## 1. Introduction: Analysing and Designing Algorithms

- When solving a problem, we are well advised first to construct an exact model in terms of which we can express allowed solutions.
- Finding such a model is already half the solution. Any branch of mathematics or science can be called into service to help model the problem domain, e.g.:
  - Simultaneous linear equations (finding currents in electrical circuits, finding stresses in frames made of concrete beams)
  - Differential equations (predicting population growth, predicting the rate at which chemicals will react)
  - Formal grammars (compiling programming languages, database queries)
  - Graphs (transportation problems, optimal scheduling)
- Once we have a suitable mathematical model, we can specify a solution in terms of that model.



## Introductory Example

- Suppose we model our problem domain by a sequence of numbers $A = a_1, a_2, \ldots, a_n$ and the solution consists in sorting them.
- Functional specification in terms of an abstract program (pseudocode):

  $A \leftarrow$ any $A'$ such that permutation ($A'$, $A$)    ascending ($A'$)
- Functional specification in terms of a relation:

  permutation ($A'$, $A$)    ascending ($A'$)
- Functional specification in terms of pre- and postcondition:

  {A=X} sort {permutation (X, A)    ascending (A)}

  Note that the "logical variable" X is necessary to express the input-output relationship.
- Definitions:

  permutation: smallest relation satisfying for any $x$, $s$, $t_1$,$t_2$

  permutation ( , )

  permutation ( $x$ $\circ$ $s$, $t_1 \circ$ $x$ $\circ$ $t_2$) $\Leftrightarrow$ permutation ($s$, $t_1 \circ t_2$)

  ascending (s) $\Leftrightarrow$ (  $i$ | $1 \leq i <$ length [s] • $s[i] \leq s_{i+1}$)

## Program Development…

- For our purpose, all three functional specifications are equivalent; we will switch between them as convenient.
- We have allowed ourselves a simple formulation by abstracting from the way how the sequence is supposed to be input to and output from the computer.
- Given this specification, we can develop a solution by stepwise refinement:
- Insertion-Sort (A)

```
for j ← 2 to length[A] do
    "Insert A[j] in sorted sequence A[1..j-1]"
```

- "Insert A[j] in sorted sequence A[1..j-1]" specified by:

```
ascending (A[1..j-1])
permutation (A[1..j], A'[1..j])
ascending (A'[1..j])
```



## … Program Development

- "Insert A[j] in sorted sequence A[1..j-1]" is refined by:

```
key ← A[j]
i ← j-1
while i>0    A[i]>key do
    A[i+1] ← A[i]
    i ← i-1
A[i+1] ← key
```

- The final solution is obtained by composing the refined part(s).
- This algorithm can now be implemented in a variety of programming languages.

## Implementation in Pascal

- **type** T = **array** [1..N] **of** integer;

- **procedure** insertionsort (**var** A: T);

```
    var key, j, i: integer;
begin
    for j := 2 to N do
        begin
            key := A[j];
            i := j-1;
            while (i>0) and (A[i]>key) do
                begin
                    A[i+1] := A[i];
                    i := i-1
                end;
            A[i+1] := key
        end
end;
```

11

## Computing Resources

- These functional specification do not restrict the resources needed for by the algorithm, e.g. <u>time</u> and <u>memory</u>.
- Without further <u>non-functional requirements</u>, practically useless solutions would be allowed.
- What are the resources needed by Insertion-Sort?
  - memory: 1 element (key) + 2 integers (i,j)
  - time: ?
- In this course, we will focus on methods for determining the running time of algorithms under various circumstances.
- We shall assume a generic one-processor <u>random-access machine</u> (RAM) model of computation.

12

## Analyzing Insertion-Sort…

- Insertion-Sort (A)

| | | cost | times |
|---|---|---|---|
| 1 | for j ← 2 to length[A] do | $c_1$ | n |
| 2 | key ← A[j] | $c_2$ | n-1 |
| 3 | «insert A[j] in A[1..j-1]» | 0 | n-1 |
| 4 | i ← j-1 | $c_4$ | n-1 |
| 5 | while i>0   A[i]>key do | $c_5$ | $(\sum j \mid 2 \le j \le n \cdot t_j)$ |
| 6 | A[i+1] ← A[i] | $c_6$ | $(\sum j \mid 2 \le j \le n \cdot t_j-1)$ |
| 7 | i ← i-1 | $c_7$ | $(\sum j \mid 2 \le j \le n \cdot t_j-1)$ |
| 8 | A[i+1] ← key | $c_8$ | n-1 |

- The running time depends on the <u>size of the input</u>: let n = length[A]
- For each basic operation, we model its running time by a <u>cost</u> $c_i$.
- Let $t_j$ be the number of times the while condition in line 5 is tested for that value of j
- The total running time T(n) is:

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 (\sum j \mid 2 \le j \le n \cdot t_j) + c_6 (\sum j \mid 2 \le j \le n \cdot t_j -1) + c_7 (\sum j \mid 2 \le j \le n \cdot t_j -1) + c_8 (n-1)$$



## Best Case for Insertion-Sort

- Even for a fixed input size, the running time depends on which input of that size is given.

- <u>Best case</u> occurs if the array is already sorted:
  A[i] ≤ key in line 5, and $t_j$=1.

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

- T(n) is a <u>linear function</u> of n,
  T(n) = a n + b for some a, b

## Worst Case for Insertion-Sort

- Worst case occurs if the array is in reverse sorted order: we must compare A[j] with all A[j-1], …, A[1], so $t_j = j$

- Noting that

  $(\sum j|2 \le j \le n \cdot j) = n(n+1)/2 - 1$

  $(\sum j|2 \le j \le n \cdot j-1) = n(n-1)/2$

  we get for T(n) in that case:

  $$T(n) = (c_5/2 + c_6/2 + c_7/2)\, n^2 + (c_1 + c_2 + c_4 + c_5/2 + c_6/2 + c_7/2 + c_8)\, n - (c_2 + c_4 + c_5 + c_8)$$

- T(n) is a quadratic function of n,

  $T(n) = a\, n^2 + b\, n + c$ for some a, b, c

15

## Average Case for Insertion-Sort

- For the average case, we randomly choose n numbers.

- When comparing A[j] with A[1], …, A[j-1], in average half of the elements are less than A[j]. Hence $t_j = j / 2$.

- The analysis of T(n) in that case is similar to the worst case, except for the factor 2.

- In the average case, T(n) is also a quadratic function of n.

16

**Algorithms …**

- To summarize, an algorithm is a program for a (possibly abstract) machine, for which we can ensure the correctness in terms of the model of the problem domain.
- Besides correctness, we are interested in the worst-case and the average-case running time with respect to an abstract computer, the random access machine.
- We mostly consider only the worst-case running time:
  - It gives an upper bound on the running time.
  - The worst case occurs often, e.g. searching an element which is not present.
  - The average case is often roughly as bad as the worst case.
  - It is not clear what the average input is.
- For analyzing the running time and for implementing the algorithm, it has to be in a sufficiently refined form so that constructs can be faithfully mapped to the available machine (programming language).

17

**… Algorithms**

- When comparing the running time of algorithms, we often consider only the order of T(n), for example quadratic ($n^2$), n lg n, or linear (n), since for sufficiently large n this is decisive term.
- Other properties are also relevant. For example, Insertion-Sort behaves naturally in that it is faster if the array is already partially sorted, in particular if we have a sorted sequences with just a couple of unsorted elements at the end.
- Insertion-Sort has an average running time in the order of $n^2$, other sorting algorithms do better. However, other faster sorting algorithms do not behave as naturally as Insertion-Sort.
- In designing algorithms, we prefer for loops to while loops. They guarantee termination and make running time easier to analyze:

```
for i ← a to b do S(i) =
    skip                          if a>b
    S(a) ; for i ← a+1 to b do S(i)   otherwise
```

18

## Order of Growth

- The worst case running time for Insertion-Sort by is

  $T(n) = a\,n^2 + b\,n + c$

  for some constants a, b, c, which depend on the actual costs $c_i$. In doing so, we have abstracted from the actual costs.
- We further simplify the running time by leaving out b n + c, since for large values of n, is it insignificant compared to a $n^2$.
- We make a final simplification by leaving out the factor and only keeping $n^2$. We say that order of growth of T(n) is $n^2$, formally,

  $T(n) = \Theta(n^2)$

  for several reasons:
  - Many algorithms can be classified as computationally intensive simply by considering their order of growth.
  - The constant factors can be more precisely determined by experiments than by analysis.
  - The analysis is easier or only possible by considering only the order of growth.

19

## Growth Rate and Constant Factors

20

## Growth Rates of Common Functions

Suppose each operation takes 1 nanoseconds ($10^{-9}$ seconds)

| n | lg n | n | n lg n | $n^2$ | $2^n$ | n! |
|---|---|---|---|---|---|---|
| 10 | 0.003μs | 0.01μs | 0.033μs | 0.1μs | 1μs | 3.63ms |
| 20 | 0.004μs | 0.02μs | 0.086μs | 0.4μs | 1ms | 77.1years |
| 30 | 0.005μs | 0.02μs | 0.147μs | 0.9μs | 1sec | $>10^{15}$years |
| 100 | 0.007μs | 0.1μs | 0.644μs | 10μs | $>10^{13}$years | |
| 10,000 | 0.013μs | 10μs | 130μs | 100ms | | |
| 1,000,000 | 0.020μs | 1ms | 19.92μs | 16.7min | | |

- For $n < 10$, the difference is insignificant.
- $\Theta$ (n!) algorithms are useless well before $n = 20$.
- $\Theta$ ($2^n$) algorithms are practical for $n < 40$.
- $\Theta$ ($n^2$) and $\Theta$ (n lg n) are both useful, but $\Theta$ (n lg n) is significantly faster.



## The Divide-And-Conquer Approach

- Many algorithms follow the divide-and-conquer approach:
  - Divide the problem into a number of subproblems
  - Conquer the subproblems by solving them recursively. If the subproblems are small enough, solve them directly.
  - Combine the solutions to the subproblem into the solution for the original problem
- The algorithms are often more easily expressed as recursive algorithms.

## Merge-Sort…

- Merge-Sort is a divide-and-conquer algorithm:
  - Divide: Divide an n-element sequence into two subsequences of approximately n/2 elements.
  - Conquer: Sort the subsequences recursively.
  - Combine: Merge the two sorted subsequences to produce the sorted sequence.
- For allowing a recursive formulation, we pass the whole sequence and the bounds of the subsequence which has to be sorted as parameter.
- Merge-Sort (A, p, r)

```
if p < r then
    q <- ë(p + r) / 2û
    Merge-Sort (A, p, q)
    Merge-Sort (A, q+1, r)
    Merge (A, p, q, r)
```



## … Merge-Sort

- The auxiliary procedure Merge (A, p, q, r) merges the sorted subsequences A[p..q] and A[q+1..r]. It can be specified by:

  $p \leq q \quad q \leq r$
  ascending (A[p..q])    ascending (A[q+1..r])
  permutation (A'[p..r], A[p..r])
  ascending (A'[p..r])

- We assume that it can be implemented in $\Theta$ (n), for example by using an auxiliary sequence.
- The entire sequence A can be sorted by calling

  Merge-Sort (A, 1, length[A])

- What is the running time of Merge-Sort?

## Analyzing Divide-And-Conquer Algorithms

- Let $T(n)$ be the running time for a problem of size n.
- If the problem is small, say $n \le c$, we assume that a direct solution takes constant time:

  $T(n) = \Theta(1) \quad \text{if } n \le c$
- Otherwise, we divide the problem into a subproblems, each of which is 1/b the size of the original. Suppose it takes D(n) time to divide the problem and C(n) time to combine the solutions.

  $T(n) = a\,T(n/b) + D(n) + C(n) \quad \text{if } n > c$
- Such equations are called recurrences. They are common in analyzing the running time of algorithms. We will study solutions of recurrences later.

25

## Analysis of Merge-Sort

- Divide: Just computes the middle of the subsequence, thus takes constant time:

  $D(n) = \Theta(1)$
- Conquer: We solve 2 subproblems of size approximately n/2:

  $a = 2, \quad b = 2$
- Combine: Merge takes $\Theta(n)$:

  $C(n) = \Theta(n)$
- Noting that $\Theta(n) + \Theta(1)$ is still $\Theta(n)$, we get:

  $$T(n) = \begin{array}{ll} \Theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \Theta(n) & \text{if } n > 1 \end{array}$$
- Later we will see that:

  $T(n) = \Theta(n \lg n)$

26

### Remarks on Merge-Sort

- Merge-Sort has a running time of Θ (n lg n)
  Insertion-Sort has a running time of Θ ($n^2$)
- This implies that for sufficiently large n, Merge-Sort is superior to Insertion-Sort. For a certain Pascal implementation, Merge-Sort is 7 times faster than Insertion-Sort for n=256 and 11 times faster for n=512.
- Merge-Sort takes approximately the same amount of time whether the sequence is sorted, random, or inversely sorted.
- Procedure Merge requires n elements extra memory. However, since the sequences are accessed only sequentially, Merge-Sort is better suited for external sorting. Later we will see that 3 or 4 files are sufficient.
- Versions of Merge exists which do not require extra memory at the cost of additional moves. However, other internal sorting algorithms are superior even to the fastest version of Merge-Sort

27

### Complexity of Algorithms vs. Problems

- The running time of an algorithm is also referred to as its time complexity.
  The memory required by an algorithm is also referred to as its space complexity.
- Merge-Sort has a time complexity of Θ (n lg n)
  Insertion-Sort has a time complexity of Θ ($n^2$)

QUESTION: What is the time complexity of the fastest sorting algorithm?

ANSWER: We will see that it is Θ (n lg n), i.e. no algorithm can be faster than Θ (n lg n), although algorithms may still differ in their constant factors.

- We therefore can say that the problem of sorting has a time complexity of Θ (n lg n).

28