Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

Praca dyplomowa magisterska

# PRZEGLĄD I PORÓWNANIE METOD TWORZENIA TABLIC SUFIKSÓW ORAZ ICH EFEKTYWNA IMPLEMENTACJA W JĘZYKU JAVA

Michał Nowak

Promotor
dr inż. Dawid Weiss

Poznań, 2009

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

# Spis treści

# Rozdział 1

# Wprowadzenie

Drzewa sufiksów oraz tablice sufiksów, czyli tablice leksykograficznie posortowanych sufiksów pewnego ciągu symboli, znajdują wiele praktycznych zastosowań:

- w problemach przetwarzania tekstu, nazywanych żargonowo *stringology*, takich jak dopasowywanie, przeszukiwanie, wyszukiwanie powtarzających się podciągów i maksymalnych sekwencji [MM90, Gus97] oraz wielu innych;
- w bioinformatyce [AKO02], gdzie pytania dotyczące kodu genetycznego można sprowadzić do rozwiązania problemów operujących na ciągach znaków;
- w kompresji danych [BW94], gdzie wykorzystywane są do wyszukiwania powtarzających się sekwencji symboli i do obliczania transformacji Burrowsa-Wheelera (ang. *Burrows-Wheeler transform*, *BWT*), która jest krokiem wstępnym takich metod kompresji danych, jak np. `bzip2`.

Większość z powyższych problemów daje się efektywnie rozwiązać przy pomocy drzew sufiksów (ang. *suffix tree*). Udowodniono jednakże [AKO04], że tablice sufiksów są w wielu przypadkach równie dobrą strukturą danych co drzewa sufiksów, a ponieważ ich konstrukcja jest zwykle obarczona mniejszym kosztem pamięciowym i czasowym (mimo tej samej teoretycznej złożoności $\mathcal{O}(n)$), to właśnie tablice sufiksów cieszą się coraz większym powodzeniem w praktyce.

Istnieje wiele algorytmów tworzenia tablic sufiksów, w większości opublikowanych wraz z implementacjami w języku `C` lub `C++`. O ile od strony teoretycznej większość z tych algorytmów jest efektywna (liniowa), o tyle w praktyce ich efektywność zależy od aspektów dostępu do pamięci oraz wielkości słownika symboli. Brakuje również implementacji tych algorytmów w językach wysokiego poziomu, takich jak język Java. Tym brakiem właśnie motywujemy potrzebę opracowania w języku Java biblioteki algorytmów tworzenia tablic sufiksów. Przeniesienie samych algorytmów do języka Java wymaga zwrócenia szczególnej uwagi na jego cechy specyficzne w stosunku do języków niskiego poziomu – różnice w dostępie do pamięci (brak wskaźników, relokowalne struktury danych), zarządzanie pamięcią przez maszynę wirtualną (*garbage collector*), czy też fakt, że kod programu poddawany jest bezustannej obserwacji i kompilacji w czasie rzeczywistym (*just in time compilation*).

## 1.1 Cel i zakres pracy

Głównym celem tej pracy jest **wybór najlepszych algorytmów tworzenia tablic sufiksów oraz ich efektywna implementacja w postaci biblioteki napisanej w języku Java**. W zakres pracy wliczony jest również przegląd obecnie dostępnej literatury poświęconej metodom tworzenia tablic sufiksów oraz opis najciekawszych spośród nich. Docelowo powinna powstać więc pewna taksonomia opierająca się na schemacie działania algorytmów i umożliwiająca ich porównanie i odniesienie względem siebie.

Drugim celem pracy jest dokładne przetestowanie i analiza utworzonej implementacji i określenie jej wydajności na różnych maszynach wirtualnych oraz na różnych platformach sprzętowych. Uzyskane wyniki eksperymentalne skonfrontowane zostaną z istniejącymi wynikami testów wydajnościowych implementacji tych samych algorytmów w językach `C` i `C++`.

## 1.2 Struktura pracy

W rozdziałach drugim i trzecim przedstawiono przegląd literatury dziedzinowej; rozdział drugi zawiera podstawy teoretyczne wprowadzające czytelnika w tematykę tablic sufiksów, a trzeci poświęcony jest klasyfikacji i przeglądowi metod tworzenia tablic sufiksów. Rozdział czwarty zawiera opisy wybranych algorytmów. W rozdziale piątym podane są wyniki empirycznych testów wydajnościowych wykonanych w różnych środowiskach i na różnych maszynach wirtualnych. Rozdział szósty stanowi podsumowanie pracy.

# Rozdział 2

# Podstawy teoretyczne

## 2.1 Podstawowe pojęcia

Niech $\Sigma$ oznacza skończony zbiór o rozmiarze $|\Sigma| \geq 1$, zwany dalej alfabetem [PST07]. Elementami tego zbioru są symbole (w przypadku tekstu zwykle utożsamiane z pojedynczymi literami lub znakami). Alfabet określa się jako indeksowalny, jeżeli istnieje permutacja indeksów taka, że $\forall_{i=0..|\Sigma|-1}$ : $a_0 < a_1 < \ldots < a_{|\Sigma|-1}$ (symbole mogą być uporządkowane).

Niech $x$ oznacza ciąg symboli (pojęcia symbol, znak i litera będą stosowane zamiennie) o długości $n$. Element ciągu $x$ występujący na pozycji $i$ oznacza się jako $x[i]$, $i \in \langle 0, n-1 \rangle$. Przez $x[a..b]$, $0 \leq a \leq b \leq n-1$ rozumie się podciąg $x$ rozpoczynający się na pozycji $a$, a kończący na pozycji $b$ [Gus97]. Warto zauważyć, że $x = x[0..n-1]$.

*Prefiks* to taki podciąg $x$, którego pierwszy element jest jednocześnie pierwszym elementem ciągu $x$, tj. $x[0..b]$, $0 \leq b \leq n-1$. *Sufiks* oznacza taki podciąg $x$, którego ostatni element jest ostatnim znakiem ciągu $x$, tj. $x[a..n-1], 0 \leq a \leq n-1$. Przez $S_i^x$ (lub $S_i$ w przypadku braku niejednoznaczności) rozumie się sufiks $x[i..n-1]$. Ciąg $x$ jest jednocześnie swoim najdłuższym prefiksem jak i sufiksem [Gus97].

*Konkatenacją* dwóch ciągów $x$ i $y$ o długości $n$ i $m$ nazwiemy ciąg $z$, dla którego $z = x \circ y = x[0]x[1]..x[n-1]y[0]y[1]..y[m-1]$.

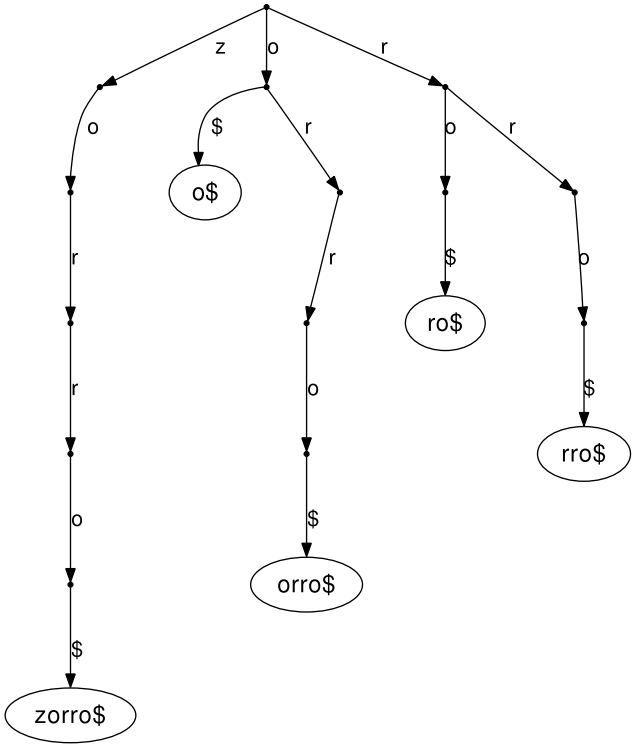
## 2.2 Struktury danych

*Drzewo trie* (ang. *trie*) [Lar99] jest typem drzewa przeszukiwań, w którym węzłom nie odpowiadają klucze, lecz ich fragmenty. W dalszej części pracy rozpatrywane będą drzewa, których kluczami są ciągi symboli z pewnego alfabetu.[1] W tego typu drzewach krawędzie etykietowane są symbolami tego alfabetu, a „wartość" klucza danego węzła wynika z jego pozycji i jest konkatenacją etykiet krawędzi leżących na ścieżce prowadzącej od korzenia drzewa do tego węzła. Wszystkie podwęzły danego węzła mają wspólny prefiks równy wartości klucza ich rodzica. Nieco inny wariant takiego drzewa, nazwany *drzewem skompresowanym* (ang. *radix tree, Patricia trie*) [Lar99], polega na tym, że węzły posiadające tylko jednego potomka są z nimi łączone, a symbole z usuniętych krawędzi są konkatenowane (zob. rys. 2.1).

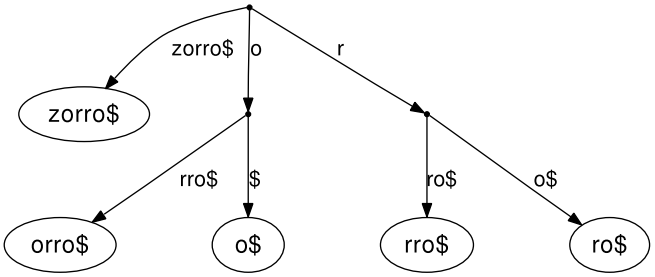
*Drzewo sufiksów* (ang. *suffix tree*) [Gus97] dla ciągu symboli $x$ z alfabetu $\Sigma$ jest skompresowanym drzewem zbudowanym na zbiorze wszystkich sufiksów $x$ o następujących cechach:

- krawędzie są etykietowane niepustymi ciągami symboli,
- każdy sufiks jest reprezentowany w drzewie jako ścieżka od korzenia do liścia,

[1] W nomenklaturze polskiej zarówno *tree*, jak i *trie* nazywane są drzewami [Knu02], str. 528. O ile nie będzie powiedziane inaczej, przez drzewo będziemy rozumieli drzewo *trie*.




RYSUNEK 2.1: Drzewo *trie* dla zbioru sufiksów wyrazu „zorro" z dodanym unikalnym symbolem końcowym $. Dla przejrzystości pominięto łuk od korzenia do liścia o wartości $.




RYSUNEK 2.2: Drzewo sufiksów wyrazu „zorro$". Dla przejrzystości pominięto łuk od korzenia do liścia o wartości $.

- wszystkie węzły wewnętrzne drzewa posiadają co najmniej dwóch potomków.

Zwyczajowo na końcu ciągu $x$ umieszczany jest znak specjalny $, leksykograficznie mniejszy od wszystkich symboli z alfabetu $\Sigma$. Dzięki temu zabiegowi żaden sufiks ciągu $x$ nie będzie prefiksem innego sufiksu, co zapewnia zachowanie wszystkich wyżej wymienionych własności (w przeciwnym przypadku sufiksy mogłyby kończyć się w węzłach wewnętrznych drzewa). Rysunek 2.2 prezentuje drzewo sufiksów dla sekwencji „zorro$". Trzy „klasyczne" algorytmy tworzenia drzew sufiksów omówione zostały w publikacjach [Wei73], [McC76] i [Ukk95]. Ich porównanie można znaleźć w pracy [GK97].

W literaturze powszechne jest etykietowanie sufiksów pozycją ich pierwszego symbolu w słowie $x$ – przez sufiks o etykiecie $i$ rozumie się podciąg $x[i..n-1]$. *Tablica sufiksów* (ang. *suffix array*) [PST07] słowa $x$ jest tablicą etykiet sufiksów uporządkowaną rosnąco według porządku leksykograficznego sufiksów. Tablicę sufiksów słowa $x$ oznacza się $SA_x$ lub $SA$ jeżeli pominięcie identyfikatora słowa nie wprowadza niejednoznaczności. Formalnie, $SA[j] = i \iff x[i..n-1]$ jest $j$-tym sufiksem słowa $x$

| $j$ | $SA[j] = i$ | $x[i..n-1]$ |
|---|---|---|
| 0 | 5 | $ |
| 1 | 4 | o$ |
| 2 | 1 | orro$ |
| 3 | 3 | ro$ |
| 4 | 2 | rro$ |
| 5 | 0 | zorro$ |

RYSUNEK 2.3: Tablica sufiksów dla sufiksów ciągu „zorro$”.

| | 0 | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|---|
| $x = [$ | z | o | r | r | o | $ | ] |
| $SA = [$ | 5 | 4 | 1 | 3 | 2 | 0 | ] |
| $ISA = [$ | 5 | 2 | 4 | 3 | 1 | 0 | ] |
| $lcp = [$ | - | 0 | 1 | 0 | 1 | 0 | ] |

RYSUNEK 2.4: Tablica sufiksów *SA*, odwrotna tablica sufiksów *ISA* oraz tablica najdłuższych prefiksów *lcp* dla sekwencji „zorro$”.

według porządku leksykograficznego. Warto zauważyć, że *SA* jest zawsze permutacją liczb $0..n-1$. Rysunki 2.3 oraz 2.4 prezentują tablicę sufiksów sekwencji symboli „zorro$”.

Dla tablicy sufiksów można zbudować *tablicę najdłuższych wspólnych podciągów* (ang. *longest common prefix array*) [PST07] o długości $n$, której elementy $lcp[i], i = 1..n-1$ oznaczają długość najdłuższego wspólnego prefiksu sufiksów $SA[i]$ i $SA[i-1]$. Przykładowa tablica najdłuższych wspólnych prefiksów zaprezentowana jest na rysunku 2.4. Tablicę *lcp* można obliczyć w czasie liniowym znając *SA* zgodnie z metodami omówionymi w artykułach [KAA01] i [Man04].

Strukturą komplementarną do tablicy sufiksów *SA* jest *odwrotna tablica sufiksów* (ang. *inverse suffix array*) [PST07] oznaczana jako *ISA*. Jest ona permutacją liczb $0..n-1$ spełniającą zależność: $ISA[i] = j \iff SA[j] = i$. Przykład odwrotnej tablicy sufiksów znajduje się na rysunku 2.4. Wartość $k$-tego wpisu w tablicy *SA* oznacza identyfikator sufiksu na $k$-tej pozycji w porządku leksykograficznym; $k$-ty wpis w tablicy *ISA* oznacza pozycję sufiksu $k$ w tablicy *SA* (oraz w porządku leksykograficznym).

Przez *h-grupę* (ang. *h-group*) [PST07] rozumie się podzbiór sufiksów słowa $x$ o wspólnym prefiksie długości $h > 0$. Podział sufiksów na *h-grupy* otrzymuje się poprzez ich częściowe uporządkowanie ze względu na wartość $h$ pierwszych symboli sufiksu. Proces ten nosi nazwę *h-sortowania* (ang. *h-sort*), w jego wyniku sufiksy uporządkowane są według *h-porządku* (ang. *h-order*). Sufiksy należące do jednej *h-grupy* są sobie równe pod względem *h-porządku*. Algorytmy *h-sortowania* są zazwyczaj stabilne, czyli zachowują wcześniejszy porządek sufiksów. Każdej *h-grupie* można przypisać pewien identyfikator (ang. *h-rank*). W zależności od potrzeb algorytmu, wartości identyfikatorów dobierane są na jeden z trzech sposobów:

1. grupa identyfikowana jest pozycją pierwszego jej elementu w przybliżonej tablicy sufiksów – głowa (ang. *head*) grupy,

2. grupa identyfikowana jest pozycją ostatniego elementu w przybliżonej tablicy sufiksów – ogon (ang. *tail*) grupy,

3. każdej z *h-grup* (nawet jednoelementowym) przypisywane są rosnące identyfikatory zgodnie z kolejnością ich pojawienia się w $SA_h$.

Wyniki *h-sortowania* zachowuje się w *przybliżonej tablicy sufiksów* (ang. *approximate suffix array*)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x = [$ | a | b | e | a | c | a | d | a | b | e | a | \$ | ] |
| $SA_1 = [$ | 11 | (0 | 3 | 5 | 7 | 10) | (1 | 8) | 4 | 6 | (3 | 9) | ] |
| $ISA_1 = [$ | 1 | 6 | 10 | 1 | 8 | 1 | 9 | 1 | 6 | 10 | 1 | 0 | ] |
| lub [ | 5 | 7 | 11 | 5 | 8 | 5 | 9 | 5 | 7 | 11 | 5 | 0 | ] |
| lub [ | 1 | 3 | 5 | 1 | 3 | 1 | 4 | 1 | 2 | 5 | 1 | 0 | ] |

RYSUNEK 2.5: Przykłady przybliżonej tablicy sufiksów oraz przybliżonej odwrotnej tablicy sufiksów dla $h = 1$. *H-grupy* wyróżnione zostały nawiasami. Na rysunku przedstawione są 3 wersje tablicy $ISA_1$ różniące się metodami przypisywania identyfikatorów grupom (wypisane w kolejności przedstawienia w tekście). Źródło: [PST07].

[PST07] oznaczanej $SA_h$. Możliwe jest również wyznaczenie *przybliżonej odwrotnej tablicy sufiksów* (ang. *approximate inverse suffix array*) [PST07] $ISA_h$. $SA_h$ jest permutacją liczb $0..n-1$, natomiast $ISA_h$ może zawierać wartości powtarzające się. Wynika to z tego, że każdemu sufiksowi należącemu do danej *h-grupy* przypisywany jest jej identyfikator. Przykład przybliżonej tablicy sufiksów znajduje się na rysunku 2.5.

W niektórych publikacjach przyjęto inną konwencję nazewniczą, *h-grupa* nazywana jest tam *h-kubełkiem* lub po prostu *kubełkiem* (ang. *bucket*). Tablica $ISA_h$ nazywana jest *tablicą wskaźników na kubełki* (ang. *bucket pointer array*) [Sch07].

# Rozdział 3

# Podział metod tworzenia tablic sufiksów

Tablice sufiksów zostały przedstawione w roku 1990 przez Udiego Manbera i Gene'a Mayersa wraz z pierwszym algorytmem ich tworzenia [MM90]. Zestawienie ważniejszych algorytmów tworzenia tablic sufiksów znajduje się w tabeli 3.1. Autorzy pracy [PST07] podsumowują obecny stan wiedzy na temat tworzenia tablic sufiksów w trzech punktach:

- istnieją algorytmy efektywne pamięciowo o złożoności liniowo zależnej od długości wejścia, np. *smaller-larger*, *skew*,
- istnieją algorytmy szybsze w praktycznych zastosowaniach pomimo swojej nieliniowej pesymistycznej złożoności, np. *bpr*, *improved two-stage*, *deep shallow*.
- każdy problem, którego rozwiązanie można znaleźć z pomocą drzew sufiksów może zostać rozwiązany w tym samym czasie przy pomocy tablic sufiksów [AKO04].

Niezrealizowanym do tej pory celem jest zaprojektowanie takiego algorytmu tworzenia tablic sufiksów, który (za [PST07]):

- posiada minimalną złożoność asymptotyczną $\Theta(n)$,
- jest szybki *w praktyce*, to znaczy na dużych rzeczywistych zbiorach danych, np. [A],
- jest *lekki* – wymaga niewiele więcej pamięci ponad ilość potrzebną do przechowania $x$ i $SA_x$.

Autorzy prac [PST07] oraz [Sch07] podjęli próbę stworzenia taksonomii metod tworzenia tablic sufiksów. Proponowane przez nich metody są do siebie bardzo podobne, a klasyfikacja proponowana przez [PST07] została rozbudowana w pracy [Sch07].

W dalszej części pracy omówiona zostanie klasyfikacja proponowana przez Klausa-Bernda Schürmanna [Sch07] poszerzona o algorytm *improved two-stage*.

## 3.1 Podział metod ze względu na proces sortowania sufiksów

W pracy [Sch07] zaproponowano dwa ortogonalne typy klasyfikacji algorytmów: pierwszy z nich dzieli algorytmy ze względu na przebieg procesu sortowania sufiksów; drugi ze względu na sposób wykorzystania zależności między sufiksami. Podział algorytmów według obu typów klasyfikacji przedstawiony jest w tabeli 3.2.

Klasyfikacja algorytmów ze względu na proces sortowania sufiksów opiera się na odpowiedzi na pytanie: „które sufiksy są przetwarzane w pierwszej kolejności i jak przebiega dalej proces sortowania?". Algorytmy dzielone są następnie na dwie podgrupy: wykorzystujące *polepszanie kubełków* (ang. *bucket refinement*) oraz *sortowanie zredukowanych ciągów znaków* (ang. *reduced string sorting*).

| Algorytm | Publikacja | Złożoność obliczeniowa | Złożoność pamięciowa |
|---|---|---|---|
| *bpr* | [Sch07] | $\mathcal{O}(\frac{n^2}{\log n})$ | $9-10n$ |
| *cache* | [Sew00] | $\mathcal{O}(n^2 \log n)$ | $9n$ |
| *copy* | [Sew00] | $\mathcal{O}(n^2 \log n)$ | $5n$ |
| *deep shallow* | [MF04] | $\mathcal{O}(n^2 \log n)$ | $5n$ |
| *difference-cover* | [BK03] | $\mathcal{O}(n \log n)$ | $5-6n$ |
| *improved two-stage* | [MP07] | $\mathcal{O}(n^2 \log n)$ | $5n$ |
| *odd-even* | [KSPP05] i [KJP04] | $\mathcal{O}(n \log\log n)$ | – |
| *prefix-doubling* | [MM90] | $\mathcal{O}(n \log n)$ | $8n$ |
| *qsufsort* | [LS99] | $\mathcal{O}(n \log n)$ | $8n$ |
| *skew* | [KS03] | $\mathcal{O}(n)$ | $10-13n$ |
| *smaller-larger* | [KA05] | $\mathcal{O}(n)$ | $7-10n$ |
| *two-stage* | [IT99] | $\mathcal{O}(n^2 \log n)$ | $5n$ |

TABLICA 3.1: Zestawienie ważniejszych metod tworzenia tablic sufiksów. Złożoność pamięciowa podana jest dla sytuacji, gdy jeden symbol zajmuje 1 jednostkę pamięci a jedna liczba – 4 jednostki pamięci. Źródło: opracowanie własne na podstawie [PST07] i [Sch07].

### 3.1.1 Polepszanie kubełków

Pod pojęciem polepszania kubełków kryje się utworzenie *k-porządku* sufiksów na podstawie wyliczonego wcześniej *h-porządku* sufiksów ($k > h$).

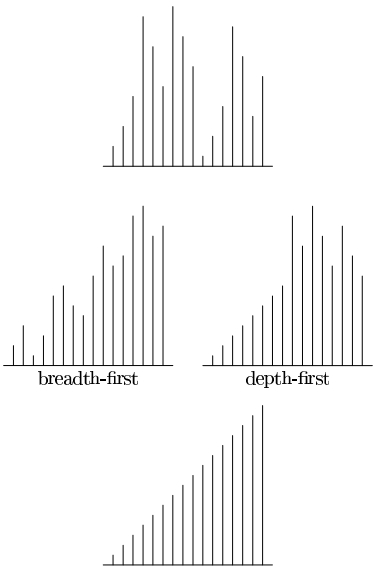
Pierwszy typ algorytmów wykorzystujących polepszanie kubełków opiera się na ogólnych metodach sortowania ciągów znaków (które nie wykorzystują wiedzy o zależnościach między sufiksami). Algorytmy tego typu wykonują *h-sortowanie* dla zwiększającego się $h$ dopóki wszystkie *h-grupy* (kubełki) nie będą zawierały tylko jednego elementu. Algorytmy należące do tej kategorii to *MSD radix sort* omówiony w pracy [MBM93] i *multikey quicksort* opisany w [BS97].

Drugi typ algorytmów polepszania kubełków opiera się na następującej obserwacji: jeżeli dwa sufiksy słowa $x$ $x[u,n-1]$ i $x[v,n-1]$ mają wspólny prefiks długości $\ell$, to ich porządek można wywnioskować na podstawie porządku ich $\ell$*-następców* $x[u+\ell,n-1]$ i $x[v+\ell,n-1]$ (przykładem tego procesu jest drugi etap algorytmu *two-stage*). Algorytmy z tej grupy można podzielić na dwa dalsze podtypy:

- algorytmy wykonujące *polepszanie wszerz* (ang. *breadth-first refinement*). Metody te dokonują *h-sortowania* ze zwiększającym się $h$ aż do uzyskania jednoelementowych kubełków (*h-grup*). Do tej grupy algorytmów należą *prefix-doubling* [MM90] i *qsufsort* [LS99].

- algorytmy wykonujące *polepszanie wgłąb* (ang. *depth-first refinement*). Metody te wykorzystują następujący schemat polepszania kubełków: dla dowolnego $h$, sortowanie sufiksów kolejnego kubełka następuje dopiero po tym, gdy poprzedni kubełek jest w pełni posortowany, czyli wszystkie jego podkubełki są jednoelementowe. Na przykład, dla ciągu „AAABBABBBAAABBAB" po *1-sortowaniu* grupa sufiksów rozpoczynających się od symbolu *B* będzie sortowana dopiero wtedy, gdy będzie ustalony porządek sufiksów rozpoczynających się od *A*; podczas sortowania tej grupy porządek sufiksów o prefiksie „AB" będzie ustalony dopiero po posortowaniu sufiksów rozpoczynających się od „AA" itd. Metody tego typu wykorzystują naiwne metody typu pierwszego w sytuacji, gdy porządek $\ell$*-następców* jest jeszcze nieznany. Do tej kategorii algorytmów należą między innymi: *two-stage*, *copy*, *cache* oraz *deep shallow*. W ogólności metody tego typu są bardzo szybkie w praktyce, pomimo ich wysokiej złożoności sięgającej nawet $\mathcal{O}(n^2 \log n)$.

Rysunek 3.1 przedstawia etapy procesu polepszania kubełków dla ciągu „AAABBABBBAAABBAB". Każdy sufiks reprezentowany jest przez pionową linię której wysokość oznacza jego pozycję według porządku leksykograficznego: niska linia oznacza sufiks leksykograficznie mniejszy od sufiksu repre-




RYSUNEK 3.1: Etapy algorytmu polepszania kubełków dla wyrazu „AAABBABBBAAABBAB". Źródło: [Sch07].

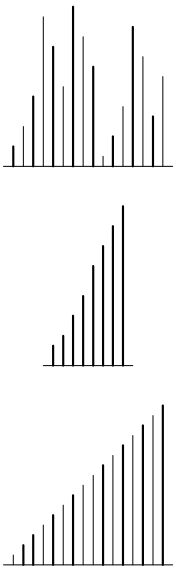
| Typ algorytmu | strategia *push* | strategia *pull* |
|---|---|---|
| Polepszanie wszerz | *prefix-doubling* | *qsufsort* |
| Polepszanie wgłąb | *two-stage*<br>*copy*<br>*deep-shallow*<br>*improved two-stage* | *cache*<br>*bpr* |
| Sortowanie zred. ciągów znaków | *skew*<br>*odd-even*<br>*smaller-larger* | *difference-cover* |

TABLICA 3.2: Podział algorytmów tworzenia tablic sufiksów według taksonomii zaproponowanej w pracy [Sch07]. Źródło: opracowanie własne w oparciu o [Sch07].

zentowanego przez wyższą linię. Na rysunku 3.1 widać (od góry): stan początkowy, stan w trakcie działania algorytmu (w przypadku polepszania wszerz jest to stan po posortowaniu według prefiksu długości 2; dla polepszania wgłąb jest to stan po uporządkowaniu sufiksów rozpoczynających się od symbolu „A"), stan końcowy – posortowane sufiksy.

### 3.1.2 Sortowanie zredukowanych ciągów znaków

Metody tego typu działają według następującego schematu: na początku wybierany jest podzbiór *sub* identyfikatorów sufiksów, odpowiadające im sufiksy są potem sortowane na podstawie prefiksów ustalonej długości. Następnie każdemu z tych sufiksów przypisywany jest pewien klucz odpowiadający jego porządkowi wg. prefiksów. Następnie tworzony jest taki ciąg $t^{sub}$ długości $|sub|$ zawierający wcześniej przydzielone klucze, którego $SA_{t^{sub}}$ odpowiada porządkowi leksykograficznemu sufiksów identyfikowanych zbiorem *sub*. Tablica $SA_{t^{sub}}$ może być obliczana rekurencyjnie tym samym algorytmem, lub z wykorzystaniem efektywnego algorytmu sortowania ciągów znaków ([BM93], [MBM93], [H], [BS97], [SZ04]). W pełni uporządkowane sufiksy ze zbioru *sub* służą potem do posortowania pozostałych sufiksów. Ostatecznie budowana jest pełna tablica sufiksów. Do algorytmów realizujących



RYSUNEK 3.2: Etapy algorytmu sortowania zredukowanych ciągów znaków dla wyrazu „AAABBABBBAAABBAB". Źródło: [Sch07].

powyższy schemat należą: *difference-cover*, *skew*, *odd-even* oraz *smaller-larger*.

Rysunek 3.2 przedstawia etapy działania algorytmu sortowania zredukowanych ciągów znaków. Podobnie jak na rysunku 3.1, sufiksy reprezentowane są przez pionowe linie, których wysokość oznacza miejsce sufiksu w porządku leksykograficznym. Sufiks reprezentowany przez mniejszą linię jest leksykograficznie mniejszy od sufiksu reprezentowanego przez wyższą linię. Na rysunku przedstawiono po kolei (od góry): stan początkowy; stan algorytmu w sytuacji gdy posortowane są sufiksy o nieparzystych identyfikatorach; stan końcowy.

## 3.2 Podział ze względu na sposób wykorzystania zależności między sufiksami

Drugim sposobem podziału zaproponowanym w pracy [Sch07] jest podział algorytmów tworzenia tablic sufiksów ze względu na sposób wykorzystania zależności między sufiksami. Jeżeli dwa sufiksy $x[u, n-1]$ i $x[v, n-1]$ mają wspólny prefiks długości $\ell$, to ich porządek można ustalić na podstawie porządku sufiksów $x[u+\ell, n-1]$ i $x[v+\ell, n-1]$. Rozróżnia się dwie metody wykorzystywania tej zależności: metodę *pull* i metodę *push*. Terminy *pull* i *push* pochodzą z terminologii systemów informacyjnych, oznaczają różne strategie komunikacji pomiędzy nadawcą wiadomości i jej odbiorcą. Metoda *push* oznacza sytuację, kiedy dostawca wiadomości nawiązuje komunikację. Odwrotna sytuacja następuje w strategii *pull* – tutaj to odbiorca zgłasza żądanie inicjalizacji komunikacji.

W odniesieniu do algorytmów tworzenia tablic sufiksów metoda *push* polega na tym, że wykorzystuje porządek wcześniej obliczonych grup sufiksów do ustalenia porządku grup sufiksów *$\ell$-poprzedników,* tzn. po ustaleniu porządku sufiksów $x[u+\ell, n-1]$ i $x[v+\ell, n-1]$ obliczany jest porządek sufiksów $x[u, n-1]$ i $x[v, n-1]$. Wzorcowym przykładem wykorzystania tej techniki jest algorytm *two-stage*.

Metoda *pull* jest wykorzystywana w procesie sortowania opierającego się o porównania. Podczas porównywania sufiksów $x[u, n-1]$ i $x[v, n-1]$ sprawdzany jest porządek sufiksów $x[u+\ell, n-1]$ i $x[v+\ell, n-1]$. Algorytm *qsufsort* jest najlepszym przykładem zastosowania tej techniki.

# Rozdział 4

# Wybrane metody tworzenia tablic sufiksów

W poniższym rozdziale omówione są wybrane metody tworzenia tablic sufiksów. Algorytmy wybrane zostały na podstawie wyników testów wydajnościowych zaprezentowanych w pracach [PST07] i [Sch07] oraz publikowanych na stronach [F] i [E]. Algorytmy uzyskujące dobre wyniki na dużych zbiorach danych – szybkie *w praktyce* pochodzą z rodziny algorytmów wykorzystujących *polepszanie wszerz*. W poniższym zestawieniu opisane zostały również algorytmy reprezentujące inne grupy metod tworzenia tablic sufiksów: *polepszanie wgłąb* i *sortowanie zredukowanych ciągów znaków*.

## 4.1 Algorytm *skew*

Algorytm *skew* został opracowany przez Petera Sandersa i Juhę Kärkkäinena [KS03]. Dostępna jest również implementacja tego algorytmu ich autorstwa [G]. Opisywana metoda pochodzi z rodziny algorytmów wykorzystujących *sortowanie zredukowanych ciągów znaków*. Algorytm wykonuje 3 główne kroki:

1. Tworzenie tablicy sufiksów zbudowanej z sufiksów $S_i$ o indeksach $i \bmod 3 \neq 0$. Problem jest redukowany do tworzenia tablicy sufiksów ciągu długości 2/3 rozmiaru ciągu wejściowego, a następnie rozwiązywany rekurencyjne.

2. Tworzenie tablicy sufiksów zbudowanej z sufiksów pominiętych w pierwszym kroku z wykorzystaniem tablic sufiksów z kroku 1.

3. Połączenie zbudowanych tablic w jedną.

Pierwszy (i najbardziej czasochłonny) krok algorytmu polega na posortowaniu sufiksów $S_i$ dla $i \bmod 3 \neq 0$, czyli utworzeniu tablicy sufiksów $SA^{12}$. Jeżeli w wyniku *h-sortowania* dla $h = 3$ wszystkie grupy są jednoelementowe, to ten etap algorytmu się kończy. W przeciwnym przypadku każdemu sufiksowi $S_i$ nadawany jest identyfikator $x_i^{'} \in [1, 2n/3]$ będący identyfikatorem jego *3-grupy* powstałej po *3-sortowaniu*. Następnie tworzony jest ciąg $x^{12} = [x_i^{'} : i \bmod 3 = 1] \circ [x_i^{'} : i \bmod 3 = 2]$, którego tablica sufiksów $SA_{x^{12}}$ obliczana jest rekurencyjnie. Tablica $SA^{12}$ wypełniana jest zgodnie z poniższym wzorem ($n_1$ oznacza liczbę sufiksów o etykietach $i$ takich, że $i \bmod 3 = 1$):

$$SA^{12}[i] = \begin{cases} 1 + 3k & \textit{jeżeli } k = SA_{x^{12}}[i] < n_1, \\ 2 + 3(k - n_1) & \textit{w przeciwnym wypadku.} \end{cases}$$

Drugi krok algorytmu polega na utworzeniu tablicy sufiksów $SA^0$ złożonej z sufiksów $S_i$, gdzie $i \bmod 3 = 0$. Tablica ta powstaje w wyniku sortowania par $(x[i], S_{i+1})$. Ponieważ porządek sufiksów

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x$ = [ | b | a | d | d | a | d | d | a | c | c | a | $ | ] |
| $S_i, i \bmod 3 = 0$ = [ | 0 | | | 3 | | | 6 | | | 9 | | | ] |
| $S_i, i \bmod 3 \neq 0$ = [ | | 1 | 2 | | 4 | 5 | | 7 | 8 | | 10 | 11 | ] |
| $x[i..i+2], i \bmod 3 \neq 0$ = [ | | add | dda | | add | dda | | acc | cca | | a$- | $– | ] |

*3-sortowanie* sufiksów o etykietach $i \bmod 3 \neq 0$ (wartości są indeksami po posortowaniu).

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x'$ = [ | | 3 | 5 | | 3 | 5 | | 2 | 4 | | 1 | 0 | ] |

Tworzenie ciągu $x^{12} = [x'_i : i \bmod 3 = 1] \circ [x'_i : i \bmod 3 = 2]$.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $x^{12}$ = [ | 3 | 3 | 2 | 1 | 5 | 5 | 4 | 0 | ] |

Obliczanie $SA_{x^{12}}$ i $SA^{12}$.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $SA_{x^{12}}$ = [ | 7 | 3 | 2 | 1 | 0 | 6 | 5 | 4 | ] |
| $ISA_{x^{12}}$ = [ | 4 | 3 | 2 | 1 | 7 | 6 | 5 | 0 | ] |
| $SA^{12}$ = [ | 11 | 10 | 7 | 4 | 1 | 8 | 5 | 2 | ] |

Sufiksy $S_i$, $i \bmod 3 = 1$ oraz $S_j$, $j = i - 1$ zgodnie z kolejnością w $SA^{12}$.

| | | | | | |
|---|---|---|---|---|---|
| [ | 10 | 7 | 4 | 1 | ] |
| [ | 9 | 6 | 3 | 0 | ] |
| $x[j]$ = [ | c | d | d | b | ] |

Stabilne *1-sortowanie* sufiksów $S_j$.

| | | | | | |
|---|---|---|---|---|---|
| $SA^0$ = [ | 0 | 9 | 6 | 3 | ] |

Scalanie tablic $SA^0$ i $SA^{12}$.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $SA$ = [ | 11 | 10 | 7 | 4 | 1 | 0 | 9 | 8 | 6 | 3 | 5 | 2 | ] |

RYSUNEK 4.1: Przykład tworzenia tablicy sufiksów według algorytmu *skew*. Źródło: opracowanie własne.

$S_{i+1}$ jest zawarty w $SA^{12}$, do znalezienia $SA^0$ wystarczy posortować stabilnie elementy $SA^{12}[j]$ reprezentujące sufiksy $SA_{i+1}, i \bmod 3 = 0$ według wartości $x[i]$. Można tego dokonać w czasie liniowym jednym krokiem sortowania kubełkowego.

Trzeci krok algorytmu polega na połączeniu tablic $SA^{12}$ i $SA^0$ w jedną tablicę sufiksów. Porównywanie sufiksów $S_j$, gdzie $j \bmod 3 = 0$ i $S_i, i \bmod 3 \neq 0$ odbywa się na jeden z dwóch sposobów:

- Jeżeli $i \bmod 3 = 1$, to porządek sufiksów $S_i$ i $S_j$ można ustalić porównując pary $(x[i], S_{i+1})$ oraz $(x[j], S_{j+1})$. Ponieważ $i + 1 \bmod 3 = 2$ i $j + 1 \bmod 3 = 1$ porządek sufiksów $S_{i+1}$ i $S_{j+1}$ można ustalić na podstawie ich pozycji w $SA_{x^{12}}$. Ta pozycja może być ustalana w czasie stałym, jeżeli obliczona zostanie tablica $ISA_{x^{12}}$.

- Analogicznie, jeżeli $i \bmod 3 = 2$, to porównywane są trójki $(x[i], x[i+1], S_{i+2})$ i $(x[j], x[j+1], S_{j+2})$, przy czym sufiksy $S_{i+2}$ i $S_{j+2}$ są zastępowane odpowiednimi wpisami z tablicy $ISA_{x^{12}}$.

## 4.2 Algorytm *qsufsort*

Algorytm *qsufsort* został przedstawiony w pracy [LS99] autorstwa Jespera Larrsona i Kunihiko Sadakane. Autorzy pracę udostępniają również implementację w języku `C` [L]. Opisywana metoda należy do rodziny algorytmów wykonujących polepszanie wgłąb, opiera się na algorytmie *prefix-doubling*.

Sortowanie sufiksów wykonywane jest algorytmem *ternary split quicksort* [BM93]. Algorytm *qsufsort* wykorzystuje również twierdzenie opublikowane w pracy [KMR72]: dla obliczonych $SA_h$ i $ISA_h$ posortowanie sufiksów $S_i$ według par $(ISA_h[i], ISA_h[i+h])$, $i + h \leq n$ tworzy *2h-porządek* sufiksów, czyli porządek według prefiksów długości $2h$.[1] Wynika to z tego, że do ustalenia porządku sufiksów

[1] Sufiksy $S_i, i > n - h$ są zawsze w pełni uporządkowane.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x$ = [ | a | b | e | a | c | a | d | a | b | e | a | $ | ] |
| $SA_1$ = [ | 11 | (0 | 3 | 5 | 7 | 10) | (1 | 8) | 4 | 6 | (2 | 9) | ] |
| $ISA_1$ = [ | 5 | 7 | 11 | 5 | 8 | 5 | 9 | 5 | 7 | 11 | 5 | 0 | ] |
| $L$ = [ | -1 | 5 | | | | | 2 | | -2 | | 2 | | ] |
| $SA_2$ = [ | 11 | 10 | (0 | 7) | 3 | 5 | (1 | 8) | 4 | 6 | (2 | 9) | ] |
| $ISA_2$ = [ | 3 | 7 | 11 | 4 | 8 | 5 | 9 | 3 | 7 | 11 | 2 | 0 | ] |
| $L$ = [ | -2 | | 2 | | -2 | | 2 | | -2 | | 2 | | ] |
| $SA_4$ = [ | 11 | 10 | (0 | 7) | 3 | 5 | 8 | 1 | 4 | 6 | 9 | 2 | ] |
| $ISA_4$ = [ | 3 | 7 | 11 | 4 | 8 | 5 | 9 | 3 | 6 | 10 | 1 | 0 | ] |
| $L$ = [ | -2 | | 2 | | -8 | | | | | | | | ] |
| $SA_8$ = [ | 11 | 10 | 7 | 0 | 3 | 5 | 8 | 1 | 4 | 6 | 9 | 2 | ] |
| $ISA_8$ = [ | 3 | 7 | 11 | 4 | 8 | 5 | 9 | 2 | 6 | 10 | 1 | 0 | ] |
| $L$ = [ | -12 | | | | | | | | | | | | ] |

RYSUNEK 4.2: Przebieg algorytmu *qsufsort* dla słowa „abeacadabea". Źródło: opracowanie własne na podstawie [PST07].

o wspólnym prefiksie długości $h$ wykorzystujemy porządek ich *h-następców*, którzy są również posortowani według h pierwszych znaków, co daje w konsekwencji porządek sufiksów według prefiksów długości $2h$.

Na początku działania algorytm wykonuje *1-sort*, w wyniku którego powstają $SA_1$ i $ISA_1$. Numery grup w odwróconej tablicy sufiksów przypisywane są poprzez wybór ogona (pozycji ostatniego sufiksu w tablicy $SA_h$) danej grupy. Algorytm *qsufsort* utrzymuje również tablicę $L$ długości $n$ używaną w celu określania rozmiarów grup. Wartość $L[j] = d$ oznacza, że grupa rozpoczynająca się na pozycji $j$ ma $d$ elementów. Wartości ujemne w tablicy $L$ oznaczają ciągi jednoelementowych grup (na przykład $-2$ oznacza dwie jednoelementowe grupy).

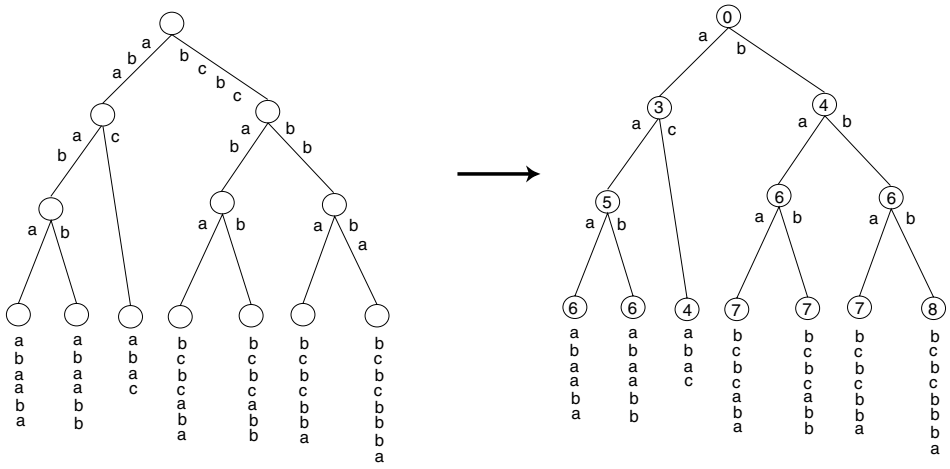
Przeglądanie tablicy $L$ od lewej do prawej umożliwia omijanie grup jednoelementowych w procesie polepszania kubełków. Każda *h-grupa* sortowana jest osobno, jej sufiksy $S_i$ porównywane są na podstawie wartości $ISA_h[i+h]$. Po posortowaniu wszystkich grup otrzymujemy *2h-porządek*. Algorytm kończy swoje działanie, gdy wszystkie h-grupy są jednoelementowe, czyli gdy $L[0] = -n$. W przeciwnym wypadku $h$ jest podwajane, a proces polepszania kubełków wykonywany po raz kolejny.

Algorytm *qsufsort* da się optymalizować pod kątem redukcji zużycia pamięci. Możliwe jest kompletne wyeliminowanie tablicy $L$. Do wyznaczania *h-grup* zawierających więcej niż 1 element można wykorzystać tablicę *ISA*, rozmiar grupy wyznacza się wtedy na podstawie jej pierwszego elementu. Jeżeli $j$ oznacza pozycję pierwszego elementu grupy w $SA$, a $i = SA_h[j]$, to jej rozmiar wynosi $ISA_h[i] - j + 1$. Złożoność pamięciowa algorytmu może również zostać zredukowana poprzez nadpisanie ciągu wejściowego i wykorzystanie tego obszaru pamięci do przechowywania *ISA* po obliczeniu $SA_1$.

## 4.3 Algorytm *deep shallow*

Algorytm *deep shallow* jest rozwinięciem algorytmu *copy* opracowanym przez Paolo Ferraginę i Giovanniego Manziniego. Opublikowany został w pracy [MF04]. Kod algorytmu w języku `C` dostępny jest pod adresem [J], jego autorem jest Giovanni Manzini.

Pierwszym krokiem algorytmu jest uporządkowanie sufiksów według dwóch pierwszych znaków (*2-sortowanie*). Każdy z utworzonych w ten sposób kubełków sortowany jest algorytmem *multikey quicksort* [BS97], który jest przerywany po osiągnięciu poziomu rekurencji równemu $L$, czyli jeżeli wewnątrz kubełka istnieje grupa sufiksów o wspólnym prefiksie długości $L$. Podejście to nosi nazwę sortowania płytkiego (ang. *shallow sorting*).




RYSUNEK 4.3: Drzewo skompresowane oraz odpowiadające mu drzewo *blind trie* zbudowane na zbiorze sekwencji *abaaba, abaabb, abac, bcbcaba, bcbcabb, bcbcbba, bcbcbbba*. Źródło: [MF04].

Sortowanie znajdowanych sufiksów o wspólnym prefiksie długości $L$ nazwane zostało przez autorów algorytmu *deep shallow* sortowaniem głębokim (ang. *deep sort*). Przebieg działania głębokiego sortowania zależy od wielkości zbioru sortowanych sufiksów. Jeżeli liczność zbioru nie przekracza zadanej wartości $B$, to sufiksy są sortowane algorytmem *blind sort*. W przeciwnym przypadku do posortowania sufiksów wykorzystane zostanie zmodyfikowany algorytm *ternary split quicksort* [BM93]. Autorzy sugerują użycie wartości $B = n \times 0.0005$ jako progu wielkości zbioru.

Algorytm *blind sort* opiera swoje działanie na strukturze danych nazywanej *blind trie* [FG98], która jest typem skompresowanego drzewa którego węzły wewnętrzne przechowują liczby określające długość wspólnego prefiksu węzłów potomnych (jeżeli węzeł zawiera liczbę $k$, to jego węzły potomne różnią się na pozycji $k+1$). Przykładowe drzewo *blind trie* znajduje się na rysunku 4.3.

Algorytm *blind sort* tworzy drzewo *blind trie*, a następnie przegląda je od lewej do prawej uzyskując w ten sposób porządek leksykograficzny sekwencji podanych na wejściu (w drzewie *blind trie* węzły potomne danego węzła są uporządkowane). Poważną wadą metody *blind sort* jest jej złożoność pamięciowa, sięgająca nawet $36m$, gdzie $m$ oznacza liczbę ciągów do posortowania.

Do sortowania zbiorów większych niż $B$ autorzy algorytmu *deep shallow* użyli zmodyfikowanego algorytmu *ternary split quicksort*, opisanego w pracy [BM93]. Dokonali oni następujących zmian:

1. Jeżeli na dowolnym etapie rekursji zbiór sufiksów jest mniejszy niż $B$, to do jego sortowania użyty zostaje algorytm *blind sort*.

2. Podczas fazy podziału sufiksów obliczane są $L_S$ i $L_L$, czyli długość najdłuższego wspólnego prefiksu elementu osiowego (ang. *pivot*) i zbioru elementów mniejszych ($L_S$) oraz większych ($L_L$) od elementu osiowego. Dzięki temu podczas sortowania sufiksów w tych zbiorach można pomijać prefiksy długości $L_S$ lub $L_L$.

Paolo Ferragina i Giovanni Manzini w pracy [MF04] wymieniają trzy zalety dwuetapowego podejścia do problemu sortowania sufiksów zastosowanego w algorytmie *deep shallow*:

1. Szybkie wykrywanie grup sufiksów o długim wspólnym prefiksie.

2. Rozmiar stosu użytego podczas rekurencji w fazie sortowania płytkiego jest ograniczony parametrem $L$ i nie zależy od rozmiaru wejścia.

3. Jeżeli długość najdłuższego wspólnego prefiksu sufiksów jednego kubełka nie przekracza $L$, to uporządkowywane są one efektywnym algorytmem sortowania ciągu znaków (*multikey quicksort*).

## 4.4 Algorytm *two-stage*

Hideo Itoh i Hozumi Tanaka zaproponowali w pracy [IT99] algorytm tworzenia tablic sufiksów o nazwie *two-stage*. Algorytm ten dzieli sufiksy na dwie kategorie: sufiksy typu $A$ to wszystkie sufiksy $S_i$ spełniające nierówność $x[i] > x[i+1]$, sufiksy typu $B$ to sufiksy spełniające nierówność $x[i] \leq x[i+1]$.

Algorytm rozpoczyna swoje działanie od obliczenia $SA_1$. Początek i koniec każdej grupy zapamiętywany jest w tablicach *head* i *tail*. Długość tych tablic odpowiada wielkości słownika symboli. Sufiksy w każdej *1-grupie* są uporządkowywane w taki sposób, żeby sufiksy typu $A$ były przed sufiksami typu $B$. Wynika to z tego, że jeżeli sufiksy $S_i$ typu $A$ i $S_j$ typu $B$ mają wspólny prefiks długości jednego symbolu, to sufiks $S_i$ poprzedza sufiks $S_j$ w porządku leksykograficznym. Indeksy pierwszych sufiksów typu $B$ każdej grupy zapamiętywane są w tablicy *part* długości $\sigma$. Kolejnym krokiem algorytmu jest sortowanie sufiksów typu $B$ wewnątrz każdej grupy. Autorzy pracy [IT99] sugerują użycie do tego celu algorytmu [BS97], nie jest to jednak konieczne i można do tego celu użyć innego algorytmu sortowania ciągów znaków.

Następnie ustalany jest porządek sufiksów typu $A$. Tablica $SA$ przeglądana jest od lewej do prawej, znajdując wartości $i = SA[j], j = 0,1,\ldots,n-1$. Jeżeli $S_{i-1}$ jest jeszcze nieuporządkowanym sufiksem typu $A$, to umieszczany jest na początku grupy do której należy. Odpowiednia wartość w tablicy *head* jest potem inkrementowana. Po jednokrotnym przejrzeniu $SA$ sufiksy są już w pełni posortowane, co kończy algorytm (przykład na rysunku 4.4).

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x$ = [ | b | a | d | d | a | d | d | a | c | c | a | $ | ] |
| type = [ | A | B | B | A | B | B | A | B | B | A | A | B | ] |
| | | | | | | *1-sort* | | | | | | | |
| $SA$ = [ | 11 | (– | 1 | 4 | 7) | (–) | (– | 8) | (– | – | 2 | 5) | ] |
| type = [ | B | A | B | B | B | A | A | B | A | A | B | B | ] |
| | | | | | | sortowanie sufiksów typu $B$ | | | | | | | |
| $SA$ = [ | 11 | (– | 7 | 4 | 1) | (–) | (– | 8) | (– | – | 5 | 2) | ] |
| | | | | | | wstawianie sufiksów typu $A$ | | | | | | | |
| $SA$ = [ | 11 | 10 | 7 | 4 | 1 | – | – | 8 | – | – | 5 | 2 | ] |
| $SA$ = [ | 11 | 10 | 7 | 4 | 1 | – | 9 | 8 | – | – | 5 | 2 | ] |
| $SA$ = [ | 11 | 10 | 7 | 4 | 1 | – | 9 | 8 | 6 | – | 5 | 2 | ] |
| $SA$ = [ | 11 | 10 | 7 | 4 | 1 | – | 9 | 8 | 6 | 3 | 5 | 2 | ] |
| $SA$ = [ | 11 | 10 | 7 | 4 | 1 | 0 | 9 | 8 | 6 | 3 | 5 | 2 | ] |

RYSUNEK 4.4: Przebieg algorytmu *two-stage* dla słowa „baddaddacca”. Nieuporządkowane sufiksy typu $A$ oznaczane są znakiem „–”. Źródło: opracowanie własne na podstawie [PST07].

## 4.5 Algorytm *improved two-stage*

Algorytm *improved two-stage* opisany został w pracy [MP07], której autorami są Simon Puglisi i Michael Maniscalco. Kwestia autorstwa tego algorytmu pozostaje niejasna, metoda ta posiada aż trzy niezależne implementacje powstałe przed publikacją artykułu:

- *archon* [I] autorstwa Dimy Małyszewa,

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x$ = [ | e | d | a | b | d | c | c | d | e | e | d | a | b | $ | ] |
| type = [ | A | A | B | $B^*$ | A | B | B | $B^*$ | A | A | A | $B^*$ | A | | ] |
| | | | | | | Obliczanie wielkości *2-grup* | | | | | | | | | |
| $SA_2$ = [ | – | (– | –) | – | – | – | – | (– | –) | – | – | (– | –) | – | ] |
| | | | | | | Sortowanie sufiksów typu $B^*$ | | | | | | | | | |
| $SA$ = [ | – | 11 | – | – | 3 | – | – | – | – | – | 7 | – | – | – | ] |
| | | | | | | Wstawianie sufiksów typu B | | | | | | | | | |
| $SA$ = [ | – | 11 | – | – | 3 | – | 6 | – | – | – | 7 | – | – | – | ] |
| $SA$ = [ | – | 11 | – | – | 3 | 5 | 6 | – | – | – | 7 | – | – | – | ] |
| $SA$ = [ | – | 11 | 2 | – | 3 | 5 | 6 | – | – | – | 7 | – | – | – | ] |
| | | | | | | Wstawianie sufiksów typu A | | | | | | | | | |
| $SA$ = [ | 13 | 11 | 2 | – | 3 | 5 | 6 | – | – | – | 7 | – | – | – | ] |
| $SA$ = [ | 13 | 11 | 2 | 12 | 3 | 5 | 6 | – | – | – | 7 | – | – | – | ] |
| $SA$ = [ | 13 | 11 | 2 | 12 | 3 | 5 | 6 | 10 | – | – | 7 | – | – | – | ] |
| $SA$ = [ | 13 | 11 | 2 | 12 | 3 | 5 | 6 | 10 | 1 | – | 7 | – | – | – | ] |
| $SA$ = [ | 13 | 11 | 2 | 12 | 3 | 5 | 6 | 10 | 1 | 4 | 7 | – | – | – | ] |
| $SA$ = [ | 13 | 11 | 2 | 12 | 3 | 5 | 6 | 10 | 1 | 4 | 7 | 9 | – | – | ] |
| $SA$ = [ | 13 | 11 | 2 | 12 | 3 | 5 | 6 | 10 | 1 | 4 | 7 | 9 | 0 | – | ] |
| $SA$ = [ | 13 | 11 | 2 | 12 | 3 | 5 | 6 | 10 | 1 | 4 | 7 | 9 | 0 | 8 | ] |

RYSUNEK 4.5: Przebieg algorytmu *divsufsort* dla słowa „edabdccdeedab". Źródło: opracowanie własne na podstawie [Mor05].

- *divsufsort* [K] której autorem jest Yuta Mori,
- *msufsort* [F] autorstwa Michaela Maniscalco.

Wymienione implementacje algorytmu *improved two-stage* napisane zostały w języku `C++`.

Na początku sufiksy są dzielone na dwie kategorie: sufiksy typu $A$ to sufiksy $S_i$ spełniające nierówność $S_i > S_{i+1}$, sufiksy typu $B$ to sufiksy spełniające nierówność $S_i \leq S_{i+1}$. Sufiks o identyfikatorze $n-1$ należy do obu grup. Następnie, sufiksy $S_i$ typu $B$ których następny sufiks $S_{i+1}$ jest typu $A$ oznaczane są jako sufiksy typu $B^*$.

Kolejnym krokiem algorytmu jest znalezienie granic grup które powstałyby po *2-sortowaniu*. Następnie tworzony jest *2-porządek* sufiksów typu $B^*$. *2-sortowanie* wszystkich sufiksów nie jest konieczne w tym momencie, a jego pominięcie zmniejsza czas działania algorytmu. Sufiksy typu $B^*$ należące do jednej grupy są sortowane zgodnie z metodami przedstawionymi w pracach [BS97, MF04] i umieszczane na początkowych pozycjach wewnątrz kubełka w tablicy $SA$.

Porządek sufiksów typu $B$ ustalany jest poprzez przeglądanie tablicy $SA$ od prawej do lewej. Dla każdego sufiksu $SA[i]$ odczytanego podczas przeglądania tablicy sufiks $SA[i]-1$ jest wstawiany na ostatnią pustą pozycję wewnątrz swojej grupy jeżeli jest sufiksem typu $B$.

Wstawianie sufiksów typu $A$ wykonywane jest w sposób identyczny jak w algorytmie *two-stage* (przykład przedstawiono na rysunku 4.5).

## 4.6 Algorytm *bpr*

Klaus-Bernd Schürmann w pracy [Sch07] zaproponował algorytm *bpr* (*bucket pointer refinement*, polepszanie wskaźników na kubełki). Algorytm ten jest poprawioną wersją metody przedstawionej w pracy [SS07] napisanej przez Klausa-Bernda Schürmanna i Jensa Stoye. Kod algorytmu *bpr* w języku `C++` dostępny jest na stronie [M].

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| $x$ = [ | D | E | B | D | E | B | D | E | A | ] |
| | Podział na kubełki, $q = 2$ | | | | | | | | | |
| | A | BD | | | DE | | EA | EB | | |
| $SA$ = [ | 8 | **(2** | **5)** | (0 | 3 | 6) | 7 | (1 | 4) | ] |
| $ISA$ = [ | 5 | 8 | 2 | 5 | **8** | 2 | 5 | **6** | 0 | ] |
| | Po posortowaniu kubełka BD | | | | | | | | | |
| $SA$ = [ | 8 | 5 | 2 | **(0** | **3** | **6)** | 7 | (1 | 4) | ] |
| $ISA$ = [ | 5 | 8 | **2** | 5 | 8 | **1** | 5 | 6 | **0** | ] |
| | Po posortowaniu kubełka DE | | | | | | | | | |
| $SA$ = [ | 8 | 5 | 2 | 6 | 3 | 0 | 7 | **(1** | **4)** | ] |
| $ISA$ = [ | 5 | 8 | 2 | **4** | 8 | 1 | **3** | 6 | 0 | ] |
| | Po posortowaniu kubełka EB, koniec algorytmu | | | | | | | | | |
| $SA$ = [ | 8 | 5 | 2 | 6 | 3 | 0 | 7 | 4 | 1 | ] |
| $ISA$ = [ | 5 | 8 | 2 | 4 | 7 | 1 | 3 | 6 | 0 | ] |

RYSUNEK 4.6: Przebieg algorytmu *bpr* dla słowa „DEBDEBDEA". Elementy wyróżnione w tablicy *SA* przeznaczone do uporządkowania w danym kroku, elementy wyróżnione w tablicy *ISA* to klucze wykorzystywane do sortowania kubełka. Źródło: opracowanie własne na podstawie [Sch07].

Pierwszym krokiem algorytmu jest posortowanie sufiksów według $q$-pierwszych znaków, czyli *q-sortowanie*. Wartość parametru $q$ obliczana jest na podstawie liczby różnych symboli (czyli wielkości alfabetu) w ciągu wejściowym, jej wartość maleje wraz ze wzrostem wielkości słownika sekwencji wejściowej. Następnie obliczana jest przybliżona odwrotna tablica sufiksów *ISA* (kubełki identyfikowane są pozycją ich ostatniego elementu w przybliżonej tablicy sufiksów), nazywana w pracy [Sch07] *tablicą wskaźników na kubełki*.

Algorytm *bpr* działa według schematu *polepszania kubełków wgłąb* i korzysta z techniki *pull*. Sufiksy $S_i$ i $S_j$ należące do jednej *h-grupy* porównywane są poprzez porównanie wartości $ISA[h+i]$ i $ISA[h+j]$.

Każda ze znalezionych na początku grup sortowana jest algorytmem *ternary split quicksort* [BM93]. Parametr $h$ przyjmuje początkowo wartość $q$. Algorytm ten wybiera jeden z sufiksów danej grupy jako element osiowy (ang. *pivot*) $S_p$ o wartości klucza $K_p = ISA[h+p]$. Sufiksy danej grupy dzielone są na trzy podgrupy: sufiksy o wartości klucza mniejszej, równej lub większej niż $K_p$. Każda z tych podgrup jest następnie w ten sam sposób sortowana, przy czym dla grupy sufiksów o wartości klucza równej elementowi osiowemu wartość parametru $h$ zwiększana jest o $q$, ponieważ wszystkie jej sufiksy mają wspólny prefiks długości $h+q$. Wynika to z tego, że elementy kubełka o wspólnym prefiksie długości $h$ sortowane są na podstawie pozycji ich *h-następników*, dla których nie znamy długości wspólnego prefiksu. Wiadomo tylko tyle, że były posortowane na początku działania algorytmu według $q$ pierwszych znaków. Podobnie jak w algorytmie *qsufsort*, równa wartość klucza $ISA[h+i]$ i $ISA[h+j]$ oznacza że sufiksy $S_i$ i $S_j$ mają wspólny prefiks długości $h+q$. Fundamentalna różnica między algorytmami *bpr* i *qsufsort* polega na tym, że pierwszy z nich realizuje schemat *polepszania kubełków wgłąb*, a drugi *polepszania wszerz*. Dzięki temu, algorytm *qsufsort* w kolejnej iteracji może podwajać wartość parametru $h$.

Istotną cechą algorytmu *bpr* jest to, że po każdym podziale grupy na podgrupy wykonywana jest aktualizacja wpisów w tablicy *ISA* odpowiadających jej elementom, zgodnie ze wzorem $ISA[i] = pozycja\ ostatniego\ elementu\ podgrupy$. Na rysunku 4.6 przedstawiony został przebieg działania algorytmu *bpr* na przykładzie ciągu „DEBDEBDEA".

# Rozdział 5

# Testy wydajnościowe

## 5.1 Wstęp

W poniższym rozdziale przedstawione zostaną wyniki testów wydajnościowych implementacji algorytmów opisanych w poprzedniej części pracy (poza algorytmem *two-stage*, który został rozwinięty i poprawiony przez *improved two-stage*). Implementacja w języku `Java` została oparta o oryginalny kod, udostępniony przez autorów. Podstawą implementacji metody *improved two-stage* był kod algorytmu *divsufsort* napisany przez Yutę Moriego [K].

Algorytmy tworzenia tablic sufiksów zaimplementowane zostały w ten sposób, żeby na wejściu przyjmowały sekwencje liczb całkowitych typu `int` zajmujących w języku `Java` 4 bajty. Spowodowało to zwiększenie złożoności pamięciowej algorytmów (wartości podane w tabeli 3.1 zakładały wejście w postaci sekwencji jednobajtowych elementów). Powodem zwiększenia rozmiaru pojedynczego elementu wejściowego było umożliwienie tworzenia tablic sufiksów dla sekwencji symboli z alfabetów o dużym rozmiarze. Nie jest to jednak możliwe w praktyce w przypadku wszystkich algorytmów; rzeczywista złożoność pamięciowa implementacji wybranych metod oraz ich zależność od rozmiaru alfabetu wejściowego opisane zostały w tabeli 5.1. Testowane implementacje algorytmów *bpr*, *deep-shallow* i *qsufsort* dla zaoszczędzenia pamięci nadpisywały ciąg wejściowy. Na potrzeby testów wydajnościowych zaimplementowano również naiwny algorytm tworzenia tablic sufiksów opierający się o zwykły algorytm sortujący *quicksort*. W dalszej części rozdziału metoda naiwna nazywana będzie *naive sort* lub NS.

Wykonane testy algorytmów można podzielić na dwie główne kategorie: testy na wejściu generowanym losowo, oraz testy na wejściu wczytywanym z plików. Testy drugiego typu wykonywane były na kilku różnych maszynach wirtualnych:

- maszyna Java HotSpot 64-Bit Server VM 11.0-b16 firmy Sun Microsystems, oznaczana w dalszej części pracy jako `sun`,
- maszyna IBM J9 VM 2.4 firmy IBM, oznaczana w dalszej części pracy jako `ibm`,
- BEA JRockit(R) R27.5.0-110_o-99226-1.6.0_03 firmy BEA (aktualnie przejęta przez Oracle), oznaczana w dalszej części pracy jako `jrockit`,
- Apache Harmony DRLVM 11.2.0 tworzona przez Apache Software Foundation, oznaczana w dalszej części pracy jako `harmony`.

Testy przeprowadzone zostały na kilku komputerach testowych o następujących parametrach:

- dwurdzeniowy procesor Athlon 5200 o prędkości 2.6 GHz, 64-bitowy system operacyjny OpenSuSE, jądro w wersji 2.6.22.19-0.2-default,

| Nazwa | Złożoność pamięciowa |
|---|---|
| *skew* | $16n$ |
| *bpr* | $4\|\Sigma\|^3 + 12n$ |
| *deep-shallow* | $4\|\Sigma\|^2 + (x+8)n$ |
| *divsufsort* | $4\|\Sigma\|^2 + 8n$ |
| *qsufsort* | $8n$ |
| *naive sort* | $8n$ |

TABLICA 5.1: Zaimplementowane algorytmy tworzenia tablic sufiksów. Parametr $n$ oznacza długość wejścia, parametr $|\Sigma|$ oznacza wielkość alfabetu sekwencji wejściowej. Algorytm *deep-shallow* wymaga alfabetu o rozmiarze nie większym niż 256, zużycie pamięci przez ten algorytm zależy od wielkości budowanego drzewa *blind trie*, które maksymalnie może zawierać $n$ elementów. Rozmiar jednego elementu drzewa wynosi $x = 48$ bajtów na 64-bitowej maszynie wirtualnej firmy Sun. Algorytmy *bpr* i *divsufsort* nie mają sztywnych ograniczeń na rozmiar alfabetu, ale ze względu na wydajność nie powinny być używane na dużych alfabetach.

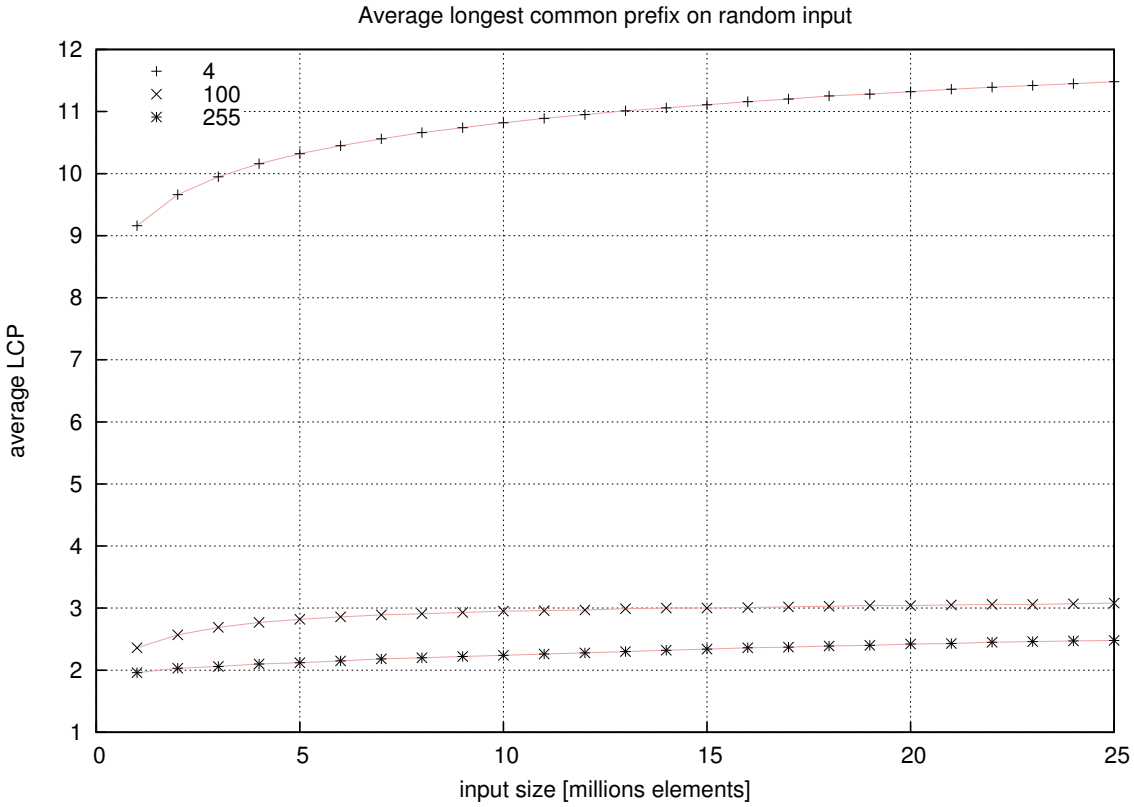
- czterordzeniowy procesor Intel Xeon x3230 o prędkości 2.66 GHz, 64-bitowy system operacyjny OpenSuSE, wersja jądra 2.6.22.19-0.2-default,
- jednordzeniowy Intel Pentium 4 o prędkości 3 GHz, 32-bitowy system operacyjny Windows XP.

Wszystkie obliczenia przeprowadzone zostały z parametrami maszyn wirtualnych odpowiadającym opcjom „`-Xmx2g -server`” – zwiększają one maksymalny rozmiar sterty do 2Gb, a maszyna wirtualna pracuje w trybie „serwerowym” (agresywna optymalizacja JIT).

Czas działania algorytmów mierzony był przez klasę uruchamiającą testy, poprzez liczenie różnicy wartości zwracanych przez metodę `System.currentTimeMillis()` przed i po obliczeniach (tzw. *wall time*). Do mierzenia zużycia pamięci wykorzystano specjalnie w tym celu napisany aspekt [B]. Mierzone było zużycie pamięci wewnątrz wirtualnej maszyny na początku działania algorytmu, pod jego koniec oraz po zakończeniu wybranych metod wewnątrz klas. Mierzenie zużycia pamięci w języku `Java` jest trudne, bowiem maszyna wirtualna nie pozwala na dokładne oszacowanie wszystkich dokonanych alokacji pamięci. Do celów testowych posłużono się programowaniem aspektowym i „wpleciono” (ang. *code weaving*) dynamiczne instrukcje mierzące różnicę w aktualnej ilości zaalokowanej pamięci względem startu algorytmu. Nie jest to oszacowanie idealne, ale pozwala na zgrubne stwierdzenie ile pamięci zużywa dany algorytm.

Test na jednej instancji wejścia (pliku lub wygenerowanej sekwencji o ustalonych parametrach) powtarzany był 10 razy, a jego wynikiem jest średnia z zebranych pomiarów. Testy których wyniki zbierano poprzedzano kilkoma uruchomieniami algorytmu. Celem tego zabiegu było wyeliminowanie błędów pomiarowych wynikających z wolniejszego działania kodu, który nie został skompilowany do kodu natywnego przez maszynę wirtualną. Testy na wejściu generowanym losowo były poprzedzone 10 rundami „rozgrzewki", natomiast przed testem na wejściu wczytywanym z pliku algorytm uruchamiany był 5 razy.

W dalszej części rozdziału prezentowane są wyniki pochodzące z pomiarów na komputerze z procesorem Xeon x3230. Wyniki z pozostałych maszyn są niemal identyczne pod względem porównywania i rankingu algorytmów, dlatego też w wielu miejscach je pominięto (wszystkie wyniki są na dołączonej do pracy płycie CD). Samo porównanie efektywności wykonania programów w różnych maszynach wirtualnych znajduje się w rozdziale 5.3.2.




RYSUNEK 5.1: Średnie *lcp* w zależności od długości wejścia dla alfabetów o wielkości 4, 100 i 255 symboli.

## 5.2 Testy wydajnościowe na losowo generowanym wejściu

Celem testów na losowo generowanym wejściu jest zbadanie rzeczywistej zależności algorytmów od długości wejścia, wielkości alfabetu oraz średniego *lcp* ciągu wejściowego. Testy przeprowadzono tylko na jednej maszynie wirtualnej (`sun`). Ziarno generatora liczb losowych otrzymywało identyczną wartość przed testem każdego algorytmu by zapewnić powtarzalność wyników.

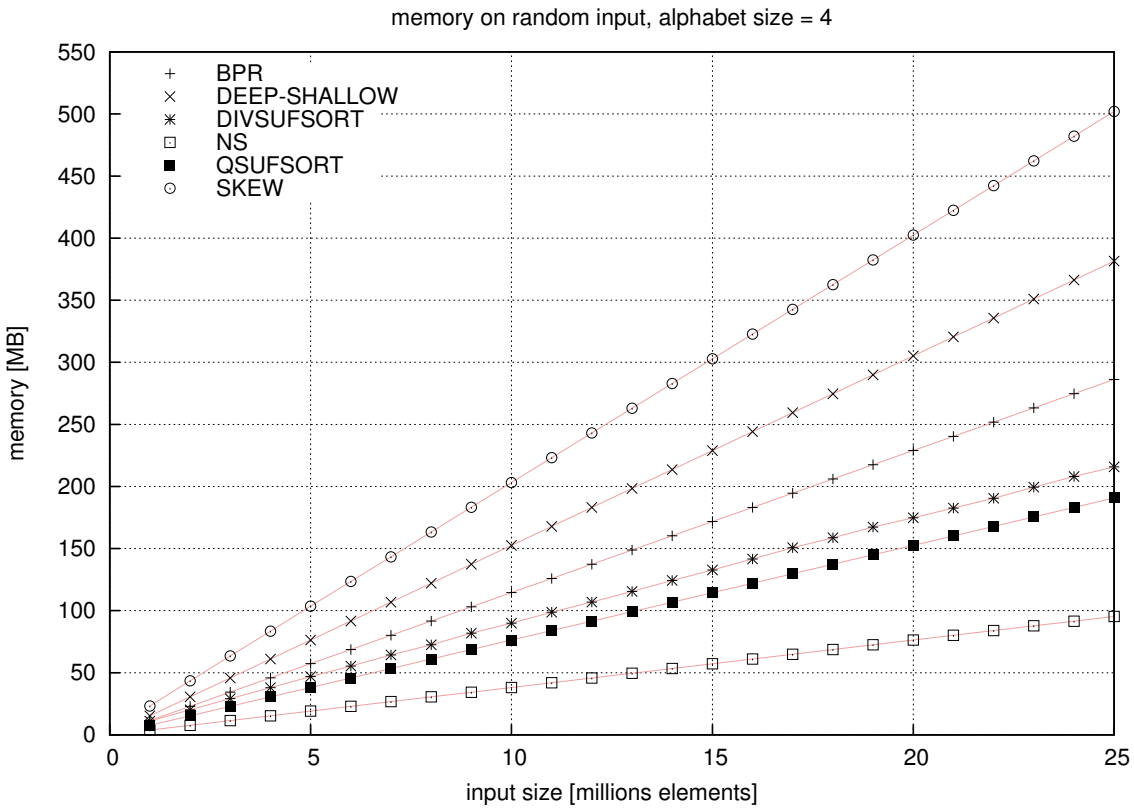
### 5.2.1 Wejście o zmiennej długości i stałej wielkości alfabetu

Testy algorytmów na losowym wejściu o zmiennej długości i stałej wielkości alfabetu powtórzono trzy razy. Każdy test wykonany był dla wejścia generowanego z alfabetu wielkości 4,100 i 255 elementów. Rysunek 5.1 przedstawia średnią wartość *lcp* generowanych ciągów wejściowych. Z rysunku wynika, że wartość średniego *lcp* jest mniej więcej odwrotnie proporcjonalna do wielkości alfabetu.

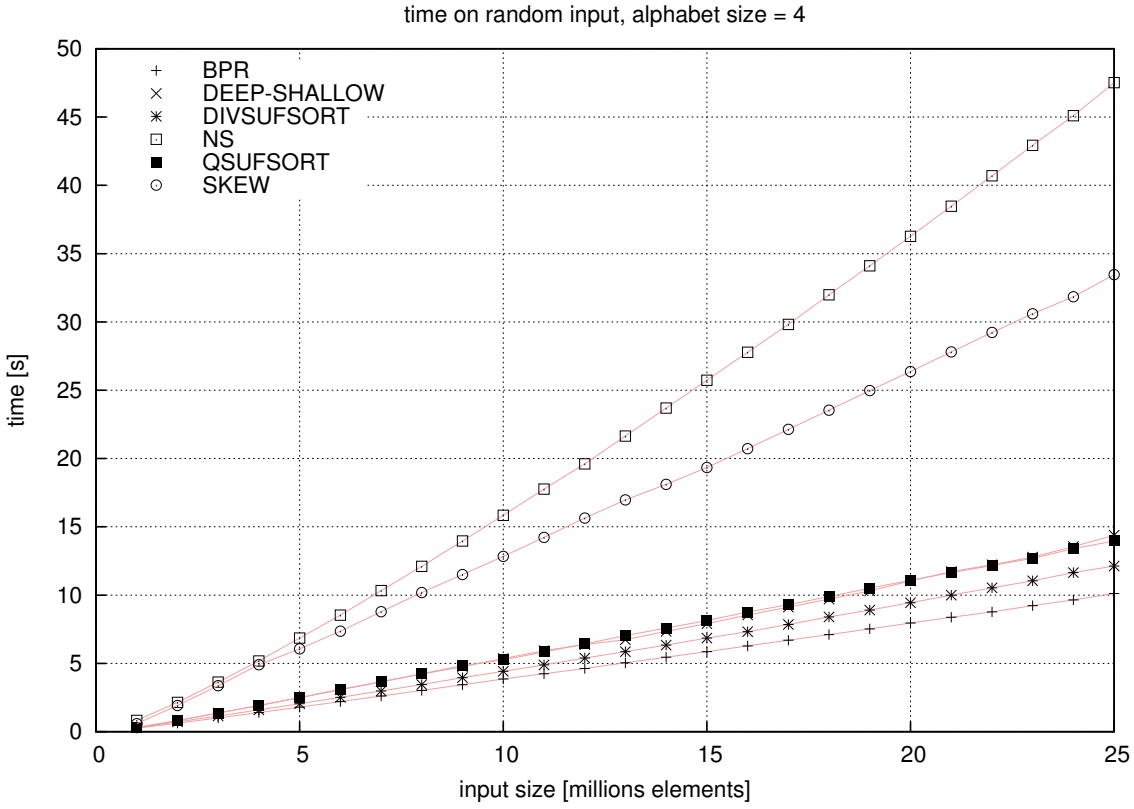
Rysunki 5.3 i 5.2 przedstawiają czas działania algorytmów oraz zużycie pamięci dla alfabetu wielkości 4. Algorytmy *qsufsort* i *naive sort* uzyskały najlepszy wynik złożoności pamięciowej. Zgodnie z przewidywaniami, algorytmy *skew* i *deep shallow* wypadł najgorzej w tej kwestii. Najszybszym algorytmem okazał się być algorytm *bpr*. Wyraźnie wolniejsze od pozostałych algorytmów są algorytmy *skew* i *naive sort*.

Wyniki testów wejścia generowanego z alfabetu wielkości 100 przedstawione są na rysunku 5.5 i 5.4. Rezultaty testu czasu działania algorytmu są niemal identyczne jak poprzedniego. Jedyną różnicą jest to, że algorytm *bpr* nie uzyskał najlepszego wyniku lecz znalazł się wśród kilku algorytmów uzyskujących bardo dobry wynik, zajął również więcej pamięci.

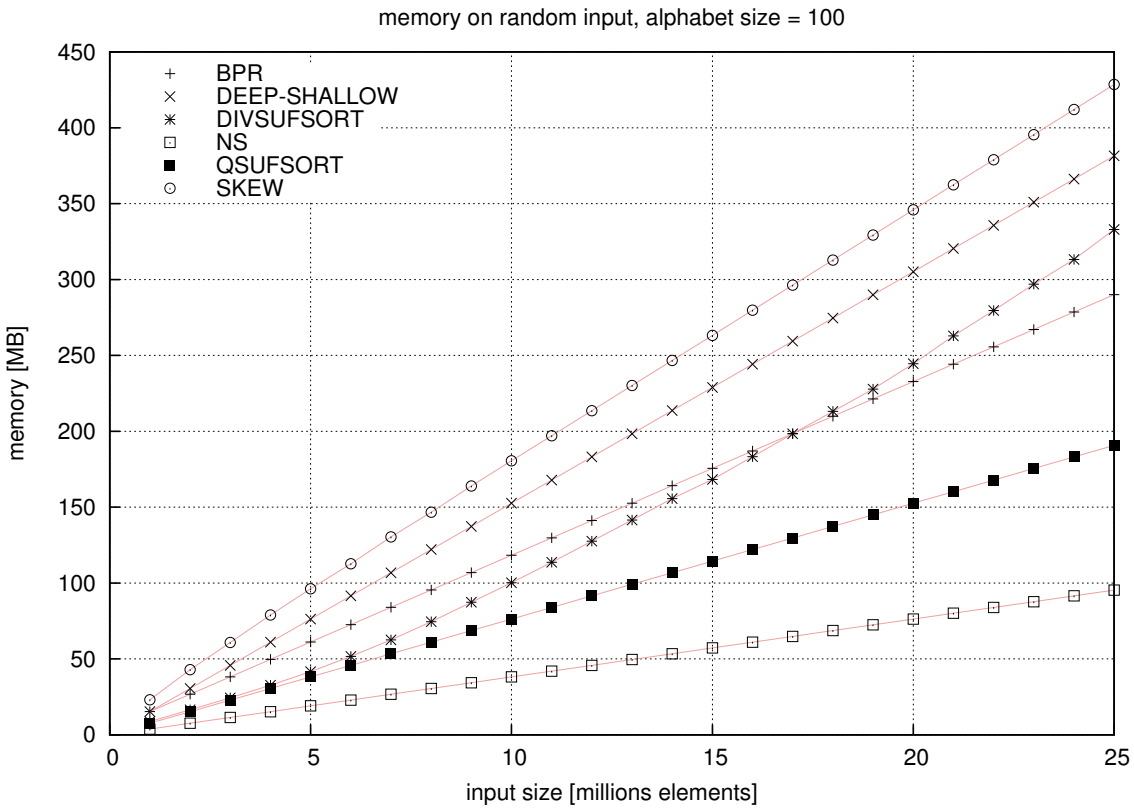
Rysunki 5.7 i 5.6 prezentują wyniki testów wejścia generowanego z alfabetu wielkości 255. Wy-



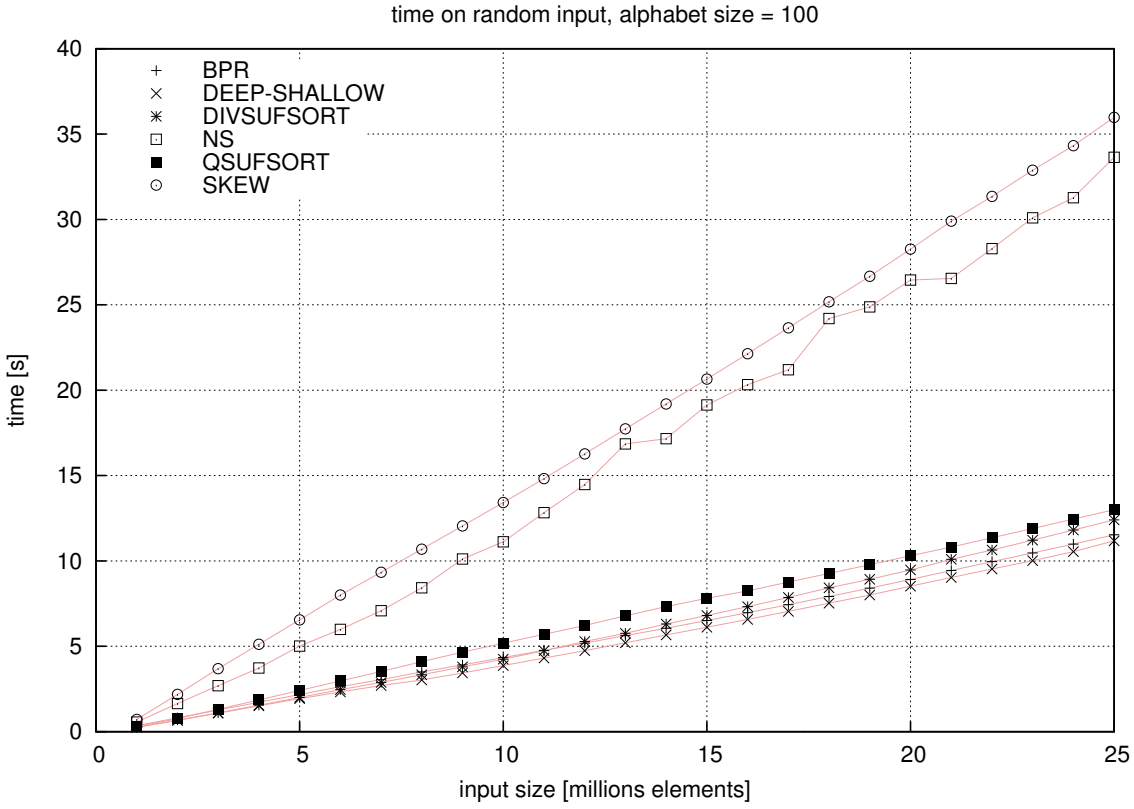

RYSUNEK 5.2: Zużycie pamięci w zależności od długości wejścia generowanego z alfabetu wielkości 4.



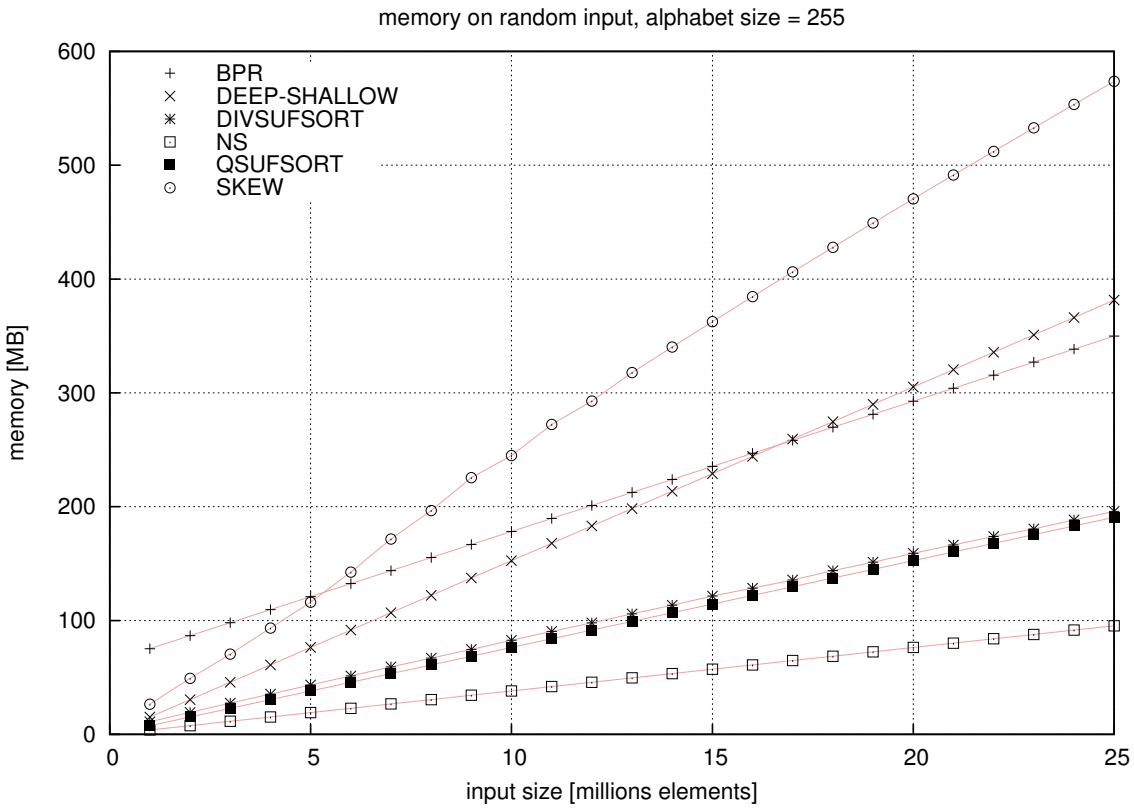

RYSUNEK 5.3: Czas działania algorytmów w zależności od długości wejścia generowanego z alfabetu wielkości 4.




RYSUNEK 5.4: Zużycie pamięci w zależności od długości wejścia generowanego z alfabetu wielkości 100.




RYSUNEK 5.5: Czas działania algorytmów w zależności od długości wejścia generowanego z alfabetu wielkości 100.



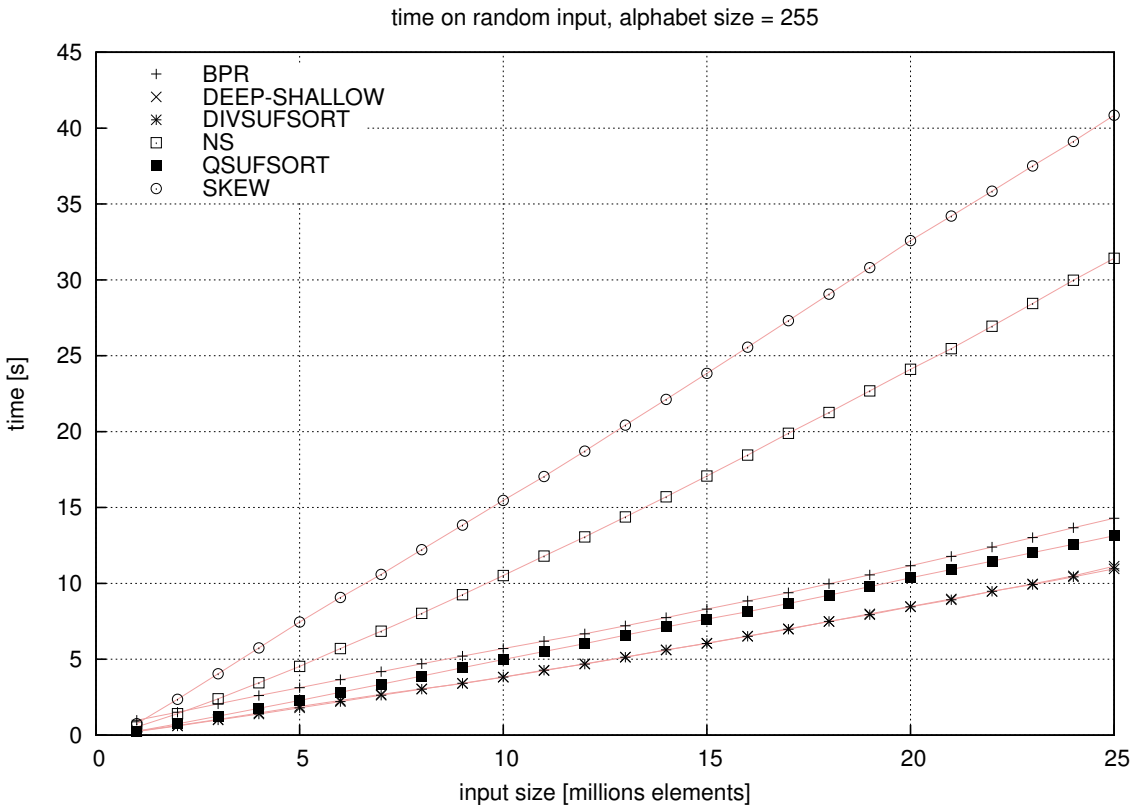

RYSUNEK 5.6: Zużycie pamięci w zależności od długości wejścia generowanego z alfabetu wielkości 255.

nik tego testu są również podobne do poprzedników. Algorytm *bpr* uzyskał jeszcze gorszy wynik niż w poprzednich testach, zajął więcej pamięci i działał wolniej od większości algorytmów.

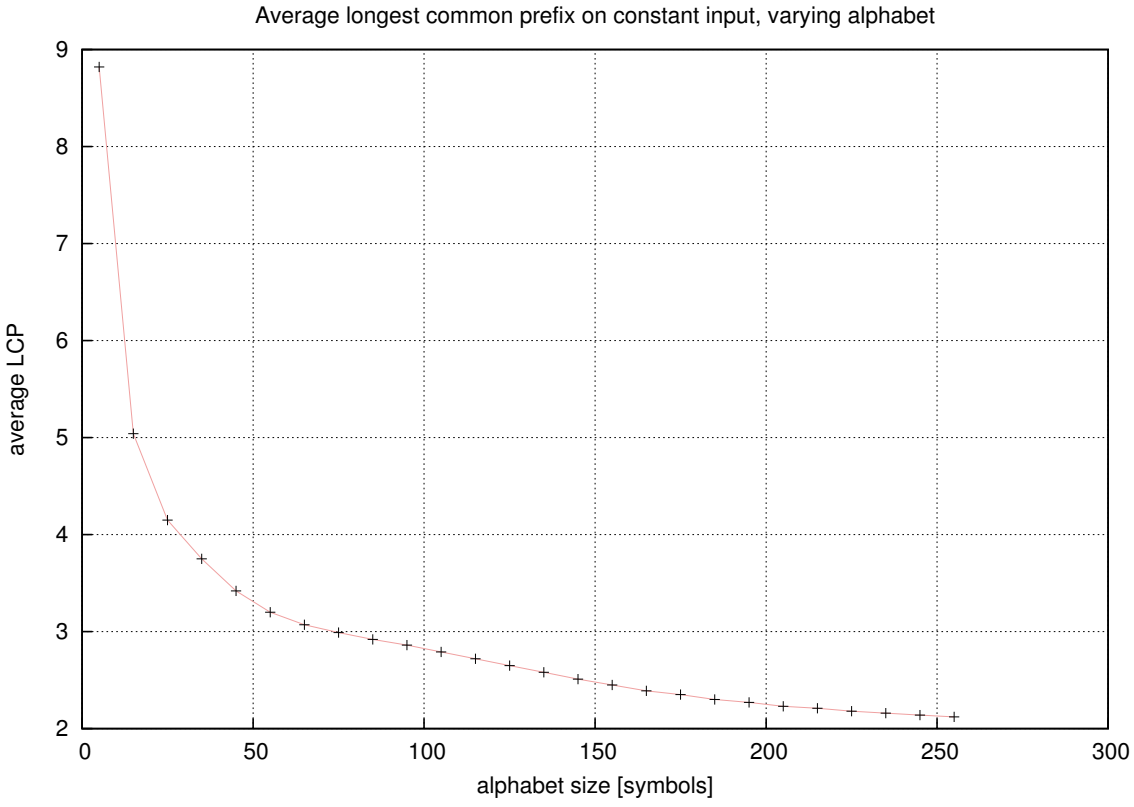
### 5.2.2 Wejście o stałej długości i zmiennej wielkości alfabetu

Wejście generowane na potrzeby testu miało długość 5 000 000 elementów. Wykres 5.8 przedstawia średnie *lcp* wygenerowanego wejścia w zależności od wielkości alfabetu. Wykres 5.9 przedstawia czasy działania algorytmów na generowanym wejściu.

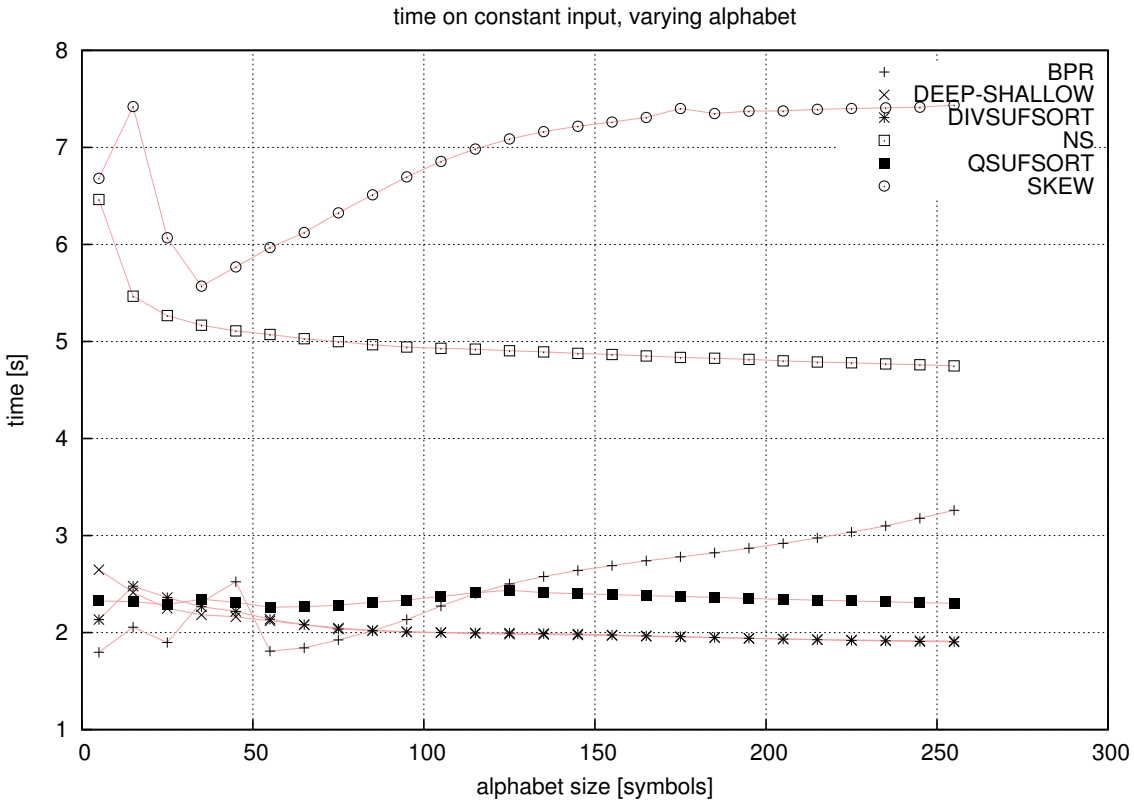
Wyniki testu pokazują które algorytmy zależą od średniego *lcp* i wielkości alfabetu sekwencji wejściowej. Algorytm *bpr* jest tego najlepszym przykładem – im większa jest jedna z tych wartości, tym gorzej sobie radzi. Algorytm *skew* uzyskuje lepsze wyniki dla sekwencji o większej wartości średniego *lcp*. Pozostałe algorytmy nie wykazują większej zależności pomiędzy czasem ich działania a wielkością alfabetu i średnim *lcp*.




RYSUNEK 5.7: Czas działania algorytmów w zależności od długości wejścia generowanego z alfabetu wielkości 255.




RYSUNEK 5.8: Średnie *lcp* generowanego wejścia w zależności od wielkości alfabetu.




RYSUNEK 5.9: Czas działania algorytmów w zależności od wielkości alfabetu.

## 5.3 Testy wydajnościowe na wejściu wczytywanym z plików

Zaimplementowane algorytmy przetestowane zostały na dwóch zestawach plików (korpusach) specjalnie przygotowanych do testowania algorytmów tworzenia tablic sufiksów. Pierwszy z nich nosi nazwę `Gauntlet`, powstał z inicjatywy Michaela Maniscalco. Ideą stojącą za stworzeniem tego korpusu było zebranie plików o nietypowej strukturze w celu testowania algorytmów na szczególnych przypadkach wejścia. Wkład w powstanie tego zbioru plików wnieśli Yuta Mori, Simon Puglisi oraz Graham Houston. Korpus dostępny jest do pobrania pod adresem [C].

Drugi korpus testowy opracowany został przez Giovanniego Manziniego. W jego skład wchodzą rzeczywiste pliki o dużym rozmiarze pochodzące z różnych źródeł. Zestaw plików dostępny jest do pobrania pod adresem [D]. Tabela 5.2 prezentuje charakterystykę plików obu korpusów.

| Plik | Rozmiar | Średnie *lcp* |
|---|---|---|
| `abac` | 200 000 | 99 997 |
| `abba` | 10 500 600 | 2 773 939 |
| `book1x20` | 15 375 420 | 6 938 159 |
| `fib_s14930352` | 14 930 352 | 3 940 597 |
| `fss10` | 12 078 908 | 2 454 179 |
| `fss9` | 2 851 443 | 579 353 |
| `houston` | 3 840 000 | 52 083 |
| `paper5x80` | 981 924 | 239 421 |
| `test1` | 2 097 152 | 1 048 064 |
| `test2` | 2 097 152 | 1 048 064 |
| `test3` | 2 097 152 | 984 064 |

| Plik | Rozmiar | Średnie *lcp* |
|---|---|---|
| `chr22.dna` | 34 553 758 | 1 979 |
| `etext99` | 105 277 340 | 1 108 |
| `gcc-3.0.tar` | 86 630 400 | 8 603 |
| `howto` | 39 422 105 | 267 |
| `jdk13c` | 69 728 899 | 678 |
| `linux-2.4.5.tar` | 116 254 720 | 479 |
| `rctail96` | 114 711 151 | 282 |
| `rfc` | 116 421 901 | 93 |
| `sprot34.dat` | 109 617 186 | 89 |
| `w3c2` | 104 201 579 | 42 299 |

TABLICA 5.2: Charakterystyka plików wchodzących w skład korpusu `Gauntlet` (po lewej) i korpusu Giovanniego Manziniego (po prawej). Rozmiar i średnie *lcp* podano w bajtach.

### 5.3.1 Szczegółowe wyniki na maszynie wirtualnej firmy Sun

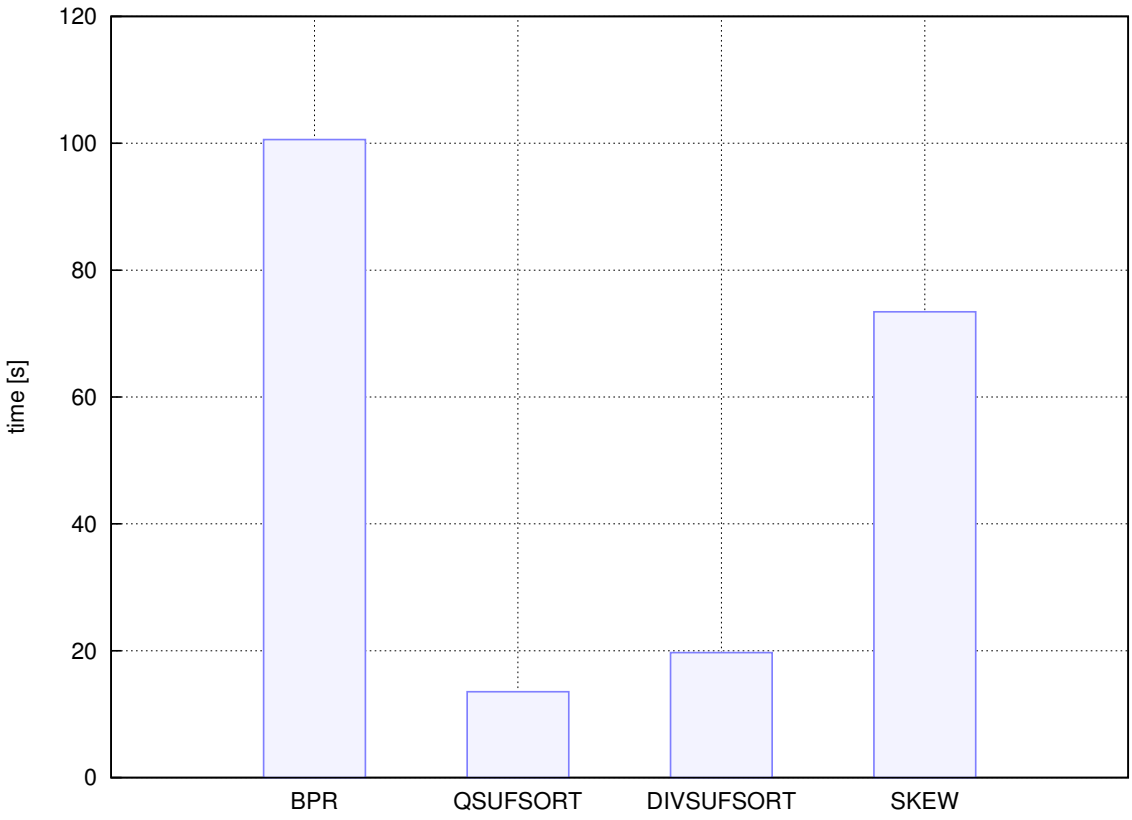
Ze względu na wysokie podobieństwo wyników, w poniższym rozdziale prezentowane są szczegółowe rezultaty tylko z jednej maszyny wirtualnej (`sun`). Podobnie jak w przypadku testów na różnych komputerach, wyniki testów na różnych maszynach wirtualnych są identyczne w sensie rankingu algorytmów.

Algorytm *deep-shallow* został pominięty w poniższych testach ze względu na nieakceptowalnie długi czas działania. Testy z jego udziałem wykazały wielokrotnie gorszy czas działania algorytmu od najgorszego z pozostałych. Również ze względu na czas działania algorytmu pominięto metodę naiwną. Algorytmy *skew* i *bpr* podczas przetwarzania niektórych plików kończyły się wyjątkiem braku pamięci. Takie przypadki oznaczone zostały w zestawieniach znakiem –, błąd działania algorytmu na jednym z plików danego korpusu powodował pominięcie tej metody w podsumowaniu testów na całym zbiorze plików.

Wyniki testów na korpusie `The Gauntlet` przedstawione zostały w tabeli 5.3. Rysunek 5.10 obrazuje porównanie sumy czasów działania algorytmów na wszystkich plikach korpusu (poza tymi algorytmami, które błędnie zakończyły działanie). Analogiczne wyniki testów na korpusie Giovanniego Manziniego przedstawione są w tabeli 5.4 oraz rysunku 5.11. Najlepszym algorytmem okazał się być *qsufsort*, który uzyskał najlepszy wynik na znakomitej większości plików. Drugie miejsce przypadło algorytmowi *divsufsort*. Pozostałe algorytmy uzyskiwały znacznie gorsze wyniki lub nie kończyły obliczeń.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `abac` | 1.07 | 0.01 | **0.00** | 0.03 |
| `abba` | 3.80 | 2.49 | **2.34** | 12.09 |
| `book1x20` | **3.36** | 4.11 | 3.44 | 23.23 |
| `fib_s14930352` | 9.68 | 6.08 | **3.27** | 15.88 |
| `fss10` | 5.18 | 4.84 | **2.65** | 13.01 |
| `fss9` | 1.01 | 0.74 | **0.57** | 2.14 |
| `houston` | 2.29 | **0.18** | 0.43 | 1.20 |
| `paper5x80` | 0.14 | 0.12 | **0.06** | 0.47 |
| `test1` | 2.37 | 0.40 | **0.21** | 2.17 |
| `test2` | 0.81 | 0.30 | **0.21** | 2.18 |
| `test3` | 70.87 | 0.45 | **0.36** | 1.04 |
| Total | 100.58 | 19.70 | **13.55** | 73.44 |

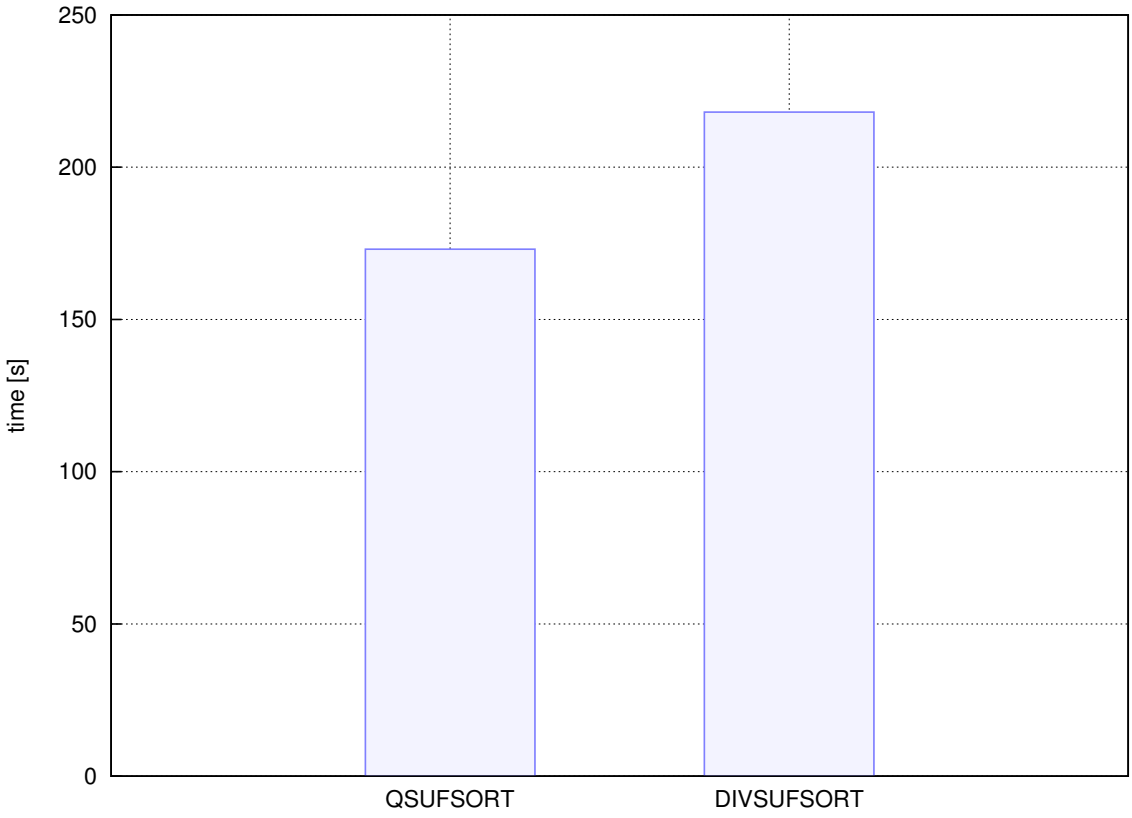
TABLICA 5.3: Czas działania algorytmów na plikach z korpusu `Gauntlet`.




RYSUNEK 5.10: Sumaryczny czas działania algorytmów na plikach z korpusu `Gauntlet`.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `chr22.dna` | **6.85** | 8.56 | 7.70 | 64.78 |
| `etext99` | — | 30.66 | **25.56** | — |
| `gcc-3.0.tar` | 20.73 | 18.62 | **14.22** | — |
| `howto` | 8.07 | 9.03 | **7.45** | 83.77 |
| `jdk13c` | 15.63 | 15.44 | **11.58** | — |
| `linux-2.4.5.tar` | — | 23.94 | **19.98** | — |
| `rctail96` | — | 29.96 | **23.89** | — |
| `rfc` | — | 26.96 | **23.78** | — |
| `sprot34.dat` | — | 31.66 | **22.24** | — |
| `w3c2` | — | 23.25 | **16.71** | — |
| Total | | 218.09 | **173.10** | |

TABLICA 5.4: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego.




RYSUNEK 5.11: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego.

### 5.3.2 Porównanie czasów działania algorytmów na różnych maszynach wirtualnych

Poniżej przedstawione są wyniki podsumowań czasów działania algorytmów na korpusach testowych. Jak już wspomniano wcześniej, ranking algorytmów jest identyczny dla każdej maszyny wirtualnej. Celem prezentowania zestawień jest znalezienie takiej maszyny wirtualnej, na której algorytmy działają najszybciej. Tabele 5.6 i 5.5 zawierają czasy działań algorytmów na plikach z korpusów Giovanniego Manziniego i `The Gauntlet`. Ze względu na bardzo niskie wartości odchylenia standardowego nie umieszczono go tabelach. Wizualne porównanie sum tych wyników znajduje się na rysunkach 5.13 i 5.12.

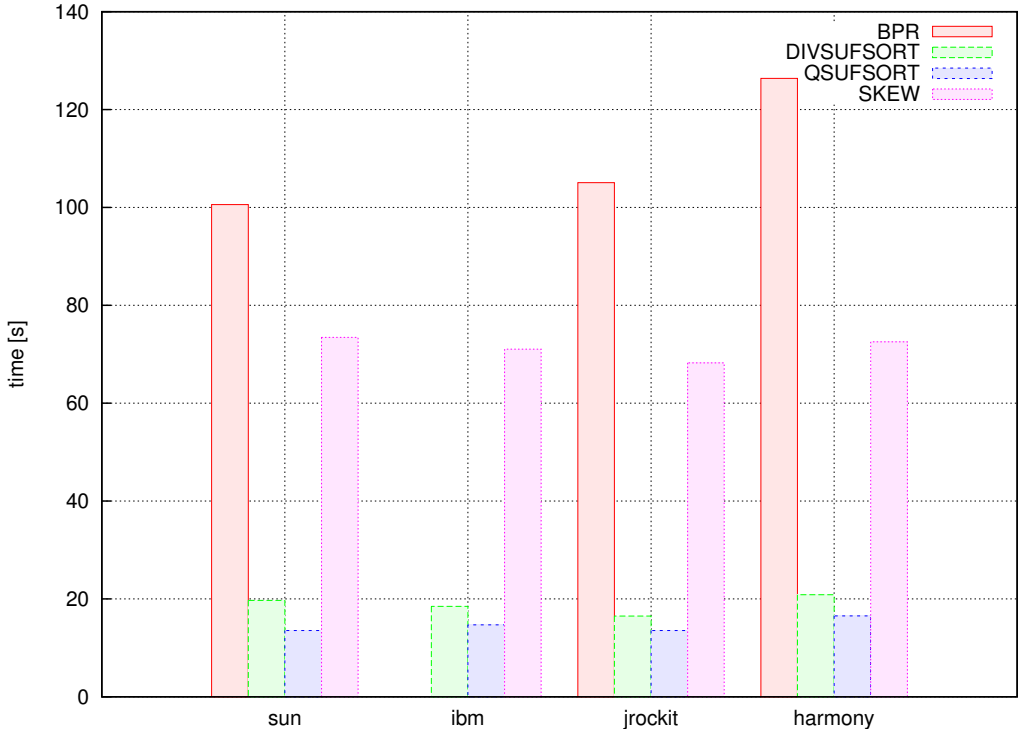
Testowane maszyny wirtualne wypadły bardzo podobnie. Na maszynie `harmony` algorytmy działały wolniej, wyniki pozostałych trzech maszyn są do siebie zbliżone. Spośród tych wyników wyróżnia się jednak wynik testu algorytmu *divsufsort* na korpusie Manziniego przeprowadzonego na maszynie wirtualnej `jrockit` – dwukrotnie gorszy niż na pozostałych maszynach. Wyniki w tabeli A.4 pokazują wolniejsze działania algorytmu na każdym z plików wejściowych. Analiza zapisów przebiegu działania algorytmu wykazała bardzo duże odchylenie standardowe czasu działania algorytmu, co sugeruje wpływ maszyny wirtualnej (*JIT*, *garbage collector*) lub jakiś inny proces obliczeniowy działające w tle i zaburzający pomiar czasu.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `sun` | 100.58 | 19.70 | 13.55 | 73.44 |
| `ibm` | — | 18.49 | 14.71 | 71.02 |
| `jrockit` | 105.05 | 16.49 | 13.55 | 68.24 |
| `harmony` | 126.39 | 20.85 | 16.51 | 72.53 |

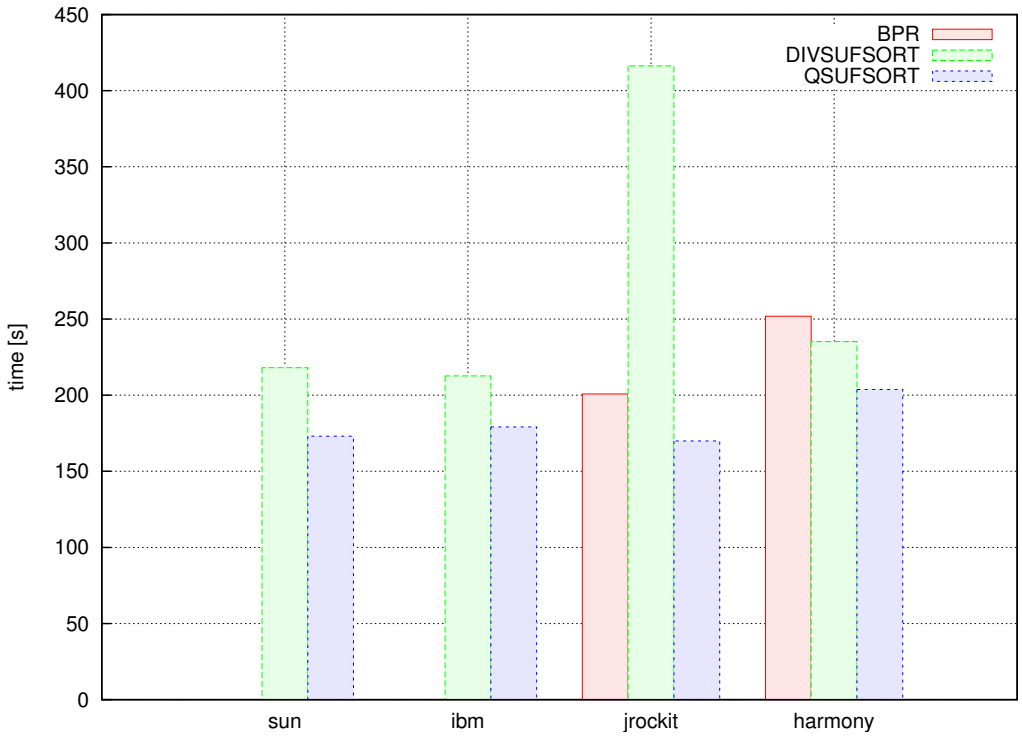
TABLICA 5.5: Czasy działania algorytmów na korpusie `Gauntlet` na różnych maszynach wirtualnych. Podane wartości wyrażone są w sekundach.

| | *bpr* | *divsufsort* | *qsufsort* |
|---|---|---|---|
| `sun` | — | 218.09 | 173.10 |
| `ibm` | — | 212.62 | 179.12 |
| `jrockit` | 200.84 | 416.30 | 170.00 |
| `harmony` | 251.82 | 235.16 | 203.76 |

TABLICA 5.6: Czasy działania algorytmów na korpusie Giovanniego Manziniego na różnych maszynach wirtualnych. Podane wartości wyrażone są w sekundach.




RYSUNEK 5.12: Porównanie czasu działania algorytmów na korpusie `Gauntlet` na różnych maszynach wirtualnych.




RYSUNEK 5.13: Porównanie czasu działania algorytmów na korpusie Giovanniego Manziniego na różnych maszynach wirtualnych.

## 5.4 Analiza wyników

Najlepszym spośród zaimplementowanych algorytmów okazał się algorytm *qsufsort*. Cechuje się on odpornością na specyficzne typy danych wejściowych, nie zależy w dużym stopniu ani od rozmiaru alfabetu sekwencji wejściowej ani od wartości średniego *lcp*. Dodatkowym atutem jest brak użycia dodatkowej pamięci ponad tą wymaganą do przechowania wejścia i wyjścia algorytmu (oraz stosu zużytego podczas rekurencyjnego sortowania). Na drugim miejscu znalazł się algorytm *divsufsort* ustępujący zwycięzcy zarówno pod względem czasu działania, jak i zużycia pamięci. Pozostałe algorytmy wypadły dużo gorzej niż wymieniona dwójka.

Co ciekawe, publikowane w internecie wyniki testów wydajnościowych implementacji algorytmów w języku `C++` [F, E] dają nieco inny obraz rankingu tych algorytmów. Najlepsze wyniki uzyskują tam implementacje algorytmu *improved two-stage*, czyli *archon*, *divsufsort* i *msufsort*. Algorytm *qsufsort* uzyskuje dobre, choć wyraźnie gorsze wyniki. Wytłumaczenie przyczyn tego faktu leży zapewne w różnicach między językami `C++` a `Java` (oraz działaniem kodu natywnego i kodu uruchamianego na maszynie wirtualnej). Algorytm *qsufsort* jest prostym algorytmem, zarówno koncepcyjnie jak i implementacyjnie. Przepisanie go z `C++` do `Javy` nie było trudne. Algorytm *divsufsort* jest dużo bardziej złożony (jego implementacja w języku `Java` jest o ponad 2000 linii kodu dłuższa od *qsufsort*). Z powodu wykorzystywania wskaźników oraz instrukcji `#define` w oryginalnej implementacji, kod w języku `Java` różni się od oryginału w wielu miejscach. Wydaje się, że dominujące dla szybkości działania różnice to:

- allokacja złożonych struktur danych (i tablic lokalnych) następuję w języku `Java` na stercie, a nie na stosie; oprócz samego narzutu allokacji dochodzi tu również koszt procesu oczyszczania pamięci (*garbage collector*);
- wskaźniki z języków niskopoziomowych muszą być modelowane jako indeksy nad tablicami typów prostych, co w języku `Java` wiąże się z dodatkowym narzutem sprawdzenia czy indeks nie przekracza rozmiaru tablicy;
- rozmiar kodu natywnego generowanego dynamicznie w języku `Java` prawdopodobnie przekraczał wielokrotnie rozmiar kodu skompilowanego w języku `C`, powodując gorsze użytkowanie pamięci podręcznej procesora.

Algorytm *skew* jako jedyny spośród testowanych algorytmów posiadał liniową teoretyczną złożoność obliczeniową. W testach wydajnościowych wypadł jednak bardzo słabo, zarówno w testach implementacji w języku `Java`, jak i w języku `C++`. Przyczyną takiego zachowania algorytmu jest jego bardzo duża złożoność pamięciowa zwiększana dodatkowo przez rekurencyjne wywoływanie algorytmu. Ponadto, rekurencja wiąże się z dodatkowymi kosztami związanymi z obsługą stosu. Należy również pamiętać, że podane złożoności pozostałych algorytmów są wartościami pesymistycznymi, które są rzadko osiągane w praktyce.

# Rozdział 6

# Podsumowanie i kierunki dalszego rozwoju

Nadrzędnym celem niniejszej pracy było napisanie (w formie biblioteki) implementacji wybranych algorytmów tworzenia tablic sufiksów w celu analizy ich efektywności i zachowania w języku wysokiego poziomu, jakim jest język `Java`. Zadanie to obejmowało zapoznanie się z dostępną literaturą poświęconą tematyce tablic sufiksów, wybór najciekawszych algorytmów oraz ich opisanie. Dodatkowo, w wyniku pracy powinna była powstać klasyfikacja algorytmów tworzenia tablic sufiksów.

Największym wyzwaniem podczas tworzenia pracy było znalezienie najlepszych spośród algorytmów tworzenia tablic sufiksów. Bardzo pomocne były w tym opracowania zawierające zestawienia algorytmów [PST07, Sch07] oraz publikowane w internecie wyniki testów wydajnościowych [F, E]. Pewnym problemem podczas implementacji algorytmów było tłumaczenie do języka `Java` takich konstrukcji języka `C++`, jak instrukcje `define`, wskaźniki oraz inne polecenia preprocesora.

Podsumowując, należy stwierdzić, że wszystkie zakładane cele zostały zrealizowane. Przegląd literatury, opisy algorytmów oraz ich klasyfikacja znalazły się w tekście tej pracy, podobnie jak wyniki testów wydajnościowych powstałej implementacji. Według naszej wiedzy niniejsza praca jest pierwszą publikacją w języku polskim poświęconą w całości tematyce algorytmów tworzenia tablic sufiksów. Zaimplementowana biblioteka programowa jest zaś na dzień dzisiejszy jedyną taką pozycją dedykowaną algorytmom tworzenia tablic sufiksów napisaną w języku `Java`. Kod źródłowy biblioteki jest udostępniony na licencji BSD, można go pobrać ze strony projektu: `http://www.jsuffixarrays.org`.

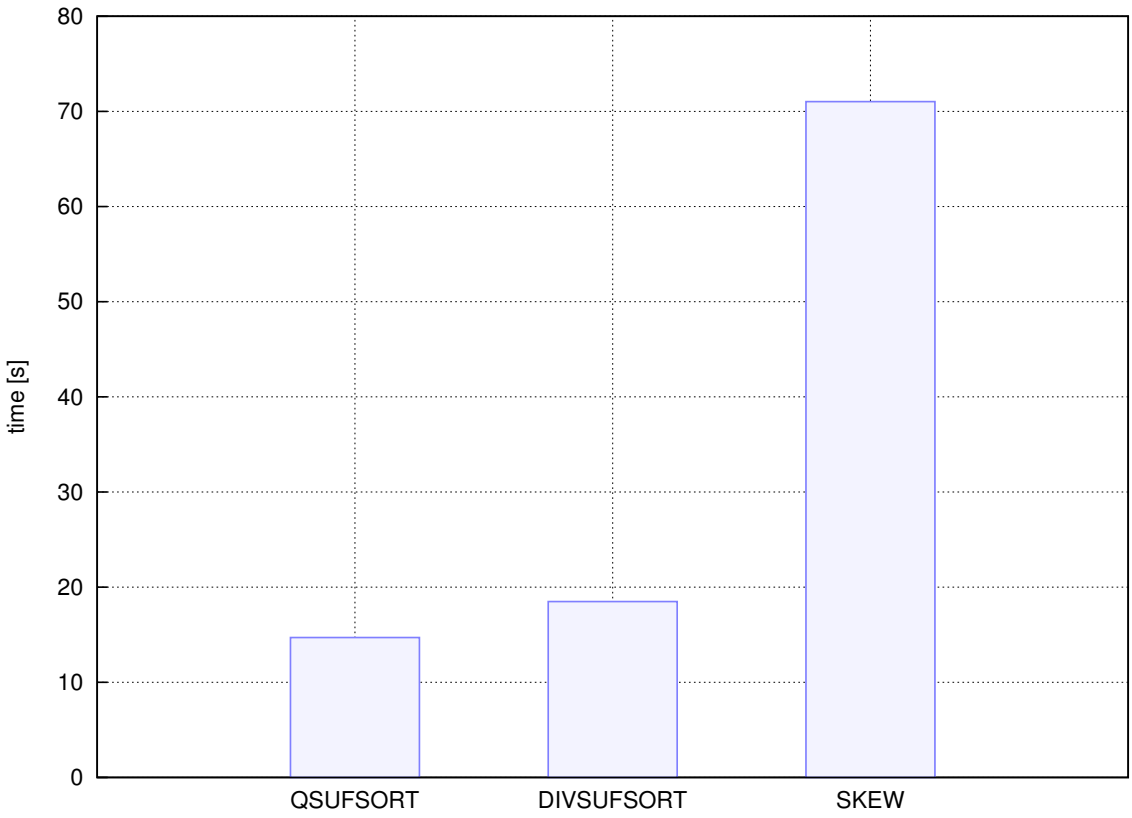
Przewidywany dalszy rozwój biblioteki zakłada opracowanie algorytmu dobierającego najlepszą metodę tworzenia tablic sufiksów na podstawie charakterystyki danych wejściowych (rozmiaru alfabetu i rozkładu symboli w danych wejściowych). Interesującym kierunkiem rozwoju biblioteki jest również stworzenie własnej implementacji algorytmu *improved two-stage*, dostosowanej do specyfiki języka `Java`.

# Dodatek A

# Wyniki dla pozostałych maszyn wirtualnych

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `abac` | — | 0.01 | **0.01** | 0.03 |
| `abba` | 4.13 | 2.76 | **2.51** | 11.71 |
| `book1x20` | **3.48** | 3.76 | 3.67 | 23.45 |
| `fib_s14930352` | 10.47 | 5.48 | **3.59** | 14.46 |
| `fss10` | 5.73 | 4.40 | **2.94** | 12.38 |
| `fss9` | 1.06 | 0.68 | **0.66** | 2.11 |
| `houston` | — | **0.20** | 0.47 | 1.23 |
| `paper5x80` | 0.18 | 0.12 | **0.07** | 0.43 |
| `test1` | 2.39 | 0.37 | **0.21** | 2.16 |
| `test2` | 0.86 | 0.32 | **0.21** | 2.10 |
| `test3` | 71.43 | 0.38 | **0.38** | 0.95 |
| Total | | 18.49 | **14.71** | 71.02 |

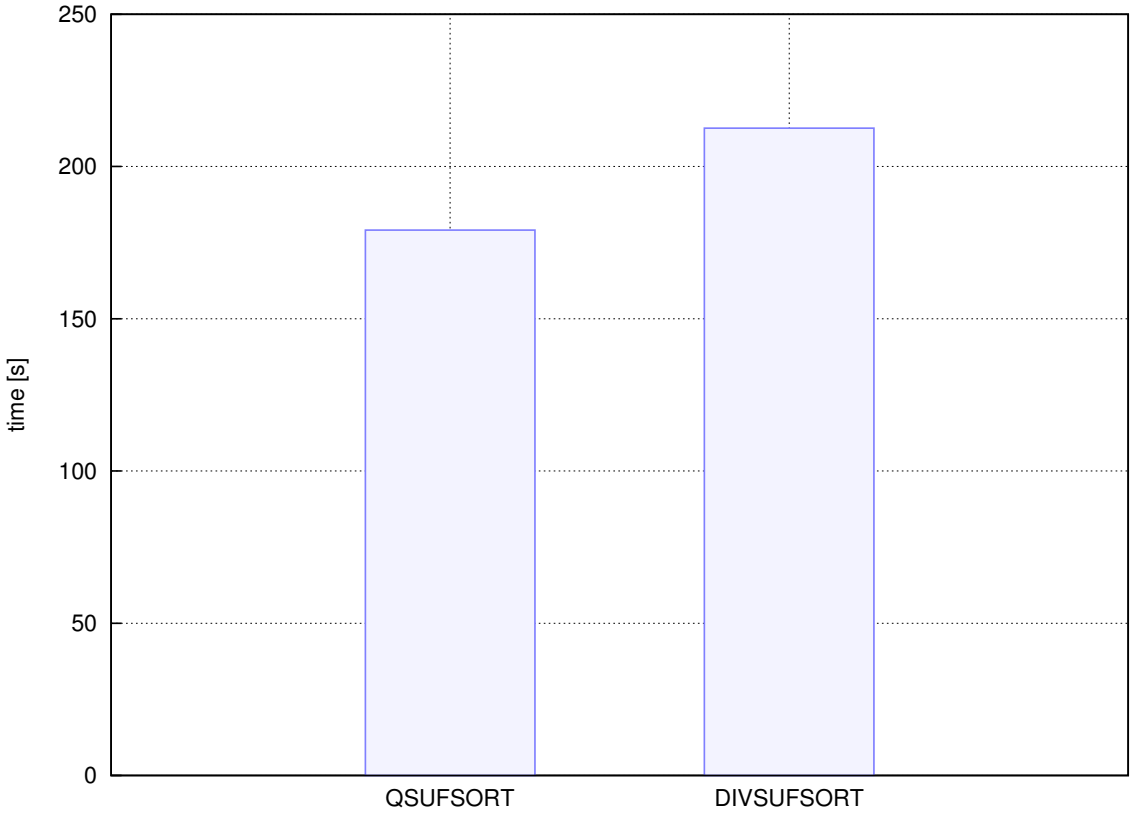
TABLICA A.1: Czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `ibm`.




RYSUNEK A.1: Sumaryczny czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `ibm`.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `chr22.dna` | **7.31** | 8.54 | 8.15 | 60.73 |
| `etext99` | 28.21 | 29.20 | **26.35** | — |
| `gcc-3.0.tar` | — | 17.25 | **14.81** | — |
| `howto` | 8.44 | 10.33 | **7.94** | 79.66 |
| `jdk13c` | 16.44 | 14.74 | **12.36** | — |
| `linux-2.4.5.tar` | 25.60 | 24.09 | **20.30** | — |
| `rctail96` | 30.55 | 28.08 | **24.60** | — |
| `rfc` | 28.93 | 25.92 | **24.43** | — |
| `sprot34.dat` | 28.55 | 28.94 | **22.89** | — |
| `w3c2` | 24.87 | 25.55 | **17.29** | — |
| Total | | 212.62 | **179.12** | |

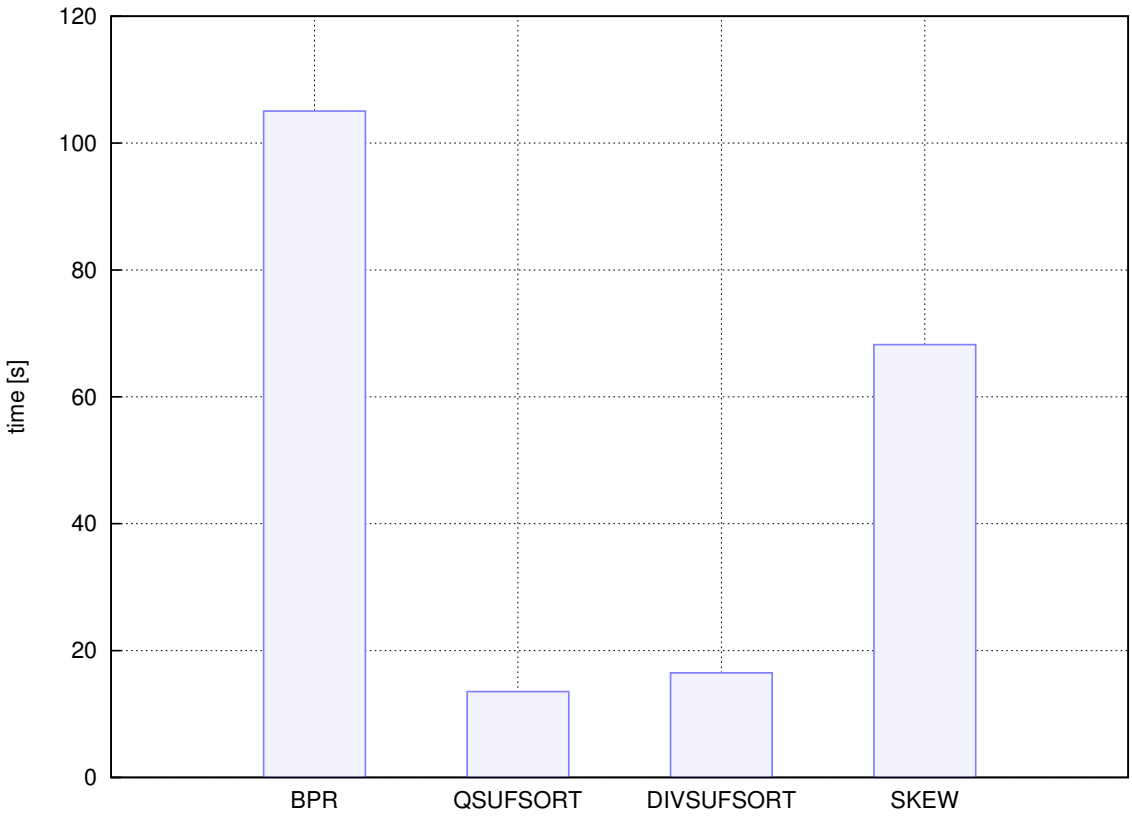
TABLICA A.2: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `ibm`.




RYSUNEK A.2: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `ibm`.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `abac` | 1.04 | 0.01 | **0.01** | 0.05 |
| `abba` | 4.23 | **2.19** | 2.26 | 11.23 |
| `book1x20` | **3.14** | 3.29 | 3.40 | 22.52 |
| `fib_s14930352` | 10.05 | 4.94 | **3.32** | 14.09 |
| `fss10` | 5.36 | 3.88 | **2.69** | 12.34 |
| `fss9` | 1.05 | 0.63 | **0.58** | 1.89 |
| `houston` | 2.63 | **0.24** | 0.44 | 0.96 |
| `paper5x80` | 0.18 | 0.15 | **0.08** | 0.42 |
| `test1` | 2.26 | 0.41 | **0.20** | 1.91 |
| `test2` | 0.71 | 0.34 | **0.20** | 1.91 |
| `test3` | 74.40 | 0.42 | **0.37** | 0.91 |
| Total | 105.05 | 16.49 | **13.55** | 68.24 |

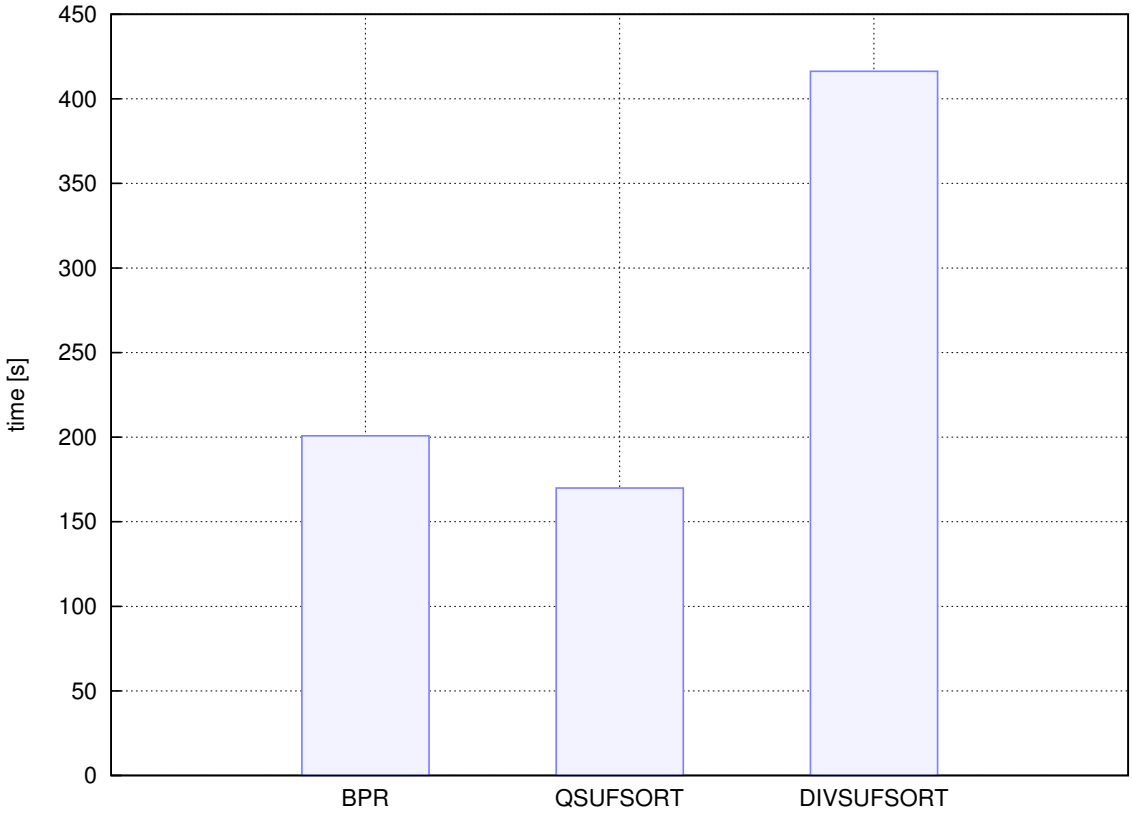
TABLICA A.3: Czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `jrockit`.




RYSUNEK A.3: Sumaryczny czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `jrockit`.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `chr22.dna` | **6.22** | 7.62 | 7.29 | 59.27 |
| `etext99` | 25.38 | 59.93 | **25.21** | — |
| `gcc-3.0.tar` | 19.56 | 37.32 | **14.07** | — |
| `howto` | 7.43 | 8.05 | **7.20** | 78.05 |
| `jdk13c` | 14.50 | 24.81 | **11.44** | — |
| `linux-2.4.5.tar` | 23.68 | 57.76 | **19.53** | — |
| `rctail96` | 28.61 | 56.08 | **23.49** | — |
| `rfc` | 26.42 | 53.69 | **23.39** | — |
| `sprot34.dat` | 26.51 | 65.53 | **21.90** | — |
| `w3c2` | 22.53 | 45.49 | **16.43** | — |
| Total | 200.84 | 416.30 | **169.95** | |

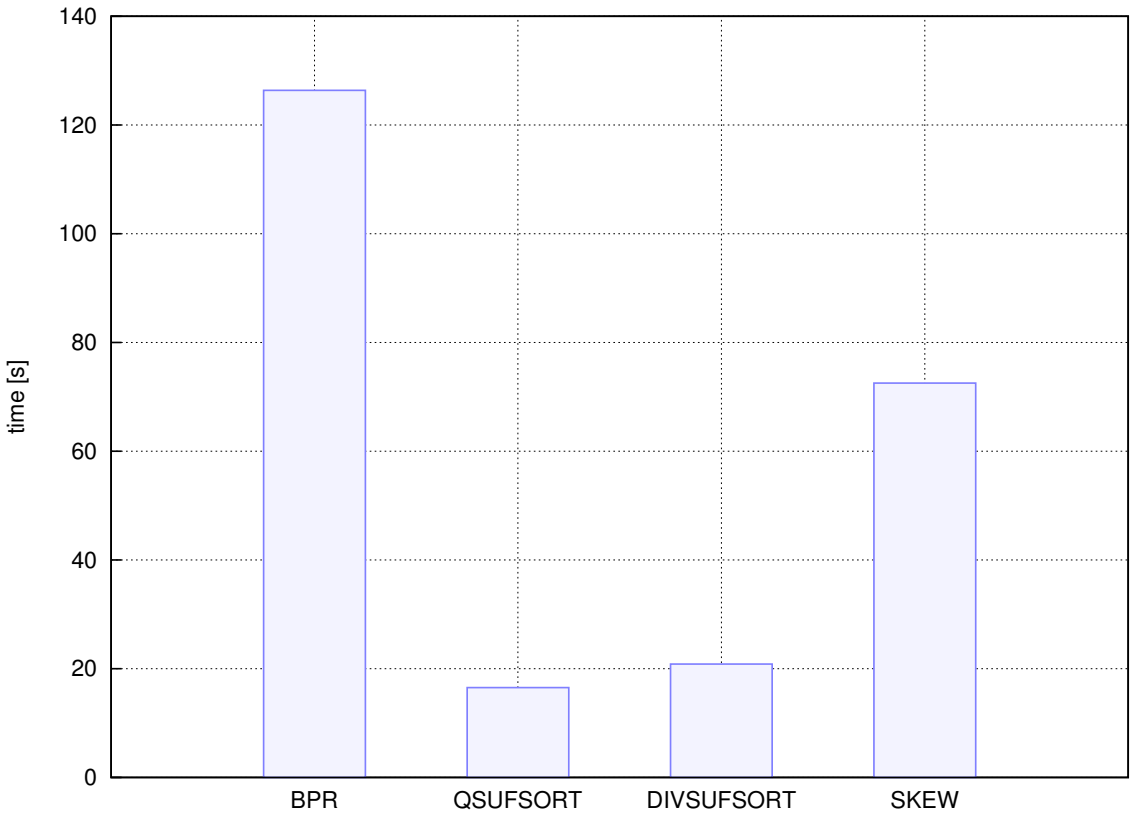
TABLICA A.4: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `jrockit`.




RYSUNEK A.4: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `jrockit`.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `abac` | 1.83 | 0.05 | **0.03** | 0.06 |
| `abba` | 4.41 | **2.82** | 2.85 | 11.62 |
| `book1x20` | **3.85** | 4.36 | 4.24 | 23.35 |
| `fib_s14930352` | 11.77 | 6.18 | **4.15** | 15.18 |
| `fss10` | 6.50 | 4.87 | **3.04** | 12.26 |
| `fss9` | 1.31 | 0.80 | **0.69** | 2.32 |
| `houston` | 4.05 | **0.23** | 0.53 | 1.45 |
| `paper5x80` | 0.19 | 0.15 | **0.07** | 0.53 |
| `test1` | 3.14 | 0.49 | **0.25** | 2.27 |
| `test2` | 0.96 | 0.36 | **0.23** | 2.28 |
| `test3` | 88.37 | 0.54 | **0.42** | 1.22 |
| Total | 126.39 | 20.85 | **16.51** | 72.53 |

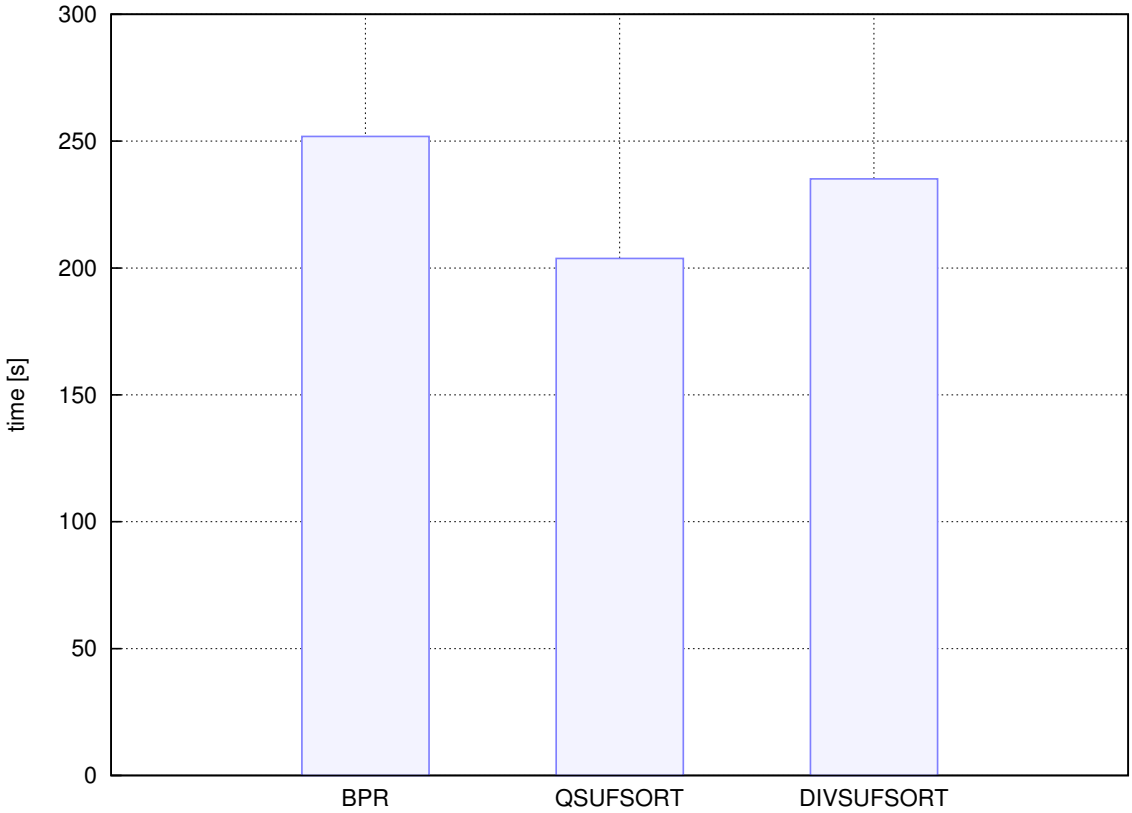
TABLICA A.5: Czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `harmony`.




RYSUNEK A.5: Sumaryczny czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `harmony`.

| | *bpr* | *divsufsort* | *qsufsort* | *skew* |
|---|---|---|---|---|
| `chr22.dna` | **7.93** | 9.15 | 8.38 | 63.94 |
| `etext99` | 32.45 | 32.38 | **29.48** | — |
| `gcc-3.0.tar` | 25.45 | 19.74 | **16.61** | — |
| `howto` | 9.71 | 9.92 | **8.46** | 82.36 |
| `jdk13c` | 18.77 | 16.84 | **13.59** | — |
| `linux-2.4.5.tar` | 29.32 | 27.61 | **24.55** | — |
| `rctail96` | 35.18 | 32.71 | **27.64** | — |
| `rfc` | 32.69 | 29.21 | **28.99** | — |
| `sprot34.dat` | 32.81 | 31.97 | **25.60** | — |
| `w3c2` | 27.51 | 25.63 | **20.47** | — |
| Total | 251.82 | 235.16 | **203.76** | |

TABLICA A.6: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `harmony`.




RYSUNEK A.6: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `harmony`.

# Literatura


[AKO02] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. *WABI '02: Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, strony 449–463, London, UK, 2002. Springer-Verlag.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, marzec 2004.

[BK03] Stefan Burkhardt, Juha Kärkkäinen. Fast lightweight suffix array construction and checking. R. Baeza-Yates, E. Chávez, M. Crochemore, redaktorzy, *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003*, wolumen 2676 serii *Lecture Notes in Computer Science*, strony 55–69, Morelia, Michoacán, Mexico, czerwiec 2003. Springer.

[BM93] Jon L. Bentley, M. Douglas McIlroy. Engineering a sort function. *Software Practice Experience*, 23(11):1249–1265, 1993.

[BS97] Jon L. Bentley, Robert Sedgewick. Fast algorithms for sorting and searching strings. *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, strony 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[BW94] M. Burrows, D. J. Wheeler. A block-sorting lossless data compression algorithm. Raport instytutowy 124, Digital SRC Research Report, 1994.

[FG98] Paolo Ferragina, Roberto Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46:236–280, 1998.

[GK97] Robert Giegerich, Stefan Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, styczeń 1997.

[IT99] Hideo Itoh, Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. *SPIRE '99: Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, strona 81, Washington, DC, USA, 1999. IEEE Computer Society.

[KA05] Pang Ko, Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, czerwiec 2005.

[KAA01] Toru Kasai, Hiroki Aftmura, Setsue Arikawa. Efficient substring traversal with suffix arrays, 2001.

[KJP04] Dong K. Kim, Junha Jo, Heejin Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. *Lecture Notes in Computer Science*, 3059:301–314, kwiecień 2004.

[KMR72] Richard M. Karp, Raymond E. Miller, Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, strony 125–136, New York, NY, USA, 1972. ACM.

[Knu02] Donald E. Knuth. *Sztuka programowania (Tom 3, Sortowanie i wyszukiwanie)*. Wydawnictwa Naukowo Techniczne, Warszawa, 2002.

[KS03] Juha Kärkkäinen, Peter Sanders. Simple linear work suffix array construction. *Proc. 13th International Conference on Automata, Languages and Programming*. Springer, 2003.

[KSPP05] Dong K. Kim, Jeong S. Sim, Heejin Park, Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2-4):126–142, czerwiec 2005.

[Lar99] Jessper N. Larsson. *Structures of String Matching and Data Compression*. Praca doktorska, Department of Comp. Science, Lund University, 1999.

[LS99] N. Jesper Larsson, Kunihiko Sadakane. Faster suffix sorting. Raport instytutowy LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, maj 1999.

[Man04] Giovanni Manzini. Two space saving tricks for linear time lcp array computation. *Algorithm Theory - SWAT 2004*, strony 372–383, 2004.

[MBM93] Peter M. Mcilroy, Keith Bostic, M. Douglas Mcilroy. Engineering radix sort. *Computing Systems*, 6:5–27, 1993.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, kwiecień 1976.

[MF04] Giovanni Manzini, Paolo Ferragina. Engineering a lightweight suffix array construction algorithm (extended abstract), 2004.

[MM90] Udi Manber, Gene Myers. Suffix arrays: a new method for on-line string searches. *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, strony 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[Mor05] Yuta Mori. Short description of improved two-stage suffix sorting algorithm. `http://homepage3.nifty.com/wpage/software/itssort.txt`, 2005.

[MP07] Michael A. Maniscalco, Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *J. Exp. Algorithmics*, 12, 2007.

[PST07] Simon J. Puglisi, William. F. Smyth, Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

[Sch07] Klaus-Bernd Schürmann. *Suffix Arrays in Theory and Practice*. Praca doktorska, Faculty of Technology of Bielefeld University, Germany, 2007.

[Sew00] Julian Seward. On the performance of bwt sorting algorithms. *IEEE Data Compression Conference*, strony 173–182, marzec 2000.

[SS07] Klaus-Bernd Schürmann, Jens Stoye. An incomplex algorithm for fast suffix array construction. *Softw. Pract. Exper.*, 37(3):309–329, 2007.

[SZ04] Ranjan Sinha, Justin Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*, 9(es), 2004.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, strony 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

# Zasoby internetowe


[A] Project Gutenberg.
http://www.gutenberg.net

[B] The AspectJ Project.
http://www.eclipse.org/aspectj/

[C] The Gauntlet (Universal Robustness Corpus).
http://www.michael-maniscalco.com/testset/gauntlet/

[D] Manzini's Large Corpus.
http://www.mfn.unipmn.it/~manzini/lightweight/corpus/

[E] Yuta Mori, Suffix Array Construction Benchmark
http://homepage3.nifty.com/wpage/benchmark/index.html

[F] Michael Maniscalco, The MSufSort Algorithm.
http://www.michael-maniscalco.com/msufsort.htm

[G] Peter Sanders, Skew algorithm.
http://www.mpi-inf.mpg.de/~sanders/programs/suffix/

[H] M. Douglas McIlroy, ssort.c
http://cm.bell-labs.com/cm/who/doug/source.html

[I] Dmitry A. Malyshev, Archon
http://kvgate.com/index.php?root/comp/arch/archon/

[J] Giovanni Manzini, A Lightweight Suffix Array and BWT Construction Algorithm
http://web.unipmn.it/~manzini/lightweight/ds.tgz

[K] Yuta Mori, libdivsufsort project homepage.
http://code.google.com/p/libdivsufsort/

[L] N.Jesper Larsson, qsufsort.c
http://www.larsson.dogma.net/qsufsort.c

[M] Klaus-Bernd Schürmann i Jens Stoye, bpr downloadpage.
http://bibiserv.techfak.uni-bielefeld.de/download/tools/bpr.html



Instytut Informatyki, Wydział Informatyki i Zarządzania
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ mnowak-masterthesis,
    author = "Michał Nowak",
    title = "{Przegląd i porównanie metod tworzenia tablic sufiksów oraz ich efektywna implementacja w języku Java}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2009",
}
```