Wyszukiwanie wzorca w tekście

algorytm Sunday'a (Quick Search algorithm)

Piotr Beling

Uniwersytet Łódzki

2017
(ostatnia aktualizacja: 2019)

Problem wyszukiwania wzroca

- Niech T (dalej nazywany tekstem) i W (dalej nazywany wzorcem) będą ciągami znaków (wyrazami nad pewnym alfabetem $\mathbb{A}$).
- Założenie: Łańcuchy indeksujemy od 0, np. poprawnymi indeksami T są: $0, 1, \ldots, |T| - 1$, gdzie $|T|$ oznacza długość słowa T.
- Powiemy że W występuje w (pasuje do) T na pozycji p (albo że W jest fragmentem T występującym na pozycji p) jeśli: $W[i] = T[p + i]$ dla wszystkich $i = 0, 1, \ldots, |W| - 1$.
- Problem: sprawdzić czy W występuje w tekście T i, jeśli tak, to na której pozycji.
- Czasami żąda się znalezienia wszystkich wystąpień W w T (taki wariant będziemy rozpatrywali dalej), czasami zaś tylko pierwszego (o najmniejszym indeksie).

Problem wyszukiwania wzroca

- Niech T (dalej nazywany tekstem) i W (dalej nazywany wzorcem) będą ciągami znaków (wyrazami nad pewnym alfabetem $\mathbb{A}$).
- Założenie: Łańcuchy indeksujemy od 0, np. poprawnymi indeksami T są: $0, 1, \ldots, |T| - 1$, gdzie $|T|$ oznacza długość słowa T.
- Powiemy że W występuje w (pasuje do) T na pozycji p (albo że W jest fragmentem T występującym na pozycji p) jeśli: $W[i] = T[p + i]$ dla wszystkich $i = 0, 1, \ldots, |W| - 1$.
- Problem: sprawdzić czy W występuje w tekście T i, jeśli tak, to na której pozycji.
- Czasami żąda się znalezienia wszystkich wystąpień W w T (taki wariant będziemy rozpatrywali dalej), czasami zaś tylko pierwszego (o najmniejszym indeksie).

Problem wyszukiwania wzroca

- Niech T (dalej nazywany tekstem) i W (dalej nazywany wzorcem) będą ciągami znaków (wyrazami nad pewnym alfabetem $\mathbb{A}$).
- Założenie: Łańcuchy indeksujemy od 0, np. poprawnymi indeksami T są: $0, 1, \ldots, |T| - 1$, gdzie $|T|$ oznacza długość słowa T.
- Powiemy że W występuje w (pasuje do) T na pozycji p (albo że W jest fragmentem T występującym na pozycji p) jeśli: $W[i] = T[p + i]$ dla wszystkich $i = 0, 1, \ldots, |W| - 1$.
- Problem: sprawdzić czy W występuje w tekście T i, jeśli tak, to na której pozycji.
- Czasami żąda się znalezienia wszystkich wystąpień W w T (taki wariant będziemy rozpatrywali dalej), czasami zaś tylko pierwszego (o najmniejszym indeksie).

Problem wyszukiwania wzroca

- Niech T (dalej nazywany tekstem) i W (dalej nazywany wzorcem) będą ciągami znaków (wyrazami nad pewnym alfabetem $\mathbb{A}$).
- Założenie: Łańcuchy indeksujemy od 0, np. poprawnymi indeksami T są: $0, 1, \dots, |T| - 1$, gdzie $|T|$ oznacza długość słowa T.
- Powiemy że W występuje w (pasuje do) T na pozycji p (albo że W jest fragmentem T występującym na pozycji p) jeśli: $W[i] = T[p + i]$ dla wszystkich $i = 0, 1, \dots, |W| - 1$.
- Problem: sprawdzić czy W występuje w tekście T i, jeśli tak, to na której pozycji.
- Czasami żąda się znalezienia wszystkich wystąpień W w T (taki wariant będziemy rozpatrywali dalej), czasami zaś tylko pierwszego (o najmniejszym indeksie).

Problem wyszukiwania wzroca

- Niech T (dalej nazywany tekstem) i W (dalej nazywany wzorcem) będą ciągami znaków (wyrazami nad pewnym alfabetem $\mathbb{A}$).
- Założenie: Łańcuchy indeksujemy od 0, np. poprawnymi indeksami T są: $0, 1, \ldots, |T| - 1$, gdzie $|T|$ oznacza długość słowa T.
- Powiemy że W występuje w (pasuje do) T na pozycji p (albo że W jest fragmentem T występującym na pozycji p) jeśli: $W[i] = T[p + i]$ dla wszystkich $i = 0, 1, \ldots, |W| - 1$.
- Problem: sprawdzić czy W występuje w tekście T i, jeśli tak, to na której pozycji.
- Czasami żąda się znalezienia wszystkich wystąpień W w T (taki wariant będziemy rozpatrywali dalej), czasami zaś tylko pierwszego (o najmniejszym indeksie).

matchesAt

Następująca funkcja sprawdza czy W występuje w T na pozycji p:

```
fun matchesAt(T, p, W):
  for i ← 0, 1, ..., |W|-1:
    if W[i] ≠ T[p+i]: return False
  return True
```

By nie przekroczyć zakresu indeksów:

- p musi być liczbą nieujemną,
- równocześnie musi spełniać $p + |W| - 1 < |T|$, by dla maksymalnej wartości $i = |W| - 1$ liczba $p + i$ była poprawnym indeksem T (t.j. $p + i < |T|$).

Podsumowując, p musi spełniać: $0 \leq p \leq |T| - |W|$.

Złożność obliczniowa:

- czasowa: od 1 do $|W|$ porównań znaków,
- pamięciowa: $O(1)$.

matchesAt

Następująca funkcja sprawdza czy W występuje w T na pozycji p:

```
fun matchesAt(T, p, W):
  for i ← 0, 1, ..., |W|-1:
    if W[i] ≠ T[p+i]: return False
  return True
```

By nie przekroczyć zakresu indeksów:

- p musi być liczbą nieujemną,
- równocześnie musi spełniać $p + |W| - 1 < |T|$, by dla maksymalnej wartości $i = |W| - 1$ liczba $p + i$ była poprawnym indeksem T (t.j. $p + i < |T|$).

Podsumowując, p musi spełniać: $0 \leq p \leq |T| - |W|$.

Złożność oblicznioowa:

- czasowa: od 1 do $|W|$ porównań znaków,
- pamięciowa: $O(1)$.

matchesAt

Następująca funkcja sprawdza czy W występuje w T na pozycji p:

```
fun matchesAt(T, p, W):
  for i ← 0, 1, ..., |W|-1:
    if W[i] ≠ T[p+i]: return False
  return True
```

By nie przekroczyć zakresu indeksów:

- p musi być liczbą nieujemną,
- równocześnie musi spełniać $p + |W| - 1 < |T|$, by dla maksymalnej wartości $i = |W| - 1$ liczba $p + i$ była poprawnym indeksem T (t.j. $p + i < |T|$).

Podsumowując, p musi spełniać: $0 \leq p \leq |T| - |W|$.

Złożność oblicznioowa:

- czasowa: od 1 do $|W|$ porównań znaków,
- pamięciowa: $O(1)$.

matchesAt

Następująca funkcja sprawdza czy W występuje w T na pozycji p:

```
fun matchesAt(T, p, W):
  for i ← 0, 1, ..., |W|-1:
    if W[i] ≠ T[p+i]: return False
  return True
```

By nie przekroczyć zakresu indeksów:

- p musi być liczbą nieujemną,
- równocześnie musi spełniać $p + |W| - 1 < |T|$, by dla maksymalnej wartości $i = |W| - 1$ liczba $p + i$ była poprawnym indeksem T (t.j. $p + i < |T|$).

Podsumowując, p musi spełniać: $0 \leq p \leq |T| - |W|$.

Złożność oblicznioowa:

- czasowa: od 1 do $|W|$ porównań znaków,
- pamięciowa: $O(1)$.

matchesAt

Następująca funkcja sprawdza czy W występuje w T na pozycji p:

```
fun matchesAt(T, p, W):
  for i ← 0, 1, ..., |W|-1:
    if W[i] ≠ T[p+i]: return False
  return True
```

By nie przekroczyć zakresu indeksów:

- p musi być liczbą nieujemną,
- równocześnie musi spełniać $p + |W| - 1 < |T|$, by dla maksymalnej wartości $i = |W| - 1$ liczba $p + i$ była poprawnym indeksem T (t.j. $p + i < |T|$).

Podsumowując, p musi spełniać: $0 \leq p \leq |T| - |W|$.

Złożność oblicnziowa:

- czasowa: od 1 do $|W|$ porównań znaków,
- pamięciowa: $O(1)$.

Algorytm naiwny

Algorytm naiwny sprawdza czy wzorzec W występuje w tekście T dla (wszystkich) kolejnych pozycji $p = 0, 1, \ldots, |T| - |W|$.

```
for p ← 0, 1, ..., |T|-|W|:
  if matchesAt(T, p, W): report(p)
```

(przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Algorytm wywołuje `matchesAt` $|T| - |W| + 1$ razy, zaś każde z tych wywołań wykonuje od 1 do $|W|$ porównań znaków. Stąd jego złożoność czasowa mierzona liczbą porównań znaków:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$,
- optymistyczna: $|T| - |W| + 1 = O(|T|)$.

Złożoność pamięciowa: $O(1)$.

Algorytm naiwny

Algorytm naiwny sprawdza czy wzorzec W występuje w tekście T dla (wszystkich) kolejnych pozycji $p = 0, 1, \ldots, |T| - |W|$.

```
for p ← 0, 1, ..., |T|-|W|:
  if matchesAt(T, p, W): report(p)
```

(przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Algorytm wywołuje `matchesAt` $|T| - |W| + 1$ razy, zaś każde z tych wywołań wykonuje od 1 do $|W|$ porównań znaków.
Stąd jego złożoność czasowa mierzona liczbą porównań znaków:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$,
- optymistyczna: $|T| - |W| + 1 = O(|T|)$.

Złożoność pamięciowa: $O(1)$.

Algorytm naiwny

Algorytm naiwny sprawdza czy wzorzec W występuje w tekście T dla (wszystkich) kolejnych pozycji $p = 0, 1, \ldots, |T| - |W|$.

```
for p ← 0, 1, ..., |T|-|W|:
   if matchesAt(T, p, W): report(p)
```

(przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Algorytm wywołuje `matchesAt` $|T| - |W| + 1$ razy, zaś każde z tych wywołań wykonuje od 1 do $|W|$ porównań znaków.

Stąd jego złożoność czasowa mierzona liczbą porównań znaków:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$,
- optymistyczna: $|T| - |W| + 1 = O(|T|)$.

Złożoność pamięciowa: $O(1)$.

Algorytm naiwny

Algorytm naiwny sprawdza czy wzorzec W występuje w tekście T dla (wszystkich) kolejnych pozycji $p = 0, 1, \ldots, |T| - |W|$.

```
for p ← 0, 1, ..., |T|-|W|:
  if matchesAt(T, p, W): report(p)
```

(przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Algorytm wywołuje `matchesAt` $|T| - |W| + 1$ razy, zaś każde z tych wywołań wykonuje od 1 do $|W|$ porównań znaków.
Stąd jego złożoność czasowa mierzona liczbą porównań znaków:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$,
- optymistyczna: $|T| - |W| + 1 = O(|T|)$.

Złożoność pamięciowa: $O(1)$.

Algorytm naiwny

Algorytm naiwny sprawdza czy wzorzec W występuje w tekście T dla (wszystkich) kolejnych pozycji $p = 0, 1, \ldots, |T| - |W|$.

```
for p ← 0, 1, ..., |T|-|W|:
    if matchesAt(T, p, W): report(p)
```

(przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Algorytm wywołuje `matchesAt` $|T| - |W| + 1$ razy, zaś każde z tych wywołań wykonuje od 1 do $|W|$ porównań znaków.
Stąd jego złożoność czasowa mierzona liczbą porównań znaków:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$,
- optymistyczna: $|T| - |W| + 1 = O(|T|)$.

Złożoność pamięciowa: $O(1)$.

Algorytm naiwny

Algorytm naiwny sprawdza czy wzorzec W występuje w tekście T dla (wszystkich) kolejnych pozycji $p = 0, 1, \ldots, |T| - |W|$.

```
for p ← 0, 1, ..., |T|-|W|:
  if matchesAt(T, p, W): report(p)
```

(przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Algorytm wywołuje `matchesAt` $|T| - |W| + 1$ razy, zaś każde z tych wywołań wykonuje od 1 do $|W|$ porównań znaków. Stąd jego złożoność czasowa mierzona liczbą porównań znaków:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$,
- optymistyczna: $|T| - |W| + 1 = O(|T|)$.

Złożoność pamięciowa: $O(1)$.

Algorytm naiwny - przykład

$\downarrow^{p=0}$

T `ACBCDABABBDB`

W `ABA`

`matchesAt(T, 0, W)` sprawdza czy wzorzec W występuje w tekście T na pozycji $p = 0$:

- pierwsza litera się zgadza, $A = A$,
- druga litera się nie zgadza, $C \neq B$, więc `matchesAt(T, 0, W)` zwraca `False`.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 0

Algorytm naiwny - przykład

$\downarrow^{p=0}$

T `ACBCDABABBDB`

W `ABA`

`matchesAt(T, 0, W)` sprawdza czy wzorzec W występuje w tekście T na pozycji $p = 0$:

- pierwsza litera się zgadza, $A = A$,
- druga litera się nie zgadza, $C \neq B$, więc `matchesAt(T, 0, W)` zwraca `False`.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 1

Algorytm naiwny - przykład

$\downarrow^{p=0}$

T `ACBCDABABBDB`

W `ABA`

`matchesAt(T, 0, W)` sprawdza czy wzorzec W występuje w tekście T na pozycji $p = 0$:

- pierwsza litera się zgadza, $A = A$,
- druga litera się nie zgadza, $C \neq B$, więc `matchesAt(T, 0, W)` zwraca `False`.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 2

Algorytm naiwny - przykład

$\downarrow^{p=0+1=1}$

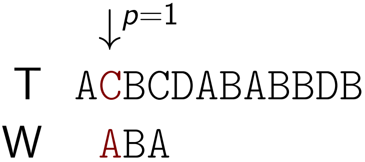
T ACBCDABABBDB

W ABA

Następnie p jest zwiększane (wzorzec jest przesuwany) o 1 pozycję.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 2

Algorytm naiwny - przykład



$\downarrow^{p=1}$

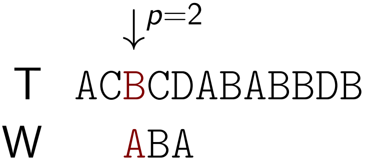
T ACBCDABABBDB

W ABA

`matchesAt(T, 1, W)` zwraca `False`
(W nie występuje w T na pozycji $p = 1$),
ponieważ $C \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 3

Algorytm naiwny - przykład



$\downarrow^{p=2}$

T `ACBCDABABBDB`

W `ABA`

Dla kolejnej pozycji $p = 2$,
`matchesAt(T, 2, W)` też zwraca `False`,
ponieważ $B \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 4

Algorytm naiwny - przykład

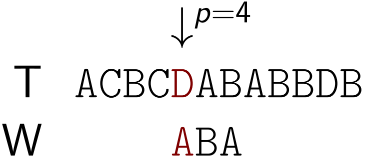
$\downarrow^{p=3}$

```
T  ACBCDABABBDB
W     ABA
```

Podobnie `matchesAt(T, 3, W)` zwraca `False`, ponieważ $C \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 5

Algorytm naiwny - przykład



$\downarrow^{p=4}$

T ACBCDABABBDB

W ABA

Także `matchesAt(T, 4, W)` zwraca `False`,
ponieważ $D \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 6

Algorytm naiwny - przykład

$\downarrow^{p=5}$

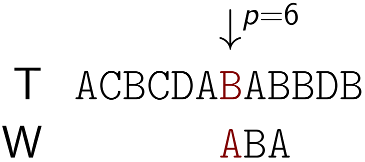
T ACBCDABABBDB

W ABA

`matchesAt(T, 5, W)` zwraca `True`
(W występuje w T na pozycji $p = 5$).

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 9

Algorytm naiwny - przykład



$\downarrow^{p=6}$

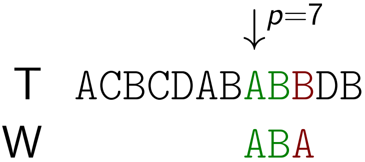
T ACBCDABABBDB

W ABA

Algorytm szuka kolejnych wystąpień wzorca w tekście...
`matchesAt(T, 6, W)` zwraca `False`,
ponieważ $B \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 10

Algorytm naiwny - przykład



$\downarrow^{p=7}$

T ACBCDABABBDB

W ABA

`matchesAt(T, 7, W)` zwraca `False`,
ponieważ $B \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 13

Algorytm naiwny - przykład

```
           ↓p=8
T  ACBCDABABBDB
W          ABA
```

Podobnie `matchesAt(T, 8, W)` zwraca `False`, ponieważ $B \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 14

Algorytm naiwny - przykład

$\downarrow^{p=9}$

T ACBCDABABBDB

W ABA

I także `matchesAt(T, 9, W)` zwraca `False`,
ponieważ $B \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 15

Algorytm naiwny - przykład

$\downarrow^{p=10}$

T ACBCDABABBDB

W ABA

Algorytm kończy pracę, gdyż dla ewentualnej, kolejnej wartości $p = 10$:

$$\underbrace{p}_{=10} > \underbrace{|T| - |W|}_{=9}.$$

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 15

Algorytm Sunday'a

- Algorytm Sunday'a różni się od naiwnego tym, że po każdym wywołaniu `matchesAt` próbuje zwiększać p o więcej niż 1.
- W tym celu analizuje znak stojący bezpośrednio za fragmentem T do którego próbowano dopasować wzorzec, t.j. znak o indeksie $p + |W|$. Oznaczmy $z = T[p + |W|]$.
- Jeśli z nie występuje we wzorcu W, to p można zwiększyć aż o $|W| + 1$ (mniejsze zwiększenie p doprowadziłoby do sytuacji, że w porównywanym fragmencie T znalazłby się znak z niewystępujący w W, więc `matchesAt` musiałoby zwrócić `False`).
- Jeśli z występuje w W, to p jest zwiększane w taki sposób, by ostatnie wystąpienie z w W było porównywane przez `matchesAt` z pozycją $p + |W|$ (literą z) w T.

Algorytm Sunday'a

- Algorytm Sunday'a różni się od naiwnego tym, że po każdym wywołaniu `matchesAt` próbuje zwiększać p o więcej niż 1.
- W tym celu analizuje znak stojący bezpośrednio za fragmentem T do którego próbowano dopasować wzorzec, t.j. znak o indeksie $p + |W|$. Oznaczmy $z = T[p + |W|]$.
- Jeśli z nie występuje we wzorcu W, to p można zwiększyć aż o $|W| + 1$ (mniejsze zwiększenie p doprowadziłoby do sytuacji, że w porównywanym fragmencie T znalazłby się znak z niewystępujący w W, więc `matchesAt` musiałoby zwrócić `False`).
- Jeśli z występuje w W, to p jest zwiększane w taki sposób, by ostatnie wystąpienie z w W było porównywane przez `matchesAt` z pozycją $p + |W|$ (literą z) w T.

Algorytm Sunday'a

- Algorytm Sunday'a różni się od naiwnego tym, że po każdym wywołaniu `matchesAt` próbuje zwiększać p o więcej niż 1.
- W tym celu analizuje znak stojący bezpośrednio za fragmentem T do którego próbowano dopasować wzorzec, t.j. znak o indeksie $p + |W|$. Oznaczmy $z = T[p + |W|]$.
- Jeśli z nie występuje we wzorcu W, to p można zwiększyć aż o $|W| + 1$ (mniejsze zwiększenie p doprowadziłoby do sytuacji, że w porównywanym fragmencie T znalazłby się znak z niewystępujący w W, więc `matchesAt` musiałoby zwrócić `False`).
- Jeśli z występuje w W, to p jest zwiększane w taki sposób, by ostatnie wystąpienie z w W było porównywane przez `matchesAt` z pozycją $p + |W|$ (literą z) w T.

Algorytm Sunday'a

- Algorytm Sunday'a różni się od naiwnego tym, że po każdym wywołaniu `matchesAt` próbuje zwiększać p o więcej niż 1.
- W tym celu analizuje znak stojący bezpośrednio za fragmentem T do którego próbowano dopasować wzorzec, t.j. znak o indeksie $p + |W|$. Oznaczmy $z = T[p + |W|]$.
- Jeśli z nie występuje we wzorcu W, to p można zwiększyć aż o $|W| + 1$ (mniejsze zwiększenie p doprowadziłoby do sytuacji, że w porównywanym fragmencie T znalazłby się znak z niewystępujący w W, więc `matchesAt` musiałoby zwrócić `False`).
- Jeśli z występuje w W, to p jest zwiększane w taki sposób, by ostatnie wystąpienie z w W było porównywane przez `matchesAt` z pozycją $p + |W|$ (literą z) w T.

Algorytm Sunday'a – przykład

$\downarrow^{p=0}$

T `ACBCDABABBDB`

W `ABA`

`matchesAt(T, 0, W)` zwraca `False`
(W nie występuje w T na pozycji $p = 0$),
ponieważ $C \neq B$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 2

Algorytm Sunday'a – przykład

$\downarrow^{p=0}$

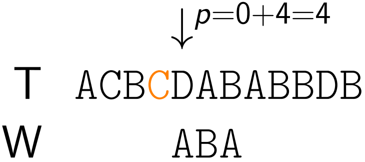
T `ACBCDABABBDB`

W `ABA`

Ponieważ wzorzec W nie zawiera litery C,
to p można zwiększyć (wzorzec można przesunąć)
o $|W| + 1 = 4$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 2

Algorytm Sunday'a – przykład



$\downarrow p$=0+4=4

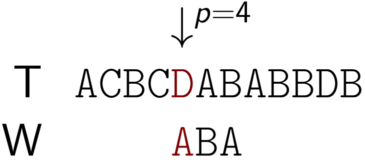
T ACBCDABABBDB

W ABA

Ponieważ wzorzec W nie zawiera litery C,
to p można zwiększyć (wzorzec można przesunąć)
o $|W| + 1 = 4$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 2

Algorytm Sunday'a – przykład



`matchesAt(T, 4, W)` zwraca `False`
(W nie występuje w T na pozycji $p = 4$),
ponieważ $D \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 3

Algorytm Sunday'a – przykład

$\downarrow^{p=4}$

T `ACBCDABABBDB`

W `    ABA`

Wzorzec W zawiera literę A, więc p jest zwiększane (wzorzec jest przesuwany) o 1, bo wtedy ostatnie wystąpienie A w W pokryje się z rozpatrywanym A w tekście T.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 3

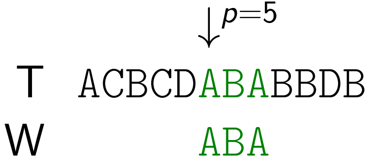
Algorytm Sunday’a – przykład

```
        ↓p=4+1=5
T  ACBCDABABBDB
W       ABA
```

Wzorzec *W* zawiera literę *A*, więc *p* jest zwiększane (wzorzec jest przesuwany) o 1, bo wtedy ostatnie wystąpienie *A* w *W* pokryje się z rozpatrywanym *A* w tekście *T*.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 3

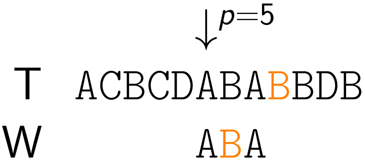
Algorytm Sunday'a – przykład



`matchesAt(T, 5, W)` zwraca `True`
(W występuje w T na pozycji $p = 5$).

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 6

Algorytm Sunday'a – przykład



$\downarrow^{p=5}$

T ACBCDABABBDB

W ABA

Algorytm szuka kolejnych wystąpień wzorca w tekście...
Wzorzec W zawiera literę B, więc p jest zwiększane (wzorzec jest przesuwany) o 2, bo wtedy B zawarte w W pokryje się z rozpatrywanym B w tekście T.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 6

Algorytm Sunday'a – przykład

↓ $p=5+2=7$

T `ACBCDABABBDB`

W `       ABA`

Algorytm szuka kolejnych wystąpień wzorca w tekście...
Wzorzec W zawiera literę B, więc p jest zwiększane (wzorzec jest przesuwany) o 2, bo wtedy B zawarte w W pokryje się z rozpatrywanym B w tekście T.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 6

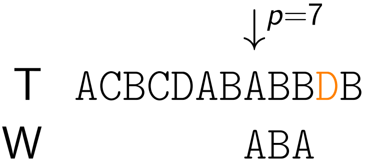
Algorytm Sunday’a – przykład

```
          ↓p=7
T   ACBCDABABBDB
W          ABA
```

`matchesAt(T, 7, W)` zwraca `False`
(W nie występuje w T na pozycji $p = 7$),
ponieważ $B \neq A$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 9

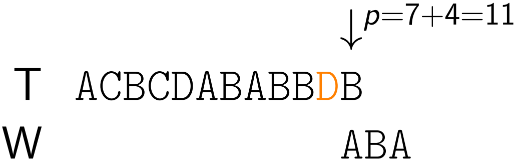
Algorytm Sunday'a – przykład



Ponieważ wzorzec W nie zawiera litery D,
to p można zwiększyć (wzorzec można przesunąć)
o $|W| + 1 = 4$.

$|T| = 12$, $|W| = 3$,
liczba wykonanych porównań znaków: 9

Algorytm Sunday'a – przykład



$\downarrow p=7+4=11$

T ACBCDABABBDB

W ABA

Ponieważ wzorzec W nie zawiera litery D,
to p można zwiększyć (wzorzec można przesunąć)
o $|W|+1=4$.
Algorytm kończy pracę gdyż $\underbrace{p}_{=11} > \underbrace{|T|-|W|}_{=9}$.

$|T|=12$, $|W|=3$,
liczba wykonanych porównań znaków: 9

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

  $$|W| - \mathrm{lastp}(z),$$

  gdzie $\mathrm{lastp}(z)$ to:

  - indeks ostatniego wystąpieniu z we wzorcu W,
  - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

  $$|W| - \text{lastp}(z),$$

  gdzie $\text{lastp}(z)$ to:
  - indeks ostatniego wystąpieniu z we wzorcu W,
  - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

$$|W| - \mathrm{lastp}(z),$$

gdzie $\mathrm{lastp}(z)$ to:

  - indeks ostatniego wystąpieniu z we wzorcu W,
  - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

  $$|W| - \mathrm{lastp}(z),$$

  gdzie $\mathrm{lastp}(z)$ to:

  - indeks ostatniego wystąpieniu z we wzorcu W,
  - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

$$|W| - \mathrm{lastp}(z),$$

gdzie $\mathrm{lastp}(z)$ to:

  - indeks ostatniego wystąpieniu z we wzorcu W,
  - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

  $$|W| - \mathrm{lastp}(z),$$

  gdzie $\mathrm{lastp}(z)$ to:
  - indeks ostatniego wystąpieniu z we wzorcu W,
  - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

  $$|W| - \text{lastp}(z),$$

  gdzie $\text{lastp}(z)$ to:
    - indeks ostatniego wystąpieniu z we wzorcu W,
    - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – przesuwanie wzorca

- Jak poprzednio, przez $z = T[p + |W|]$ oznaczmy znak stojący w T bezpośrednio za fragmentem do którego próbowano dopasować wzorzec.
- Jeśli z jest ostatnim znakiem we wzorcu W (t.j. ma w nim indeks $|W| - 1$), to p można zwiększyć o 1,
- jeśli przedostatnim (o indeksie $|W| - 2$), to o 2, itd.
- Jeśli z występuje w W jedynie pod indeksem 0, to p można zwiększyć o $|W|$.
- Jeśli z nie występuje w W, to p można zwiększyć o $|W| + 1$.
- Podsumowując p można zwiększyć o

  $$|W| - \text{lastp}(z),$$

  gdzie $\text{lastp}(z)$ to:

  - indeks ostatniego wystąpieniu z we wzorcu W,
  - albo -1 jeśli W nie zawiera z.

Algorytm Sunday'a – wstępne przetwarzanie wzorca

- Dla efektywności algorytmu kluczowe jest szybkie (najlepiej w czasie stałym) wyznaczanie wartości funkcji lastp.
- Można tego dokonać tablicując wszystkie jej wartości, przed rozpoczęciem właściwego przeszukiwania:

  ```
  lastp ← { -1, -1, ..., -1 }
  for i ← 0, 1, ..., |W|-1: lastp[W[i]] ← i
  ```

  `lastp` jest tablicą o wielkości alfabetu, indeksowaną jego znakami (np. kodami ASCII). Złożoność czasowa to $|\mathbb{A}| + |W|$ przypisań, gdzie $|\mathbb{A}|$ to rozmiar alfabetu.
- W przypadku dużego alfabetu, można zaimplementować `lastp` za pomocą tablicy haszującej, zawierającej wartości tylko dla znaków występujących w W (dalej założymy, że odczyt kluczy niezawartych w tej tablicy daje wartość -1). Wielkość takiej tablicy będzie proporcjonalna do liczby różnych liter zawartych w W, zaś czas na jej wypełnienie proporcjonalny do $|W|$.

Algorytm Sunday'a – wstępne przetwarzanie wzorca

- Dla efektywności algorytmu kluczowe jest szybkie (najlepiej w czasie stałym) wyznaczanie wartości funkcji lastp.
- Można tego dokonać tablicując wszystkie jej wartości, przed rozpoczęciem właściwego przeszukiwania:

  ```
  lastp ← { -1, -1, ..., -1 }
  for i ← 0, 1, ..., |W|-1: lastp[W[i]] ← i
  ```

  `lastp` jest tablicą o wielkości alfabetu, indeksowaną jego znakami (np. kodami ASCII). Złożoność czasowa to $|\mathbb{A}| + |W|$ przypisań, gdzie $|\mathbb{A}|$ to rozmiar alfabetu.
- W przypadku dużego alfabetu, można zaimplementować `lastp` za pomocą tablicy haszującej, zawierającej wartości tylko dla znaków występujących w W (dalej założymy, że odczyt kluczy niezawartych w tej tablicy daje wartość -1). Wielkość takiej tablicy będzie proporcjonalna do liczby różnych liter zawartych w W, zaś czas na jej wypełnienie proporcjonalny do $|W|$.

Algorytm Sunday'a – wstępne przetwarzanie wzorca

- Dla efektywności algorytmu kluczowe jest szybkie (najlepiej w czasie stałym) wyznaczanie wartości funkcji lastp.
- Można tego dokonać tablicując wszystkie jej wartości, przed rozpoczęciem właściwego przeszukiwania:

  ```
  lastp ← { -1, -1, ..., -1 }
  for i ← 0, 1, ..., |W|-1: lastp[W[i]] ← i
  ```

  `lastp` jest tablicą o wielkości alfabetu, indeksowaną jego znakami (np. kodami ASCII). Złożoność czasowa to $|\mathbb{A}| + |W|$ przypisań, gdzie $|\mathbb{A}|$ to rozmiar alfabetu.
- W przypadku dużego alfabetu, można zaimplementować `lastp` za pomocą tablicy haszującej, zawierającej wartości tylko dla znaków występujących w W (dalej założymy, że odczyt kluczy niezawartych w tej tablicy daje wartość -1). Wielkość takiej tablicy będzie proporcjonalna do liczby różnych liter zawartych w W, zaś czas na jej wypełnienie proporcjonalny do $|W|$.

Algorytm Sunday'a – wstępne przetwarzanie wzorca

- Dla efektywności algorytmu kluczowe jest szybkie (najlepiej w czasie stałym) wyznaczanie wartości funkcji lastp.
- Można tego dokonać tablicując wszystkie jej wartości, przed rozpoczęciem właściwego przeszukiwania:

  ```
  lastp ← { -1, -1, ..., -1 }
  for i ← 0, 1, ..., |W|-1: lastp[W[i]] ← i
  ```

  `lastp` jest tablicą o wielkości alfabetu, indeksowaną jego znakami (np. kodami ASCII). Złożoność czasowa to $|\mathbb{A}| + |W|$ przypisań, gdzie $|\mathbb{A}|$ to rozmiar alfabetu.
- W przypadku dużego alfabetu, można zaimplementować `lastp` za pomocą tablicy haszującej, zawierającej wartości tylko dla znaków występujących w W (dalej założymy, że odczyt kluczy niezawartych w tej tablicy daje wartość -1). Wielkość takiej tablicy będzie proporcjonalna do liczby różnych liter zawartych w W, zaś czas na jej wypełnienie proporcjonalny do $|W|$.

Algorytm Sunday'a – przykład wypełniania tablicy lastp

- Prześledźmy proces wypełniania tablicy `lastp` dla przykładowego wzorca W=ABA i alfabetu $\mathbb{A} = \{A, B, C, D\}$.
- Po wykonaniu `lastp ←{ -1, -1, ..., -1 }` tablica zawiera same `-1`:

indeks:	A	B	C	D
wartość:	-1	-1	-1	-1

.

- Następnie wykonywane są przypisania `lastp[W[i]] ←i` dla kolejnych i wynoszących: $0, 1, 2 = \underbrace{|W|}_{3} - 1$:
- po lastp$[\underbrace{W[0]}_{A}] \leftarrow 0$, lastp=

indeks:	A	B	C	D
wartość:	0	-1	-1	-1

,

- po lastp$[\underbrace{W[1]}_{B}] \leftarrow 1$, lastp=

indeks:	A	B	C	D
wartość:	0	1	-1	-1

,

- po lastp$[\underbrace{W[2]}_{A}] \leftarrow 2$, lastp=

indeks:	A	B	C	D
wartość:	2	1	-1	-1

.

Algorytm Sunday'a – przykład wypełniania tablicy lastp

- Prześledźmy proces wypełniania tablicy `lastp` dla przykładowego wzorca W=ABA i alfabetu $\mathbb{A} = \{A, B, C, D\}$.
- Po wykonaniu `lastp ←{ -1, -1, ..., -1 }` tablica zawiera same −1:

indeks:	A	B	C	D
wartość:	-1	-1	-1	-1

.

- Następnie wykonywane są przypisania `lastp[W[i]] ←i` dla kolejnych i wynoszących: $0, 1, 2 = \underbrace{|W|}_{3} - 1$:
- po $\text{lastp}[\underbrace{W[0]}_{A}] \leftarrow 0$, lastp=

indeks:	A	B	C	D
wartość:	0	-1	-1	-1

,

- po $\text{lastp}[\underbrace{W[1]}_{B}] \leftarrow 1$, lastp=

indeks:	A	B	C	D
wartość:	0	1	-1	-1

,

- po $\text{lastp}[\underbrace{W[2]}_{A}] \leftarrow 2$, lastp=

indeks:	A	B	C	D
wartość:	2	1	-1	-1

.

Algorytm Sunday'a – przykład wypełniania tablicy lastp

- Prześledźmy proces wypełniania tablicy `lastp` dla przykładowego wzorca W=ABA i alfabetu $\mathbb{A} = \{A, B, C, D\}$.
- Po wykonaniu `lastp ←{ -1, -1, ..., -1 }` tablica zawiera same `-1`:

indeks:	A	B	C	D
wartość:	-1	-1	-1	-1

.

- Następnie wykonywane są przypisania `lastp[W[i]] ←i` dla kolejnych i wynoszących: $0, 1, 2 = \underbrace{|W|}_{3} - 1$:

- po lastp$[\underbrace{W[0]}_{A}] \leftarrow 0$, lastp=

indeks:	A	B	C	D
wartość:	0	-1	-1	-1

,

- po lastp$[\underbrace{W[1]}_{B}] \leftarrow 1$, lastp=

indeks:	A	B	C	D
wartość:	0	1	-1	-1

,

- po lastp$[\underbrace{W[2]}_{A}] \leftarrow 2$, lastp=

indeks:	A	B	C	D
wartość:	2	1	-1	-1

.

Algorytm Sunday'a – przykład wypełniania tablicy lastp

- Prześledźmy proces wypełniania tablicy `lastp` dla przykładowego wzorca W=ABA i alfabetu $\mathbb{A} = \{A, B, C, D\}$.
- Po wykonaniu `lastp ←{ -1, -1, ..., -1 }` tablica zawiera same `-1`:

indeks:	A	B	C	D
wartość:	-1	-1	-1	-1

.

- Następnie wykonywane są przypisania `lastp[W[i]] ←i` dla kolejnych i wynoszących: $0, 1, 2 = \underbrace{|W|}_{3} - 1$:
- po lastp$[\underbrace{W[0]}_{A}] \leftarrow 0$, lastp=

indeks:	A	B	C	D
wartość:	0	-1	-1	-1

,

- po lastp$[\underbrace{W[1]}_{B}] \leftarrow 1$, lastp=

indeks:	A	B	C	D
wartość:	0	1	-1	-1

,

- po lastp$[\underbrace{W[2]}_{A}] \leftarrow 2$, lastp=

indeks:	A	B	C	D
wartość:	2	1	-1	-1

.

Algorytm Sunday'a – przykład wypełniania tablicy lastp

- Prześledźmy proces wypełniania tablicy `lastp` dla przykładowego wzorca W=ABA i alfabetu $\mathbb{A} = \{A, B, C, D\}$.
- Po wykonaniu `lastp ←{ -1, -1, ..., -1 }` tablica zawiera same `-1`:

indeks:	A	B	C	D
wartość:	-1	-1	-1	-1

.

- Następnie wykonywane są przypisania `lastp[W[i]] ←i` dla kolejnych i wynoszących: $0, 1, 2 = \underbrace{|W|}_{3} -1$:

- po lastp$[\underbrace{W[0]}_{A}] \leftarrow 0$, lastp=

indeks:	A	B	C	D
wartość:	0	-1	-1	-1

,

- po lastp$[\underbrace{W[1]}_{B}] \leftarrow 1$, lastp=

indeks:	A	B	C	D
wartość:	0	1	-1	-1

,

- po lastp$[\underbrace{W[2]}_{A}] \leftarrow 2$, lastp=

indeks:	A	B	C	D
wartość:	2	1	-1	-1

.

Algorytm Sunday'a – przykład wypełniania tablicy lastp

- Prześledźmy proces wypełniania tablicy `lastp` dla przykładowego wzorca W=ABA i alfabetu $\mathbb{A} = \{A, B, C, D\}$.
- Po wykonaniu `lastp ←{ -1, -1, ..., -1 }` tablica zawiera same `-1`:

indeks:	A	B	C	D
wartość:	-1	-1	-1	-1

.

- Następnie wykonywane są przypisania `lastp[W[i]] ←i` dla kolejnych i wynoszących: $0, 1, 2 = \underbrace{|W|}_{3} - 1$:

- po lastp$[\underbrace{W[0]}_{A}] \leftarrow 0$, lastp=

indeks:	A	B	C	D
wartość:	0	-1	-1	-1

,

- po lastp$[\underbrace{W[1]}_{B}] \leftarrow 1$, lastp=

indeks:	A	B	C	D
wartość:	0	1	-1	-1

,

- po lastp$[\underbrace{W[2]}_{A}] \leftarrow 2$, lastp=

indeks:	A	B	C	D
wartość:	2	1	-1	-1

.

Algorytm Sunday'a – pseudokod

Pseudokod właściwego przeszukiwania (po wypełnieniu `lastp`):

```
p ← 0
while p ≤ |T|-|W|:
    if matchesAt(T, p, W): report(p)
    p ← p + |W|
    if p<|T|: p ← p - lastp[T[p]]
```

(ponownie przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Złożoność czasowa tej części algorytmu:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$ porównań (i pomijalna liczba $|T| - |W| + 1$ odczytów z `lastp`),
- optymistyczna: $\Theta\left(\frac{|T|}{|W|}\right)$ ($\left\lfloor\frac{|T|+1}{|W|+1}\right\rfloor$ porównań i $\left\lfloor\frac{|T|}{|W|+1}\right\rfloor$ odczytów z `lastp`).

Algorytm jest bardzo szybki w praktycznych zastosowaniach.

Algorytm Sunday'a – pseudokod

Pseudokod właściwego przeszukiwania (po wypełnieniu `lastp`):

```
p ← 0
while p ≤ |T|-|W|:
  if matchesAt(T, p, W): report(p)
  p ← p + |W|
  if p<|T|: p ← p - lastp[T[p]]
```

(ponownie przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Złożoność czasowa tej części algorytmu:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$ porównań (i pomijalna liczba $|T| - |W| + 1$ odczytów z `lastp`),
- optymistyczna: $\Theta\left(\frac{|T|}{|W|}\right)$ $\left(\left\lfloor\frac{|T|+1}{|W|+1}\right\rfloor\right.$ porównań i $\left\lfloor\frac{|T|}{|W|+1}\right\rfloor$ odczytów z `lastp`).

Algorytm jest bardzo szybki w praktycznych zastosowaniach.

Algorytm Sunday'a – pseudokod

Pseudokod właściwego przeszukiwania (po wypełnieniu `lastp`):

```
p ← 0
while p ≤ |T|-|W|:
    if matchesAt(T, p, W): report(p)
    p ← p + |W|
    if p<|T|: p ← p - lastp[T[p]]
```

(ponownie przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Złożoność czasowa tej części algorytmu:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$ porównań (i pomijalna liczba $|T| - |W| + 1$ odczytów z `lastp`),
- optymistyczna: $\Theta\left(\frac{|T|}{|W|}\right)$ ($\left\lfloor\frac{|T|+1}{|W|+1}\right\rfloor$ porównań i $\left\lfloor\frac{|T|}{|W|+1}\right\rfloor$ odczytów z `lastp`).

Algorytm jest bardzo szybki w praktycznych zastosowaniach.

Algorytm Sunday'a – pseudokod

Pseudokod właściwego przeszukiwania (po wypełnieniu `lastp`):

```
p ← 0
while p ≤ |T|-|W|:
    if matchesAt(T, p, W): report(p)
    p ← p + |W|
    if p<|T|: p ← p - lastp[T[p]]
```

(ponownie przyjęto, że znalezione wystąpienia sygnalizowane są za pomocą wywołań `report`)

Złożoność czasowa tej części algorytmu:

- pesymistyczna: $(|T| - |W| + 1)|W| = O(|T||W|)$ porównań (i pomijalna liczba $|T| - |W| + 1$ odczytów z `lastp`),
- optymistyczna: $\Theta\left(\frac{|T|}{|W|}\right)$ $\left(\left\lfloor\frac{|T|+1}{|W|+1}\right\rfloor\right.$ porównań i $\left\lfloor\frac{|T|}{|W|+1}\right\rfloor$ odczytów z `lastp`).

Algorytm jest bardzo szybki w praktycznych zastosowaniach.

Algorytm Sunday'a – uwaga implementacyjna

Warunek `p<|T|` w linii

```
if p<|T|: p ← p - lastp[T[p]]
```

zabezpiecza przed próbą odczytu z nieprawidłowego indeksu (równego $|T|$) łańcucha T.

Jeśli na końcu T jest strażnik (np. `NULL` – jak w łańcuchach w C/C++) umożliwiający odczyt $T[|T|]$, to warunek `p<|T|` można pominąć, upraszczając powyższą linię do:

```
p ← p - lastp[T[p]]
```

Algorytm Sunday'a – uwaga implementacyjna

Warunek `p<|T|` w linii

```
if p<|T|: p ← p - lastp[T[p]]
```

zabezpiecza przed próbą odczytu z nieprawidłowego indeksu (równego $|T|$) łańcucha T.

Jeśli na końcu T jest strażnik (np. `NULL` – jak w łańcuchach w C/C++) umożliwiający odczyt $T[|T|]$, to warunek `p<|T|` można pominąć, upraszczając powyższą linię do:

```
p ← p - lastp[T[p]]
```

Bibliografia

- D. M. Sunday *A Very Fast Substring Search Algorithm*, Communications of the ACM, 33, 8, 132-142 (1990) https://csclub.uwaterloo.ca/∼pbarfuss/p132-sunday.pdf
- H. W. Lang *Sunday algorithm*, http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/sundayen.htm
- C. Charras, T. Lecroq *EXACT STRING MATCHING ALGORITHMS – Animation in Java: Quick Search algorithm*, http://www-igm.univ-mlv.fr/∼lecroq/string/node19.html