2 Introduction

- Class is about designing and analyzing algorithms
  - Algorithm: A well-defined procedure that transfers an input to an output.
    * Not a program (but often specified like it): An algorithm can often be implemented in several ways.
  - Design: We will study methods/ideas/tricks for developing (fast!) algorithms.
  - Analysis: Abstract/mathematical comparison of algorithms (without actually implementing them).

- Math is needed in three ways:
  - Formal specification of problem
  - Analysis of correctness
  - Analysis of efficiency (time, memory use,...)

- Hopefully the class will show that algorithms matter!

3 Algorithm example: Insertion-sort

3.1 Specification

- Input: $n$ integers in array $A[1..n]$
- Output: $A$ sorted in increasing order

3.2 Insertion-sort algorithm

```
FOR $j = 2$ to $n$ DO
  $key = A[j]$
  $i = j - 1$
  WHILE $i > 0$ and $A[i] > key$ DO
    $A[i + 1] = A[i]$
    $i = i - 1$
  OD
  $A[i + 1] = key$
OD
```

- NOTE: Algorithm shows example of the (Pascal like) pseudo-code we will sometimes used to describe algorithms.

Example:
5 2 4 6 1 3  j=2  i=1  key=2
5 5 4 6 1 3  i=0
2 5 4 6 1 3

2 5 4 6 1 3  j=3  i=2  key=4
2 5 5 6 1 3  i=1
2 4 5 6 1 3

2 4 5 6 1 3  j=4  i=3  key=6
2 4 5 6 1 3

2 4 5 6 1 3  j=5  i=4  key=1
2 4 5 6 6 3  i=3
2 4 5 5 6 3  i=2
2 4 4 5 6 3  i=1
2 2 4 5 6 3  i=0
1 2 4 5 6 3

1 2 4 5 6 3  j=6  i=5  key=3
1 2 4 5 6 6  i=4
1 2 4 5 5 6  i=3
1 2 4 4 5 6  i=2
1 2 3 4 5 6

3.3 Correctness

- Induction:
  - The Invariant “A[1..j-1] is sorted” holds at the beginning of each iteration of FOR-loop.
  - When j=n+1 we have correct output.

3.4 Analysis

- We want to predict the resource use of the algorithm.
- We can be interested in different resources
  - but normally running time.
- To analyze running time we need mathematical model of a computer:
Random-access machine (RAM) model:
- Memory consists of infinite array
- Instructions executed sequentially one at a time
- All instructions take unit time:
  * Load/Store
  * Arithmetics (e.g. +, -, *, /)
  * Logic (e.g. >)

- Running time of an algorithm is the number of RAM instructions it executes.
- RAM model not completely realistic, e.g.
  - memory not infinite (even though we often imagine it is when we program)
  - not all memory accesses take same time (cache, main memory, disk)
  - not all arithmetic operations take same time (e.g. multiplications expensive)
  - instruction pipelining
  - other processes
- But RAM model often enough to give relatively realistic results (if we don’t cheat too much).

- Running time of insertion-sort depends on many things
  - How sorted the input is
  - How big the input it
  - ...
- Normally we are interested in running time as a function of input size
  - in insertion-sort: \( n \).
- We don’t really want to count every RAM instruction
  - Let us analyze insertion-sort by assuming that line \( i \) in the program use \( c_i \) RAM instructions.
  - How many times are each line of the program executed?
    * Let \( t_j \) be the number of times line 4 (the WHILE statement) is executed in the \( j \)'th iteration.

```
FOR \( j = 2 \) to \( n \) DO
    \( key = A[j] \)
    \( i = j - 1 \)
    WHILE \( i > 0 \) and \( A[i] > key \) DO
        \( A[i + 1] = A[i] \)
        \( i = i - 1 \)
    OD
    \( A[i + 1] = key \)
OD
```

<table>
<thead>
<tr>
<th>cost</th>
<th>times</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 )</td>
<td>( n )</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>( c_4 )</td>
<td>( \sum_{j=2}^{n} t_j )</td>
</tr>
<tr>
<td>( c_5 )</td>
<td>( \sum_{j=2}^{n} (t_j - 1) )</td>
</tr>
<tr>
<td>( c_6 )</td>
<td>( \sum_{j=2}^{n} (t_j - 1) )</td>
</tr>
<tr>
<td>( c_7 )</td>
<td>( n - 1 )</td>
</tr>
</tbody>
</table>
• Running time: (depends on $t_j$)
  \[
  T(n) = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n - 1)
  \]

  - Best case: $t_j = 1$ (already sorted)
    \[
    T(n) = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_7(n - 1)
    = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)
    = k_1 n - k_2
    \]

  **Linear function of $n$**

  - Worst case: $t_j = j$ (sorted in decreasing order)
    \[
    T(n) = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j - 1) + c_6 \sum_{j=2}^{n} (j - 1) + c_7(n - 1)
    = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 (\frac{n(n+1)}{2}) - 1 + c_5 (\frac{(n-1)n}{2}) + c_6 (\frac{(n-1)n}{2}) + c_7(n - 1)
    = (c_1/2 + c_5/2 + c_6/2)n^2 + (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7)n - (c_2 + c_3 + c_4 + c_7)
    = k_3 n^2 + k_4 n - k_5
    \]

  **Quadratic function of $n$**

  Note: We used $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$ (Next week!)

  - “Average case”: Be careful! (average over what?)

  We assume $n$ numbers chosen randomly $\Rightarrow t_j = j/2$
  \[
  T(n) = k_6 n^2 + k_7 n + k_8
  \]

  Still **Quadratic function of $n$**

  • Note:

  - We will normally be interested in worst-case running time.
    * Upper bound on running time for any input.
    * For some algorithms, worst-case occur fairly often.
    * Average case often as bad as worst case (but not always!).

  - We will only consider order of growth of running time:
    * We already ignored cost of each statement and used the constants $c_i$.
    * We even ignored $c_i$ and used $k_i$.
    * We just said that best case was linear in $n$ and worst/average case quadratic in $n$.

  $\Rightarrow O$-notation (best case $O(n)$, worst/average case $O(n^2)$) (next lecture!)
4 Designing Good Algorithms: Divide and Conquer/Mergesort

4.1 Divide-and-conquer

- Can we design better than $O(n^2)$ sorting algorithm?
- We will do so using one of the most powerful algorithm design techniques.

<table>
<thead>
<tr>
<th>Divide and Conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>To Solve P:</td>
</tr>
<tr>
<td>1. Divide P into smaller problems $P_1, P_2, P_3, \ldots, P_k$.</td>
</tr>
<tr>
<td>2. Conquer by solving the (smaller) subproblems recursively.</td>
</tr>
<tr>
<td>3. Combine solutions to $P_1, P_2, \ldots, P_k$ into solution for P.</td>
</tr>
</tbody>
</table>

4.2 Merge-Sort

- Using divide-and-conquer, we can obtain a merge-sort algorithm.
  - Divide: Divide $n$ elements into two subsequences of $n/2$ elements each.
  - Conquer: Sort the two subsequences recursively.
  - Combine: Merge the two sorted subsequences.
- Assume we have procedure $\text{Merge}(A, p, q, r)$ which merges sorted $A[p..q]$ with sorted $A[q+1..r]$ in $O(r - p)$ time.
- We can sort $A[p..r]$ as follows (initially $p=1$ and $r=n$):

<table>
<thead>
<tr>
<th>Merge Sort(A,p,r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $p &lt; r$ then</td>
</tr>
<tr>
<td>$q = \lfloor (p + r) / 2 \rfloor$</td>
</tr>
<tr>
<td>$\text{MergeSort}(A,p,q)$</td>
</tr>
<tr>
<td>$\text{MergeSort}(A,q+1,r)$</td>
</tr>
<tr>
<td>$\text{Merge}(A,p,q,r)$</td>
</tr>
</tbody>
</table>
4.3 Correctness
- Induction on $n$
  - Easy assuming Merge() is correct!

4.4 Analysis
- To simplify things, let us assume that $n$ is a power of 2, i.e. $n = 2^k$ for some $k$.
- Running time of the procedure can be analyzed using a recurrence equation/relation.

$$T(n) \leq c_1 + T(n/2) + T(n/2) + c_2n$$
$$\leq 2T(n/2) + c_3n$$

$\downarrow$

$T(n) = O(n \log_2 n)$ as we will see later.

- We can also get $O(n \log_2 n)$ bound by noticing that the recursion tree has depth $\log_2 n$ and that $O(n)$ time is spent on each level.
• Note:
  - We really have $T(n) = c_1$ if $n = 1$
  - If $n \neq 2^k$ things can be complicated (We will often assume $n = 2^k$ to avoid complicated cases).

4.5 Log’s
• Base 2 logarithm comes up all the time (from now on we will always mean $\log_2 n$ when we write $\log n$).
  - Number of times we can divide $n$ by 2 to get to 1 or less.
  - Number of bits in binary representation of $n$.
  - Inverse function of $2^n = 2 \cdot 2 \cdot 2 \cdots n$ (times).
  - Way of doing multiplication by addition: $\log(ab) = \log(a) + \log(b)$

• Note:
  - $\log_a n = \frac{\log_b n}{\log_b a}$
  - $\log n << \sqrt{n} << n$