APPENDIX

B

The Language of Efficiency

B.1 Introduction and Motivation

B.2 What Do We Use for a Yardstick?

```
void SelectionSort( int[ ] A) {                          // sorts an array of integers, A,
                                                          // into increasing order
    int   maxPosition, temp, i, j;

    for (i = A.length – 1; i > 0; i–– ) {               // for each i in 1:A.length – 1
                                                          // in decreasing order of i
        maxPosition = i;

        for( j = 0;  j  <  i;  j++) {

            if (A[j] > A[maxPosition]) {         // find the position, maxPosition, of
                maxPosition = j;                 // the largest integer in A[0:i]
            }                                    // then exchange
                                                 // A[i] and A[maxPosition]
        }

        // exchange A[i] and A[maxPosition]
        temp = A[i]; A[i] = A[maxPosition]; A[maxPosition] = temp;

    }
}
```

Type of Computer	Time
Home computer	51.915
Desktop computer	11.508
Minicomputer	2.382
Mainframe computer	0.431
Supercomputer	0.087

Table B.2 Running Times in Seconds to Sort an Array of 2000 Integers

Array Size n	Home Computer	Desktop Computer
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	172.9
2000	3114.9	690.5

Table B.3 SelectionSort Running Times in Milliseconds on Two Types of Computers

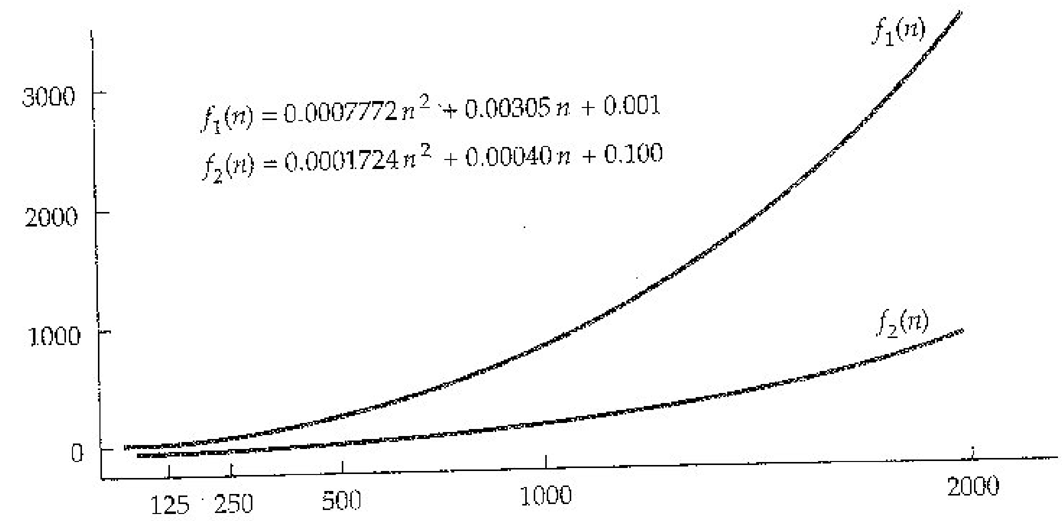

Figure B.4 Two Curves Fitting the Data in Table B.3

Adjective Name	O-Notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
$n \log n$	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$
Exponential	$O(10^n)$

Table B.7 Some Common Complexity Clas

Algorithm A stops in $f(n)$ microseconds					
$f(n)$	$n = 2$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1	1	1	1.00×10^0	1.00×10^0
$\log_2 n$	1	4	8	1.00×10^1	2.00×10^1
n	2	1.6×10^1	2.56×10^2	1.02×10^3	$1.05 \times 10^{[illegible]}$
$n \log_2 n$	2	6.4×10^1	2.05×10^3	1.02×10^4	$2.[illegible] \times 10^7$
n^2	4	2.56×10^2	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	8	4.10×10^3	1.68×10^7	1.07×10^9	$1.15 \times 10^{1[illegible]}$
2^n	4	6.55×10^4	1.16×10^{77}	1.80×10^{308}	$6.74 \times 10^{[illegible]}$

Table B.8 Running Times for Different Complexity Classes

$f(n)$	$n = 2$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1 μsec*	1 μsec	1 μsec	1 μsec	1 μsec
$\log_2 n$	1 μsec	4 μsecs	8 μsecs	10 μsecs	20 μsecs
n	2 μsecs	16 μsecs	256 μsecs	1.02 msecs	1.05 secs
$n \log_2 n$	2 μsecs	64 μsecs	2.05 msecs	10.2 msecs	21 secs
n^2	4 μsecs	25.6 μsecs	65.5 msecs	1.05 secs	1.8 wks
n^3	8 μsecs	4.1 msecs	16.8 secs	17.9 mins	36,559 yrs
2^n	4 μsecs	65.5 msecs	3.7×10^{63} yrs	5.7×10^{294} yrs	2.1×10^{315639} yrs

1 μsec = one microsecond = one millionth of a second; 1 msec = one millisecond = one thousandth of a second; sec = one second; min = one minute; wk = one week; and yr = one year.

able B.9 Running Times for Algorithm A in Different Time Units

Number of steps is	T = 1 min	T = 1 hr	T = 1 day	T = 1 wk	T = 1 yr	Ratio
n	6×10^7	3.6×10^9	8.64×10^{10}	6.05×10^{11}	3.15×10^{13}	$05 \cdot 10^6$
$n \log_2 n$	2.8×10^6	1.3×10^8	2.75×10^9	1.77×10^{10}	7.97×10^{11}	$2 \cdot 10^5$
n^2	7.75×10^3	6.0×10^4	2.94×10^5	7.78×10^5	5.62×10^6	10^3
n^3	3.91×10^2	1.53×10^3	4.42×10^3	8.46×10^3	3.16×10^4	10^2
2^n	25	31	36	39	44	1.738
10^n	7	9	10	11	13	1.735

Table B.10 Size of Largest Problem That Algorithm A Can Solve if Solution Is Computed in Time $\leq T$ at 1 Microsecond per Step

1 yr = 34 mil sec = $34 \cdot 10^6$ sec = $\cdot 10^{12}$ = 3.4×10^{13}

~~1 yr ≈ 500,000~~ sec.

1 yr ≈ 500,000 min.

Number of steps is	T = 1 min	60x T = 1 hr	1444x T = 1 day	10,080 T = 1 wk	524,160x T = 1 yr
n	6×10^7	3.6×10^9	8.64×10^{10}	6.05×10^{11}	3.15×10^{13}
$n \log_2 n$	2.8×10^6	1.3×10^8	2.75×10^9	1.77×10^{10}	7.97×10^{11}
n^2	7.75×10^3	6.0×10^4	2.94×10^5	7.78×10^5	5.62×10^6
n^3	3.91×10^2	1.53×10^3	4.42×10^3	8.46×10^3	3.16×10^4
2^n	25	31	36	39	44
10^n	7	9	10	11	13

Table B.10 Size of Largest Problem That Algorithm *A* Can Solve if Solution Is Computed in Time ≤ T at 1 Microsecond per Step

Points

- Fast-growing running time = slow programs.
- The faster computer the most wasteful for slow programs is
- These are the large inputs that, generally cause long computation. So, how the program behaves for a large input is, usually, the defining factor of its usefulness.

B.4 O-Notation—Definition and Manipulation

Definition of O-Notation: We say that $f(n)$ is $O(g(n))$ if there exist two positive constants K and n_0 such that $|f(n)| \leq K|g(n)|$ for all $n \geq n_0$.

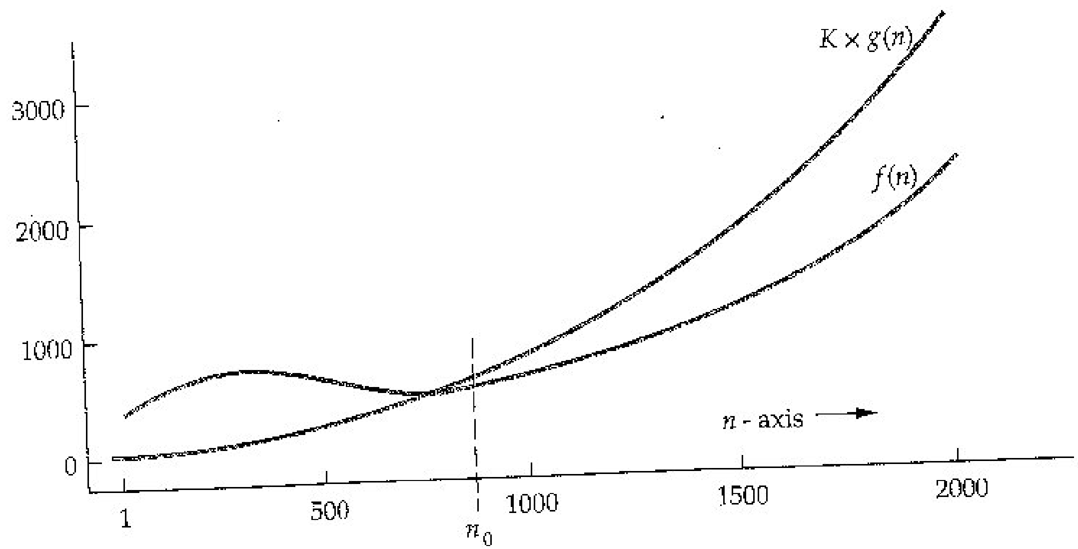

Figure B.12 Graphical Meaning of O-Notation

B.5 What O-Notation Doesn't Tell You

Measuring the running time of a program

T(n) – the running time of a program on "worst" input of size n
$T_{avg}(n)$ – the average time of a program on inputs of size n
Time is mesured in some **abstract units**, independent of particular computer, compiler, and similar factors.

slide.spxl

Rule of composition and product

Suppose that $T_1(n)$ and $T_2(n)$ are the running times of two program fragments P_1 and P_2, and that $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$. Then $T_1(n) + T_2(n)$, the running time of P_1 followed by P_2, is $O(\max(f(n), g(n)))$.

To see why, observe that for some constants c_1, c_2, n_1, and n_2, if $n \geqslant n_1$ then $T_1(n) \leqslant c_1 * f(n)$, and if $n \geqslant n_2$ then $T_2(n) < c_2 * g(n)$.

Let $n_0 = \max(n_1, n_2)$. If $n \geqslant n_0$, then
$T_1(n) + T_2(n) \leqslant c_1 * f(n) + c_2 * g(n)$.
From this we conclude that if $n \geqslant n_0$, then
$T_1(n) + T_2(n) \leqslant (c_1 + c_2) * \max(f(n), g(n))$.
Thefore, the combined running time $T_1(n) + T_2(n)$ is $O(\max(f(n), g(n)))$.

The rule for products is the following. If $T_1(n)$ and $T_2(n)$ are $O(f(n))$ and $O(g(n))$, respectively, then $T_1(n) * T_2(n)$ is $O(f(n) * g(n))$. One can prove this fact using the same ideas as in the proof of the sum rule. It follows from the product rule that $O(c * f(n))$ means the same thing as $O(f(n))$ if c is a positive constant. For example, $O(n^2/2)$ is the same as$O(n^2)$.

```
function fact( n: integer ): integer;
      { fact(n) computes n! }
      begin
         if n <= 1 then
              fact := 1
         else
              fact := n * fact(n-1)
      end; { fact }
```

Input size measure: n.
Running time: T(n).

$$T(n) = \begin{cases} c + T(n-1) & \text{if } n > 1 \\ d & \text{if } n \leq 1 \end{cases}$$

T(n) is a linear function for $n \geq 1$, because
$T(n) - T(n-1) = \text{constant}$. Therefore $T \in O(n) \cap \Omega(n)$

(We may even solve the above equation:
T(n) is linear, so it must be of the form
An + B. Easy calculus gives us $T(n) = c*n + (d - c)$).

Analysis of recursive programs - efficiency.

$n = 2^k$

```
function mergesort ( L : LIST; n : integer ) : LIST;
    { L is a list of length n . A sorted version of L
      is returned. We assume n is a power of 2. }
    var
        L1, L2: LIST
    begin
        if n = 1 then
            return (L);
        else begin
            break L into two halves, L1 and L2, each of length n/2;
            return (merge(mergesort(L1,n/2), mergesort(L2, n/2)));
        end
    end; { mergesort }
```

INPUT SIZE MEASURE: n.

Estimate the complexity of mergesort.
Assume that initiation, test, return, breaking and merge take together at most c * n time.
We will guess an asymptotic upper bound of the worst case running time T(n) of mergesort and prove it by induction.

Claim. For some constant d and each $n = 2^k$, $k \geq 1$ (which implies $n_0 = 2$), $T(n) \leq d * n * \log n$, that is to say, $T \in O(n * \log n)$.

It is sufficient to prove that for all $k \in \omega$, there exists c with:

(*) $T(2^k) \leq d * k * 2^k$.

1° For k = 1, $T(2^k) \leq 4c$, thus (*) holds if $d > 2c$.
2° Assume that (*) holds for all $k < m$ (the induction hypothesis).
$T(2^m) \leq 2(T(2^{m-1})) + c * 2^m <$
(by induction hypothesis)
$2 * d * (m-1) * 2^{m-1} + c * 2^m = c((m-1) * 2^m + 2^m) =$
$= c * m * 2^m$, which means that (*) holds also for k = m.

General rules

1. The running time of each assignment, read, and write statement can usually be taken to be O(1). There are a few exceptions, such as in PL/I, where assignments can involve arbitrarily large arrays, and in any language that allows function calls in assignement statements.

2. The running time of a sequence of statements is determined by the sum rule. That is, the running time of the sequence is, to within a constant factor, the largest running time of any statement in the sequence.

3. The running time of an if-statement may be estimated as the running time of the conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally O(1).
The time for an if-then-else construct may be estimated as the time to evalutae the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.

4. The time to execute a loop is the sum, over all times around the loop, of the time to execute the body and the time to evaluate the condition for termination (usually the latter is O(1)). Often this time is, neglecting constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure. The number of iterations around a loop is usually clear, but there are times when the number of iterations cannot be computed precisely.

slide . sol 2