Resource Consumption Analysis of Algorithms (Lecture 2)

Anil Seth

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.
- Processor time is directly proportional to the time taken by the program before giving the output.

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.
- Processor time is directly proportional to the time taken by the program before giving the output.

  A program consuming less processor time gives its output faster.

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.
- Processor time is directly proportional to the time taken by the program before giving the output.

  A program consuming less processor time gives its output faster.
- An algorithm is more (time/space) efficient if its implementation consumes lesser time/space resources.

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.
- Processor time is directly proportional to the time taken by the program before giving the output.

  A program consuming less processor time gives its output faster.
- An algorithm is more (time/space) efficient if its implementation consumes lesser time/space resources.
- We would like to estimate efficiency of an algorithm from its description, that is without implementing it.

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.
- Processor time is directly proportional to the time taken by the program before giving the output.

  A program consuming less processor time gives its output faster.
- An algorithm is more (time/space) efficient if its implementation consumes lesser time/space resources.
- We would like to estimate efficiency of an algorithm from its description, that is without implementing it.
- This is called time/space complexity analysis of an algorithm.

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.
- Processor time is directly proportional to the time taken by the program before giving the output.

  A program consuming less processor time gives its output faster.
- An algorithm is more (time/space) efficient if its implementation consumes lesser time/space resources.
- We would like to estimate efficiency of an algorithm from its description, that is without implementing it.
- This is called time/space complexity analysis of an algorithm.
- It is also useful in comparing different algorithms for the same problem.

Resources

- Executing a program on a computer consumes resources, for example: processor time and memory space.
- Processor time is directly proportional to the time taken by the program before giving the output.

  A program consuming less processor time gives its output faster.
- An algorithm is more (time/space) efficient if its implementation consumes lesser time/space resources.
- We would like to estimate efficiency of an algorithm from its description, that is without implementing it.
- This is called time/space complexity analysis of an algorithm.
- It is also useful in comparing different algorithms for the same problem.

Time Complexity

- By far, the most commonly analyzed resource is the time taken by an algorithm.

Time Complexity

- By far, the most commonly analyzed resource is the time taken by an algorithm.
- Time complexity is equal to summation of the time taken in (also called cost of) each step, performed by the algorithm before giving the output.

Time Complexity

- By far, the most commonly analyzed resource is the time taken by an algorithm.
- Time complexity is equal to summation of the time taken in (also called cost of) each step, performed by the algorithm before giving the output.
- Therefore to analyze time complexity, we need to assign cost to each step in our model of computation.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.
- Each register is of unbounded length. That is, it can hold an arbitrary integer.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.
- Each register is of unbounded length. That is, it can hold an arbitrary integer.
- Each memory location has a unique address.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.
- Each register is of unbounded length. That is, it can hold an arbitrary integer.
- Each memory location has a unique address.

  It can be accessed by providing its address.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.
- Each register is of unbounded length. That is, it can hold an arbitrary integer.
- Each memory location has a unique address.

  It can be accessed by providing its address.
- A program is a finite sequence of instructions and executes sequentially, instruction by instruction.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.
- Each register is of unbounded length. That is, it can hold an arbitrary integer.
- Each memory location has a unique address.

  It can be accessed by providing its address.
- A program is a finite sequence of instructions and executes sequentially, instruction by instruction.

  A pointer called program counter, tracks the current instruction in the program being executed.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.
- Each register is of unbounded length. That is, it can hold an arbitrary integer.
- Each memory location has a unique address.

  It can be accessed by providing its address.
- A program is a finite sequence of instructions and executes sequentially, instruction by instruction.

  A pointer called program counter, tracks the current instruction in the program being executed.
- Program is assumed to be outside memory so that it can't modify itself.

RAM an idealized Computer

- Random Access Machine (RAM) is an abstraction of a conventional single processor machine.
- Memory in a RAM is organized as a sequence of words (memory locations). These are also called registers.
- Each register is of unbounded length. That is, it can hold an arbitrary integer.
- Each memory location has a unique address.

  It can be accessed by providing its address.
- A program is a finite sequence of instructions and executes sequentially, instruction by instruction.

  A pointer called program counter, tracks the current instruction in the program being executed.
- Program is assumed to be outside memory so that it can't modify itself.

About RAM ...

- Each instruction in the program performs some simple operation on registers/memory locations.

About RAM ...

- Each instruction in the program performs some simple operation on registers/memory locations.
- It may be reading from a memory location, writing into a memory location, doing some arithmetic operation on registers or checking if value in some register is zero or not.

About RAM ...

- Each instruction in the program performs some simple operation on registers/memory locations.
- It may be reading from a memory location, writing into a memory location, doing some arithmetic operation on registers or checking if value in some register is zero or not.
- binary operations are done with respect to the first register. That is, it contains one of the operands and it also stores the result.

About RAM ...

- Each instruction in the program performs some simple operation on registers/memory locations.
- It may be reading from a memory location, writing into a memory location, doing some arithmetic operation on registers or checking if value in some register is zero or not.
- binary operations are done with respect to the first register. That is, it contains one of the operands and it also stores the result.
- For a given problem, we may assume the convention that input is presented in some set of registers and when the machine halts the output is also stored in some predefined set of registers.

About RAM ...

- Each instruction in the program performs some simple operation on registers/memory locations.
- It may be reading from a memory location, writing into a memory location, doing some arithmetic operation on registers or checking if value in some register is zero or not.
- binary operations are done with respect to the first register. That is, it contains one of the operands and it also stores the result.
- For a given problem, we may assume the convention that input is presented in some set of registers and when the machine halts the output is also stored in some predefined set of registers.
- Any program can, in principle, be converted to an equivalent RAM program.

Cost model for RAM

- For the cost model, we assume that the numbers stored in registers are in binary.

Cost model for RAM

- For the cost model, we assume that the numbers stored in registers are in binary.
- The cost model, assigns a cost to each operation of RAM.

Cost model for RAM

- For the cost model, we assume that the numbers stored in registers are in binary.
- The cost model, assigns a cost to each operation of RAM.
- There are two commonly use cost models.

Cost model for RAM

- For the cost model, we assume that the numbers stored in registers are in binary.
- The cost model, assigns a cost to each operation of RAM.
- There are two commonly use cost models.

  Uniform cost model and logarithmic cost model.

Cost model for RAM

- For the cost model, we assume that the numbers stored in registers are in binary.
- The cost model, assigns a cost to each operation of RAM.
- There are two commonly use cost models.

  Uniform cost model and logarithmic cost model.
- In the uniform model, each operation on registers costs unit time.

Cost model for RAM

- For the cost model, we assume that the numbers stored in registers are in binary.
- The cost model, assigns a cost to each operation of RAM.
- There are two commonly use cost models.

  Uniform cost model and logarithmic cost model.
- In the uniform model, each operation on registers costs unit time.
- In logarithmic model each operation takes time proportional to length of the values stored in the registers involved in the operation.

Cost model for RAM

- For the cost model, we assume that the numbers stored in registers are in binary.
- The cost model, assigns a cost to each operation of RAM.
- There are two commonly use cost models.

  Uniform cost model and logarithmic cost model.
- In the uniform model, each operation on registers costs unit time.
- In logarithmic model each operation takes time proportional to length of the values stored in the registers involved in the operation.

Cost model for RAM Continued ...

- The uniform model abstracts the conventional computer if the numbers stored in registers are bounded. For example, say, they can be represented with 64 bits.

Cost model for RAM Continued ...

- The uniform model abstracts the conventional computer if the numbers stored in registers are bounded. For example, say, they can be represented with 64 bits.
- While RAM model is theoretical, cost on RAM model relates well with the cost of running the program on an actual (single processor) machine.

Cost model for our psuedo language

- All our pseudo-programs can be translated easily into a RAM program.

Cost model for our psuedo language

- All our pseudo-programs can be translated easily into a RAM program.
- A variable corresponds to a memory location. Its value at any point in execution is content of this memory location.

Cost model for our psuedo language

- All our pseudo-programs can be translated easily into a RAM program.
- A variable corresponds to a memory location. Its value at any point in execution is content of this memory location.
- RAM cost model translates to cost model for our (pseudo) programming language instructions as follows.

Cost model for our psuedo language

- All our pseudo-programs can be translated easily into a RAM program.
- A variable corresponds to a memory location. Its value at any point in execution is content of this memory location.
- RAM cost model translates to cost model for our (pseudo) programming language instructions as follows.
- If values in variables stored are bounded then cost of comparison, increment, decrement and arithmetic operations is unit. (This corresponds to uniform cost model of RAM).

Cost model for our psuedo language

- All our pseudo-programs can be translated easily into a RAM program.
- A variable corresponds to a memory location. Its value at any point in execution is content of this memory location.
- RAM cost model translates to cost model for our (pseudo) programming language instructions as follows.
- If values in variables stored are bounded then cost of comparison, increment, decrement and arithmetic operations is unit. (This corresponds to uniform cost model of RAM).

Cost model for our psuedo language continued ...

- If values in variables stored are unbounded then cost of operations is as follows.
  (Here $|x|$ is the number of bits needed to write x).

Cost model for our psuedo language continued ...

- If values in variables stored are unbounded then cost of operations is as follows.
  (Here $|x|$ is the number of bits needed to write x).
  - Cost of $x + 1, x - 1$ is $|x|$.

Cost model for our psuedo language continued ...

- If values in variables stored are unbounded then cost of operations is as follows.
  (Here $|x|$ is the number of bits needed to write x).
    - Cost of $x + 1, x - 1$ is $|x|$.
    - Cost of $x \leq y$ is $|x| + |y|$.

Cost model for our psuedo language continued ...

- If values in variables stored are unbounded then cost of operations is as follows.
  (Here $|x|$ is the number of bits needed to write x).
    - Cost of $x + 1, x - 1$ is $|x|$.
    - Cost of $x \leq y$ is $|x| + |y|$.
    - Cost of $x + y$ is $|x| + |y|$.

Cost model for our psuedo language continued ...

- If values in variables stored are unbounded then cost of operations is as follows.
  (Here $|x|$ is the number of bits needed to write x).
    - Cost of $x+1, x-1$ is $|x|$.
    - Cost of $x \leq y$ is $|x|+|y|$.
    - Cost of $x+y$ is $|x|+|y|$.
    - Cost of $x*y$ is $|x| \cdot |y|$.

Cost model for our psuedo language continued ...

- If values in variables stored are unbounded then cost of operations is as follows.
  (Here $|x|$ is the number of bits needed to write x).
  - Cost of $x + 1, x - 1$ is $|x|$.
  - Cost of $x \leq y$ is $|x| + |y|$.
  - Cost of $x + y$ is $|x| + |y|$.
  - Cost of $x * y$ is $|x| \cdot |y|$.
  - Cost of checking $x > 0$ is $|x|$.

Cost model for our psuedo language continued ...

- If values in variables stored are unbounded then cost of operations is as follows.
  (Here $|x|$ is the number of bits needed to write x).
  - Cost of $x+1, x-1$ is $|x|$.
  - Cost of $x \leq y$ is $|x|+|y|$.
  - Cost of $x+y$ is $|x|+|y|$.
  - Cost of $x * y$ is $|x| \cdot |y|$.
  - Cost of checking $x > 0$ is $|x|$.
  - Cost of assignment $x = y$ is $max\{|x|, |y|\}$.

Complexity measure as a function

- The time taken by an algorithm depends on the input. So, the time complexity of an algorithm is expected to be a function of actual input to the algorithm.

Complexity measure as a function

- The time taken by an algorithm depends on the input. So, the time complexity of an algorithm is expected to be a function of actual input to the algorithm.
- However, it is more convenient to define the time complexity as a function of *size* of the input (to algorithm), rather than of the input itself.

Complexity measure as a function

- The time taken by an algorithm depends on the input. So, the time complexity of an algorithm is expected to be a function of actual input to the algorithm.
- However, it is more convenient to define the time complexity as a function of *size* of the input (to algorithm), rather than of the input itself.
- This results in a more succinct description of the time complexity function.

Complexity measure as a function

- The time taken by an algorithm depends on the input. So, the time complexity of an algorithm is expected to be a function of actual input to the algorithm.
- However, it is more convenient to define the time complexity as a function of *size* of the input (to algorithm), rather than of the input itself.
- This results in a more succinct description of the time complexity function.

Input Size

- Size of the input is roughly the number of bits required to specify the input.

Input Size

- Size of the input is roughly the number of bits required to specify the input.
- For our analysis, it is sufficient to take size as a number *proportional* to the number of bits required.

Input Size

- Size of the input is roughly the number of bits required to specify the input.
- For our analysis, it is sufficient to take size as a number *proportional* to the number of bits required.
- This allows us to choose a more convenient/natural parameter of the input.

Input Size

- Size of the input is roughly the number of bits required to specify the input.
- For our analysis, it is sufficient to take size as a number *proportional* to the number of bits required.
- This allows us to choose a more convenient/natural parameter of the input.
- To an extent, choice of size of input depends on the context we are interested in.

Input Size

- Size of the input is roughly the number of bits required to specify the input.
- For our analysis, it is sufficient to take size as a number *proportional* to the number of bits required.
- This allows us to choose a more convenient/natural parameter of the input.
- To an extent, choice of size of input depends on the context we are interested in.

Example: Time complexity Analysis of Insert

- In this case, the input instance is (A, i, n).

Example: Time complexity Analysis of Insert

- In this case, the input instance is (A, i, n).
- We assume that each array element is stored in a single (or in a fixed number of) memory words.

Example: Time complexity Analysis of Insert

- In this case, the input instance is (A, i, n).
- We assume that each array element is stored in a single (or in a fixed number of) memory words.
- So the size of the Array is proportional to n.

Example: Time complexity Analysis of Insert

- In this case, the input instance is (A, i, n).
- We assume that each array element is stored in a single (or in a fixed number of) memory words.
- So the size of the Array is proportional to n.
- We take the input size as n, the number of elements in the array.

Example: Time complexity Analysis of Insert

- In this case, the input instance is (A, i, n).
- We assume that each array element is stored in a single (or in a fixed number of) memory words.
- So the size of the Array is proportional to n.
- We take the input size as n, the number of elements in the array.
- Comparisons and shifting on such elements is done in unit time. Uniform cost model is justified.

Example: Time complexity Analysis of Insert

- In this case, the input instance is (A, i, n).
- We assume that each array element is stored in a single (or in a fixed number of) memory words.
- So the size of the Array is proportional to n.
- We take the input size as n, the number of elements in the array.
- Comparisons and shifting on such elements is done in unit time. Uniform cost model is justified.

Example: Time complexity Analysis of Insert

If body of the while loop is executed h times then the total cost of executing each instruction of Insert(A, i, n) is show below.

```
Insert (A,i,n)
  k=A[i] // cost $c_1$
  j=i // cost $c_2$
  while (j<n) and (A[j+1] < k) do // cost $(h+1)c_3$
    A[j] = A[j+1]  //cost $h \cdot c_4$
    j = j+1  //cost $h \cdot c_5$
  A[j]=k //cost 1
```

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$
- $= d_1 + d_2 \cdot h$, for some constants d_1, d_2.

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$
- $= d_1 + d_2 \cdot h$, for some constants d_1, d_2.
- What does h depend on?

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$
- $= d_1 + d_2 \cdot h$, for some constants d_1, d_2.
- What does h depend on?
  - It is the number of elements in $A[i+1], \ldots, A[n]$ which are smaller than $A[i]$.

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$
- $= d_1 + d_2 \cdot h$, for some constants d_1, d_2.
- What does h depend on?
  - It is the number of elements in $A[i+1], \ldots, A[n]$ which are smaller than $A[i]$.
  - The number h ranges between 0 and $n - i$, depending on A.

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$
- $= d_1 + d_2 \cdot h$, for some constants d_1, d_2.
- What does h depend on?
    - It is the number of elements in $A[i+1], \ldots, A[n]$ which are smaller than $A[i]$.
    - The number h ranges between 0 and $n - i$, depending on A.
- We may like our bound to be independent of actual array A and instead depend on n only.

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$
- $= d_1 + d_2 \cdot h$, for some constants d_1, d_2.
- What does h depend on?
  - It is the number of elements in $A[i+1], \ldots, A[n]$ which are smaller than $A[i]$.
  - The number h ranges between 0 and $n - i$, depending on A.
- We may like our bound to be independent of actual array A and instead depend on n only.
- We can do it in the following three ways.

Example: Time complexity Analysis of Insert

- This sums up to $c_1 + c_2 + c_3 + h \cdot (c_3 + c_4 + c_5)$
- $= d_1 + d_2 \cdot h$, for some constants d_1, d_2.
- What does h depend on?
  - It is the number of elements in $A[i+1], \ldots, A[n]$ which are smaller than $A[i]$.
  - The number h ranges between 0 and $n - i$, depending on A.
- We may like our bound to be independent of actual array A and instead depend on n only.
- We can do it in the following three ways.

Best and Worst Case Analysis

- **Best case analysis:** This refers to taking the complexity as the least number of steps performed (the best case) among all inputs of size n.

Best and Worst Case Analysis

- **Best case analysis:** This refers to taking the complexity as the least number of steps performed (the best case) among all inputs of size n.
    - In our example this corresponds to $h = 0$ and is d_1.

Best and Worst Case Analysis

- **Best case analysis:** This refers to taking the complexity as the least number of steps performed (the best case) among all inputs of size n.
  - In our example this corresponds to $h = 0$ and is d_1.
- **Worst case analysis:** This refers to the largest number of steps performed among all inputs of size n.

Best and Worst Case Analysis

- **Best case analysis:** This refers to taking the complexity as the least number of steps performed (the best case) among all inputs of size n.
  - In our example this corresponds to $h = 0$ and is d_1.
- **Worst case analysis:** This refers to the largest number of steps performed among all inputs of size n.
  - This in our example is $d_1 + d_2 n - d_2 i$.

Best and Worst Case Analysis

- **Best case analysis:** This refers to taking the complexity as the least number of steps performed (the best case) among all inputs of size n.
  - In our example this corresponds to $h = 0$ and is d_1.
- **Worst case analysis:** This refers to the largest number of steps performed among all inputs of size n.
  - This in our example is $d_1 + d_2n - d_2i$.

Average Case Analysis

- **Average case analysis** corresponds to the number of steps required, averaged over all inputs of size n.

Average Case Analysis

- **Average case analysis** corresponds to the number of steps required, averaged over all inputs of size n.
- We show this in our example assuming that all numbers are distinct.

Average Case Analysis

- **Average case analysis** corresponds to the number of steps required, averaged over all inputs of size n.
- We show this in our example assuming that all numbers are distinct.
- By probability theory, it is equally likely for $A[i]$ to fit in any of the $(n - i + 1)$ positions in the sorted array.

Average Case Analysis

- **Average case analysis** corresponds to the number of steps required, averaged over all inputs of size n.
- We show this in our example assuming that all numbers are distinct.
- By probability theory, it is equally likely for $A[i]$ to fit in any of the $(n-i+1)$ positions in the sorted array.
- So the average time is

Average Case Analysis

- **Average case analysis** corresponds to the number of steps required, averaged over all inputs of size n.
- We show this in our example assuming that all numbers are distinct.
- By probability theory, it is equally likely for $A[i]$ to fit in any of the $(n - i + 1)$ positions in the sorted array.
- So the average time is

$$= \frac{1}{n - i + 1} \sum_{h=0}^{n-i} (d_1 + h \cdot d)$$

Average Case Analysis

- **Average case analysis** corresponds to the number of steps required, averaged over all inputs of size n.
- We show this in our example assuming that all numbers are distinct.
- By probability theory, it is equally likely for $A[i]$ to fit in any of the $(n-i+1)$ positions in the sorted array.
- So the average time is

$$= \frac{1}{n-i+1}\sum_{h=0}^{n-i}(d_1 + h \cdot d)$$

$$= d_1 + \frac{d_2}{2}(n-i)$$

Average Case Analysis

- **Average case analysis** corresponds to the number of steps required, averaged over all inputs of size n.
- We show this in our example assuming that all numbers are distinct.
- By probability theory, it is equally likely for $A[i]$ to fit in any of the $(n-i+1)$ positions in the sorted array.
- So the average time is

$$= \frac{1}{n-i+1}\sum_{h=0}^{n-i}(d_1 + h \cdot d)$$

$$= d_1 + \frac{d_2}{2}(n-i)$$

Comparison of different time complexity Analyses

- The three types of analyses discussed above are applicable to any algorithm. Let us briefly compare them.

Comparison of different time complexity Analyses

- The three types of analyses discussed above are applicable to any algorithm. Let us briefly compare them.
- Best case analysis is of use when we want a *lower bound* on the time required by the algorithm on all inputs.
  This is of limited use in practice.

Comparison of different time complexity Analyses

- The three types of analyses discussed above are applicable to any algorithm. Let us briefly compare them.
- Best case analysis is of use when we want a *lower bound* on the time required by the algorithm on all inputs.
  This is of limited use in practice.
- Average case analysis is an accurate measure of the performance of the algorithm.

Comparison of different time complexity Analyses

- The three types of analyses discussed above are applicable to any algorithm. Let us briefly compare them.
- Best case analysis is of use when we want a *lower bound* on the time required by the algorithm on all inputs.
  This is of limited use in practice.
- Average case analysis is an accurate measure of the performance of the algorithm.
  - But it could be mathematically very challenging even for simple algorithms.

Comparison of different time complexity Analyses

- The three types of analyses discussed above are applicable to any algorithm. Let us briefly compare them.
- Best case analysis is of use when we want a *lower bound* on the time required by the algorithm on all inputs. This is of limited use in practice.
- Average case analysis is an accurate measure of the performance of the algorithm.
  - But it could be mathematically very challenging even for simple algorithms.
  - Example above, where it is easy, is not indicative of usual scenarios.

Comparison of different time complexity Analyses

- The three types of analyses discussed above are applicable to any algorithm. Let us briefly compare them.
- Best case analysis is of use when we want a *lower bound* on the time required by the algorithm on all inputs.
  This is of limited use in practice.
- Average case analysis is an accurate measure of the performance of the algorithm.
  - But it could be mathematically very challenging even for simple algorithms.
  - Example above, where it is easy, is not indicative of usual scenarios.

Comparison of different time complexity Analyses

- Worst case analysis provides an upper bound on performance of the algorithm on all inputs. It not as accurate as average case analysis but is mathematically manageable in most cases.

Comparison of different time complexity Analyses

- Worst case analysis provides an upper bound on performance of the algorithm on all inputs. It not as accurate as average case analysis but is mathematically manageable in most cases.
- Worst case analysis is used most commonly.

Comparison of different time complexity Analyses

- Worst case analysis provides an upper bound on performance of the algorithm on all inputs. It not as accurate as average case analysis but is mathematically manageable in most cases.
- Worst case analysis is used most commonly.
- We almost always bench-mark performance of our algorithms by their worst case time complexity.

Comparison of different time complexity Analyses

- Worst case analysis provides an upper bound on performance of the algorithm on all inputs. It not as accurate as average case analysis but is mathematically manageable in most cases.
- Worst case analysis is used most commonly.
- We almost always bench-mark performance of our algorithms by their worst case time complexity.

Example: Complexity Analysis of Insertion Sort

- Having estimated the worst case running time of `subroutine Insert`(A, i, n) as $d_1 + d_2n - d_2i$,
  we are now ready to analyze running time of Insertion_Sort.

Example: Complexity Analysis of Insertion Sort

- Having estimated the worst case running time of `subroutine Insert`(A, i, n) as $d_1 + d_2 n - d_2 i$,
  we are now ready to analyze running time of Insertion_Sort.
- Let us recall the algorithm.

Example: Complexity Analysis of Insertion Sort

- Having estimated the worst case running time of `subroutine Insert`(A, i, n) as $d_1 + d_2 n - d_2 i$,
  we are now ready to analyze running time of Insertion_Sort.
- Let us recall the algorithm.

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n-i)$
```

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n-i)$
```

Its worst case running time is

$$\leq \sum_{i=n-1}^{1} [d_1 + d_2(n-i)]$$

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n-i)$
```

Its worst case running time is

$$\leq \sum_{i=n-1}^{1} [d_1 + d_2(n-i)] = d_1(n-1) + d_2 \sum_{j=1}^{n-1} j$$

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n-i)$
```

Its worst case running time is

$$\leq \sum_{i=n-1}^{1} [d_1 + d_2(n-i)] = d_1(n-1) + d_2 \sum_{j=1}^{n-1} j$$

$$= d_1(n-1) + d_2 \frac{n(n-1)}{2}$$

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n - i)$
```

Its worst case running time is

$$\leq \sum_{i=n-1}^{1} [d_1 + d_2(n - i)] = d_1(n-1) + d_2 \sum_{j=1}^{n-1} j$$

$$= d_1(n-1) + d_2 \frac{n(n-1)}{2} = \frac{d_2}{2} n^2 + (d_1 - \frac{d_2}{2})n - d_1$$

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n-i)$
```

Its worst case running time is

$$\leq \sum_{i=n-1}^{1} [d_1 + d_2(n-i)] = d_1(n-1) + d_2 \sum_{j=1}^{n-1} j$$

$$= d_1(n-1) + d_2 \frac{n(n-1)}{2} = \frac{d_2}{2} n^2 + (d_1 - \frac{d_2}{2})n - d_1$$

- So, the worst case run-time of Insertion_Sort(A,n) is bounded by $d_3 n^2 + d_4 n + d_5$,

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n-i)$
```

Its worst case running time is

$$\leq \sum_{i=n-1}^{1} [d_1 + d_2(n-i)] = d_1(n-1) + d_2 \sum_{j=1}^{n-1} j$$

$$= d_1(n-1) + d_2 \frac{n(n-1)}{2} = \frac{d_2}{2} n^2 + (d_1 - \frac{d_2}{2})n - d_1$$

- So, the worst case run-time of Insertion_Sort(A,n) is bounded by $d_3 n^2 + d_4 n + d_5$, for some constants d_3, d_4 and d_5, with $d_3 > 0$.

Example: Complexity Analysis of Insertion Sort

Insertion Sort

```
Insertion_Sort(A,n)
  for i=n-1 downto 1 do
  Insert(A,i,n) \\worst case cost $d_1 + d_2(n-i)$
```

Its worst case running time is

$$\leq \sum_{i=n-1}^{1} [d_1 + d_2(n-i)] = d_1(n-1) + d_2 \sum_{j=1}^{n-1} j$$

$$= d_1(n-1) + d_2 \frac{n(n-1)}{2} = \frac{d_2}{2} n^2 + (d_1 - \frac{d_2}{2})n - d_1$$

- So, the worst case run-time of Insertion_Sort(A,n) is bounded by $d_3 n^2 + d_4 n + d_5$, for some constants d_3, d_4 and d_5, with $d_3 > 0$.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.
- Asymptotically the above expression is $d_3 n^2$.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.
- Asymptotically the above expression is $d_3 n^2$.
- Further, it is customary to drop the constant multiple in front of n^2.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.
- Asymptotically the above expression is $d_3 n^2$.
- Further, it is customary to drop the constant multiple in front of n^2.
- It is subsumed in the term dependent on n for all large enough n, we are more interested in form of the function of n.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.
- Asymptotically the above expression is $d_3 n^2$.
- Further, it is customary to drop the constant multiple in front of n^2.
- It is subsumed in the term dependent on n for all large enough n, we are more interested in form of the function of n.
- Exact analysis is frequently difficult and ignoring a constant factor makes things easier.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.
- Asymptotically the above expression is $d_3 n^2$.
- Further, it is customary to drop the constant multiple in front of n^2.
- It is subsumed in the term dependent on n for all large enough n, we are more interested in form of the function of n.
- Exact analysis is frequently difficult and ignoring a constant factor makes things easier.
- We summarize this as saying that (worst case) time complexity of Insertion_Sort is $O(n^2)$.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.
- Asymptotically the above expression is $d_3 n^2$.
- Further, it is customary to drop the constant multiple in front of n^2.
- It is subsumed in the term dependent on n for all large enough n, we are more interested in form of the function of n.
- Exact analysis is frequently difficult and ignoring a constant factor makes things easier.
- We summarize this as saying that (worst case) time complexity of Insertion_Sort is $O(n^2)$.
- Precise meaning of $O(\cdot)$ notation will be explained in the next lecture.

Example: Complexity Analysis of Insertion Sort

- As n increases, term n^2 dominates over other terms.
- Asymptotically the above expression is d_3n^2.
- Further, it is customary to drop the constant multiple in front of n^2.
- It is subsumed in the term dependent on n for all large enough n, we are more interested in form of the function of n.
- Exact analysis is frequently difficult and ignoring a constant factor makes things easier.
- We summarize this as saying that (worst case) time complexity of Insertion_Sort is $O(n^2)$.
- Precise meaning of $O(\cdot)$ notation will be explained in the next lecture.

Exact Complexity Analysis of Insertion Sort

Unlike a common situation, in case of insertion_sort, an exact time complexity analysis is also possible.

Exact Complexity Analysis of Insertion Sort

Unlike a common situation, in case of insertion_sort, an exact time complexity analysis is also possible.

- **Definition:**
  - An inversion in an array A is a pair (i, j) s.t. $i < j$ and $A[i] > A[j]$.

Exact Complexity Analysis of Insertion Sort

Unlike a common situation, in case of insertion_sort, an exact time complexity analysis is also possible.

- **Definition:**
  - An inversion in an array A is a pair (i, j) s.t. $i < j$ and $A[i] > A[j]$.
  - The number of inversions in an array A of size n is

Exact Complexity Analysis of Insertion Sort

Unlike a common situation, in case of insertion_sort, an exact time complexity analysis is also possible.

- **Definition:**
  - An inversion in an array A is a pair (i, j) s.t. $i < j$ and $A[i] > A[j]$.
  - The number of inversions in an array A of size n is

    $$|\{(i, j) \mid i, j \leq n, (i, j) \text{ is an inversion of } A\}|$$

Exact Complexity Analysis of Insertion Sort

Unlike a common situation, in case of insertion_sort, an exact time complexity analysis is also possible.

- **Definition:**
  - An inversion in an array A is a pair (i, j) s.t. $i < j$ and $A[i] > A[j]$.
  - The number of inversions in an array A of size n is

    $$|\{(i, j) \mid i, j \leq n, (i, j) \text{ is an inversion of } A\}|$$
- Question: How many inversions are there in array [34,-9,-7,100,1]? List them.

Exact Complexity Analysis of Insertion Sort

Unlike a common situation, in case of insertion_sort, an exact time complexity analysis is also possible.

- **Definition:**
    - An inversion in an array A is a pair (i, j) s.t. $i < j$ and $A[i] > A[j]$.
    - The number of inversions in an array A of size n is

      $$|\{(i, j) \mid i, j \leq n, (i, j) \text{ is an inversion of } A\}|$$
- Question: How many inversions are there in array [34,-9,-7,100,1]? List them.
- **Exercise:** Show that running time of Insertion_Sort(A,n) is proportional to n+(number of inversions in A).

Exact Complexity Analysis of Insertion Sort

Unlike a common situation, in case of insertion_sort, an exact time complexity analysis is also possible.

- **Definition:**
  - An inversion in an array A is a pair (i, j) s.t. $i < j$ and $A[i] > A[j]$.
  - The number of inversions in an array A of size n is

    $$|\{(i, j) \mid i, j \leq n, (i, j) \text{ is an inversion of } A\}|$$
- Question: How many inversions are there in array [34,-9,-7,100,1]? List them.
- **Exercise:** Show that running time of Insertion_Sort(A,n) is proportional to n+(number of inversions in A).

Exercises

- Question: Based on result of the previous exercise, what is the running time of Insertion_Sort(A,n), when A is sorted/reverse sorted?

Exercises

- Question: Based on result of the previous exercise, what is the running time of Insertion_Sort(A,n), when A is sorted/reverse sorted?
- Exercise (optional) [assumes basic probability theory]: What is the expected number of inversions in an array of n distinct elements? Using this and the result in previous exercise find out the average case time complexity of Insertion_Sort(A,n).