BROWN
Computer Science

CS1570: Design and Analysis of Algorithms

Lecture 1: Computational model and proof techniques

Lorenzo De Stefani

Fall 2020

Outline

- The RAM computational model
- Insertion Sort
- Types of algorithmic analysis
- Asymptotic notation
- Recap on proofs
  - Induction
  - Contradiction
  - Construction

Design and analysis of Algorithms

- Analysis: predict the cost of an algorithms in terms of resources and performance
  - Lower bounds to the time/space requirement of algorithms
  - They are a gauge of the quality of algorithms implementations
- Design: construction of algorithms which minimize the running time and memory utilization
  - Upper bounds, ideally matching the corresponding lower bounds

A mathematical approach to algorithm analysis

- We evaluate the performance of algorithms in an hardware-independent mathematical model
  - We count the number of operations
  - Number of memory locations
- General analysis which is "portable" to every hardware platform
- Actual performance (i.e., execution time) ultimately depends on hardware specifications (e.g., FLOPS, Bus bandwidth, Cache size)

The RAM computational model

Generic Random Access Machine (RAM)

- Operations are executed sequentially
- We assume the computational unit to be equipped with a set of primitive operations
  - Algebraic, Logic, Comparisons, Calls to functions,...
- We assume each computation to require the same unit time (an *atomic* time amount)
  - Removes hardware-specific considerations

The RAM computational model

Generic Random Access Machine (RAM)

- We assume that all memory locations can be accessed in the same time
  - Flat memory model – no memory access latency!
- Moving values from one memory location to the other requires unit time
- Memory atomic unit referred as memory word
  - We assume that each word can hold an input value or an intermediate result in the computation
  - We shall see that this can be a particularly problematic assumption ☺
- The size of the memory correspond to the number of memory words in the memory

The problem of sorting

- Input: a sequence $(a_1, a_2, \dots, a_n)$ of numbers
- Output: a permutation $(a'_1, a'_2, \dots, a'_n)$ of these values such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Fundamental problem in computing
  - Even if you do not notice your CPU is running sorting all the time
- Example: $(8,2,4,9,3,6) \rightarrow (2,3,4,6,8,9)$

Insertion sort

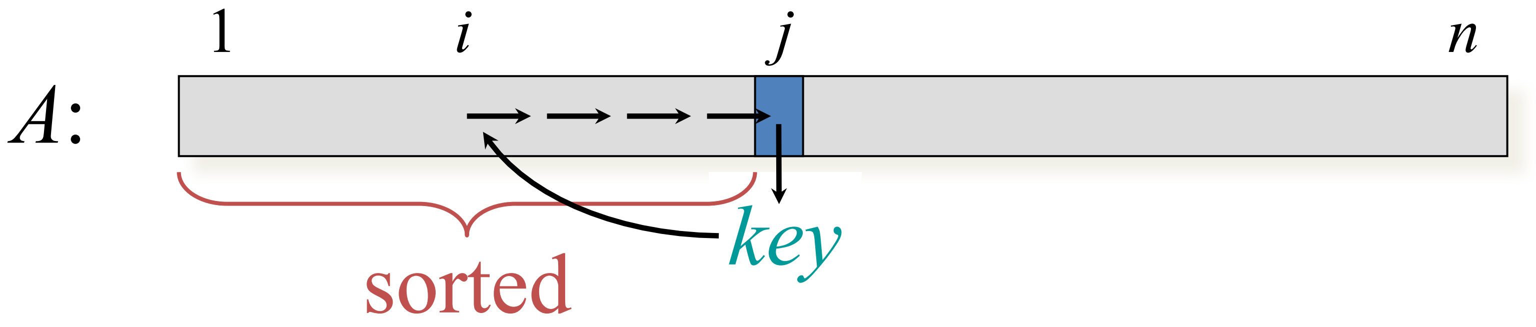
```
INSERTION-SORT $(A, n)$        $\triangleright$ $A[1 \,.\,.\, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```

Pseudocode notation

- Used for simplified algorithm presentation
- Language independent
- Similar to python script

Insertion sort

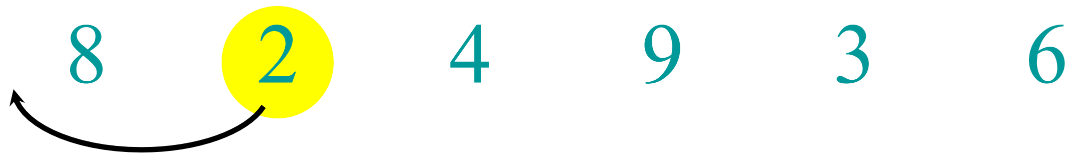
```
INSERTION-SORT $(A, n)$        ⊳ $A[1 \mathinner{.\,.} n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



Insertion sort: example

8 2 4 9 3 6

Insertion sort: example



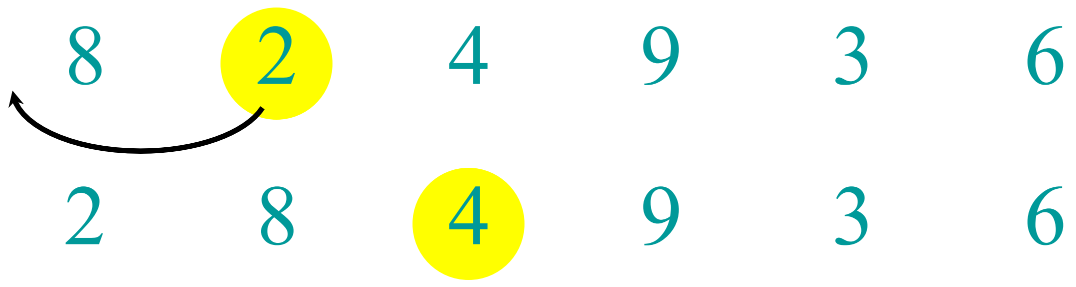
In this step:

- 1 comparison
- 3 memory assignments

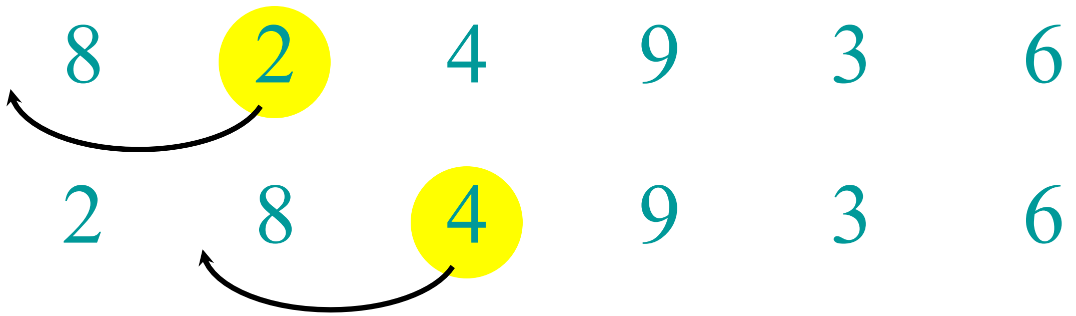
Total:

- 1 comparison
- 3 memory assignments

Insertion sort: example



Insertion sort: example



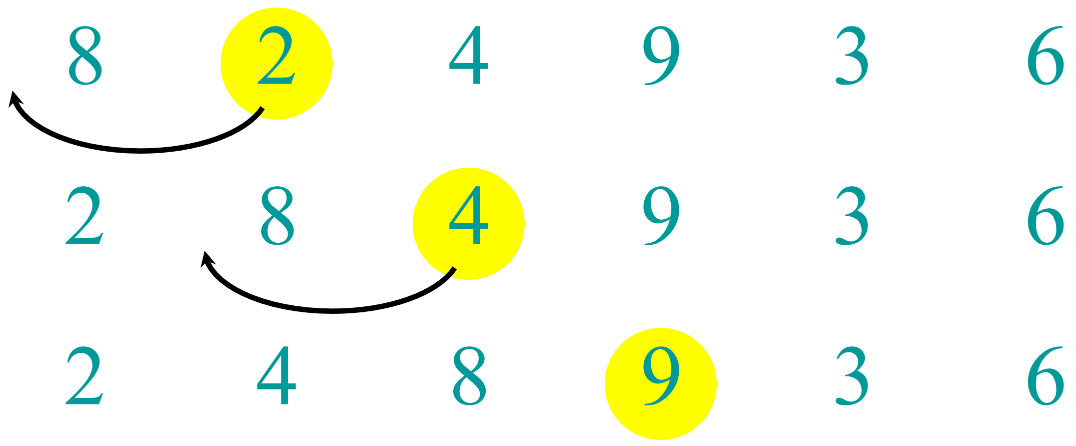
In this step:

- 2 comparison
- 3 memory assignments

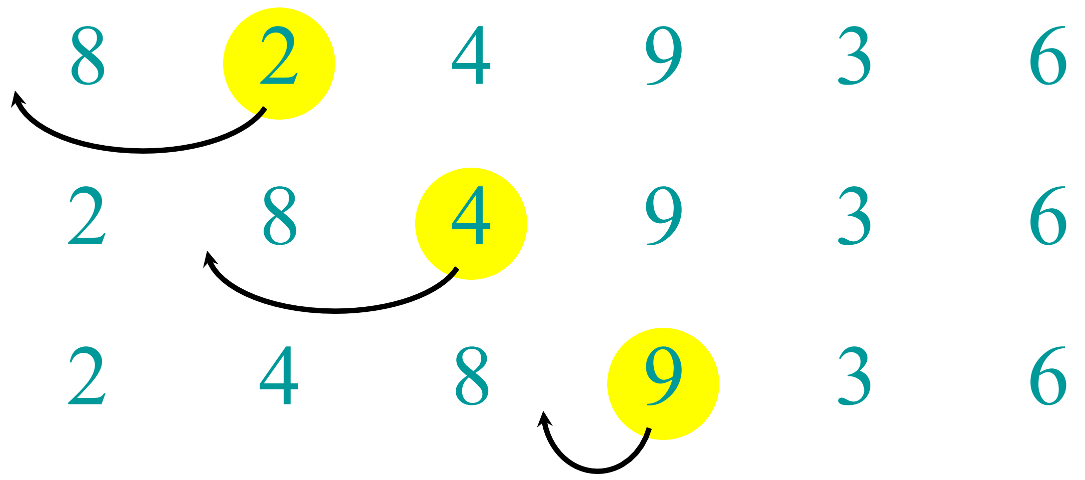
Total:

- 3 comparison
- 6 memory assignments

Insertion sort: example



Insertion sort: example



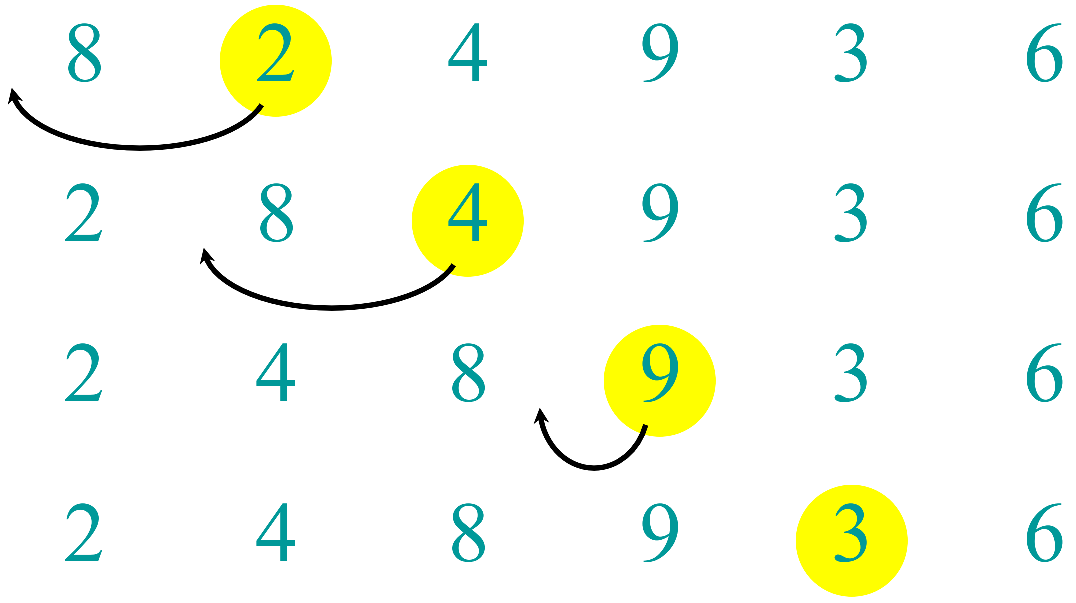
In this step:

- 1 comparison
- 2 memory assignments

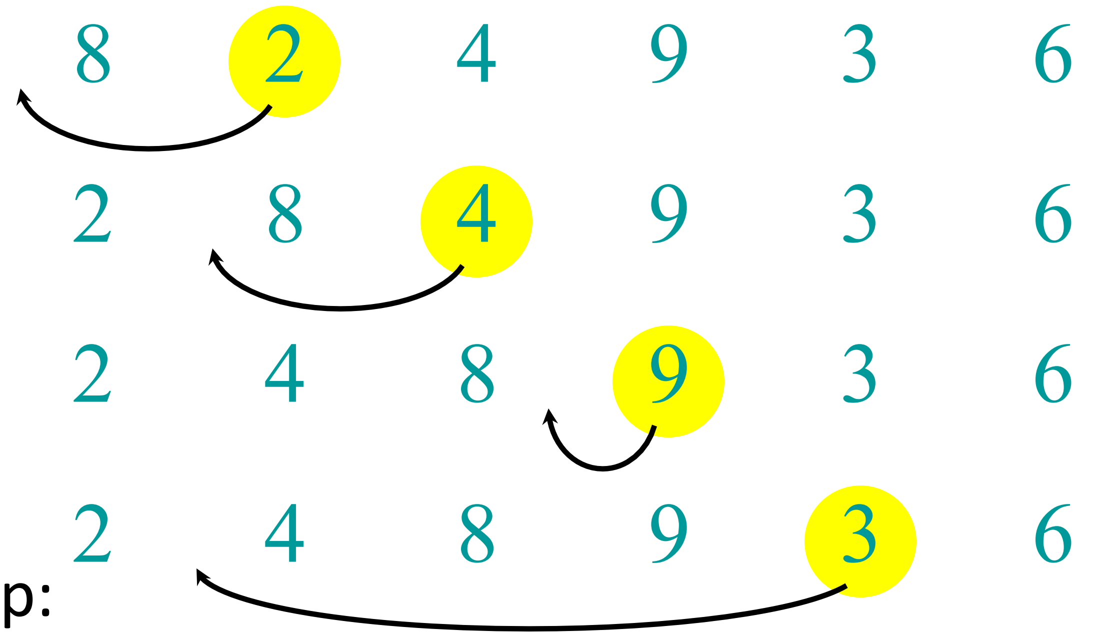
Total:

- 4 comparison
- 8 memory assignments

Insertion sort: example



Insertion sort: example



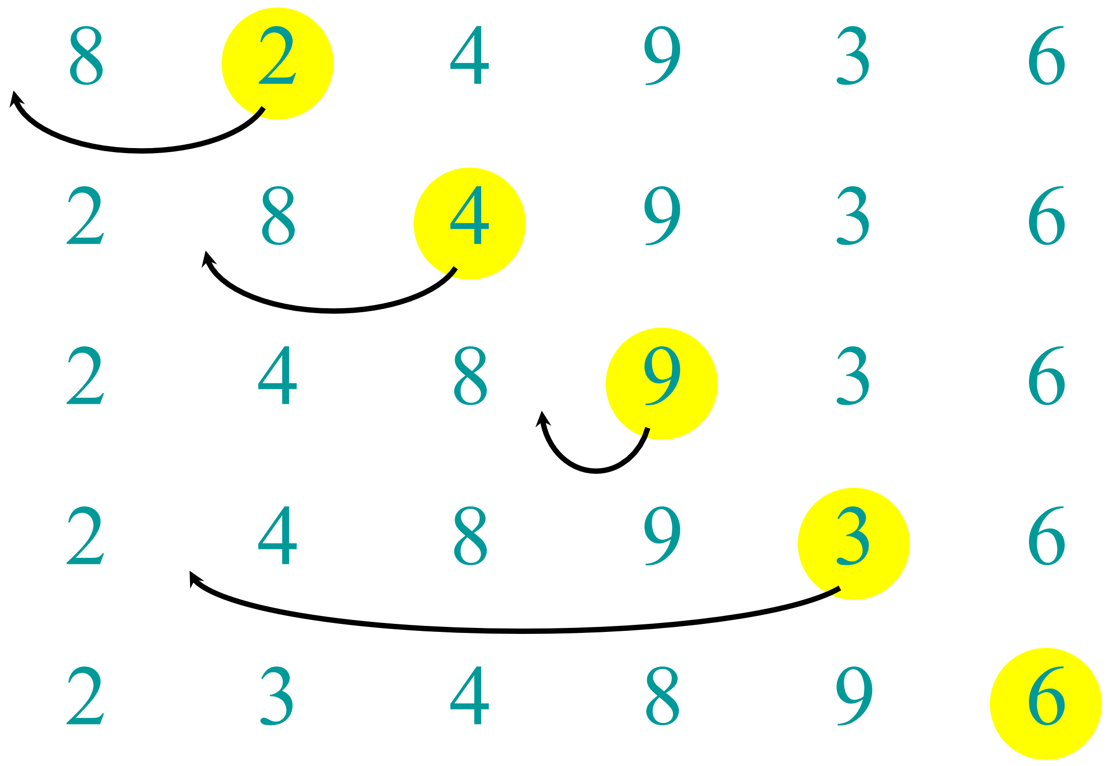
In this step:

- 4 comparison
- 5 memory assignments

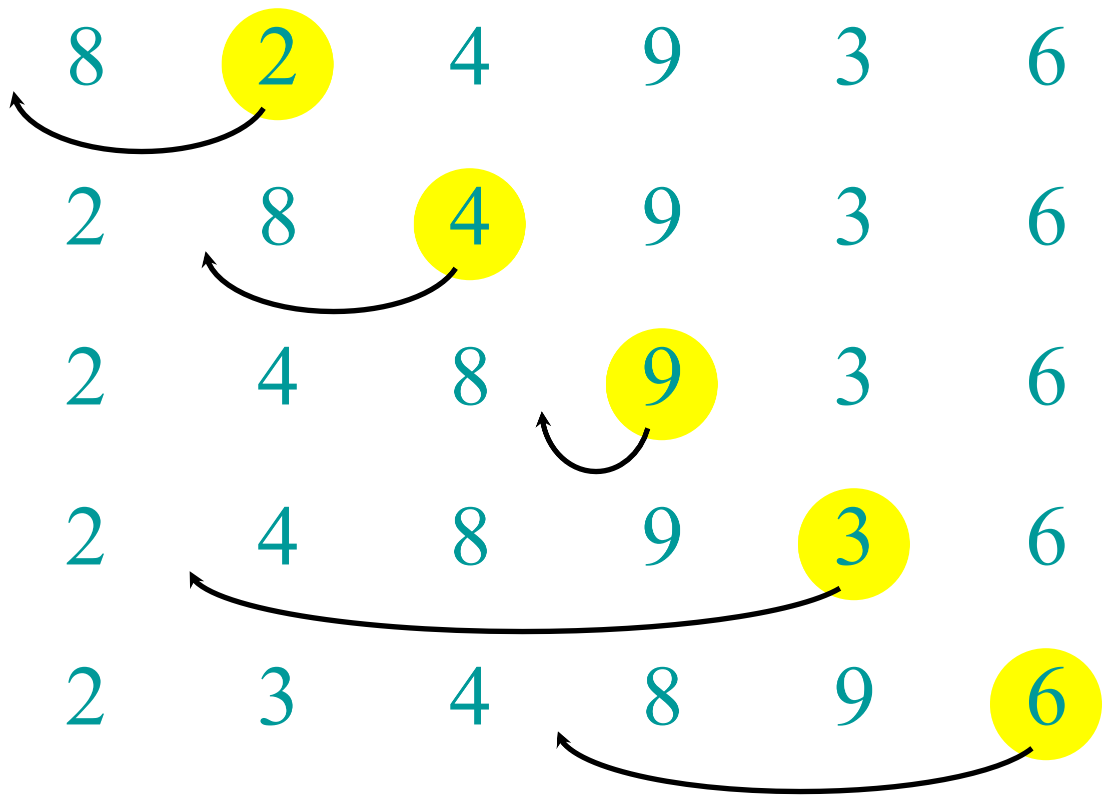
Total:

- 8 comparison
- 13 memory assignments

Insertion sort: example



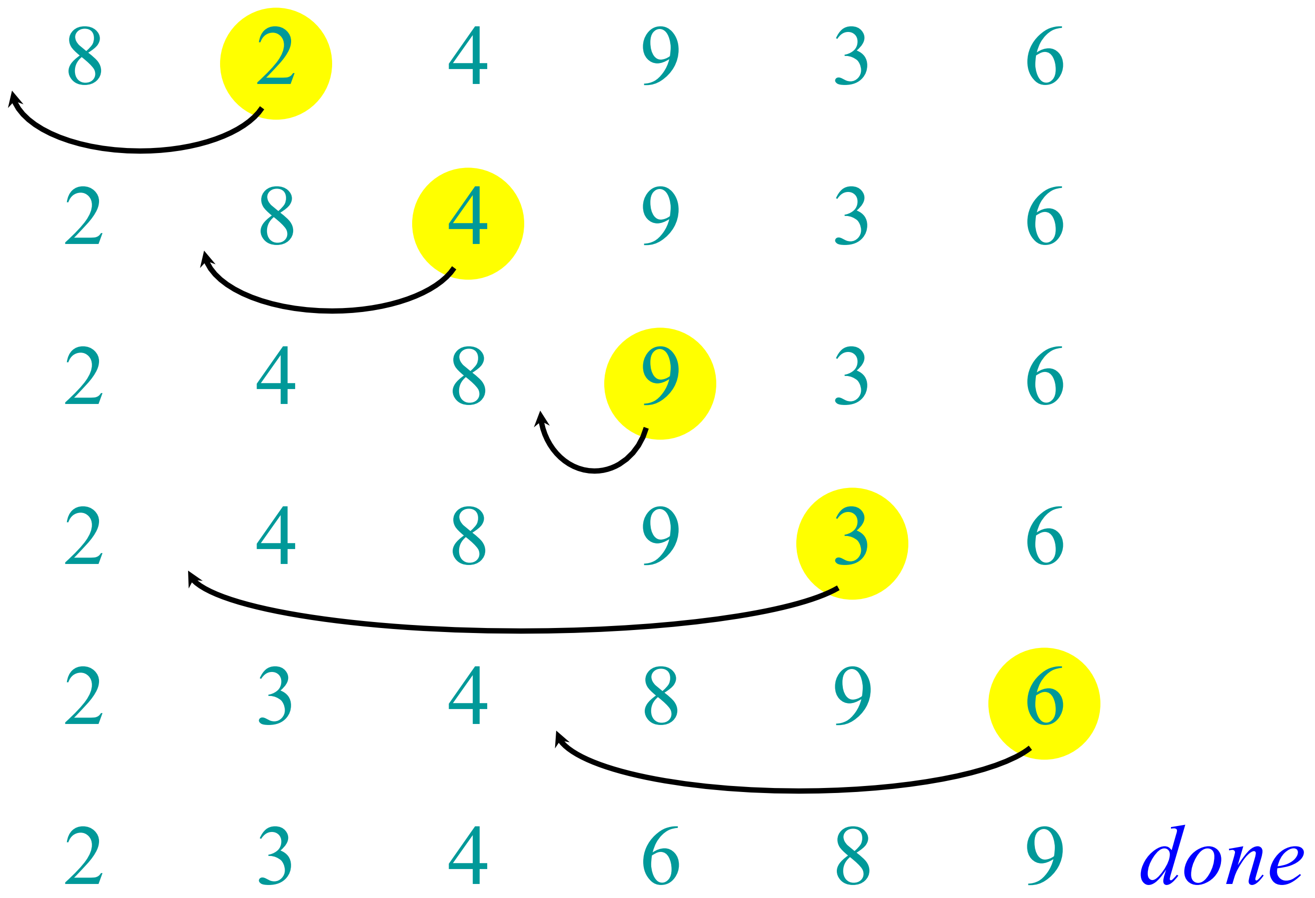
Insertion sort: example



In this step:

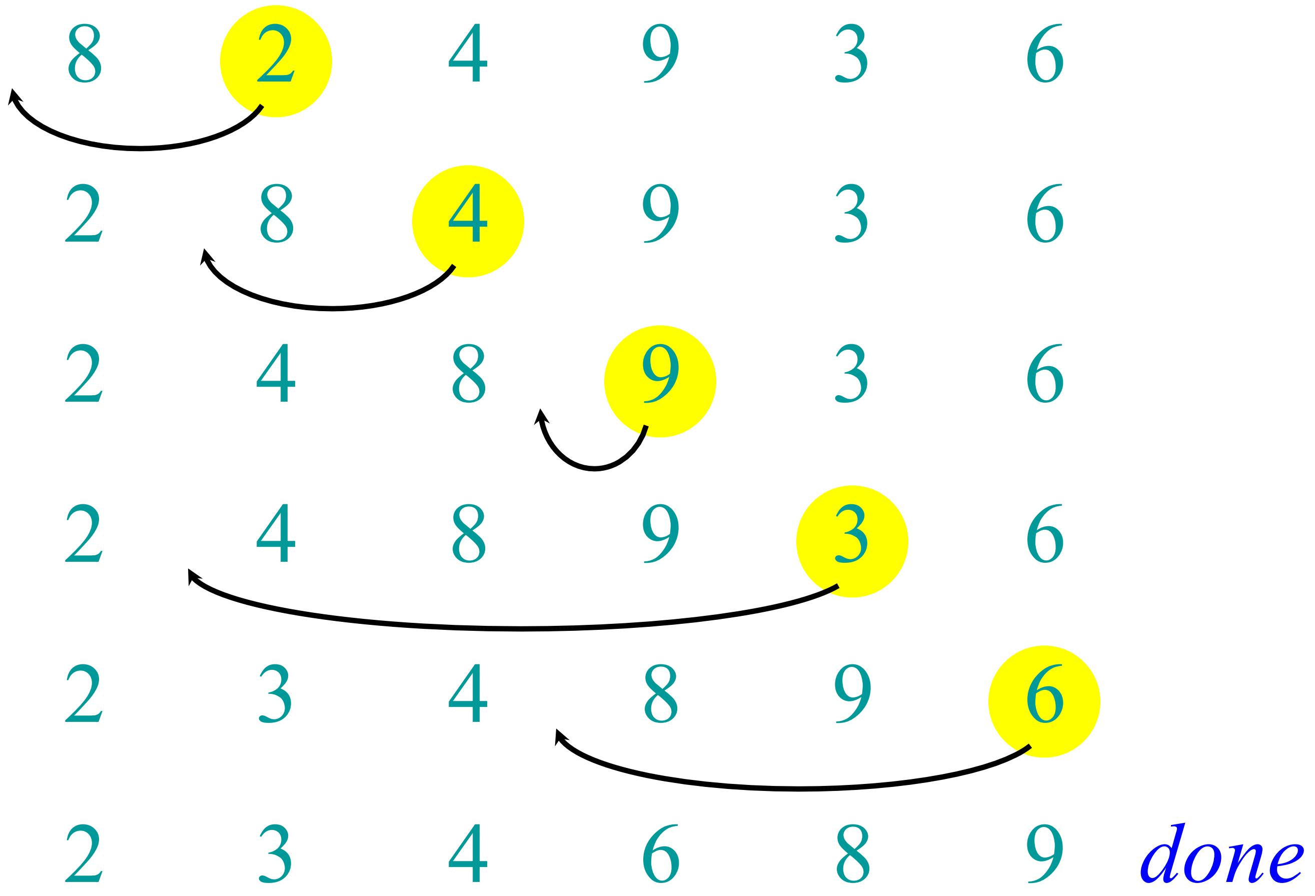
- 3 comparison
- 4 memory assignments

Insertion sort: example



Total:
- 11 comparison
- 17 memory assignments

Insertion sort: example



Total memory words used = 7 (6 for input + 1 for each i, j, key)

Running time

- The running time for an execution of the algorithm depends on the *size* of input
  - "Size" is a loaded term" generally depends on the context (E.g., for sorting #of items in the list)
- We characterize running time and memory use with respect to the size of the input
  - Input size generally denoted as n
  - $T_A(n)$ denote the execution time of algorithm A on input of size n
- We want to obtain upper bounds to execution time for an algorithm for any input of a given size

Types of analysis

- Worst case: $T_A(n)$ = maximum number of operations executed by algorithm A on input of size n
- Average case: $\overline{T_A}(n)$ = expected number of operations executed by algorithm A on input of size n
  - Need assumption on the statistical distribution of the input
  - Expectation with respect to the randomness of the input and of the algorithm (if any)
  - For deterministic algorithms or Montecarlo randomized algorithms, assuming all input are equally likely commonly referred as average time
  - For non-uniform input distribution and for Las Vegas random algorithm commonly referred as expected time
- Best case: we do not care about this ☺

Asymptotic analysis

- We are interested in the order of magnitude of $T(n)$ with respect to n
- Using asymptotic notation, we characterize the growth of $T(n)$ as $n \to \infty$
  - We focus only on the highest order term of the running time complexity expression
    - The high order term will dominate low order terms when the input is sufficiently large
    - We also discard constant coefficients
- Asymptotic notation is not an estimate of the true execution time!
  - It is a way of bounding $T(n)$

Big-O notation

- Let f and g be functions *f, g: N → R⁺*. Say that. *f(n) = O(g(n))* if some positive integers *c* and n_o exist such that for every integer $n \geq n_o$
  - $f(n) \leq c\, g(n)$
  - When f(n) = O(g(n)) we say that g(n) is an asymptotic upper bound for f(n), to emphasize that we are suppressing constant factors
- Intuitively, f(n) = O(g(n)) means that f is less than or equal to g if we disregard differences up to a constant factor.
- Used to formalize tight upper bounds

Big-O notation Example

Example: Consider the following running time:

- $f(n) = 4n^3 + 30n^2 + 200n + 1000$
- Then $f(n) = O(?)$
  - $f(n) = O(n^3)$

Example

- If the time complexity is:
  - $f(n) = 4n^3 + 30n^2 + 200n + 1000$, then
  - $f(n) = O(n^3)$
- Is f(n) always less than c g(n) for some n?
  - That is there exists c and n_0 such that, is $f(n) \leq c\, g(n)$, for all $n \geq n_0$?
  - Try n=0; we get $1000 \leq 0$ (c x 0)
  - Try n=10; we get $4000 + 3000 + 2000 + 1000 \leq c\ 1000$
    - $10000 \leq 1000c$. if c is 10 or more.
    - So, if c = 10, then true whenever $n \geq n_0 = 10$

Example continued

- What if we have:
  - $f(n) = 4n^3 + 30n^2 + 200n + 1000$, then
  - $f(n) = O(n^2)$?
    - Let's pick n = 10
    - $10000 \leq 100c$
      - True if c = 100. But then what if n is bigger, such as 100
      - Then we get $4{,}321{,}000 \leq 10{,}000c$
        - » Now c has to be 432.1
        - » So, this fails and hence f(n) is not $O(n^2)$.
        - » Note that you must fix c and n_o. You cannot keep changing c. If you started with a larger c, then you would have to get a bigger n, but eventually you would always cause the inequality to fail.

Example Continued

- Note that since f(n) = O(n^3), then it would trivially hold for O(n^4), O(n^5), and 2^n
- The big-O notation does not require that the upper bound be "tight" (i.e., as small as possible)
- However, when we characterizing the performance of an algorithm, a tight bound is desirable.
- Some very loose bounds are rather trivial!

Logarithms and asymptotic notation

- $\log_2 n$ and $\log_x n$, for any x, differ from each other by a constant
  - When using logarithms in Big-O notation, e.g., $O(\log n)$, we disregard the basis of the logarithm
- Note that log n grows much more slowly than any polynomial like n or n^2.
- An exponential like 2^n or 4^{2n} grows much faster than any polynomial.

Small-o notation

Let f and g be functions f, g: $N \rightarrow R^+$. We say that $f(n) = o(g(n))$ if:

$$\lim_{n \rightarrow \infty} f(n)/g(n) = 0$$

- This is equivalent to saying that $f(n) = o(g(n))$ then, for any real number $c > 0$, a number n_o exists where $f(n) < cg(n)$ for all $n \geq n_o$.
- It is a way of formalizing a loose upper bound while Big-O notation should formalize tight ones

Big vs Small o Notation

- Big-O notation says that one function is asymptotically no more than another
  - Think of it as $\leq$.
- Small-o notation says that one function is asymptotically less than another
  - This of it as $<$
  - A small-o bound is always also a big-O bound (not vice-versa)
    - Just as if $x < y$ then $x \leq y$ ($x \leq y$ does not mean that $x < y$)

Ω notation

- Let f and g be functions f, g: $N \rightarrow R^+$. Say that f(n) = Ω (g(n)) if positive integers *c* and n_o exist such that for every integer n $\geq n_o$
  - f(n) $\geq$ c g(n)
  - When f(n) = Ω (g(n)) we say that g(n) is an asymptotic lower bound for f(n), to emphasize that we are suppressing constant factors
- Intuitively, f(n) = Ω (g(n)) means that f is greater than or equal to g if we disregard differences up to a constant factor.
- Used to formalize lower bounds

Small-ω notation

Let f and g be functions *f, g: N → R⁺*. We say that *f(n) = ω (g(n))* if:

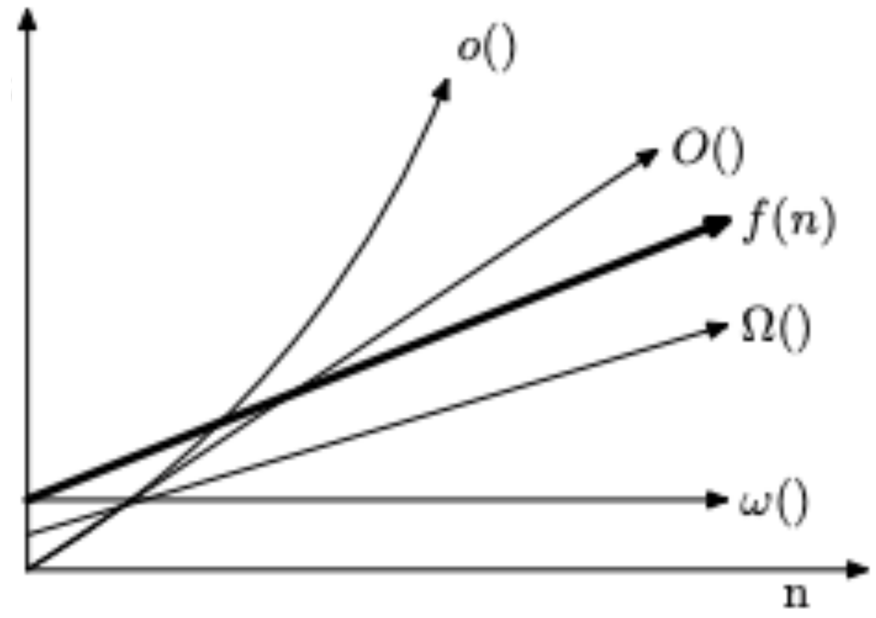
$$\lim_{n \to \infty} f(n)/g(n) = \infty$$

- This is equivalent to saying that f(n) = ω (g(n)) then, for any real number c > 0, a number n_o exists where f(n) > cg(n) for all $n \geq n_o$.
- It is a way of formalizing a loose lower bound while Big-Ω notation should formalize tight ones

θ notation

- Combines the information of big-O and Ω notation
- Let f and g be functions f, g: $N \rightarrow R^{+}$. Say that f(n) = θ (g(n)) if positive integers c_1 ,c_2 and n_o exist such that for every integer $n \geq n_o$
  - $c_1 g(n) \leq f(n) \leq c_2 g(n)$
  - If f(n) = Ω (g(n)) and f(n) = O (g(n)) then f(n) = θ (g(n))
- f(n) = θ (g(n)) tightly characterizes the asymptotic time complexity of f(n)

Relations between asymptotic



A word of caution

- So do constant terms not actually matter?
- Example: Multiplication of $n \times n$ square matrices
  - Naïve matrix multiplication algorithm A_S has complexity $O(n^3)$
  - Fastest asymptotical know algorithm A_G (Le Galle 2017) has complexity $O(n^{2.3753})$
- Which one is better?
  - A_G is asymptotically faster than A_S but...
  - due to constant terms A_G is actually faster than A_S only for values of n close to the number of atoms in the universe

Back to Insertion sort

- Worst case: Input reverse sorted

$$T(n) \leq \sum_{j=2}^{n} 2j \leq \sum_{i=2}^{n} 2n) \leq O(n^2)$$

  – At most $j-1$ comparisons in the j-th cycle iteration
  – At most $j+1$ data movements

- Average case: All permutation equally likely

$$\bar{T}(n) \leq O(n^2)$$

Proofs

Proofs are a big part of this class

- A proof is a convincing logical argument
  - Proofs in this class need to be clear, formal but not excessively
    - The level of formalism of the book is a great guideline!
- Types of Proofs
  - A $\Leftrightarrow$ B means A if and only if B
    - Prove A $\Rightarrow$ B and prove B $\Rightarrow$ A
  - Proof by counterexample (prove false via an example)
  - Proof by construction
  - Proof by contradiction
  - Proof by induction

Proof by induction

- Find a base case for the induction and prove it (e.g. i=0)
- Assume inductively the statement holds for $i \geq 0$
- Verify that the property holds for $i + 1$ in the inductive step
  - Proof should use the inductive hypothesis
- Induction very useful in the analysis of algorithms using a recursive strategy

Proofs: Example 1

- Prove for every graph G sum of degrees of all nodes is even
  - Take a minute to prove it or at least convince yourself it is true
  - This is a proof by induction
    - Their informal reasoning: every edge you add touches two vertices and increases the degree of both of these by 1 (i.e., you keep adding 2)
  - A proof by induction means showing 1) it is true for some base case and then 2) if true for any *n* then it is true for *n*+1
    - So spend a minute formulating the proof by induction
    - Base case: 0 edges in G means sum-degrees=0, is even
    - Induction step: if sum-degrees even with *n* edges then show even with n+1 edges
      - When you add an edge, it is by definition between two vertices (but can be the same). Each vertex then has its degree increase by 1, or 2 overall
      - even number + 2 = even (we will accept that for now)

Another Proof by Induction Example

- Prove that $n^2 \geq 2n$ for all n 2, 3, …
- Base case (n=2): $2^2 \geq 2x2$? Yes.
- Assume true for n=m and then show it must also be true for n=m+1
  - So we start with $m^2 \geq 2m$ and assume it is true
  - we must show that this requires $(m+1)^2 \geq 2(m+1)$
    - Rewriting we get: $m^2+2m+1 \geq 2m+2$
    - Simplifying a bit we get: $m^2 \geq 1$.
    - So, we need to show that $m^2 \geq 1$ given that $m^2 \geq 2m$
      - If $2m \geq 1$, then we are done. Is it?
      - Yes, since m itself ≥ 2

Proofs: Example 2

For any two sets A and B, $\overline{(A \cup B)} = \bar{A} \cap \bar{B}$

(Theorem 0.20, p 20)

- We prove sets are equal by showing that they have the same elements
- What proof technique to use? Any ideas?
- Prove in each direction:
  - First prove forward direction then backward directions
    - Show if element x is in one of the sets then it is in the other
  - We will do in words, but not as informal as it sounds since we are really using formal definitions of each operator

Proof: Example 2

- $\overline{(A \cup B)} = \bar{A} \cap \bar{B}$
- Forward direction (LHS → RHS):
  - Assume x $\in \overline{(A \cup B)}$
  - Then x is not in (A $\cup$ B) [defn. of complement]
  - Then x is not in A and x is not in B [defn. of union]
  - So x is in Ā and x is in $\overline{B}$ and hence is in RHS
- Backward direction (RHS → LHS)
  - Assume x $\in \bar{A} \cap \bar{B}$
  - So x $\in$ Ā and x $\in \overline{B}$ [defn. of intersection]
  - So x $\notin$ A and x $\notin$ B [defn. of complement]
  - So x not in union (A $\cup$ B) [defn. of union]
  - So x must be its complement [defn. of complement]
- So we are done!

Proofs: Example 3

- For every even number n > 2, there is a 3-regular graph with n nodes
  - A graph is k-regular if every node has degree k
- We will use a proof by construction
  - Many theorems say that a specific type of object exists. One way to prove it exists is by constructing it.
  - May sound weird, but this is by far the most common proof technique we will use in this course
    - We may be asked to show that some property is true. We may need to construct a model which makes it clear that this property is true

Proof: Example 3 continued

- Can you construct such a graph for n=4, 6, 8?
  - Try now.
  - If you see a pattern, then generalize it and that is the proof.
  - Hint: place the nodes into a circle
- Solution:
  - Place the nodes in a circle and then connect each node to the ones next to it, which gives us a 2-regular graph.
  - Then connect each node to the one opposite it and you are done. This is guaranteed to work because if the number of nodes is even, the opposite node will always get hit exactly once.
    - The text describes it more formally.
    - Note that if it was odd, this would not work.

Proof: Example 4

Jack sees Jill, who has come in from outside. Since Jill is not wet he concludes it is not raining

- This is a proof by contradiction.
    - To prove a theorem true by contradiction, assume it is false and show that leads to a contradiction
    - In this case, that translates to assume it is raining and look for contradiction
  - If we know that if it were raining then Jill would be wet, we have a contradiction because Jill is not wet.
  - That is the process, although not a very good example (what if she left the umbrella at the door!)
  - This case is perhaps a bit confusing. Lets go to a more mathematical example …

Prove Square Root of 2 Irrational

Proof by contradiction, assume it is rational

- Rational numbers can be written as m/n for integer m, n
- Assume with no loss of generality we reduce the fraction

$$\sqrt{2} = m/n$$

  - This means that m and n cannot both be even
    - If so, 2 goes into both so reduce it
- Then do some math

$$n\sqrt{2} = m$$

  - n = m
  - $2n^2 = m^2$
  - This means that m^2 is even and thus m must be even
    - Since odd x odd is odd

Prove Square Root of 2 Irrational

- So $2n^2 = m^2$ and m is even
- Any odd number can be written as 2k for some integer k, so:
  - $2n^2 = (2k)^2 = 4k^2$ Then divide both sides by 2
  - $n^2 = 2k^2$
  - But now we can say that n^2 is even and hence n must be even
- We just showed that m and n must both be even, but since we started with a reduced fraction, that is a contradiction.
  - Thus it cannot be true that $\sqrt{2}$ is rational