Zadanie 2

- Jakub Karbowski
- Jakub Szymczak

Faktoryzacja LU

Implementacja bazuje na rekurencyjnym wzorze podanym w książce "Wprowadzenie do algorytmów" s. 840.

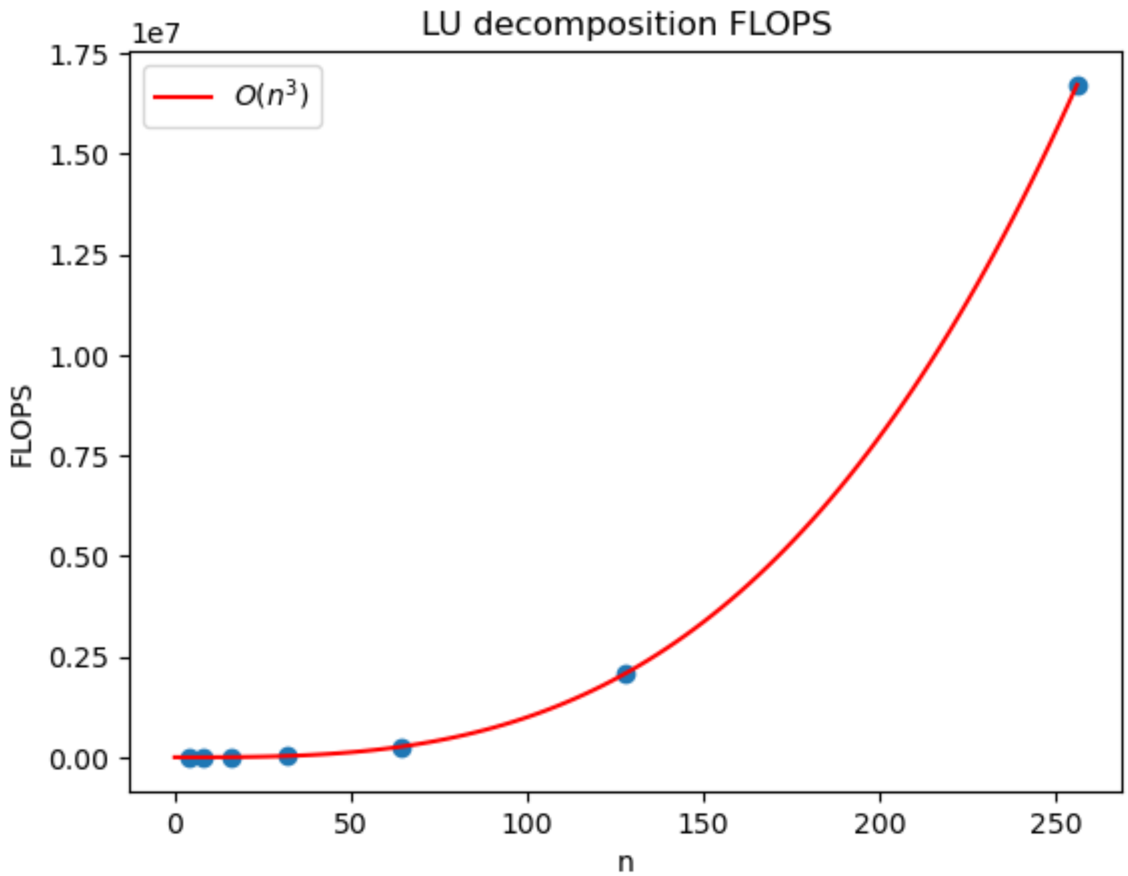
$$
\begin{aligned}
A &= \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix} && (1) \\
&= \underbrace{\begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix}}_{U} && (2)
\end{aligned}
$$

Do implementacji zastosowano język Haskell.

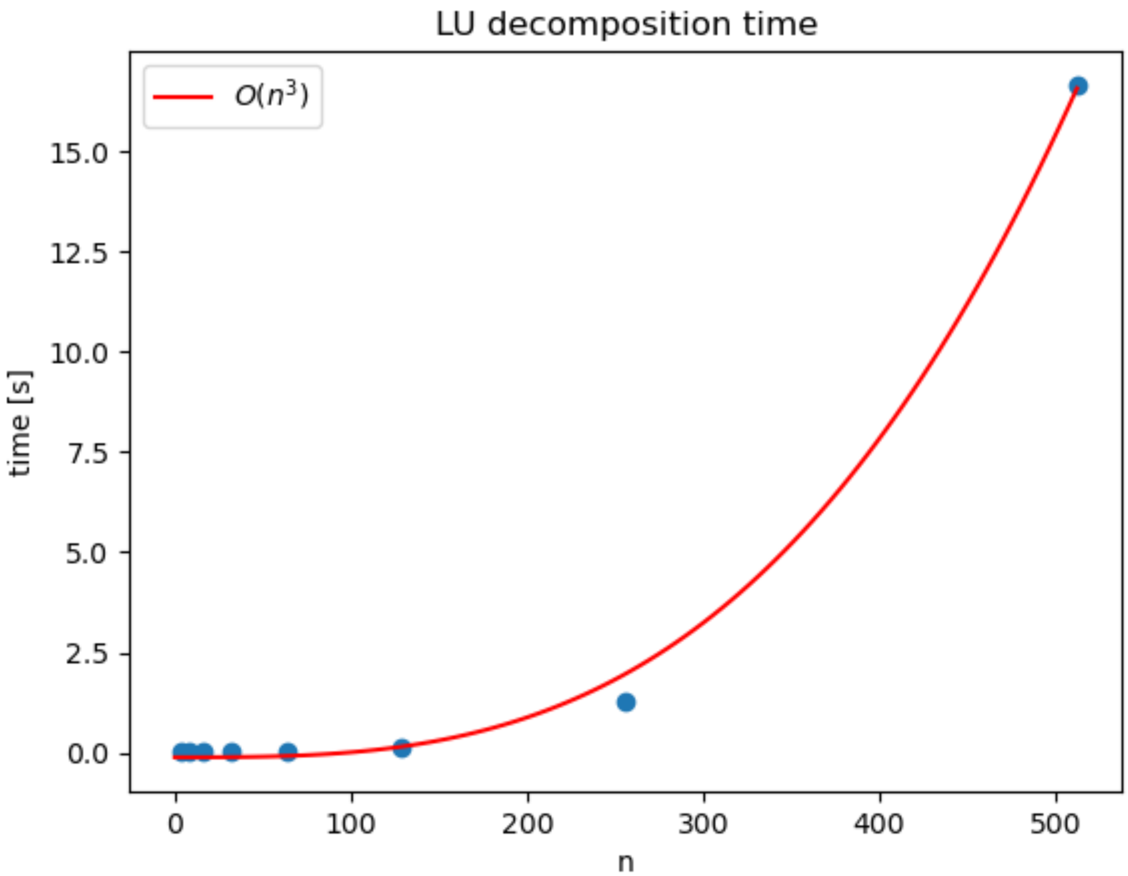
```haskell
lu [[a00]] = ([[1]], [[a00]])
lu ((a00:w):rows) = (l, u)
  where n = length w + 1
        v = map head rows
        ap = map tail rows
        (lp, up) = lu $ ap `msubm` (v `outer` w) `mdivs` a00
        l = (1 : replicate (n - 1) 0) : zipWith (:) (map (/a00) v) lp
        u = (a00 : w) : map (0:) up
```

Liczbę operacji policzono za pomocą osobnego programu symulującego wykonywanie algorytmu odwracania.

```python
def lu_ops(n):
    if n == 1: return 0
    outer = (n-1)**2
    msubm = (n-1)**2
    mdivs = (n-1)**2
    lu_rec = lu_ops(n-1)
    div_a00 = n-1
    return outer + msubm + mdivs + lu_rec + div_a00
```



Liczba operacji idealnie pokrywa się z teoretyczną złożonością $O(n^3)$.



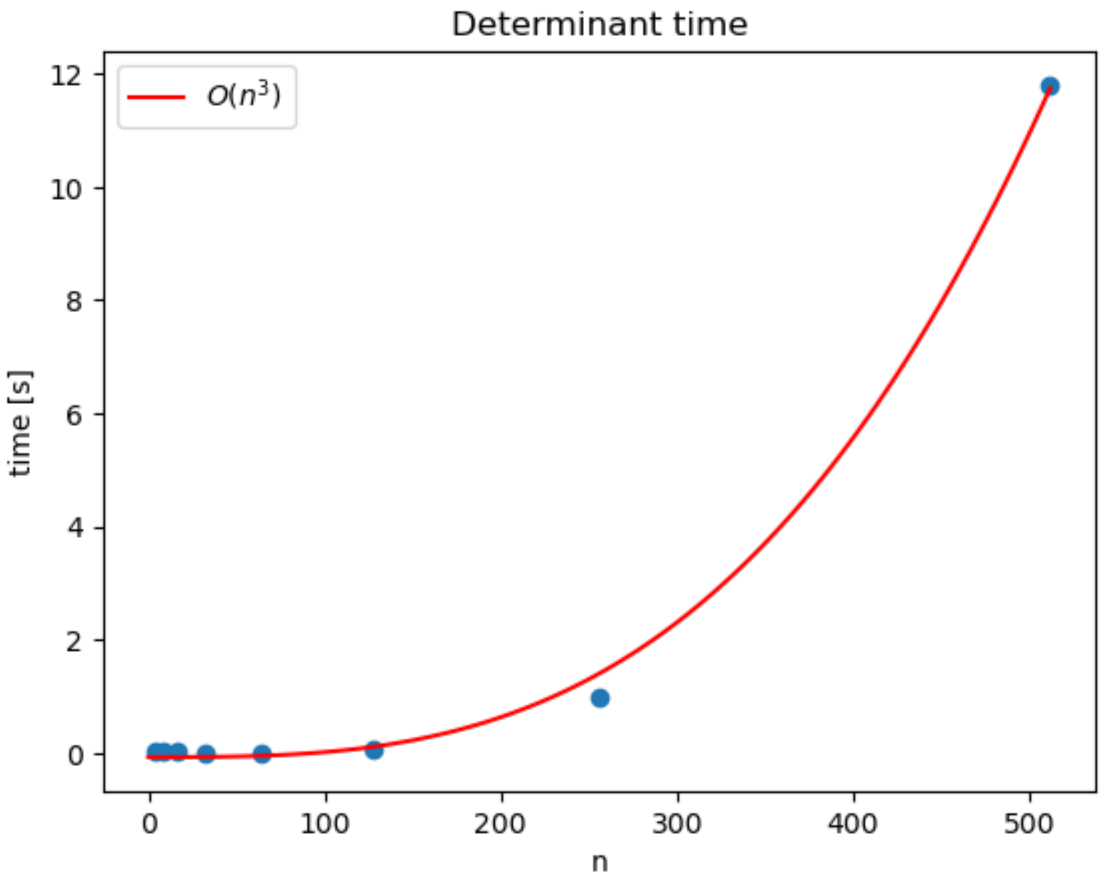
Zmierzona złożoność obliczeniowa $O(n^3)$ pokrywa się z teoretyczną. Implementacja w Haskellu utrudnia pomiar czasu, ponieważ Haskell jest leniwy. Aby "zmusić" go do obliczeń, liczymy sumę elementów macierzy wynikowych, co wpływa lekko na czas ($+O(n^2)$ co asymptotycznie nie wpływa na $O(n^3)$).

Wyznacznik

Dzięki obliczeniu faktoryzacji LU, obliczenie wyznacznika sprowadza się do wymnożenia elementów na przekątnej macierzy U.

```
det = product . diagonal . snd . lu
```

Liczba operacji jest równa liczbie operacji dla faktoryzacji LU, plus $n - 1$ dodatkowych mnożeń elementów na przekątnej.



Czas jest asymptotycznie identyczny jak w przypadku faktoryzacji LU. Liczenie wyznacznika zajmuje proporcjonalnie mniej czasu niż faktoryzacja, ponieważ nie jest konieczne zmuszanie Haskella do obliczeń przez sumowanie elementów. Dodatkowo, Haskell może w pewien sposób optymalizować graf obliczeń, pomijając niepotrzebne operacje.

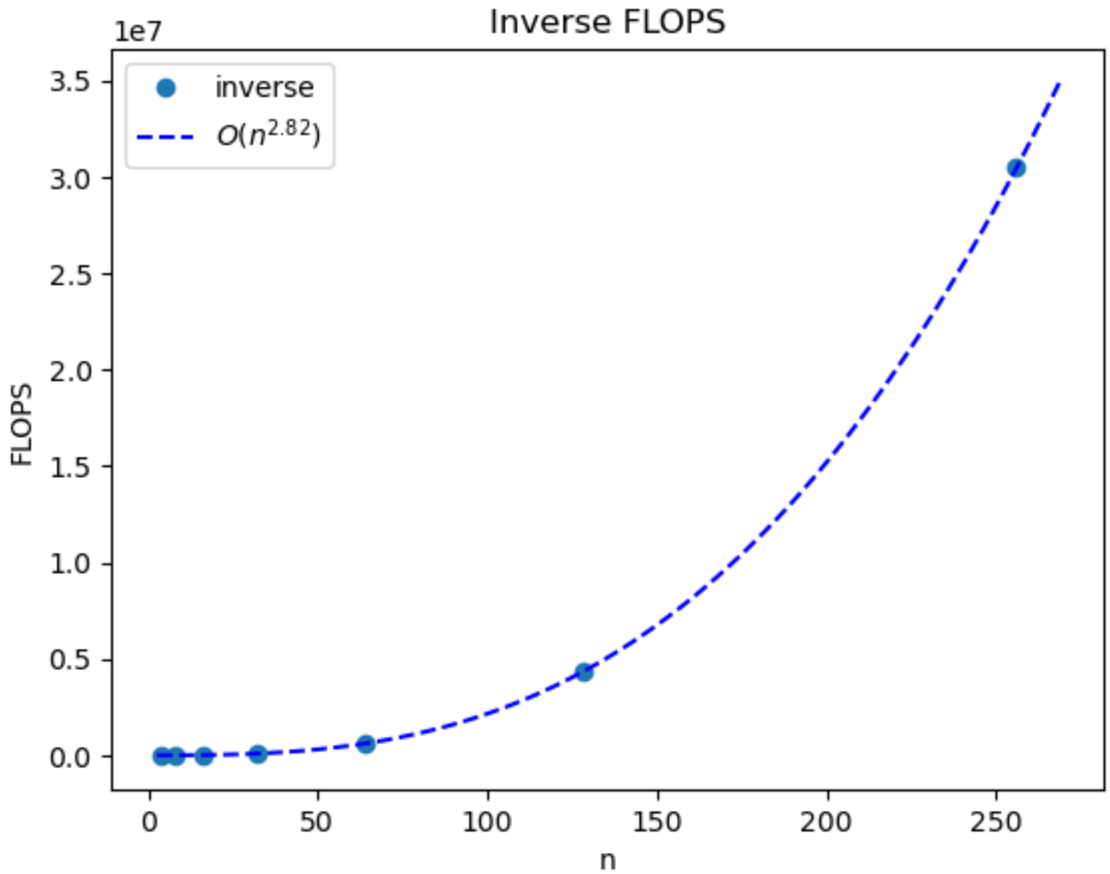
Odwracanie macierzy

Polega na podziale macierzy na 4 równe bloki i rekurencyjnym wywoływaniu aż do przypadku granicznego, gdzie odwrotnością macierzy jednoelementowej jest odwrotność jej jedynej wartości. Dokładne wzory na macierz odwrotną były wyprowadzone na wykładzie i wywodzą się eliminacji Gaussa z macierzą identycznościową z prawej strony.

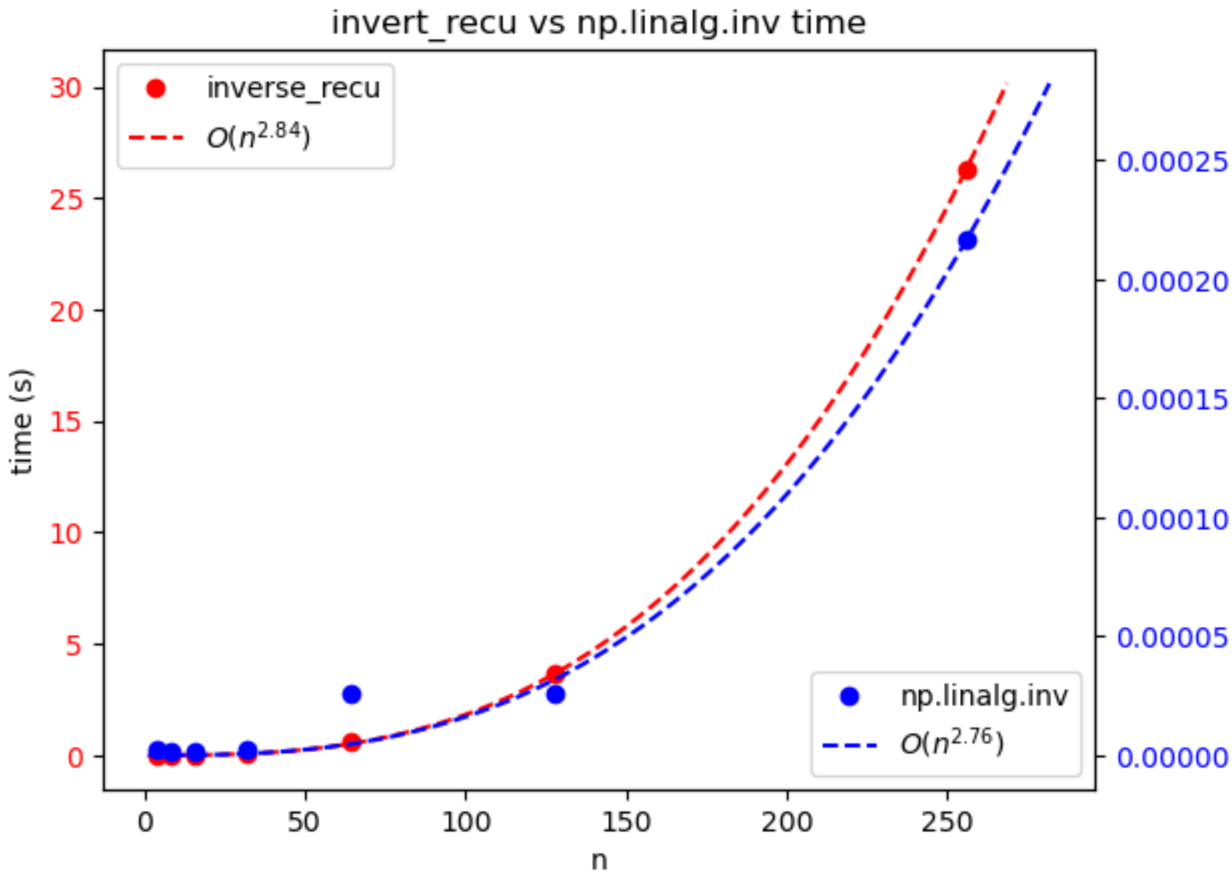
$$A^{-1} = \begin{pmatrix} A_{11}^{-1} + A_{11}^{-1} A_{12} S_{22}^{-1} A_{21} A_{11}^{-1} & -A_{11}^{-1} A_{12} S_{22}^{-1} \\ -S_{22}^{-1} A_{21} A_{11}^{-1} & S_{22}^{-1} \end{pmatrix}$$

```
def invert_matrix_recu(A):
    if A.size > 1:
        A11, A12, A21, A22 = split(A)

        A11_inv = invert_matrix_recu(A11)
        S22 = A22 - A21 * A11_inv * A12
        S22_inv = invert_matrix_recu(S22)

        B11 = A11_inv * (Identity + A12 * S22_inv * A21 * A11_inv)
        B12 = -A11_inv * A12 * S22_inv
        B21 = -S22_inv * A21 * A11_inv
        B22 = S22_inv

        return np.concatenate(
            [np.concatenate([B11, B12], axis=1), np.concatenate([B21, B22],
axis=1)], axis=0,)
    else:
        return np.full((1, 1), 1 / A[0][0])
```

```
/tmp/ipykernel_113165/4227815796.py:4: RuntimeWarning: invalid value encountered in p
ower
  return a * x**b
```



```
/tmp/ipykernel_113165/233283149.py:10: RuntimeWarning: invalid value encountered in p
ower
  return a * x**b
/tmp/ipykernel_113165/233283149.py:23: RuntimeWarning: invalid value encountered in p
ower
  return a * x**b
```



Złożoność odwracania macierzy jest bezpośrednio zależna od złożoności mnożenia macierzy, w algorytmie użyliśmy mnożenia rekurencyjnego(Strassen), więc złożoność odwracania macierzy jest taka jak złożoność mnożenia rekurencyjnego.