[i+1]=A[i]
>          i--
>     A[i+1]=key
> ```
>
> 6

Example: 5, 2, 4, 6, 1, 3



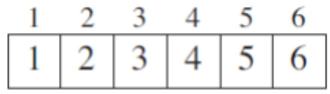
j=2, i=1…1

j=3, i=2…1

j=4, i=3…1

j=5, i=4…1

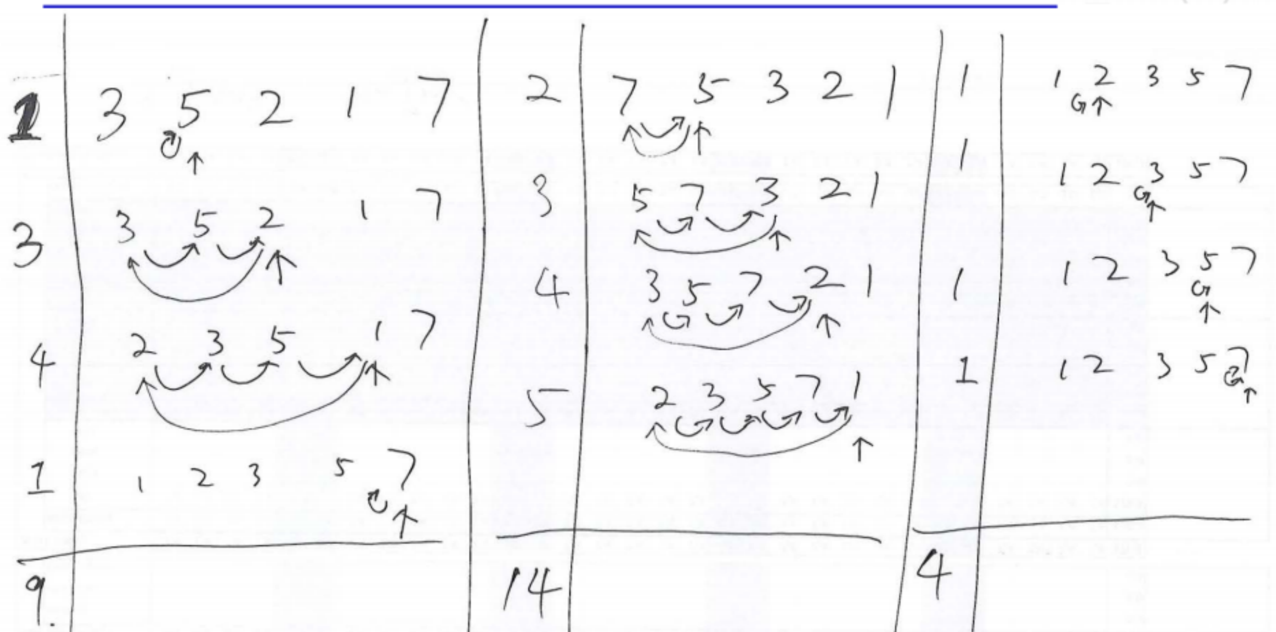
j=6, i=5…1



j=7, which is greater than n. Done!

```
INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤ A[2] ≤ ... ≤ A[n]

for j = 2 to n
   key = A[j]
   // Insert A[j] into the sorted A[1..j-1].
   i=j-1
   while i>0 and A[i]>key
       A[i+1]=A[i]
       i--
   A[i+1]=key
```



Average case Worst case Best case

Analysing the insertion sort

- How fast is insertion sort? absolute vs. relative speeds?
  - Absolut speeds depend on specific computers.
  - Relative speeds do not depend on specific computers.
  - In this course, we care about the relative speed.
  - The relationship between the running time and input size.
    - T(n): running time being a function of input size n.

The RAM model

- Instructions
  - Primitive or atomic operations.
  - Each takes constant time, depending on the machine.
  - Instructions are executed one after another.
- We consider instructions commonly found in real computers.
  - Arithmetic (add, subtract, multiply, etc.)
  - Data movement (assignment)
  - Control (branch, subroutine call, return)
  - Comparison
- Data types – integers, characters, and floats

Analysis of Insertion Sort

INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤ A[2] ≤ **…** ≤ A[n]

```
1. for j = 2 to n
2.    key = A[j]
3.    // Insert A[j] into the sorted A[1..j-1].
4.     i=j-1
5.     while i>0 and A[i]>key
6.          A[i+1]=A[i]
7.          i--
8.     A[i+1]=key
```

Cost of each instruction	How many times of each instruction will be executed.
cost	times
	Loop header executes one more time than the loop body does.
c_1	$\boxed{n}$
c_2	$\boxed{n-1}$
$c_3=0$	n-1
c_4	n-1
c_5	$\sum_{j=2}^{n} t_j$
c_6	$\sum_{j=2}^{n}(t_j-1)$
c_7	$\sum_{j=2}^{n}(t_j-1)$
c_8	n-1

- t_j is the number of times of the **while** loop test in line 5 is executed for a specific value of j.
  - t_j is the number of elements in A[1...j-1] which need to be checked in the j-th iteration of the for loop in line 5.
- t_j may be different for different j.
- t_j may be different for different input instances, e.g., best case or worst case

Run time of insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Must represent t_j in terms of n

Best case: t_j=1

$$\boxed{\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}}$$

T(n)=a*n+b
where a and b are **constants**.
Thus, we have a **linear** algorithm.

CLRS, Page 1146, Eq. A.1

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

Worst case: t_j=j

$$\boxed{\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}}$$

T(n)=c*n^2+d*n+e, where c, d, and e are **constants**.
Thus, a **quadratic** algorithm.

15

Best/Worst/Average Case

- Suppose algorithm P accepts k different input instances of size n. Let $T_i(n)$ be the time complexity of P on the i-th input instance, for 1 ≤ i ≤ k, and pi being the probability that this instance occurs.
- Worst case time complexity: $W(n) = max_{1 \leq i \leq k} Ti_(n)$
  - The **maximum** running time over all k inputs of size n
  - It is the most interesting/important!
- Average case time complexity: $A(n) = \Sigma_{1 \leq i \leq k} p_i T_i(n)$
  - The **expected** running time over all k inputs of size n
  - Need assumptions about statistical distributions of input instances.
  - E.g., uniform distribution that each instance is equally likely.
- Best case time complexity: $B(n) = min_{1 \leq i \leq k} T_i(n)$
  - The **minimum** running time over all k inputs of size n

- Can be cheating

Compare Algorthms' Efficiencies

Look at how fast T(n) grows as n grows to a very large number (to the limit). This is called **Asymtotic Complexity**.

Asymptotic Analysis

- This is the BIG IDEA of algorithmic analysis.
- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware.
  - "rounding" for numbers: 1,000,001 ≈ 1,000,000
  - "rounding" for functions: $3n^2 \approx n^2$
- Basic idea of asymptotic analysis - capturing the essence
  - Ignore machine-dependent constants.
  - Look at the growth of the running time with the size of the input in the limit, instead of the actual running time.
  - Asymptotically more efficient algorithms are best for all but very small inputs.

Theta notation Θ

- "Engineering way" of manipulating Θ notation.
  - Ignore its leading constant
    - $T(n) = 1000 * n^5 = \Theta * (n^5)$
  - Drop its lower order terms
    - $T(n) = n^5 + n^3 + lgn = \Theta(n^5)$

- How to identify lower order terms?
  - Constant < poly-logarithm < polynomial < exponential
  - c $lg^k n$ n^a b^n

- T(n) = $23n^5+12n^4+2n^3+5n^2+n+4096$
  - T(n) = Θ (n^5)
- T(n) = 50n lg n + $\lg^{10000}$n
  - T(n) = Θ (n lg n)
- T(n) = $8n^2 \lg n + 5n^2 + n$
  - T(n) = Θ (n^2 lg n)

Theta notation Θ

- Mathematical definition
  - Θ(g(n)) is a set of functions {f(n)}.
  - Θ(g(n)) ={f(n): there exists positive constants c_1, c_2, and n_0, s.t. $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$}.
- f(n)= Θ(g(n)) means f(n) ∈ Θ(g(n))

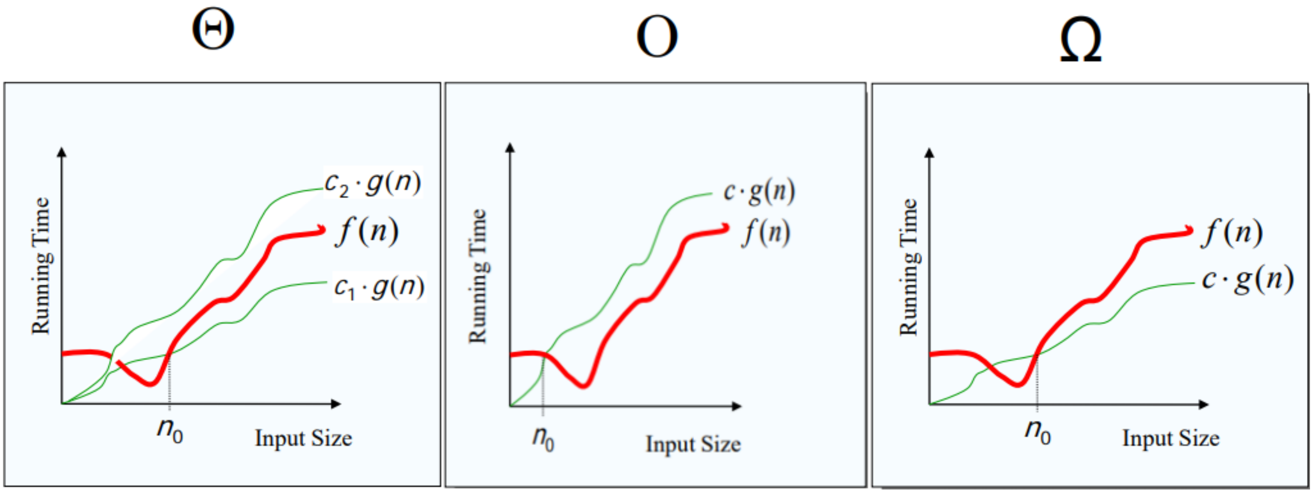
- Asymptotically tight bound
- f(n) is equal to g(n) within a constant factor.

Big-O Notation O

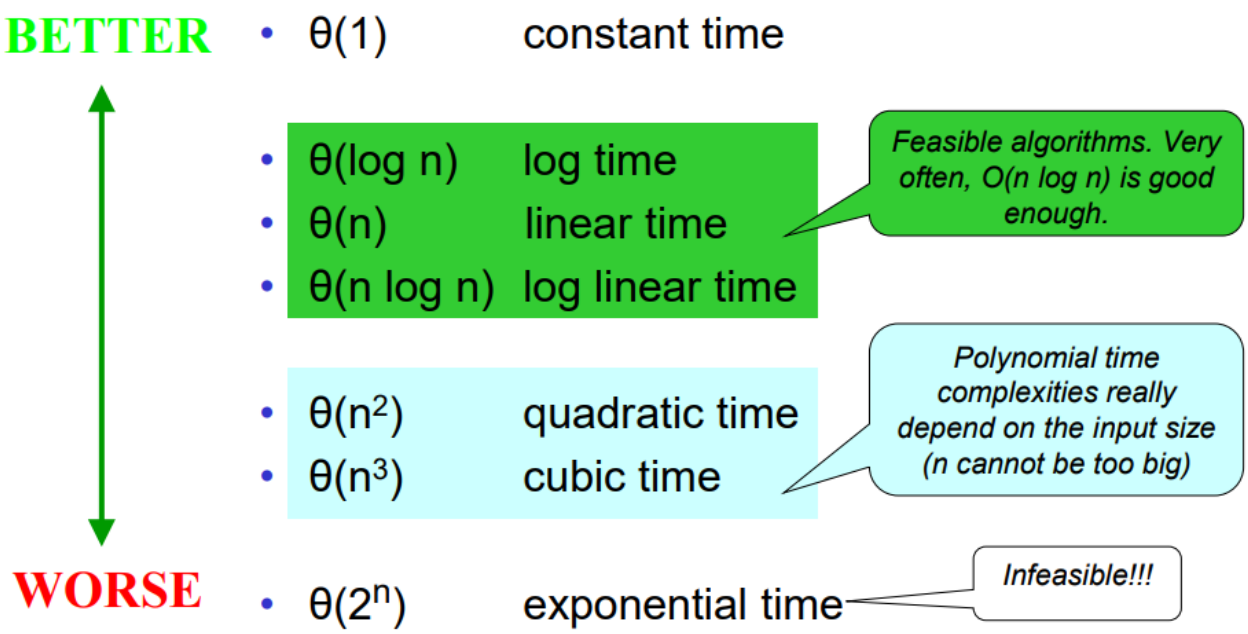
- Mathematical definition
    - O(g(n)) is a set of functions.
    - O(g(n)) ={f(n): there exists positive constants c, and n_0, s.t. $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$}.
- f(n)= O(g(n)) means f(n) ∈ O(g(n))
- Asymptotically upper bound.
- f(n) grows asymptotically slower than g(n).
- Used for worst-case analysis.

Big-Omega Notation Ω

- Mathematical definition
    - Ω(g(n)) is a set of functions.
    - Ω(g(n)) ={f(n): there exists positive constants c, and n0, s.t. $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$}.
- f(n)= Ω(g(n)) means f(n) ∈ Ω(g(n))
- Asymptotically lower bound.
- f(n) grows asymptotically faster than g(n).
- Used for best-case analysis.



Common Time Complexities



Two concepts (Complexity)

- Concrete complexity vs. abstract complexity
  - Concrete complexity refers to the results from the complexity analysis using the RAM model, including many details.
  - Abstract complexity refers to the results from the asymptotic analysis, i.e., using the theta, Big-O, and Big-Omega notation.
- Example, insertion sort
  - Concrete complexity

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)\,. \end{aligned}$$

- Abstract complexity (above ex.)
  - Worst case, average case: $\Theta(n^2)$
  - Best case: Θ(n)
- Another example, exercise from Lecture 1, CLRS, 1.2-2.
  - Insertion sort needs $8n^2$ steps vs. merge sort needs 64nlgn steps.
  - Concrete complexity: $8n^2$ vs. 64nlgn
  - Abstract complexity: $\Theta(n^2)$ vs. Θ(nlgn)