CS280, Spring 2001 **Handout 7**
February 5, 2001

1. Reading: K. Rosen *Discrete Mathematics and Its Applications*, 2.1, 2.2
2. The main message of this lecture:

> **Though real computational devices all have finite storage facilities, the adequate theoretical framework for computing has been provided by the computational models with no a priori limits on the size of programs, runtime, or memory size. Time complexity is an order-of-magnitude count of the number of important operations that the algorithm might carry out as a function of the size of the input.**

Algorithm is usually understood as a finite set of instructions for performing a computation. This is not a formal definition yet (which will be offered at CS381/481), but with the following comments is good enough for all the practical purposes of CS280.

There are several essential defaults there:

1. An algorithm is given as a set of instructions (*program*) of finite size. In our course we will be using plain English, or *pseudocode* to describe algorithms. In either case a program will be of finite length though there is no a priori bound on its size.
2. There is a computing agent (machine operating system or human), which can carry out the computation according to the program.
3. There are facilities for making, storing, and retrieving steps in a computation (*memory*).
4. The computation is carried out in a discrete stepwise fashion, without use of continuous methods or analogue devices.
5. The computation is carried forward deterministically, without resort to random methods or devices, e.g. dice.
6. There is no fixed finite bound on the size of inputs.
7. There is no fixed finite bound on the size of memory storage space available.
8. There is no fixed finite bound on the number of steps in a computation. We require only that a computation terminate after *some* finite number of steps; we do not insist on an a priori ability to estimate how long the computation will take.

In addition, there are the following *desired* properties of algorithms:

9. *Generality.* An algorithm should be applicable for all problems of the desired form.
10. *Correctness.* The algorithm should produce the correct output for each admissible input value.

An algorithm efficiency is given by a rough count of the number of important operations the algorithm performs (**time complexity**) or the computer memory required (**space complexity**) as a function of the size of the input in the **worst case** (unless the otherwise is mentioned). There are some natural guidelines for choosing which operations are essential and how to count the memory blocks, though both of the choices really depend on the details of the computational model involved. In our examples both of the measures will be specified exactly.

Example 7.1. Finding the maximum: *find the largest in a list of integers* $a_1, a_2, \ldots, a_n$.

Algorithm 1.

```
$max := a_1$
for $i := 2$ to $n$
    if $max < a_i$ then $max := a_i$
```

Complexity measure: the number of comparisons as a function of n. There are two sorts of comparisons consecutively executed in each loop step starting with $i = 2$:

a) comparing i with n to determine that the end of the list has not been reached yet

b) comparing the temporary *max* with a_i (only if (a) succeeds!).

It is clear that both of the comparisons will be performed $n-1$ times for $i = 2, 3, 4, \ldots, n$, and the comparison (a) will be performed when $i = n+1$. Altogether these give us $2(n-1)+1 = 2n-1$ comparisons, thus the time complexity of this algorithm is $O(n)$.

Example 7.2. Searching problem: *locate an element* x *in a list of distinct elements* $a_1, a_2, \ldots, a_n$ (i.g. locate a file given its filename).

Algorithm 2. Linear search algorithm

```
$i = 1$
**while** ($i \leq n$ end $x \neq a_i$)
    $i := i + 1$
**if** $i \leq n$ **then** *location* $:= i$ **else** *location* $:= 0$
```

Complexity measure = the number of comparisons as a function of n. For each loop step two comparisons are performed

a) $i = n(?)$, starting with $i = 1$, to determine whether the end of the list has been reached

b) $x = a_i(?)$, only if (a) fails.

If $x = a_i$ then the algorithm exits the loop on the condition (b) after performing $2i$ comparisons, and then makes one more comparison $i \leq n$ outside the loop. The total number of comparisons then reaches $2i + 1$. In the worst case (when x is not in the list) the algorithm performs n complete cycles of looping for $i = 1, 2, \ldots, n$ (making two comparisons in each cycle), one comparison of type (a) when $i = n+1$ to exit the loop, and one more comparison $i \leq n$ outside the loop, which brings the total number of them to $2n + 2$. Therefore, the time complexity of the linear search algorithm is $O(n)$, no matter the element x is in the list or not necessarily.

The **average time** complexity estimates the average number of operations needed when all the inputs are equally likely. Here we assume that with probability $1/n$ x occupies position $i = 1, 2, 3, \ldots, n$ thus the algorithm makes $3, 5, 6, \ldots, 2n+1$ comparisons. Then the average number of comparisons is $\frac{3}{n} + \frac{5}{n} + \frac{7}{n} + \ldots + \frac{2n+1}{n} = \frac{3+5+7+\ldots+(2n+1)}{n} = \frac{(2+1)+(4+1)+(6+1)+\ldots+(2n+1)}{n} = \frac{(2+4+6+\ldots+2n)+n}{n} = \frac{2(1+2+3+\ldots+n)+n}{n} = \frac{2[n(n+1)/2]+n}{n} = \frac{n(n+1)+n}{n} = (n+1)+1 = n+2 = O(n)$.

Algorithm 3. Binary search algorithm, used when the original array $a_1, a_2, a_3, \ldots, a_n$ is sorted (say, from smallest to largest): A) compare x to the middle term of the list, B) pick the half of the list containing x, go to A). For the pseudocode see p.104 of the textbook.

To evaluate the complexity assume that $n = 2^k$ for some integer k, i.e. $k = \log n$. Each full step of the loop takes two comparisons $i < j$ and $x > a_m$. After the first cycle of the loop the search is restricted to a half of the list i.e. to a sublist of length $n/2 = 2^k/2 = 2^{k-1}$, after the second - to a half of that, i.e. to 2^{k-2}, etc. The number of loop steps is then equal to $k = \log n$ + one more comparison to exit the loop + one more comparison outside the loop. The total # of comparisons = time complexity of the binary search = $2 \log n + 2 = O(\log n)$.

Example 7.3. The addition of two binary integers of length n takes $O(n)$ additions of bits.

Homework assignments. (due Friday 02/09).

Section 2.1: 24

Section 2.2: 2, 6b) 18