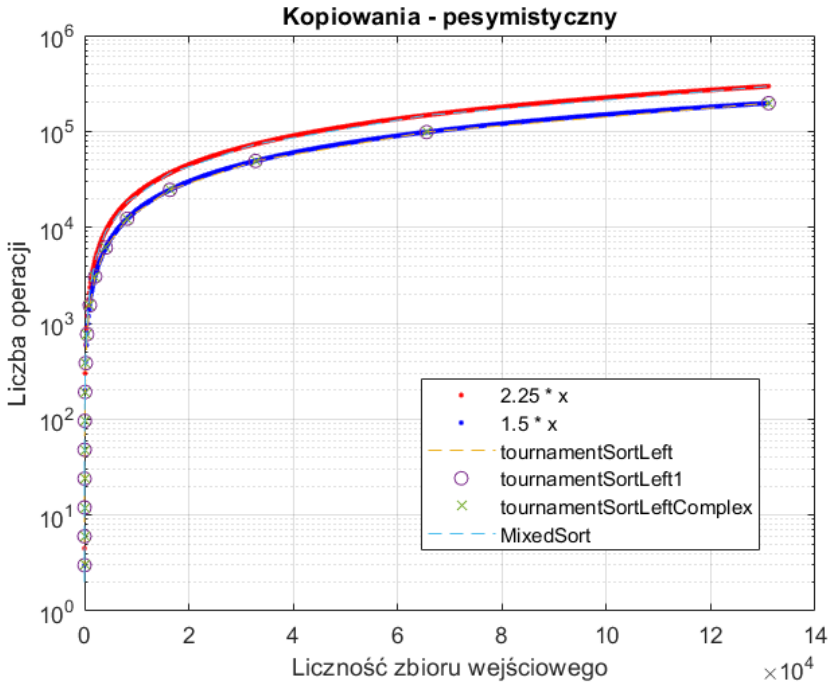
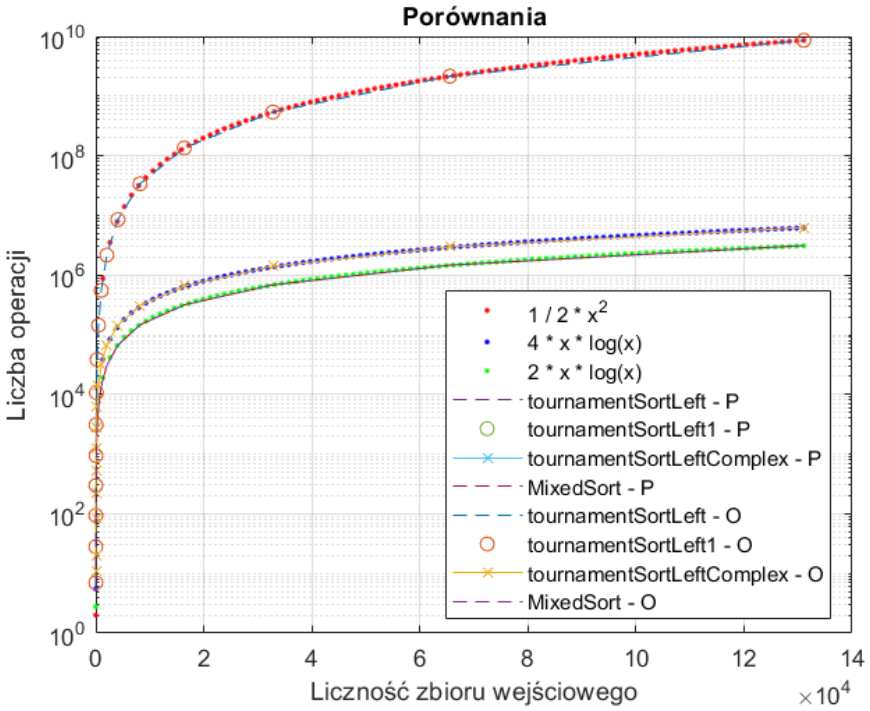
Piotr Mikołajczyk			
AISDE LAB 3	272018 Nr Indeksu	2021.04.04 – 2021.04.17 Data wykonania ćwiczenia	2021.04.17 Nominalna data oddania sprawozdania
	Punkty do wykonania	„Wykonujemy zadanie 6 z instrukcji. Należy wybrać 4 spośród następujących wersji sortowania turniejowego: tournamentSortLeft, tournamentSortLeft_1, tournamentSortLeft_2, tournamentSortLeft_12, tournamentSortLeft_Complex i MixedSort. Polecenie "zbadać wydajność" rozumiemy m. in. jako oszacowanie złożoności obliczeniowej i czasowej. Należy brać pod uwagę zarówno zbiory danych z unikatowymi kluczami jak i takie, w których zbiór wartości kluczy jest mały w porównaniu z długością sortowanego ciągu. Wnioski dotyczące przewagi różnych wersji nad innymi w różnych sytuacjach należy poprzeć analizą kodu.” / „ Z6. Zbadać wydajność i porównać różne wersje algorytmu sortowania turniejowego”	

SPRAWOZDANIE:
Badanie złożoności obliczeniowej i czasowej drzew binarnych

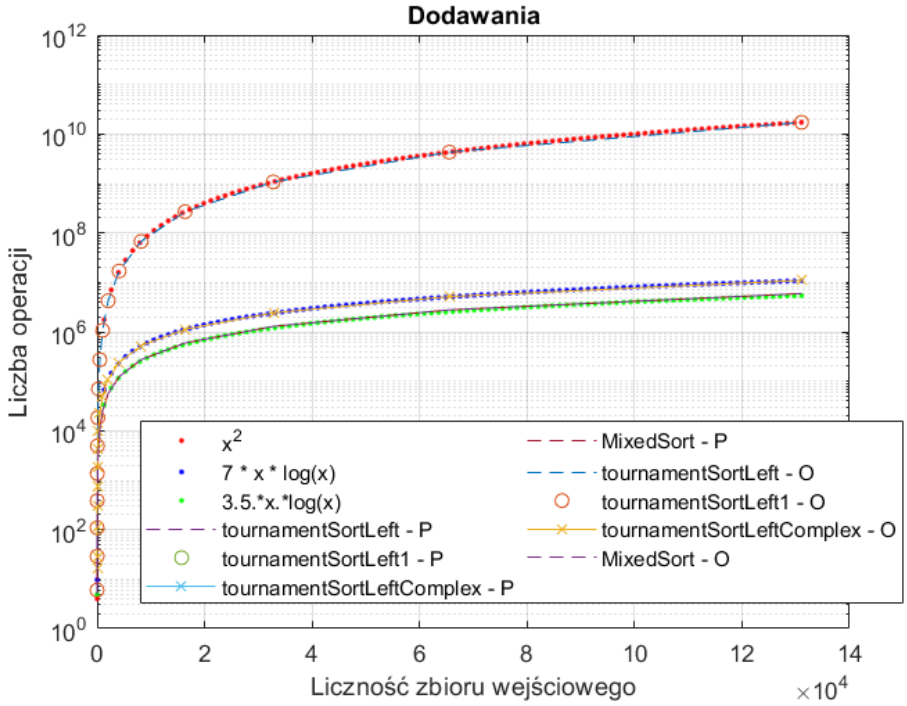
W laboratorium zbadano 4 wybrane algorytmy sortowania (z 7 dostępnych) turniejowego. Wybrano algorytmy: tournamentSortLeft, tournamentSortLeft_1, tournamentSortLeft_Complex i MixedSort.



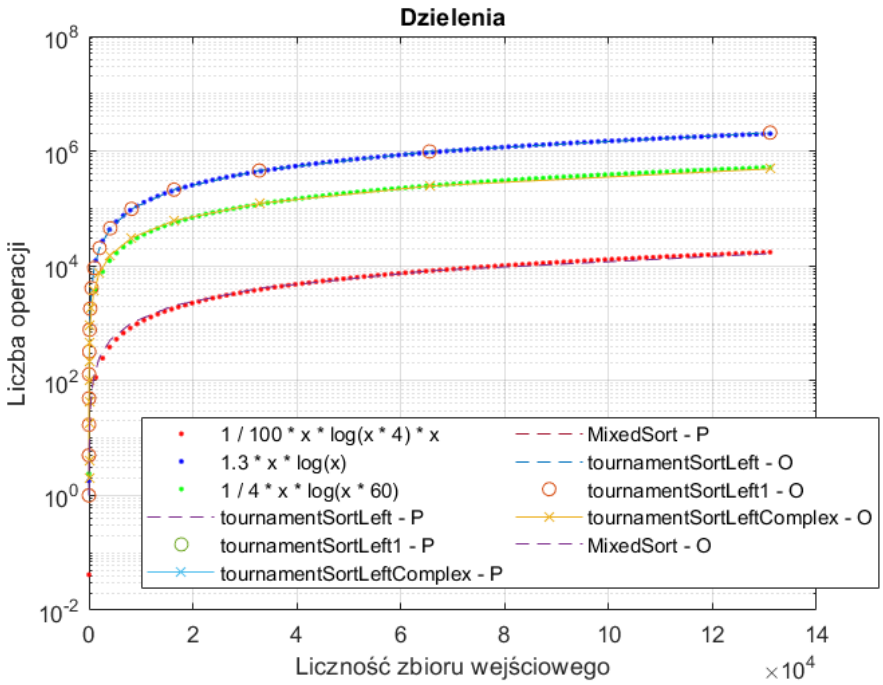
Rys. 1 – Liczba kopiowań algorytmów w zależności od liczności zbioru – przypadek pesymistyczny – wykres półlogarytmiczny



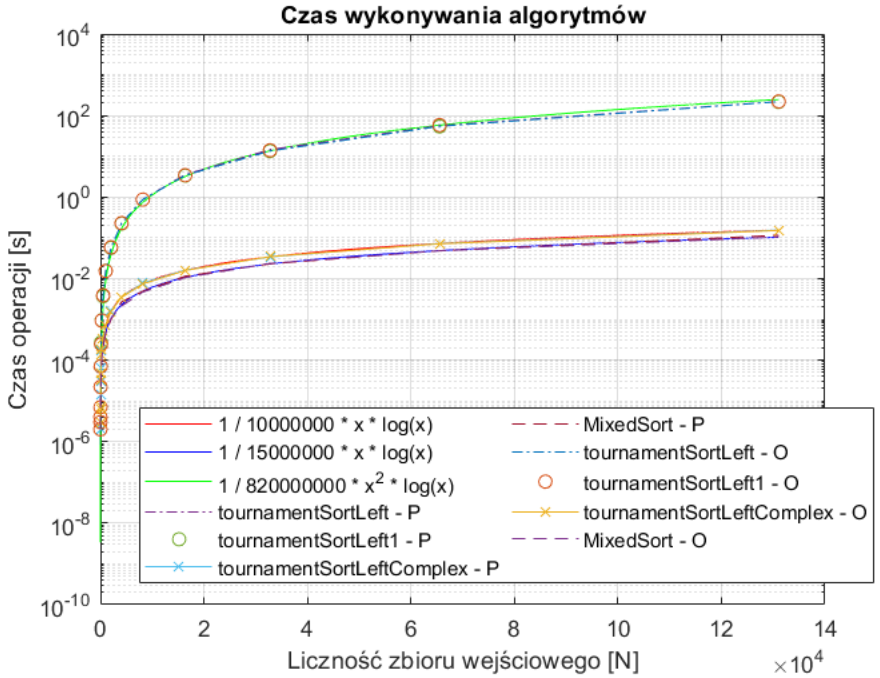
Rys. 2 – Liczba porównań algorytmów w zależności od liczności zbioru – przypadki optymistyczny i pesymistyczny – wykres półlogarytmiczny



Rys. 3 – Liczba operacji dodawania algorytmów w zależności od liczności zbioru – przypadki optymistyczny i pesymistyczny – wykres półlogarytmiczny



Rys. 4 – Liczba operacji dzielenia algorytmów w zależności od liczności zbioru – przypadki optymistyczny i pesymistyczny – wykres półlogarytmiczny



Rys. 5 – Czas wykonywania algorytmów w zależności od liczności zbioru – przypadki optymistyczny i pesymistyczny – wykres półlogarytmiczny

tournamentSortLeft

```
void Array<RECORD>::tournamentSortLeft(Iterator first, Iterator last){
   while(first < last){
      tournamentLeft(first,last);
      ++first;
   }
}

void Array<RECORD>::tournamentLeft(Iterator first, Iterator last){
   while(last > first){
      this->roundLT(first,last);
// obliczenie położenia środka tablicy (z uwzględnieniem ew. nieparzystości rozmiaru)
      last = first + ((last - first)/2);
   }
}

inline Iterator Array<RECORD>::roundLT(Iterator first, Iterator last){
   for(; first < last; ++first, --last){
      this->swapIf_a_GT_b(this->array[first], this->array[last]);
   }

   if(first == last){
      --last;
      this->swapIf_a_GT_b(this->array[last], this->array[first]);
   }
   return last;
}

bool swapIf_a_GT_b(RECORD& a, RECORD& b){ if(a > b){ swap(a,b); return true; } else return false; }
```

tournamentSortLeft_1

```
void Array<RECORD>::tournamentSortLeft_1(Iterator first, Iterator last){
   while(first < last){
      tournamentLeft(first,last);
      this->moveIndexRight(first);
      ++first;
   }
}

void Array<RECORD>::tournamentLeft(Iterator first, Iterator last){
   while(last > first){
      this->roundLT(first,last);
// obliczenie położenia środka tablicy (z uwzględnieniem ew. nieparzystości rozmiaru)
      last = first + ((last - first)/2);
   }
}

inline Iterator Array<RECORD>::roundLT(Iterator first, Iterator last){
   for(; first < last; ++first, --last){
```

```
      this->swapIf_a_GT_b(this->array[first], this->array[last]);
   }

   if(first == last){
      --last;
      this->swapIf_a_GT_b(this->array[last], this->array[first]);
   }
   return last;
}

bool swapIf_a_GT_b(RECORD& a, RECORD& b){ if(a > b){ swap(a,b); return true; } else return false; }

void Array<RECORD>::moveIndexRight(Iterator& first){
            while(array[first] == array[first+1]){ ++first; if(first == lastIndex()) break; }
}
```

tournamentSortLeft_Complex

```
void Array<RECORD>::tournamentSortLeft_Complex(Iterator first, Iterator last){
            REKURENCJA_WEJSCIE

            bool flag = false;
   Iterator mid = 0;
   while(first < last){
// turniej podstawowy
      tournamentLeft(first,last);

// turniej dodatkowy dla "przegranych" w prawej polowce tablicy
      mid = last-((last-first)/2);
      tournamentRight(mid,last);

// przesuniecie iteratorow, gdy brzegowe klucze sa rowne
      this->moveIndexRight( first );
      this->moveIndexLeft ( last  );

// turnieje "w przeciwnym kierunku" w polowkach
      tournamentRight( first+1, mid-1  );
      tournamentLeft ( mid,     last-1 );

      this->moveIndexRight( first );
      this->moveIndexLeft ( last  );

      ++first;
      --last;

// sprawdzenie, czy sortowanie w polowkach stalo sie niezalezne
                  if(this->array[mid-1] <= this->array[mid]){ flag = true; break; }

// zamiana na styku polowek (konieczna, gdy rozmiar tablicy jest nieparzysty - dla parzystego rozmiaru
// moze byc, choc nie jest krytyczna)
      swap(this->array[mid-1],this->array[mid]);
```

```
   }

   if(flag){
// sortowanie w polowkach jest niezalezne - mozna oddzielnie w nich sortowac
      tournamentSortLeft_Complex(first, mid-1);
      tournamentSortLeft_Complex(mid,   last);
  }

            REKURENCJA_WYJSCIE
}


void Array<RECORD>::tournamentRight(Iterator first, Iterator last){
   while(last > first){
      this->roundLT(first,last);
// obliczenie położenia środka tablicy (z uwzględnieniem ew. nieparzystości rozmiaru)
      first = last - ((last - first)/2);
   }
}

void Array<RECORD>::tournamentLeft(Iterator first, Iterator last){
   while(last > first){
      this->roundLT(first,last);
// obliczenie położenia środka tablicy (z uwzględnieniem ew. nieparzystości rozmiaru)
      last = first + ((last - first)/2);
   }
}

inline Iterator Array<RECORD>::roundLT(Iterator first, Iterator last){
   for(; first < last; ++first, --last){
      this->swapIf_a_GT_b(this->array[first], this->array[last]);
   }

   if(first == last){
      --last;
      this->swapIf_a_GT_b(this->array[last], this->array[first]);
   }
   return last;
}

bool swapIf_a_GT_b(RECORD& a, RECORD& b){ if(a > b){ swap(a,b); return true; } else return false; }
```

MixedSort

```
void Array<RECORD>::mixedSort(Iterator first, Iterator last){
   REKURENCJA_WEJSCIE

   // procedura nie ma sensu dla zbyt małych zbiorów
   if(last-first > 12){
      // runda turnieju deblowego przeprowadzona dwa razy (dwoma różnymi sposobami)
```

```
      Iterator mid = roundLTfour(first,last);
      Iterator mid1 = mid+1;
      roundLTfourBis(first,last);

      // selekcja największego klucza w lewej połówce tablicy (NL) i najmniejszego w prawej (NP)
      directSelectionLeft(first,mid);
      directSelectionRight(mid1,last);

      // jeżeli NL <= NP, to oba te rekordy są na swoim ostatecznym miejscu więc połówki tablicy można
procedować osobno
      if(array[mid] <= array[mid1]){
         mixedSort(first, mid-1);
         mixedSort(mid1+1, last);
      }
      else{
         // wykonanie w obu połówkach tablicy filtrowania quicksortowego:
         // największy klucz lewej połówki jest pivotem dla połówki prawej
         // najmniejszy klucz z prawej połówki jest pivotem dla lewej
         swap(array[mid],array[mid1]);
         Iterator mid0 = partitioningHoareLast(first, mid);
         Iterator mid2 = partitioningHoareFirst(mid1, last);

         // uzyskany został podział tablicy na trzy niezależne części
         mixedSort(mid0+1, mid2-1);
         mixedSort(first,  mid0-1);
         mixedSort(mid2+1, last);
      }
   }
   else insertionSort(first,last);

   REKURENCJA_WYJSCIE
}

void Array<RECORD>::insertionSort(Iterator first, Iterator last, Iterator h)
{
#if WSTAWIANIE_START_OD_PRAWEJ
   for(Iterator curr = last-h; curr >= first; curr-=h) insert(curr, last, h);
#else // START_OD_LEWEJ
   for(Iterator curr = first+h; curr <= last; curr+=h) insert(first, curr, h);
#endif
}
```

Wnioski:

Na rysunku 1 przedstawiono ilość kopiowania w zależności od liczności zbioru, dla każdego algorytmu. Algorytm MixedSort ma większą złożoność dla kopiowania niż reszta algorytmów której ilość kopiowania jest zbliżona. W przypadku optymistycznym kopiowania nie występują ponieważ sortowany ciąg jest już ułożony poprawnie. Trend ilości kopiowania jest liniowy.

Najgorzej liczba porównań wypada dla algorytmu tournamentSortLeft oraz tournamentSoftLeft1 dla wersji optymistycznych jak i pesymistycznych – ilość porównań (rys . 2) nie zależy od wielkości zbioru.

Prawdopodobnie dzieje się tak ponieważ wszystkie elementy muszą rozegrać turniej więc dla optymistycznego jak i pesymistycznego wypadku, ilość porównań jest taka sama w każdym z ww. algorytmów. Ilość porównań dla tSL oraz tSL1 jest kwadratowa natomiast reszta algorytmów ma przyrost porównań dla coraz większego zbioru wejściowego, liniowy.

Dla dodawania poszczególne algorytmy zachowują się podobnie jak dla porównań (z innymi współczynnikami). Znowu tSL oraz tSL1 mają liczbe dodawania (rys. 3) z trendem kwadratowym natomiast tSLC oraz tMS liczbe operacji przyrastającą liniowo.

Liczby zarówno dla przypadków optymistycznych jak i pesymistycznych są zbliżone dla poszczególnych algorytmów. Dla operacji dzielenia sprawa ma się nieco inaczej. tSL w wersji pesymistycznej jak i optymistyczne ma najmniej operacji tego typu wraz z wzrostem liczności zbioru, następnie tSLC i niewiele dalej tSL1 oraz MS. Odpowiedni wykres przedstawiono na rysunku 4.

Złożoność obliczeniową przedstawiono na rysunku 5. Widać z nich że liczba porównań oraz dodawania determinuje czas wykonywania się poszczególnych algorytmów – tSL oraz tSL1 wykonują się o 2 rzędy wielkości (dla sprawdzonych danych) niż reszta algorytmów oraz mają trend kwadratowy. Natomiast pozostałe algorytmy wykonują się ze złożonością liniową.

Oświadczam, że niniejsza praca stanowiąca podstawę do uznania osiągnięcia efektów uczenia się z przedmiotu Algorytmy i Struktury Danych została wykonana przez mnie samodzielnie.