Sprawozdanie zadanie 1.
Dynamiczne alokowanie pamięci

Do zadania zostały zaimplementowane 3 algorytmy:

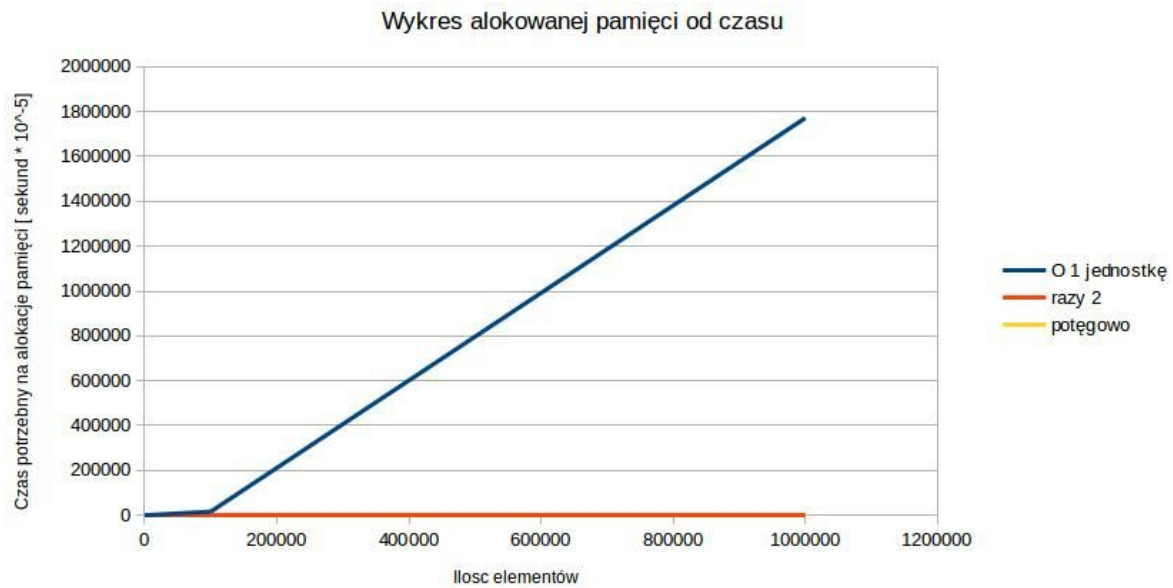
```
23  class Tablica {
24
25  public:
26    int Rozmiar = 10;
27    int * tablica = new int[Rozmiar];
28  // powiekszanie tablicy o jedno miejsce az do momentu dopasowania
29    int * Powiekszanie_O1(){
30      this->Rozmiar++;
31      int * tab = new int[this->Rozmiar];
32      for(int i=0; i<this->Rozmiar; i++) tab[i]=this->tablica[i];
33      delete [] this->tablica;
34      return tab;
35    }
36  // powiekszanie tablicy dwukrotnie
37    int * Powiekszanie_xRazy(int IleRazy){
38      this->Rozmiar*=IleRazy;
39      int * tab = new int[this->Rozmiar];
40      for(int i=0; i<=(this->Rozmiar)/IleRazy; i++) tab[i]=this->tablica[i];
41      delete [] this->tablica;
42      return tab;
43    }
44  // powiekszanie tablicy potegowo
45    int * Powiekszanie_Potega(){
46      this->Rozmiar*=this->Rozmiar;
47      int * tab = new int[Rozmiar];
48      for(int i=0; i<=(this->Rozmiar)/this->Rozmiar; i++) tab[i]=this->tablica[i];
49      delete [] this->tablica;
50      return tab;
51    }
52  //pomniejszanie tablicy do zadanego wymiaru
53    int * SetSize(int n_wymiar){
54      this->Rozmiar = n_wymiar;
55      int * tab = new int[Rozmiar];
56      for(int i=0; i<Rozmiar; i++) tab[i]=this->tablica[i];
57      delete [] this->tablica;
58      return tab;
59    }
60  };
61
```

Wszystkie algorytmy działają na tej samej zasadzie - tworzona jest lokalnie w ramach metody nowa tablica, ale jej rozmiar jest odpowiednio dla każdej z metod większy. Dla algorytmu który powiększa rozmiar tablicy “o jednen” rozmiar ten jest większy o jeden, czyli o jest zaalokowane tyle miejsca, ile zajmuje w tym przypadku zmienna typu int oraz cała poprzednio zaalokowana potrzebna pamięć. Odpowiednio dla kolejnych funkcji, gdzie możemy zwiększać rozmiar zaalokowanej pamięci o tyle razy ile będzie chciał użytkownik, (lub domyślnie 2) tyle razy więcej pamięci zostanie zaalokowane w porównaniu do poprzedniego rozmiaru naszej tablicy, alokowanie pamięci wygląda tak samo dla powiększania poprzez algorytm oparty na potęgowaniu.

Następnym krokiem jest “przepisanie”, czy też skopiowanie zawartości starej tablicy do nowo stworzonej, z większą - porządaną przez nas ilością pamięci. Po udanym skopiowaniu wartości, stara tablica jest już niepotrzebna, więc pamięć zostaje zwolniona przy użyciu operatora delete, a że chcemy skasować tablicę używamy do tego jeszcze nawiasów indeksowania.
Ostatnim krokiem który dzieje się w naszych algorytmach jest zwracanie nowej tablicy jako wskaźnik do zaalokowanej pamięci dzięki czemu nie tracimy dostępu do naszych danych.

Porównanie wyników pomiarów czasu:

Powiększanie ->	O 1 jednostkę	razy 2	potęgowo
Rozmiar\czas	czas [ms]	czas [ms]	czas [ms]
10	0,003	0	0,001
10E3	5,62	0,044	0,002
10E5	16193	1,19	0,42
10E6	1771400	11,51	3,63
10E9	-	-	-



Wnioski:

Zgodnie z przeprowadzonymi pomiarami, które udało się wykonać można wysunąć wniosek, iż najbardziej skutecznym algorytmem jest algorytm zwiększania pamięci potęgowo. Działa on najszybciej, ponieważ alokowany jest najwięcej pamięci w jednym “kroku”. Najmniej wydajnym algorytmem jest zwiększanie pamięci o “jeden”. Algorytm ten wymaga największej liczby operacji, przez co jest najbardziej złożony, w przeciwieństwie do algorytmu potęgowego, gdzie liczba operacji jest najmniejsza. Niestety testów nie udało się przeprowadzić dla rozmiaru miliarda tablicy. Jedną z przyczyn jest brak potrzebnej pamięci potrzebnej do przetworzenia takiej ilości czynności, a drugą czas wymagany na wykonanie algorytmu zwiększania “objętości” tablicy o jeden.