Special course in Computer Science: Advanced Text Algorithms

Lecture 10: Detecting text regularities

Eugen Czeizler
Department of IT, Abo Akademi
http://combio.abo.fi/teaching/textalg/

(slides originally by I. Petre, E. Czeizler, V. Rogojin)

Regularities in texts

- Regularity = a similarity between two (or more) factors of a given text.

- Examples of regularities:
  - repetition = a factor is an exact copy of another one

$$aba\,aba; \quad aba\,bbb\,aba$$

  - symmetries = a factor is a symmetric copy of another one

$$abb\,bba; \quad abb\,cdd\,bba$$

Detecting regularities

- Question: Why do we care about regularities in texts?

- There are many applications:
  - text algorithms: e.g., finding the borders of a word, finding the periods of a word.
  - the compression of a text is very successful if the text has many regularities.
  - analysis of biological sequences: repetitions in genomes, are intensively studied because, e.g., some over-repeated short segments can cause to genetic diseases.

Regularities in texts

- We will discuss here the cases when the similar factors are consecutive in the text:
  - repetitions: hotshots = $(hots)^2$
  - even palindromes: ww^R (abbbba)
  - odd palindromes: waw^R (abbcbba)

- For a word of length n, w=w[1]..w[n], the word w^R=w[n]..w[1] is called the reverse image of w.

Finding Squares

- Question: Decide (in linear time) whether a given word w contains a square factor, i.e., a non-empty factor of the form uu.

- We discuss first an O(n log n) algorithm based on a divide-and-conquer approach.

Divide-and-conquer approach (Bonus)

- Divide-and-conquer is an important algorithm design paradigm based on multi-branched recursion.
- Recursively breaks down a problem into two or more sub-problems of the same (or related) type, until these are simple enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem.
- This technique is the basis of efficient algorithms for all kinds of problems, such as
  - sorting: e.g., quicksort, merge sort,
  - multiplying large numbers: e.g. Karatsuba algorithm,
  - syntactic analysis: e.g., top-down parsers

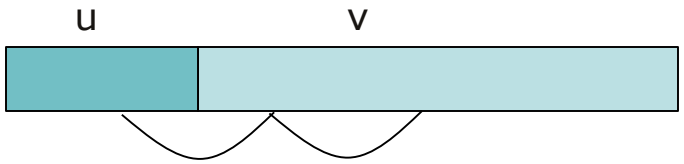
Finding squares

- Given 2 words u and v which do not contain any squares, we first build a Boolean function test(u,v) that returns:
  - 1 if uv contains a square factor and
  - 0 otherwise.

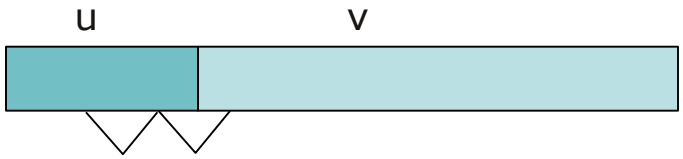
- Such a square factor has to start in u and end in v

Finding Squares

- The operation test is a composition of two smaller boolean functions: righttest and lefttest.
  - righttest searches for a square whose center is in v,



  - lefttest searches for a square whose center is in u.



The function righttest

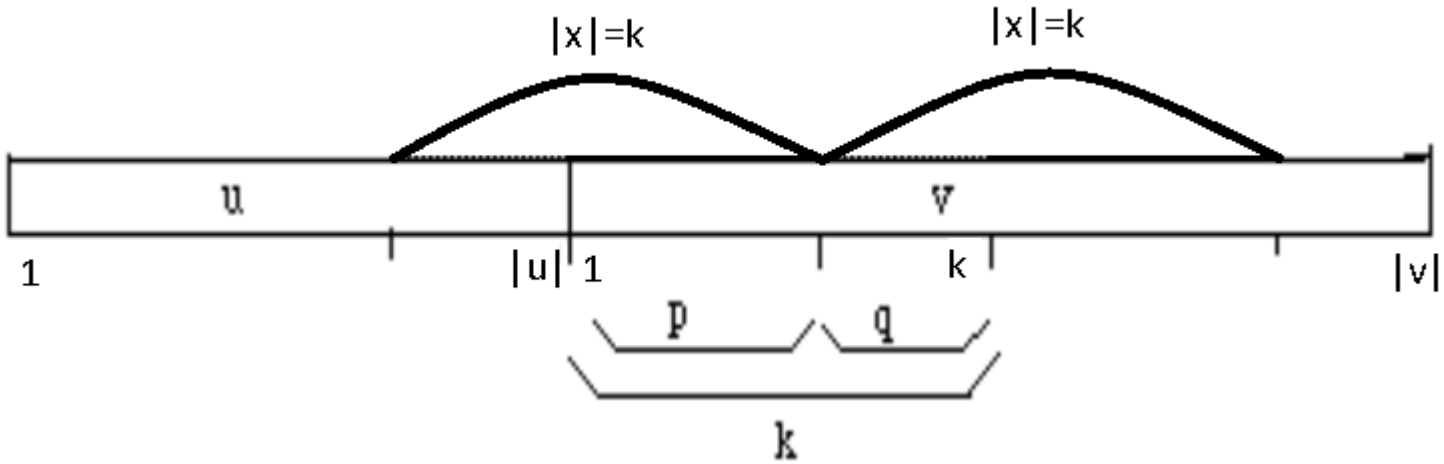
- We use two auxiliary tables related to string-matching

1) Table PREF is defined for the positions $1 \leq i \leq |v|$.
   - The value PREF[k] is the size of longest common prefix of v and v[k+1..|v|]

2) Table SUF_u is also defined for the positions $1 \leq i \leq |v|$
   - The value $SUF_u[k]$ is the size of longest suffix of v[1…k] which is also a suffix of u.

- Both of these tables can be computed in linear time with respect to |v|

The function righttest



- Suppose a square xx of size 2k occurs in uv.
- The suffix of v[1…k] of size q is also a suffix of u
  - So $q \leq SUF_u[k]$
- The prefix of v[k+1…|v|] of size p is a prefix of v
  - So, $p \leq PREF[k]$.
- Then, the existence of a square in uv, centered in v, reduces to testing for each position k in v if $SUF_u[k]+PREF[k] \geq k$

The function righttest

- Theorem: The Boolean value righttest(u,v) can be computed in O(|v|) time (independent on |u|).

Proof:

- The computation of the tables PREF and SUF_u can be done in O(|v|) time
- There exists a square centered in v if and only if for some position k, $SUF_u[k]+PREF[k] \geq k$
- This can be tested in O(|v|) time.

- Similarly, we can compute the Boolean value lefttest(u,v) in O(|u|) time.

- Theorem: The Boolean value test(u,v)=lefttest(u,v) OR righttest(u,v) can be computed in O(|u|+|v|) time

Testing for squares

- We can now write an algorithm which tests whether a word contains a square, using a divide-and-conquer technique.

$$\textit{function}\ \ SQUARE(text) : Boolean;$$

if $n > 1$ then begin

if $SQUARE(text[1..\left\lfloor \frac{n}{2} \right\rfloor])$ then return *true*;

if $SQUARE(text[\left\lfloor \frac{n}{2} \right\rfloor + 1..n])$ then return *true*;

if $test(text[1..\left\lfloor \frac{n}{2} \right\rfloor], text[\left\lfloor \frac{n}{2} \right\rfloor + 1..n])$ then return *true*;

end;

return *false*;

- The algorithm has O(n log n) time complexity.

Testing for squares

- The previous algorithm can be extended to detect all squares in a given text.
- Then then algorithm is actually optimal since there are words that contain exactly O(n log n) squares
  - Example: the Fibonacci words

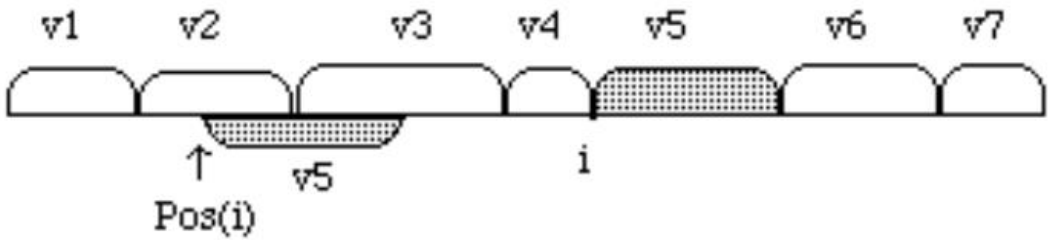
Testing for squares

- If the alphabet is fixed, then we can actually have a linear time algorithm for testing for squares.

- This algorithm uses a special factorization of words (called f-factorization) which is very useful for instance in data compression methods based on the elimination of repetitions (which will be discussed later).

F-factorizations

- Given a word w, its f-factorization is a sequence of non-empty words $(v_1,v_2,...,v_m)$ such that:
  - $v_1=w[1]$
  - For $k>1$, v_k is defined as follows: if $|v_1v_2...v_{k-1}|=i$, then v_k is the longest prefix u of $w[i+1..n]$ that occurs at least twice in $w[1..i]u$. If there is no such word u, then $v_k=w[i+1]$
  - We denote by $pos(v_k)$ the smallest position $l<i$, such that an occurrence of v_k starts at l. If there is no such position, then $pos(v_k)=0$.



Example of f-factorization

- Let w=abaabbaabbaababa of size 16.
- Then, v_1=a with $|v_1|$=1.
- Since there is no prefix u of w[2..16] that occurs at least twice in w[1]u, we set v_2=b. Then $|v_1v_2|$=2
- Similarly, and we see that u=a is a maximal prefix that occurs twice in w[1..2]u. So, we set v_3=a. Then $|v_1v_2v_3|$=3.
- w[4..16]=abbaabbaababa, and we see that u=ab is a maximal prefix that occurs twice in w[1..3]u=abaab. So, we set v_4=ab. Then $|v_1v_2v_3v_4|$=5.
- w[6..16]=baabbaababa, and we see that u=baabbaab is a maximal prefix that occurs twice in w[1..5]u=abaabbaabbaab. So, we set v_4=baabbaab. Then $|v_1v_2v_3v_4v_5|$=13.
- w[14..16]=aba which occurs 3 times in w[1..13]aba. So, we set v_5=aba.

F-factorization

- The f-factorization of a word w, and the computation of values pos(v_k) can be done in linear time using the suffix tree associated to w.

- Theorem: Let (v_1,v_2,…,v_m) be the f-factorization of a word w. Then, w contains a square iff for some k at least one of the following conditions holds:

(1) pos(v_k)+|v_k|≥| $v_1v_2…v_{k-1}$| (selfoverlapping of v_k),

(2) lefttest(v_{k-1}, v_k) or righttest(v_{k-1}, v_k),

(3) righttest($v_1v_2…v_{k-2}$, $v_{k-1}v_k$).

Testing for squares

- We can now write a linear-time algorithm testing for squares in a given text, on a fixed alphabet.

function $linear\_SQUARE(text) : Boolean;$

$compute\ the\ f-factorization\ (v_1, v_2, ..., v_m)\ of\ text;$

for $k := 1$ to m do

if (1) or (2) or (3) holds in the previous theorem then

return *true*;

return *false*;

Testing for squares in linear time

function $linear\_SQUARE(text) : Boolean$;
  $compute\ the\ f-factorization\ (v_1, v_2, ..., v_m)\ of\ text$;
  for $k := 1$ to m do
    if (1) or (2) or (3) holds in the previous theorem then
      return *true*;
  return *false*;

- The computation of the f-factorization can be done in linear time using suffix trees
- The verification for all k's of condition (1)$pos(v_k)+|v_k| \geq |v_1 v_2 \ldots v_{k-1}|$ takes linear time

Testing for squares in linear time

function $linear\_SQUARE(text) : Boolean;$

$compute\ the\ f-factorization\ (v_1, v_2, ..., v_m)\ of\ text;$

for $k := 1$ to m do

if (1) or (2) or (3) holds in the previous theorem then

return *true*;

return *false*;

- The verification for all k's of condition (2)lefttest(v_{k-1}, v_k) or righttest(v_{k-1}, v_k), takes also linear time
- The verification for some k of condition (3)righttest($v_1v_2...v_{k-2}$, $v_{k-1}v_k$) takes O(|$v_{k-1}v_k$|) so overall, for all k's the total time will be proportional to the summ of lengths of all v_k's, i.e., linear.
- Thus, the algorithm has O(|text|) time complexity.

Searching for symmetric words

- Problem 1: Decide whether a given word u has a prefix which is palindrome.
  - We discuss first the case of even palindromes, i.e., words of the form ww^R.

- There exists a brute-force algorithm for this problem having quadratic time complexity.

- We will improve it (and reach linear time complexity) by using more efficiently the information gathered by the algorithm.

Brute-force algorithm for finding even prefix-palindromes

Algorithm brute_force

$i := 1;$

while $i \leq \left\lfloor \frac{n}{2} \right\rfloor$ do begin

$j := 0;$

while $j < i$ and $text[i-j] = text[i+1+j]$ do

$j := j+1;$

if $j = i$ then return true;

$i := i+1;$

end;

return false;

- This algorithm checks whether text[1..2i] is an even palindrome by checking the following property:
- w(i,j): $text[i-j+1..i] = text[i+1..i+j]^R$

- The index i is the "center" of a potential palindrome
- The maximum value of j satisfying w(i,j) for some i, is called the radius of the palindrome centered at i

Brute-force algorithm for finding even prefix-palindromes

Algorithm brute_force

$i := 1;$

while $i \le \left\lfloor \frac{n}{2} \right\rfloor$ do begin

$j := 0;$

while $j < i$ and $text[i-j] = text[i+1+j]$ do

$j := j+1;$

if $j = i$ then return true;

$i := i+1;$

end;

return false;

- Example: Take text=abbacc

i=1

j=0 text[1]≠text[2]

i=2

j=0 text[2]=text[3]

j=1 text[1]=text[4]

j=2=i return true

The prefix palindrome is abba

i=3

j=0 text[3]≠text[4]

i=4

Finding palindromes in linear time

- The string a^m contains $\Theta(m^2)$ palindromes

- Thus, to obtain a linear time algorithm for finding palindromes, we need a suitably compact, implicit description for palindromes.

Finding palindromes in linear time

- We say that a palindrome u in a word w is maximal if the characters that precede and respectively follow u in w are different.
  - Example: abba is a maximal even palindrome in the word aabbab since it is preceeded by letter a and followed by letter b

- Then, we represent maximal palindromes by the pair of their center position in w and their length.
  - Example (cont.): the previous palindrome abba can be represented as the pair (3,2)
  - Clearly, any non-maximal palindrome is contained in a maximal one centered at the same position, and is implicitly represented by the latter.

Brute-force algorithm for finding even prefix-palindromes

- Let us denote by

Rad[i]=max{0≤j≤i | text[i-j+1..i]=text[i+1..i+j]},

i.e., the radius of a palindrome centered at value i.

- The brute-force algorithm computes these values but does use them.
  - The information is wasted since at the next iteration, the value of the index j is reset to 0

- To improve the time complexity of the brute-force algorithm we will also use this information.
  - We compute and then store the matrix Rad for further use

Computing the matrix Rad

- Since Rad[i]=max{0≤j≤i | text[i-j+1..i]=text[i+1..i+j]},

the computation of prefix-palindromes reduces to the computation of this matrix.

  - if we have an even palindrome as a prefix, i.e., text[1..2i] is an even palindrome, then we must have Rad[i]=i, i.e., text[1..i]=text[i+1..2i]

Computing the matrix Rad

- Once we already computed some values Rad[1], Rad[2], ..., Rad[i], the next combinatorial result gives us a way to compute also other entries of this matrix without comparing any symbols.

- Lemma: If the integer k is such that $1 \leq k \leq Rad[i]$ and $Rad[i-k] \neq Rad[i]-k$, then $Rad[i+k]=\min(Rad[i-k], Rad[i]-k)$

Linear time algorithm for finding even prefix-palindrome

- We are now ready to write a linear time algorithm for finding prefix palindromes
  - It was given by Manacher in 1975
- Actually it computes the shortest even prefix-palindrome.
- However, the algorithm can be easily modified to compute the radii of all maximal palindromes of a given word.

Linear time algorithm for finding even prefix-palindrome

- n=|text|
- enp represents the end pointer of the palindrome, i.e., the cursor
- mdp represents the center of a potential palindrome
- bp, i.e., the beginning pointer, is the position of the letter which is the mirror image of the current letter

- The procedure PAL returns
  - the index immediately after the shortest even prefix palindrome
  - or 0 if there is no even prefix palindrome

```
procedure PAL(text, n)
  enp := 1; PAL := 0;
  L1: enp := enp + 1; if enp = n + 1 then GOTO DONE;
  mdp := bp := enp - 1; count := 0;
  L2: while text[enp] = text[bp] do begin
      count := count + 1; enp := enp + 1; bp := bp - 1;
      if bp = 0 then begin
          pal := enp; GOTO DONE;
      end;
      if enp = n + 1 then GOTO DONE;
  end;
  Rad[mdp] := count;
  for k := 1 until count do
      if (mdp ≠ k) and (count - k ≥ 0) and (Rad[mdp - k] ≠ count - k) then
          Rad[mdp + k] := min(count - k, Rad[mdp - k])
      else begin
          mdp := mdp + k; count := count - k; bp := mdp - count; GOTO L2;
      end;
  GOTO L1;
  DONE;
  END.
```

Linear time algorithm for finding prefix-palindromes

- In each stage of the algorithm we also update the tableRad[mdp+k] for all consecutive k=1,2,... s.t. Rad[mdp-k]≠Rad[mdp]-k

- If the last such value is k', then the next value we consider is mdp=mdp+k'
  - This is similar with the shifts in the pattern-matching algorithms

```
procedure PAL(text, n)
    enp := 1; PAL := 0;
    L1: enp := enp + 1; if enp = n + 1 then GOTO DONE;
    mdp := bp := enp - 1; count := 0;
    L2: while text[enp] = text[bp] do begin
        count := count + 1; enp := enp + 1; bp := bp - 1;
        if bp = 0 then begin
            pal := enp; GOTO DONE;
        end;
        if enp = n + 1 then GOTO DONE;
    end;
    Rad[mdp] := count;
    for k := 1 until count do
        if (mdp ≠ k) and (count - k ≥ 0) and (Rad[mdp - k] ≠ count - k) then
            Rad[mdp + k] := min(count - k, Rad[mdp - k])
        else begin
            mdp := mdp + k; count := count - k; bp := mdp - count; GOTO L2;
        end;
    GOTO L1;
    DONE;
    END.
```

Linear time algorithm for finding prefix-palindromes

- In order to obtain the radii of all palindromes in a given text, we just need to eliminate the code

```
if bp = 0 then begin
        pal := enp; GOTO DONE;
    end;
```

```
procedure PAL(text, n)
    enp := 1; PAL := 0;
    L1: enp := enp + 1; if enp = n + 1 then GOTO DONE;
    mdp := bp := enp - 1; count := 0;
    L2: while text[enp] = text[bp] do begin
        count := count + 1;  enp := enp + 1;  bp := bp - 1;
        if enp = n + 1 then begin
            Rad[mdp] := count; GOTO DONE;
        end;
    end;
    Rad[mdp] := count;
    for k := 1 until count do
        if (mdp ≠ k) and (count - k ≥ 0) and (Rad[mdp - k] ≠ count - k) then
            Rad[mdp + k] := min(count - k, Rad[mdp - k])
        else begin
            mdp := mdp + k; count := count - k; bp := mdp - count; GOTO L2;
        end;
    GOTO L1;
    DONE;
    END.
```

Finding palindromes

- This algorithm can be also easily modified such that
  - It computes all odd prefix-palindromes
  - It computes the longest prefix palindrome
  - It computes the longest palindrome occurring in the text.

Composition of symmetric words

- Let P^* denote the set of words that are compositions of even palindromes

- Let PAL^* denote the set of words that are compositions of any type of palindromes (even or odd)

- Convention: One letter words are not considered to be palindromes.
  - Their symmetry is trivial

- Question: Given a word w, can we decide whether it is an even palstar (i.e., $w \in P^*$), or a palstar (i.e., $w \in PAL^*$)?

Even palstars

- Define the function

$$first(text) = \begin{cases} o, & \text{if the text does not have} \\ & \text{an even prefix-palindrome} \\ \min\{i \mid text[1..i] \text{ is an even palindrome}\}, & \text{otherwise} \end{cases}$$

- The following algorithm tests even palstars in a natural way:
  - Find the first even prefix-palindrome and cuts it from the text
  - Repeats this process as many times as possible
  - If in the end we remain with the empty word, then the initial text was an even palstar

Even palstars

- Theorem: Even palstars can be tested in linear time.

Proof:

- The function first can be computed by a simple modification of the algorithm Manacher

- It is obvious that the complexity is linear

- Is the algorithm PSTAR correct?

- The algorithm always chooses the shortest prefix even palindrome and then cuts it from the text (a Greedy approach).

```
algorithm PSTAR(text)
    s := 0;
    while s < n do begin
        if first(text[s+1..n]) = 0 then
            return false;
        s := s + first(text[s+1..n]);
    end;
    return true;
```

Even Palstars

Proof (cont.):

- It can be proved that if the text is an even palstar then we can always decompose it using the Greedy approach of this algorithm

- Define the function

parse(text)=min{s | text[1..s]∈P and text[s+1..n]∈P*}

- It can be proved that if text is a non-empty palstar, then parse(text)=first(text).

- This means that the algorithm PSTAR is correct

```
algorithm PSTAR(text)
    s := 0;
    while s < n do begin
        if first(text[s+1..n]) = 0 then
            return false;
        s := s + first(text[s+1..n]);
    end;
    return true;
```

Greedy approach (Bonus)

- A Greedy algorithm is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

- Example: Consider the travelling salesman problem:
  - given a list of cities, their pairwise distances and an initial city, the task is to find a shortest possible tour that visits each city exactly once.
  - If we apply this Greedy strategy to the traveling salesman problem we obtain the following algorithm: "At each stage visit the unvisited city nearest to the current city".

Palstars

- Let us try to extend the previous result for all palstars, i.e., compositions of both types of palindromes (odd and even)

- Define the following functions:

$$first1(text) = \begin{cases} o, & \text{if the text does not} \\ & \text{have a prefix-palindrome} \\ \min\{i \mid text[1..i] \text{ is a palindrome}\}, & \text{otherwise} \end{cases}$$

$$parse1(text) = \min\{s \mid text[1..s] \in PAL \text{ and } text[s+1..n] \in PAL^*\}$$

- Unfortunately, for palstars the equation parse1(text)=first1(text) is not always true

Palstars

- Example:
  - If text=bbabb, then parse1(bbabb)=5 and first1(bbabb)=2
  - If text=abbabba, then parse1(abbabba)=7, first1(abbabba)=4
  - If text=aabab, then parse1(aabab)=first1(aabab)=2

Palstars

- Example:
  - If text=bbabb, then parse1(bbabb)=5 and first1(bbabb)=2
    - parse1(text)=2 first1(text)+1
  - If text=abbabba, then parse1(abbabba)=7, first1(abbabba)=4
    - parse1(text)=2 first1(text)-1
  - If text=aabab, then parse1(aabab)=first1(aabab)=2
    - parse1(text)=first1(text)

- Lemma: Given a palstar word w, we have parse1(w)$\in${first1(w), 2 first1(w)-1, 2first1(w)+1}.

Palstars

- First, we define the following 2 tables:

$$F[i] = first1(text[i+1..n])$$

$$PAL[i] = \begin{cases} 1, & \text{if } text[i+1..n] \text{ is a palindrome} \\ 0, & \text{otherwise} \end{cases}$$

- Now we can write the algorithm testing whether a given text is palstar.

```
Algorithm PALSTAR(text): Boolean;
  palstar[n] := true;
  for i := n-1 down to 0 do begin
    f := F[i];
    if f = 0 then palstar[i] := false
    else if PAL[i] then palstar[i] := true
      else palstar[i] := (palstar[i + f] or
                          palstar[i + 2f - 1] or
                          palstar[i + 2f + 1])
  end;
  return palstar[0];
```

Palstars

- Theorem: Palstar membership can be tested in linear time.

Proof:

- Once we have the tables F and PAL, the algorithm PALSTAR detects palstars in linear time.

- Moreover, using the table of radii of all (even or odd) palindromes, it can be proved that both tables F and PAL can be constructed in linear time.

```
Algorithm PALSTAR(text): Boolean;
    palstar[n] := true;
    for i := n − 1 down to 0 do begin
        f := F[i];
        if f = 0 then palstar[i] := false
        else if PAL[i] then palstar[i] := true
            else palstar[i] := (palstar[i + f] or
                                palstar[i + 2f − 1] or
                                palstar[i + 2f + 1])
    end;
    return palstar[0];
```