String Matching Algorithms

String Searching

The context of the problem is to find out whether one string (called "pattern") is contained in another string. This problem correspond to a part of more general one, called "pattern recognition". The strings considered are sequences of symbols, and symbols are defined by an alphabet. The size and other features of the alphabet are important factors in the design of string-processing algorithms.

Brute-force algorithm: The obvious method for pattern matching is just to check, for each possible position in the text at which the pattern could match, whether it does in fact match. The following program searches in this way for the first occurrence of a pattern string p in a text string ***a***:

```
int brutesearch(char *p, char *a)
  {
    int i, j, M = strlen(p), N = strlen(a);
    for (i = 0, j = 0; j < M && i < N; i++, j++)
      if (a[i] != p[j]) { i -= j-1; j = -1; }
    if (j == M) return i-M; else return i;
  }
```

or

function *brutesearch*: **integer**;
  var i, j: **integer**;
  begin i:= 1; j:= 1;
  repeat **if** a[i] = p[j]
    then begin i:= i+1; j:= j+1 **end**
    else begin i:= i - j +2; j:= 1 **end**;
  until (j > M) or (i > N);
  if j > M **then** brutesearch:= i - M **else** brutesearch:= i
  end;

```
1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1
1 0 1 0 0 1 1 1
  1 0 1 0 0 1 1 1
    1 0 1 0 0 1 1 1
      1 0 1 0 0 1 1 1
        1 0 1 0 0 1 1 1
          1 0 1 0 0 1 1 1
            1 0 1 0 0 1 1 1
              1 0 1 0 0 1 1 1
                1 0 1 0 0 1 1 1
                  1 0 1 0 0 1 1 1
                    1 0 1 0 0 1 1 1
                      1 0 1 0 0 1 1 1
                        1 0 1 0 0 1 1 1
                          1 0 1 0 0 1 1 1
                            1 0 1 0 0 1 1 1
                              1 0 1 0 0 1 1 1
1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1
```

Property: *brute-force string searching can require about* M N *character comparisons.*

Knuth-Morris-Pratt Algorithm

The basic idea behind the algorithm is this:

when a mismatch is detected, our "false start" consists of characters that we know in advance (since they're in the pattern). Somehow we should be able to take advantage of this information instead of backing up the ***i*** pointer over all those known characters.

A simple example of the pattern of 10000000.

The idea is worth thinking about and the Knuth-Morris-Pratt algorithm is a generalization of it. Searching for pattern 10100111 in the string 1010100111 we detect mismatch at 5th position, but we back up at 3rd position. Surprisingly, it is always possible to arrange things so that the ***i*** pointer is never decremented.

It depends entirely on the pattern:

j	next[j]	
1	0	10100111 10100111
2	0	10100111 10100111
3	1	10100111 10100111
4	2	10100111 10100111
5	0	10100111 10100111
6	1	10100111 10100111
7	1	10100111 10100111

Restart positions for KMP search

The array ***next[M]*** will be used to determine how far to go back up when a mismatch is detected.

By the definition of the ***next*** array, the first ***next[j]*** characters at that position match the first ***next[j]*** characters of the pattern.

Very precise definition of ***next[j]***:

- **next**[1]:=0;
- if j > 1, then **next**[j] is equal to the maximal number of k < j, such that the first (k-1)-characters of pattern coincide with the least (k-1) characters of string of j-1 character.

We do not need to backup the pointer ***i***, we can simply leave the pointer unchanged and set the pointer j to next[j].

The function, which calculates the index for back up:

```
function kmpsearch: integer;
    var i, j: integer;
    begin i:= 1; j:= 1; initnext;
    repeat if (j = 0) or (a[i] = p[j])
        then begin  i:= i+1; j:= j+1 end
        else begin j:= next[j] end;
    until (j > M) or (i > N);
    if j > M then kmpsearch:= i - M else kmpsearch:= i;
    end;
```

The program is quite transparent:

```
procedure initnext;
    var i,j: integer;
    begin i:= 1; j:= 0; next[1]:= 0;
    repeat if (j = 0) or (p[i] = p[j])
        then begin i:= i+1; j:= j+1; next[i]:= j; end
        else begin j:= next[j] end;
    until i > M;
    end;
```

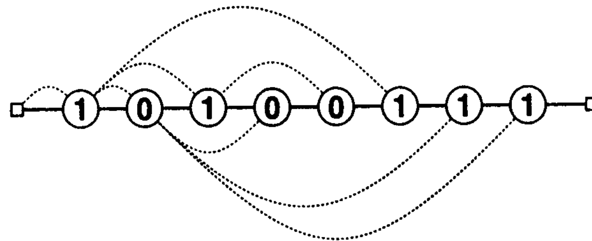
Sometime the pattern, often used, can be ”wired in” to the program (as for the pattern above):

```
    i:=0;
0: i:= i+1;
1: if a[i] <> '1' then goto 0; i:+ i+1;
2: if a[i] <> '0' then goto 1; i:+ i+1;
3: if a[i] <> '1' then goto 1; i:+ i+1;
4: if a[i] <> '0' then goto 2; i:+ i+1;
5: if a[i] <> '0' then goto 3; i:+ i+1;
6: if a[i] <> '1' then goto 1; i:+ i+1;
7: if a[i] <> '1' then goto 2; i:+ i+1;
8: if a[i] <> '1' then goto 2; i:+ i+1;
    return:= i - 8;
```

This program is a simple example of ”string matching compiler”: given a pattern, we can produce a very efficient program to scan that pattern in an arbitrarily long text string.

The program above uses just a ***few very basic operations*** to solve the string searching problem. This means that it can easily be described in terms of a very simple machine model called ***a finite-state machine***.

Figure shows the finite- state machine for the program above:



The machine consists of states (indicated by circled numbers) and transitions (indicated by lines). Each state has two transitions leaving it:

- a match transition (solid line, going right);
- a non-niatch transition (dotted line, going left).

The states are where the machine executes instructions; the transitions are the ***goto*** instructions.

When in the state labeled "***x,***" the machine can perform only one instruction:
- "if the current character is ***x*** then ***scan past*** it and take the match transition, otherwise take the non-match transition."

To "***scan past***" a character means to take the next character in the string as the ”current character”; the machine scans past characters as it matches them.

There are two exceptions to this:

- the first state always takes a match transition and scans to the next character (essentially this corresponds to scanning for the first occurrence of the first character in the pattern),
- and the last state is a "halt" state indicating that a match has been found.

Similar (but more powerful) machines can be used to help develop much more powerful pattern-matching algorithms.

Property: KMP string matching never uses more than $M + N$ character comparisons.

```
1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1
1 0 1 0 0 1 1 1
      1 0 1 0 0 1 1 1
        1 0 1 0 0 1 1 1
          1 0 1 0 0 1 1 1
                    1 0 1 0 0 1 1 1
1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1
```

KMP string matching in binary text.

This figure shows that this method certainly uses far fewer comparisons than the brute-force method for our binary example.

However, the Knuth-Morris-Pratt algorithm is not likely to be significantly faster than the brute-force method in many actual applications, because few applications involve searching for highly self-repetitive patterns in highly self-repetitive text.

However, the method does have a major practical advantage: it proceeds sequentially through the input and never "backs up" in the input. This makes it convenient for use on a large file being read in from some ***external*** device.

(Algorithms requiring backup need some complicated buffering in this situation.)

Finite automata

Many string-matching algorithms build ***a finite automaton*** that scans the text string $\boldsymbol{T}$ for all occurrences of the pattern $\boldsymbol{P}$. These string-matching automata are very efficient:

- they examine each text character exactly once, taking constant time per text character.

The time used (after the automaton is built) is therefore ***È(n)***. The time to build the automaton, however, can be large if ***alphabet*** is large.

A ***finite automaton*** M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.

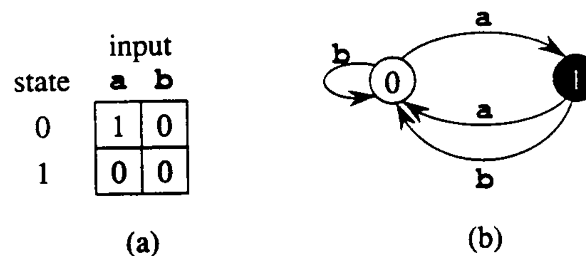
The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves

("makes a transition") from state ***q*** to state ***ä (q, a)***. Whenever its current state ***q*** is a member of ***A***, the machine ***M*** is said to have accepted the string read so far. An input that is not accepted is said to be rejected. Figure below illustrates these definitions with a simple two-state automaton.

A finite automaton ***M*** induces a function ***ö***, called the final-state function, from ***Ó****** to ***Q*** such that ***ö(ù)*** is the state ***M*** ends up in after scanning the string ***ù***. Thus, ***M*** accepts a string ***ù*** if and only if ***ö(ù)*** is in ***A***. The function ***ö*** is defined by the recursive relation:

$$
\begin{aligned}
\phi(\varepsilon) &= q_0 \,, \\
\phi(wa) &= \delta(\phi(w), a) \qquad \text{for } w \in \Sigma^*, a \in \Sigma
\end{aligned}
$$

Example:



A simple two-state finite automaton with state set ***Q = {0, 1}***, start state ***q*** $_0$ ***=0***, and input alphabet ***Ó = {a, b}***. (a) A tabular representation of the transition function ä. (b) An equivalent state-transition diagram. State 1 is the only accepting state (shown blackened). Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates ***ä(1,b) = 0***. This automaton accepts those strings that end in an odd number of ***a***'s. More precisely, a string ***x*** is accepted if and only if ***x = yz***, where ***y = å*** or ***y*** ends with a ***b***, and ***z = a*** ^{k} , where ***k*** is odd. For example, the sequence of states this automaton enters for input ***abaaa*** (including the start state) is ***(0, 1, 0, 1, 0, 1)***, and so it accepts this input. For input ***abbaa***, the sequence of states is ***(0, 1, 0, 0, 1, 0)***, and so it rejects this input.

String-matching automata

There is a string-matching automaton for every pattern ***P***; this automaton must be constructed from the pattern in a preprocessing step before it can be used to search the text string. Figure in example below illustrates this construction for the pattern ***P = ababaca***. From now on, we shall assume that ***P*** is a given fixed pattern string; for brevity, we shall not indicate the dependence upon ***P*** in our notation.

In order to specify the string-matching automaton corresponding to a given pattern ***P[1…m]***, we first define an auxiliary function ***ó***, called the suffix function corresponding to ***P***.

The function ***s*** is a mapping from ***Ó**** to ***[0, 1, . . . , m]*** such that ***ó(x)*** is the length of the longest prefix of ***P*** that is a suffix of ***x***:

$$ó(x) = max \{k: P_k » x\}$$

The suffix function ***s*** is well defined since the empty string P_0 = ***å*** is a suffix of every string.

For a pattern ***P*** of length ***m***, we have ***ó(x) = m*** if and only if ***P » x***. It follows from the definition of the suffix function that if ***x » y***, then ***ó (x) < ó (y)***.

We define the string-matching automaton corresponding to a given pattern P[1….m] as follows:

- The state set ***Q*** is ***{0, 1, … m}***. The start state q_0 is state ***0***, and state ***m*** is the only accepting state.
- The transition function ***ä*** is defined by the following equation, for any state ***q*** and character ***a***:

$$ä(q, a) = ó(P_q a)$$

Here is an intuitive rationale for defining $ä(q, a) = ó(P_q a)$ The machine maintains as an invariant of its operation that

$$f(T_i) = s(T_i)$$

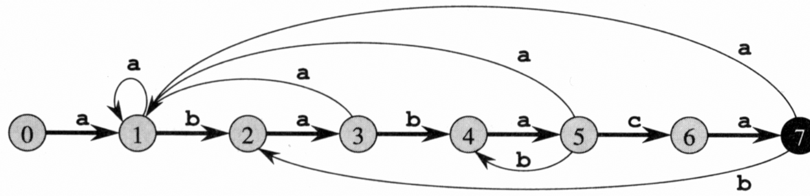
this result is proved as a ***Theorem*** below.

In words, this means that after scanning the first ***i*** characters of the text string ***T***, the machine is in state $f(T_i) = q$, where $q = s(T_i)$ is the length of the longest suffix of T_i that is also a prefix of the pattern ***P***.

If the next character scanned is ***T[i + 1] = a,*** then the machine should make a transition to state $s(T_{i+1}) = s(T_i a)$.

The proof of the theorem shows that $a(T_i a) = c(P_q a)$. That is, to compute the length of the longest suffix of $T_i a$ that is a prefix of ***P***, we can compute the longest suffix of $P_q a$ that is a prefix of ***P***. At each state, the machine only needs to know the length of the longest prefix of ***P*** that is a suffix of what has been read so far.

Therefore, setting ***d(q, a) = s (Pq a)*** maintains the desired invariant above. This informal argument will be made rigorous shortly.



(a)

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Example:

(a) A state-transition diagram for the string-matching automaton that accepts all strings ending in the string ***ababaca***. State ***0*** is the start state, and state ***7*** (shown blackened) is the only accepting state. A directed edge from state ***i*** to state ***j*** labeled ***a*** represents $\delta(i, a) = j$. The right-going edges forming the "spine" of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are not shown; by convention, if a state ***i*** has no outgoing edge labeled ***a*** for some ***a in*** Σ, then $\delta(i, a) = 0$.

(b) The corresponding transition function δ and the pattern string ***P = ababaca***. The entries corresponding to successful matches between pattern and input characters are shown shaded.

(c) The operation of the automaton on the text ***T = abababacaba***. Under each text character ***T[i]*** is given the state $\phi(T_i)$ the automaton is in after processing the prefix ***T***. One occurrence of the pattern is found, ending in position 9.

In the string-matching automaton of example above, for example, we have $\delta(5, b) = 4$. This follows from the fact that if the automaton reads a ***b*** in state ***q = 5***, then $P_q b$ = ***ababab***, and the longest prefix of ***P*** that is also a suffix of ***ababab*** is P_4 = ***abab***.

To clarify the operation of a string-matching automaton, we now give a simple, efficient program for simulating the behavior of such an automaton (represented by its transition function δ) in finding occurrences of a pattern ***P*** of length ***m*** in an input text ***T[1 … n]***. As for any string-matching automaton for a pattern of length

m, the state set *Q* is *{ 0, 1, … , m }*, the start state is *0*, and the only accepting state is state *m*.

FINITE-AUTOMATON-MATCHER *(T, d, m)*

```
1 n := length[T]
2 q := 0
3   for i := I to n
4       do q := d(q, T[I])
·           if q = m
·           then s := i - m
7             print "Pattern occurs with shift" ~s
```

The simple loop structure of FINITE-AUTOMATON-MATCHER implies that its running time on a text string of length *n* is *O(n)*. This running time, however, does not include the time required to compute the transition function *d* We address this problem later, after proving that the procedure FINITE-AUTOMATON-MATCHER operates correctly.

Consider the operation of the automaton on an input text *T[1, …, n]*. We shall prove that the automaton is in state $s(T_i)$ after scanning character *T[i]*. Since $s(T_i) = m$ if and only if P »T_i, the machine is in the accepting state *m* if and only if the pattern *P* has just been scanned. To prove this result, we make use of the following two lemmas about the suffix function *s*.

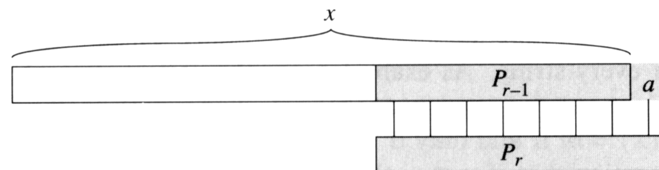


Illustration of Lemma 1.

Lemma 1 (Suffix-function inequality):
For any string *x* and character *a*, we have $s(xa) \leq s(x) + 1$.

Proof Referring to figure above, let $r = s(xa)$. *If* $r = 0$, then the conclusion $r \leq s(x) + I$ is trivially satisfied, by the nonnegativity of $s(x)$. So assume that $r > 0$. Now, P_r » *xa*, by the definition of *a*. Thus, P_{r-1}, » *x*, by dropping the *a* from the end of P_r, and from the end of *xa*. Therefore, $r - I \leq ó(x)$, since *ó(x)* is largest *k* that P_k » x.

Lemma 2 (Suffix-function recursion lemma):

For any string character a, if q = ó(x), then ó (xa) = ó (P_qa).

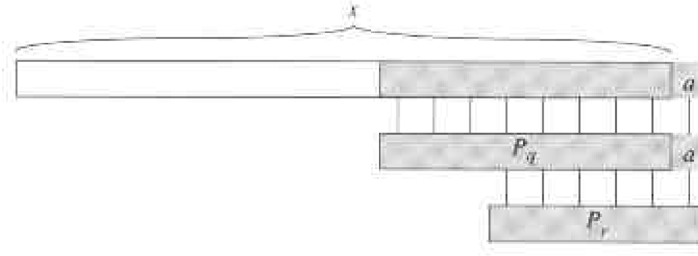


Illustration of lemma 2.

Proof: From the definition of *ó,* we have $\boldsymbol{P_q}$ **» *x*.** as figure above shows $\boldsymbol{P_q a}$**» *xa.*** If we let ***r = ó(xa),*** then $\boldsymbol{r \leq q+1}$ by lemma 1. Since $\boldsymbol{P_q a}$**» *xa,*** $\boldsymbol{P_r}$ **» *xa,*** and $\boldsymbol{P_r} \leq \boldsymbol{P_q a}$, lemma above implies that $\boldsymbol{P_r}$ **»** $\boldsymbol{P_q a}$. Therefore, $\boldsymbol{r \leq}$ ***ó(P****$_q$* ***a*)**, that is, ***ó(xa)*** $\leq$ ***ó(P****$_q$**a*)**. But we also have ***ó (P****$_q$**a*) $\leq$ *ó (xa)***, *since $P_q a$ » xa.* Thus, ***ó (xa) = ó (P****$_q$* ***a*)**.

We are now ready to prove the main theorem characterizing the behavior of a string-matching automaton on a given input text. As noted above, this theorem shows that the automaton is merely keeping track, at each step, of the longest prefix of the pattern that is a suffix of what has been read so far.

Theorem :

If ***f*** is the final-state function of a string-matching automaton for a given pattern ***P*** and ***T[1 … n]*** is input text for the automaton, then ***f(Ti) = s*** $\boldsymbol{(T_i)}$for ***i = 0, 1, … n.***

Proof: The proof is by induction on ***i***. For ***i = 0***, the theorem is trivially true, since $\boldsymbol{T_0}$ **= *e*.** Thus, ***f*** $\boldsymbol{(T_0)}$ **= *s*** $\boldsymbol{(T_0) = 0}$.

Now, we assume that ***f*** $\boldsymbol{(T_i)}$ **= *s*** $\boldsymbol{(T_i)}$ and prove that ***f*** $\boldsymbol{(T_{i+1})}$ **= *s*** $\boldsymbol{(T_{i+1})}$. Let ***q*** denote ***f*** $\boldsymbol{(T_i)}$, and let ***a*** denote ***T[i + 1]***. Then,

f $\boldsymbol{(T_{i+1})}$	=	***f*** $\boldsymbol{(T_i a)}$	(by the definitions of $\boldsymbol{T_{i+1}}$ and ***a***)
	=	***d(f*** $\boldsymbol{(T_i)}$***, a)***	(by the definition of ***f***)
	=	***d(q, a)***	(by the definition of ***q***)
	=	***s*** $\boldsymbol{(P_q a)}$	(by the definition of ***d***)
	=	***s*** $\boldsymbol{(T_i a)}$	(by lemma 2 and induction)
	=	***s*** $\boldsymbol{(T_{i+1})}$	(by the definition of $\boldsymbol{T_{i+1}}$).

By induction, the theorem is proved.

By this theorem, if the machine enters state ***q*** on line ***4***, then ***q*** is the largest value such that $\boldsymbol{P_q}$ » $\boldsymbol{T_i}$. Thus, we have ***q = m*** on line ***5*** if and only if an occurrence of the pattern ***P*** has just been scanned. We conclude that FINITE-AUTOMATON-MATCHER operates correctly.

Computing the Transition Function

The following procedure computes the transition function ***5*** from a given pattern ***P[1 … m]***.

COMPUTE-TRANSITION-FUNCTION ***(P, Ó)***

```
1   m := length [P]
2   for q := 0 to m
3       do for each character a Î Ó
4               do k := min (m + 1, q + 2)
5                   repeat k := k - 1
6                     until P_k » P_q a
7                   δ(q, a):= k
8                   return δ
```

This procedure computes ***d(q, a)*** in a straightforward manner according to its definition. The nested loops beginning on lines ***2*** and ***3*** consider all states ***q*** and characters ***a***, and lines ***4-7*** set ***d(q, a)*** to be the largest ***k*** such that P_k » $P_q a$.The code starts with the largest conceivable value of ***k***, which is min(m, q + 1), and decreases ***k*** until P_k » $P_q a$.

The running time Of COMPUTE-TRANSITION-FUNCTION is ***O(m***^{3} ***Ó)*** because the outer loops contribute a factor of ***m Ó***, the inner **repeat** loop can run at most ***m + 1*** times, and the test P_k » $P_q a$ on line ***6*** can require comparing up to ***m*** characters. Much faster procedures exist; the time required to compute ***d*** from ***P*** can be improved to ***O(m Ó)*** by utilizing some cleverly computed information about the pattern ***P***. With this improved procedure for computing ***d***, the total running time to find all occurrences of a length-m pattern in a length-n text over an alphabet ***Ó*** is ***O(n + m Ó)***

Boyer – Moore Algorithm

If "backing up" is not difficult, then a significantly faster string-searching method can be developed by scanning the pattern from ***right to left*** when trying to match it against the text.

When searching for our sample pattern 10100111, if we find matches on the eighth, seventh, and sixth character but not on the fifth, then we can immediately slide the pattern seven positions to the right, and check the fifteenth character next, because our partial match found 111, which might appear elsewhere in the pattern.

Of course, the pattern at the end does appear elsewhere in general, so we need a *next* table as above.

A right-to-left version of the ***next*** table for the pattern 101 10101 is shown in figure below: in this case next ***[j]*** is the number of character positions by which the pattern can be shifted to the right given that a mismatch in a right-to-left scan occurred on the jth character from the right in the pattern.

This is found as before, by sliding a copy of the pattern over the last ***j-1*** characters of itself from left to right, starting with the next-to-last character of the copy lined up with the last character of the pattern and stopping when all overlapping characters match.

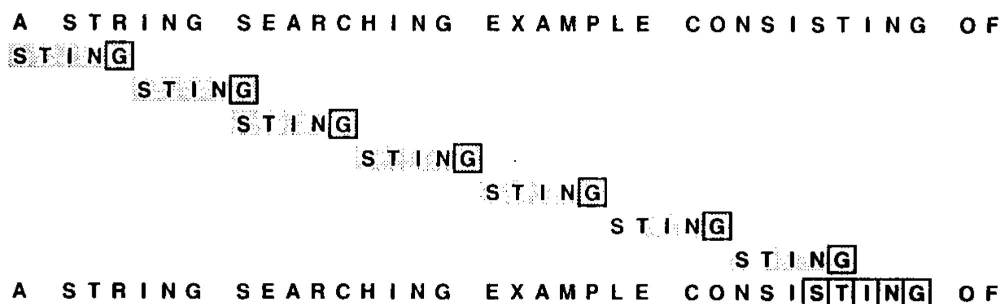
This leads directly to a program which is quite similar to the above implementation of the Knuth-Morris-Pratt method. We won't explore this in more detail because there is a quite different way to skip over characters with right-to-left.

Using "backup" more often, sometime we can expect better results, by scanning the pattern from right to left.

For the pattern above, if we scan from the right, we can expect restart positions:

j	next[j]	
2	4	10110101 10110101
3	7	10110101 10110101
4	2	10110101 10110101
5	5	10110101 10110101
6	5	10110101 10110101
7	5	10110101 10110101
8	5	10110101 10110101

Restart positions for Boyer-Moore search



Boyer-Moore string search using the mismatched character heuristics

function *mischarsearch*: **integer**;
    var i, j: **integer**;

```
        begin i:= M; j:= M; init skip;
        repeat if a[i] = p[j]
                then begin i:= i-1; j: j-1 end
                else begin if M-j+1 > skip[index(a[i])]
                        then i:= i + M - j + 1 else i:= i + skip[index(a[i])];
                j:= Mp end;
        until (j < 1) or (i > N);
mischarsearch:= i+1;
end;
```

Property: Boyer-Moore never uses more than $M + N$ character comparisons, and uses about N/M steps if the alphabet is not small and the pattern is not long.

Rabin-Karp Algorithm

A brute-force approach to string searching that we didn't examine above would be to exploit a large memory by treating each possible M -character section of the text as a key in a standard hash table.

But it is not necessary to keep a whole hash table, since the problem is set up so that only one key is being sought; all we need to do is compute the hash function for each of the possible M-character sections of the text and check if it is equal to the hash function of the pattern.

The problem with this method is that it seems at first to be just as hard to compute the hash function for M characters from the text as it is merely to check to see if they're equal to the pattern. Rabin and Karp found an easy way to get around this difficulty for the hash function:

$$h(k) = k \bmod q$$

where q (the table size) is a large prime. In this case, nothing is stored in the hash table, so q can be taken to be very large.

The method is based on computing the hash function for position i in the text given its value for position $i - 1$, and follows directly from a mathematical formulation.

Let's assume that we translate our M characters to numbers by packing them together in a computer word, which we then treat as an integer. This corresponds to writing the characters as numbers in a base-d number system, where d is the number of possible characters.

The number corresponding to $a[i], \dots a[i + M - 1]$ is thus

$$x = a[i]\, d^{M-1} + a[i+1]\, d^{M-2} + \dots + a[i+M-1]$$

and we can assume that we know the value of ***h(x) = x mod q***. But shifting one position right in the text simply corresponds to replacing

$$x \text{ by } (x - a[i]\, dM\text{-}1)d + a[i + M].$$

Using the property of ***mod*** operation (we can transpose it with all other arithmetic operations) to keep the numbers as small as possible, it leads to an efficient algorithm.

Property: Rabin-Karp pattern matching is extremely likely to be linear.

Longest Common Subsequence (LCS)

Problem is formulated as a need to find the longest common subsequence in two given sequences, and it is meet in various application so often, that for example, UNIX has a special command ***diff***, to compare files.

Algorithm will be expressed in the terms of ***set*** data structure, main operations for data are:

- ***FIND(r)*** – finds the set, containing element ***r***;
- ***SPLIT(S, T, U, r)*** – set S will be splitted into two sets, ***T*** and ***U***, and set ***T*** will contain elements, less then ***r***, and the set ***U*** – greater or equal to ***r***;
- ***MERGE(S, T, U)*** - sets ***S*** and ***T*** will be merged into the set ***U***.

The final goal is to find LCS from two sequences ***A*** and ***B***.
Algorithm begins with sets ***PLACES(a)*** for any element from the sequence ***A***:

$$PLACES(a) = \{i \text{ - number} \mid \text{that element } a \text{ is in } i^{th} \text{ position}\}.$$

This set of sets can be created efficiently by creating corresponding lists of positions. If elements of the sequence ***A*** are complicated enough, let's say lines of text file, hashing porcedures can be used.

The complexity of ***PLACES*** is expected as ***O(n)***.

PLACES being created, the ***LCS*** can be find straightforward. We present an algorithm to calculate the length of ***LCS***, the modification to find ***LCS*** is left to the students.

The algorithm begins with the consecutive elements of the sequence ***B***. For every element b_j, $0 < j < m$ of the set ***B***, we need to find the ***LCS*** for the sets $\{a_1,..a_i\}$ and $\{b_1,...b_j\}$.
We start with S_0 = {all the positions of the sequence A}.

The position indices ***i*** are joined into sets S_k, such that the length of ***LCS*** of the sets $\{a_1,...a_i\}$ and $\{b_1,... b_j\}$ is equal to ***k*** ($0 =< k =< n$). Let's note that the set S_k always has numbers in ascending order, and numbers in S_{k+1}, are always greater than numbers in the set S_k.

The algorithm is iterative. Let's say that all the sets S_k, for the position of ***j-1*** of the sequence ***B*** are already created, and now we consider position ***j*** in order to create for this position new sets S_k. We need only to modify existing sets S_k.

We consider the set ***PLACES***(b_j). For every element ***r*** of this set we need to decide if it is able to extend the ***LCS*** or not. If yes, and if both elements ***(r-1)*** and ***r*** will be in the same set S_k, then all the elements $s >= r$ have to be in the set S_{k+1}. This checking will be done by modifying sets in the following way:

- ***FIND(r)*** – we find the set containing r, (in order to find S_k needed);
- if ***FIND(r-1)*** is not equal to the set S_k, then the new element b_j is not able to increase the length of common subsequence, so nothing to modify, and the algorithm finishes this step;
- if ***FIND(r-1)*** $= S_k$, then the operation ***SPLIT*** (S_k, S_k, S_k', r) will be produced, in order to find elements from the set S_k , greater or equal then ***r***;
- ***MERGE***(S_k', S_{k+1}, S_{k+1}) – selected elements from the previous step will be moved to the set S_{k+1}.

When all the elements of the set ***B*** are processed, the length of ***LCS*** will be find in the set S_k, where ***k*** is a maximal number.

The algorithm is expected to work more efficiently, if the numbers of set ***PLACES***(b_j) will be processed in descending order, this may reduce the number of operations.

The complexity of the algorithm. The number of operations can be evaluated assymptotically by $O(p \log n)$: where ***n*** is a ***max*** of lengths of sequences, and ***p*** is a number of pairs of positions where symbols of sequences coincide. In the worst case ***p*** is equal to n^2 .

Algorithmic Analysis

Asymptotic Notation

Asymptotic analysis is used to explore the behavior of a function, or a relationship between functions, as some parameter of the function tends toward an asymptotic value. This type of analysis can be used to show that ***two functions are roughly equal to each other***, or that the value of some function ***grows asymptotically at a greater rate*** than some other function.

A notational convention used extensively in asymptotic analysis is ***O-notation***. By using ***O-notation*** we will have a means of expressing an ***asymptotic upper bound*** on a function.

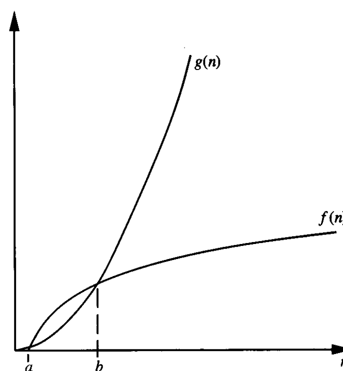
Let $\boldsymbol{Z}^+$ represent the set of nonnegative integers (i.e., natural numbers), and $\boldsymbol{R}^+$ the set of nonnegative real numbers, then this notation is defined as follows:

Definition of ***O-notation***. Assume there are two arbitrary functions $f(n)$ and $g(n)$ such that $(f(n), g(n) : \boldsymbol{Z}^+ \to \boldsymbol{R}^+)$. It is said that $f(n)$ is "***big-oh***" of $g(n)$, written as

$$f(n) = O(g(n))$$

if there exist positive constants $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that $\boldsymbol{f(n) <= cg(n)}$ for all $\boldsymbol{n \geq n_0}$.

More details and examples.



ln x <= O (x/4)

It is also important to recognize that the equality in the statement $\boldsymbol{f(n) = O(g(n))}$ is "***one-way***." Since the term $\boldsymbol{O(g(n))}$ really specifies an infinite set of functions, this statement is saying that ***f(n) belongs*** to the set of functions that can be bounded from above by a fixed multiple of $\boldsymbol{g(n)}$, when $\boldsymbol{n}$ is sufficiently large. However, for mainly historical reasons, this relationship is usually represented by an equality.

Let's derive some additional properties of the ***O-notation***. Recall that we are dealing with asymptotic upper bounds. Thus, if $\boldsymbol{f(n) = O(g(n))}$, and $\boldsymbol{h(n)}$ is a function that is larger in value than $\boldsymbol{g(n)}$ when $\boldsymbol{n}$ is sufficiently large, then $\boldsymbol{f(n) = O(h(n))}$. This means that it is perfectly correct to write that for $\boldsymbol{f(n) = n^2}$, $\boldsymbol{f(n) = O(n^3)}$. However, in this case the upper bound is not tight.

Since logarithms to the base $\boldsymbol{b}$ of $\boldsymbol{n}$ are related to logarithms to the base $\boldsymbol{a}$ of $\boldsymbol{n}$ by a constant, there is no difference between the expressions $\boldsymbol{O(lgn)}$, $\boldsymbol{O(\log_e n)}$, or $\boldsymbol{O(\log_{10} n)}$. Thus, we will simply use $\boldsymbol{O(logn)}$.

An asymptotic lower bound on a function is defined:

Definition 1.2. **Ω-notation.** Assume there are two arbitrary functions $f(n)$ and $g(n)$ such that $\{f(n), g(n) : \mathbb{Z}^+ \rightarrow \mathbb{R}^+\}$. It is said that $f(n)$ is "big-omega" of $g(n)$, written as

$$f(n) = \Omega(g(n))$$

if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

There is a relation with the previous definition of a ***O–notation***.

$$f(n) = \Omega(g(n)) \text{ if and only if } g(n) = O(f(n))$$

An asymptotic tight bound on a function:

Definition 1.3. **Θ-notation.** Assume there are two arbitrary functions $f(n)$ and $g(n)$ such that $\{f(n), g(n) : \mathbb{Z}^+ \rightarrow \mathbb{R}^+\}$. It is said that $f(n)$ is "big-theta" of $g(n)$, written as

$$f(n) = \Theta(g(n))$$

if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

A useful approximation of ***n!*** makes use of ***Q*** – ***notation*** (so-called Stirling's formula):

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Therefore, for that an expression like

$$\text{å } O(i)$$

there is only a single anonymous function (a function of i). This expression is thus *not* the same as ***O(1)*** + ***O(2)*** +   + ***O(n)***, which doesn't really have a clean interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation, as in

$$2n^2 + Q(n) = Q(n^2)$$

To interpret such equations use the following rule:

- *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.*

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ *is* asymptotically tight, but the

bound $2n = O(n^2)$ *is* not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define ***o(g(n))*** ("little-oh of ***g*** of ***n****")* as the set

$$o(g(n)) = \{ f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0$$

For example, $2n = o(n^2)$, but $2n^2 <> o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in
$f(n) = O(g(n))$, the bound $0 \leq f(n) <= cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for *all* constants $c > 0$.

Intuitively, in the o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

w-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. ω-notation is used to denote a lower bound that is not asymptotically tight. One way to define it is by

f(n)* Î w*(g (n)) if* and only *if g (n)* Î *o (f (n))

Formally, however, we define **w*(g(n))*** ("little-omega *of* ***g*** *of* ***n****")* as the set
w $(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that}$

$$0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \}.$$

The relation ***f(n)*** = **w*(g(n))*** implies that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

Comparison of functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that $f(n)$ and $g(n)$ are asymptotically positive.

Transitivity:

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)),$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)),$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n)),$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \text{ imply } f(n) = o(h(n)),$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \text{ imply } f(n) = \omega(h(n)).$$

Reflexivity:

$$f(n) = \Theta(f(n)),$$
$$f(n) = O(f(n)),$$
$$f(n) = \Omega(f(n)).$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)),$$
$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).$$

Because of these properties hold for asymptotic notations, one can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and *b:*

$$f(n) = O(g(n)) \approx a \le b,$$
$$f(n) = \Omega(g(n)) \approx a \ge b,$$
$$f(n) = \Theta(g(n)) \approx a = b,$$
$$f(n) = o(g(n)) \approx a < b,$$
$$f(n) = \omega(g(n)) \approx a > b.$$

One property of real numbers, however, does not carry over to asymptotic notation:

Trichotomy: For any two real numbers ***a*** and ***b***, exactly one of the following must hold:

$$a < b,\ a = b, \text{ or } a > b.$$

Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions ***f(n)*** and ***g(n)***, it may be the case that neither

$f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds.

For example, the functions ***n*** and $n^{1+\sin n}$ cannot be compared using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

Efficiency of Algorithms

Typically we are interested in analyzing the ***efficiency*** of a given algorithm. This involves determining the quantity of computer resources consumed by the algorithm.

The ***resources*** that are usually measured include the ***amount of memory*** and the ***amount of computational*** time required by the algorithm. Determining the efficiency of an algorithm is by no means the only figure of merit that can be used to judge an algorithm.

Of utmost importance is the ***correctness*** of the algorithm itself-that is, does the algorithm always produce the correct result for all possible input values. Suffice it to say that rigorously proving ***the correctness of a nontrivial algorithm*** can be ***an extremely difficult*** task.

A more qualitative figure of merit involves the ***program complexity*** of an algorithm, which considers both the difficulty of implementing an algorithm along with the efficiency of the algorithm itself. A measure for program complexity requires a consideration of the context in which the algorithm will be used.

In many instances, it is more advantageous to implement a simple algorithm that solves a given problem, than to implement a more complex algorithm that solves the same problem using fewer resources.

For example, if we consider the two algorithms for computing the exponential function, a straightforward approach and a more complicated recursive approach, then the first saves many efforts for small integer powers but , if exponentials to large integer powers must be computed, then a tremendous number of multiplications can be saved by using the recursive procedure, for the case where ***n = 65,536***, the straightforward approach will execute ***65,535*** multiplications, while the recursive procedure only executes 16 multiplications.

Running Time and Memory Usage

Our analysis of algorithms and data structures will mainly focus on efficiency considerations. This analysis is performed by considering the amount of resources an algorithm consumes as a function of the size of the input to the algorithm. For example, the efficiency of an algorithm might be determined by the amount of memory required, and/or by the amount of computational time required, as the size of the input to the algorithm grows.

We will use the term ***memory space*** (or simply ***space)*** to refer to the amount of memory required by an algorithm, and the term ***running time*** (or simply ***time)*** to refer to the computational time required by an algorithm.

The space occupied by an algorithm is determined by the number and sizes of the variables and data structures used by the algorithm. The time required by an algorithm is determined by the number of elementary operations that must be performed during the algorithm's execution.

In many cases, the space and time required by an algorithm are ***inversely*** related. That is, we may be able to reduce space requirements by increasing the running time, or conversely, reduce time requirements by increasing memory space. This situation is referred to as the ***space-time tradeoff.***

In many cases, we will restrict our analysis to the running time of an algorithm, and we will assume that there is sufficient memory to implement the algorithm. This is justified by noting that with current technology, computer memory is typically abundant and cheap in relation to the processor. This is not to say that issues of memory space are unimportant. For example, in certain embedded systems a processor may have a very limited amount of memory associated with it. Therefore, we wish to stress that the techniques developed below for analyzing the running time oi an algorithm can also be applied to the analysis of space requirements.

Measuring Running Time

Since the running time of an algorithm may differ depending upon the input it receives, it is calculated as a function of the input size. Therefore, one approach would be to implement a given algorithm, and for various input sizes, measure the running time using a stopwatch. Computational time measured in this fashion is said to use ***wall-clock time.***

This approach may not be considered very rigorous for a number of reasons. First, the speed with which a program executes depends upon the type of machine executing the program, as well as the skill of the programmer who implemented it. These factors, which are included when measuring running time using a wall clock, should not be considered when analyzing an algorithm.

We would like to be able to make judgments regarding the merit of an algorithm independent of a specific software implementation, or the hardware on which it is implemented. In order to perform this type of analysis, we must assume some general model of computation on which an implementation of the algorithm we are investigating will be executed. The most common model of computation used to evaluate algorithms is the ***random-access machine (RAM)***.

This model assumes the existence of a single generic processor, random-access memory, and a set of basic operations that the processor can execute. It is then

assumed that the algorithm is implemented as a computer program that will execute on this RAM model using its set of basic operations, and that the instructions are executed one after another, with ***no concurrent*** operations.

Now, to determine the running time of an algorithm we must determine the time required by each basic operation, and multiply that by how many times the operation is used when implementing the algorithm on the RAM model.
Asymptotic analysis is normally used to express the running time of an algorithm and the use of asymptotic notation in algorithmic analysis has a number of advantages. Recall that regarding an algorithm, we wish to arrive at some figure of merit that is independent of a particular software implementation or hardware platform.

The use of asymptotic notation hides the constant amounts of operations, whose values are machine specific; and distills the most important aspect of the algorithm-that its running time somehw depend with increasing input size. It gives the ***rate of growth*** of the running time, not the actual running time of an algorithm.

We will refer to the running time of an algorithm that is obtained in this fashion as the ***asymptotic running time complexity,*** or simply time complexity of the algorithm.

Because we are only interested in the asymptotic complexity of an algorithm, it will not be necessary to go through the analysis in order to determine an algorithm's running time. Instead, the running time of a nonrecursive algorithm can be determined by simply observing the behavior of any of its loop structures.

Typically, the number of iterations performed by a loop will vary with the size of the input to the algorithm. Therefore, in order to determine the running time of an algorithm, we simply count the number of basic operations that are performed inside of its loop structures.

For recursive algorithms we also need to count the number of basic operations that are performed-but in this case, there are no loop structures to analyze. Instead, we must perform an analysis to determine the number of operations that the recursive algorithm will implement as a function of the input size.

Polynomial-Time and Superpolynomial-Time Algorithms

In the previous section, we introduced the notion that algorithms can be judged according to their efficiency. This idea is actually carried a step further by computer scientists when they categorize an algorithm as either being reasonable (tractable, efficient) or unreasonable (intractable, inefficient).

The dividing line between these two categories is drawn as follows. If the time complexity of an algorithm can be expressed as $O(p(n))$ for some polynomial

function p, where n is the input size, then this algorithm is said to be a ***polynomial-time algorithm.*** Any algorithm that cannot be bounded in this fashion is called a ***superpolynomial-time algorithm***.

The motivation for categorizing algorithms as discussed above follows from this fact: As the size of the input n grows large, the running times of superpolynomial-time algorithms become exceedingly large, while the running times of polynomial-time algorithms generally remain reasonable. This phenomenon is clearly illustrated in the table below, where it compares the running times of several polynomial and superpolynomial algorithms:

Algorithm complexity	Size n					
	10	20	30	40	50	60
polynomial						
n	.00001 sec	.00002 sec	.00003 sec	.00004 sec	.00005 sec	.00006 sec
n^2	.0001 sec	.0004 sec	.0009 sec	.0016 sec	.0025 sec	.0036 sec
n^3	.001 sec	.008 sec	.027 sec	.064 sec	.125 sec	.216 sec
n^5	.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13 min
super-polynomial						
2^n	.001 sec	1.0 sec	17.9 min	12.7 days	35.7 yr	366 cen
3^n	.059 sec	58 min	6.5 yr	3855 cen	$2 \cdot 10^8$ cen	$1 \cdot 10^{13}$ cen
$n!$	3.63 sec	771 cen	$8 \cdot 10^{16}$ cen	$3 \cdot 10^{32}$ cen	$1 \cdot 10^{49}$ cen	$3 \cdot 10^{66}$ cen

Effects of improvements in computer technology on the size of the largest problem solvable in one hour for several polynomial-time and superpolynomial-time algorithms.

Algorithm complexity	Size of largest problem solvable in 1 hour		
	With present computer	With computer 100 times faster	With computer 1000 times faster
polynomial			
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
superpolynomial			
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

So if we can show that some algorithm is intractable, then it makes no sense to try to solve large problems using this algorithm--even if we have a supercomputer at our disposal. Instead, we would be better served by spending time looking for a better algorithm, one that solves a special case of the problem under

consideration, or one that solves the problem approximately. One of these approaches may yield an algorithm that is useful as well as tractable.

Algorithmic Analysis

It is important to recognize the distinction between a ***problem*** and an ***algorithm*** that solves a problem. A ***problem*** has a ***single*** problem statement that describes it in some general terms; however, there may be ***many*** different ways to solve this problem, and some of these solutions may be more efficient than others. Thus, a number of different algorithms can exist for solving a computational problem, and each of these algorithms could have a ***different*** running-time complexity.

Given a computational problem, an algorithm designer is typically interested in finding the fastest possible algorithm that solves it. In some cases, it is possible to specify a ***theoretical lower bound*** on the number of operations required to solve a problem for any possible algorithm and input instance.

We can also talk about an obvious or ***trivial lower bound*** for the number of operations required to solve a problem.

In many cases there is a so-called ***algorithmic gap*** between the theoretical lower bound for a problem, and the best available algorithm for solving that problem. In these cases, algorithm designers may work to close this gap by developing faster algorithms. If a faster algorithm is found, then it establishes a new lower bound for the fastest known algorithm that solves the problem.

If an algorithm is found that asymptotically matches the theoretical lower bound for a problem, then at least in an asymptotic sense, no algorithm could be faster.

This is not to say that nothing can be done to improve such a solution-its asymptotic running time may be "hiding" a large constant, or the asymptotic results may only apply when the input size is extremely large.

For many important problems, theoretical lower bounds have not been found. In these cases, the only basis of comparison is against the lower bound established by the fastest known algorithm for solving the problem.

In order to analyze a problem in the manner discussed above, we must be able to determine the running time of any new algorithms we develop. A little refine of this idea further by considering how to handle cases where the running time of an algorithm varies depending upon the order in which the input data is supplied to it.

Worst-case Analysis

In many algorithms, running time will vary not only for inputs of different sizes, but also for different inputs of the same size. That is, we may find that the

running times of an algorithm will vary for inputs of the same size, depending upon the initial ordering of the input data.

Typically, in the analysis of algorithms, the ***worst-case situation*** is used to ***express running*** time. This convention is actually useful for a number of reasons.

First, if we know the worst-case running time of an algorithm, then we have a guarantee that the algorithm will never take more than this time. Such a guarantee can be quite important in time-critical software applications that deal with, for example, air-traffic control, missile and aircraft guidance systems, or control of a nuclear power plant. In addition, it is often found in practical applications that the worst-case running time of an algorithm occurs frequently.

We define the ***worst-case running time*** of an algorithm to be the maximum running time of that algorithm over all possible inputs of size ***n***.

A ***warning*** should be issued. In practice we might find that an algorithm whose worst-case running time is given by, for example, ***Q (n log n)*** usually takes longer to execute than an algorithm that accomplishes the same task, but whose worst-case running time is given by ***Q*** (n^2).

It may turn out that the ***Q*** (n^2) algorithm almost always has a running time that is close to ***n***, while the ***Q (n log n)*** algorithm may almost always have a running time that is close to ***nIgn***. In spite of this possibility, we may safely state that in the majority of cases, algorithms whose worst-case running times are better in the ***Q-notation*** sense are also better in practice.

Average-case Analysis

We may also define the ***average-case running time*** of an algorithm for an input of size ***n***, to be the value that is obtained by averaging the running times of the algorithm over all possible inputs of size ***n***.

Although average-case analysis may appear to be better than worst-case analysis for measuring running time, it usually is not. Average-case analysis requires that we assume some underlying probability ***distribution for the input instances***. If this assumption is violated in practice, then the average-case analysis may not be meaningful. In addition to the difficulty of determining an appropriate probability distribution function for the input data, such average-case analysis quite often becomes mathematically intractable. Nevertheless, on occasion we will find it convenient to assume that each input instance is equally likely (i.e., that the underlying probability distribution for the input data is uniform) and proceed with an analysis of the running time of that algorithm. In summary, for the reasons stated above we will typically use the worst-case measure to express the running time complexity of an algorithm; however, we will use average-case analysis if it makes sense for a particular algorithm or application of an algorithm.