1. Proste operacje na wektorach

1 Zadanie

Uzupełnij załączony program o definicje funkcji operujących na wektorach:

1. `void linspace(double v[], double start, double stop, int n);`
   Funkcja wypełnia tablicę rzeczywistą `v n` wartościami równomiernie rozłożonymi w przedziale `[start, stop]`. Wartość n powinna być nieujemna; dla $n = 1$ funkcja wpisuje do początkowego elementu tablicy wartość `start`. Dla $n = 0$ funkcja nie zmienia zawartości tablicy.

2. `void add(double v1[], const double v2[], int n);`
   Funkcja dodaje wektory, których elementy są zapisane w początkowych `n` elementach rzeczywistych tablic `v1` i `v2`. Funkcja zapisuje wektor będący sumą na miejscu wektora `v1`.

3. `double dot_product(const double v1[], const double v2[], int n);`
   Funkcja oblicza i zwraca iloczyn skalarny wektorów `v1` i `v2` o długości n.

4. `void multiply_by_scalar(double v[], int n, double s);`
   Funkcja mnoży każdy element tablicy rzeczywistej `v` (o długości n) przez liczbę rzeczywistą `s`. Wynik mnożenia jest zapisywany w tablicy `v`.

5. `void range(double v[], double start, double step, int n);`
   Funkcja wpisuje do n początkowych elementów rzeczywistej tablicy `v` wartości ciągu arytmatycznego o postępie `step`. Postęp może mieć wartość ujemną - wtedy kolejne elementy tablicy będą stanowić ciąg malejący.

   Wartość n powinna być nieujemna. Dla $n = 1$ funkcja wpisuje do początkowego elementu tablicy wartość `start`. Dla $n = 0$ funkcja nie zmienia zawartości tablicy.

Uzupełnij też pomocniczą funkcję `void read_vector(double v[], int n)`, która czyta ze standardowego wejścia n elementową tablicę rzeczywistą `v`.

2 Wejście

Pierwszy wiersz standardowego wejścia zawiera jedną liczbę naturalną $1 \leqslant F \leqslant 5$, oznaczającą kod funkcji do wykonania, zgodny z numeracją z poprzedniego paragrafu. Kolejne wiersze są zależne od wartości F, i tak:

1. $F = 1$: druga linia wejścia zawiera jedną liczbę całkowitą (wartość $0 \leqslant n \leqslant 100$) i dwie liczby rzeczywiste (`start` i `stop`).
2. $F = 2$: druga linia zawiera jedną liczbę całkowitą $0 < n \leqslant 100$ (długość dodawanych wektorów). Kolejne dwie linie zawierają po n liczb rzeczywistych każda (elementy wektorów `v1` i `v2`).
3. $F = 3$: druga linia zawiera jedną liczbę całkowitą $0 < n \leqslant 100$ (długość mnożonych wektorów). Kolejne dwie linie zawierają po n liczb rzeczywistych każda (elementy wektorów `v1` i `v2`).
4. $F = 4$: druga linia zawiera jedną liczbę całkowitą $0 < n \leqslant 100$ (długość wektora) i jedną liczbę rzeczywistą s (przez którą mnożymy elementy wektora v). Kolejna linia zawiera n liczb rzeczywistych (elementy wektora `v`).
5. $F = 5$: druga linia wejścia zawiera jedną liczbę całkowitą (wartość $0 \leqslant n \leqslant 100$) i dwie liczby rzeczywiste (`start` i `step`).

3 Wyjście

Wyjście programu również zależy od użytej funkcji. Dla $F = 1, 2, 4, 5$ program wypisuje (w jednej linii) elementy wyznaczonego wektora. Dla $F = 3$ program wypisuje jedną liczbę rzeczywistą - iloczyn skalarny wektorów.

Wszystkie liczby rzeczywiste powinny być wyprowadzone z dwoma miejscami po kropce dziesiętnej (format "%.2f").

4 Przykłady

4.1 Wejście

```
1
11 -10 10
```

Wyjście

```
-10.00 -8.00 -6.00 -4.00 -2.00 0.00 2.00 4.00 6.00 8.00 10.00
```

4.2 Wejście

```
2
5
1 2 3 4 5.5
6 5 2 7.5 3
```

Wyjście

```
7.00 7.00 5.00 11.50 8.50
```

4.3 Wejście

```
3
5
1 2 3 4 5.5
6 5 2 7.5 3
```

Wyjście

```
68.50
```

4.4 Wejście

```
4
5 3.5
1 2 3 4 5
```

Wyjście

```
3.50 7.00 10.50 14.00 17.50
```

4.5 Wejście

```
5
5 1 -0.4
```

Wyjście

```
1.00 0.60 0.20 -0.20 -0.60
```

2 Permutacje, stos, kolejki

Zadanie 2.1. Losowanie, permutacje, sortowanie

Szablon programu należy uzupełnić o definicje 3 funkcji:

- Funkcja `int rand_from_interval(int a, int b)` – korzystając z bibliotecznej funkcji `rand()` oraz operacji dzielenia modulo – oblicza i zwraca liczbę należącą do domkniętego przedziału $[a, b]$. *Szczegóły w komentarzu szablonu programu.*
  Założenie: liczba elementów zbioru, z którego odbywa się losowanie, nie jest większa od `RAND_MAX`+1.

- Funkcja `void rand_permutation(int n, int tab[])` losowo wybiera jedną z permutacji n elementów zbioru liczb naturalnych. Elementy tego zbioru – liczby naturalne z przedziału $[0, n-1]$ – są wpisywane do tablicy `tab` w porządku rosnącym. Do losowania liczby z przedziału należy wykorzystać zdefiniowaną wcześniej funkcję `rand_from_interval()`.
  Zapisany w pseudokodzie algorytm wpisywania liczb do tablicy oraz wyboru permutacji:

  Require: $n \geqslant 0$
  for $i \leftarrow 0$ to $n-1$ **do**
  $a[i] \leftarrow i$
  end for
  for $i \leftarrow 0$ to $n-2$ **do**
  $k \leftarrow \text{random}(i, n-1)$ ▷ losowanie z przedzialu $[i, n-1]$
  $\text{swap}(a[i], a[k])$ ▷ zamiana elementow i i k tablicy a
  end for

- Funkcja `int bubble_sort(int n, int tab[])` metodą bąbelkową sortuje n elementów tablicy `tab` wg porządku od wartości najmniejszej do największej.
  Uwaga: Są dwa możliwe kierunki przeglądania elementów sortowanej tablicy - w kierunku rosnących albo malejących **indeksów** tej tablicy. Proszę zastosować ten pierwszy (od małych do dużych).
  Uwaga ta nie dotyczy kierunku uporządkowania elementów, lecz kierunku przeglądania tablicy, czyli najpierw porównujemy `t[i]` z `t[i+1]`, później `t[i+1]` z `t[i+2]` itd., a nie `t[i]` z `t[i-1]`, później `t[i-1]` z `t[i-2]` itd.
  Funkcja zwraca najmniejszą liczbę przeglądań tablicy (liczbę iteracji zewnętrznej pętli algorytmu sortowania), po której elementy tablicy są właściwie uporządkowane.

W segmencie głównym szablonu programu są zapisane trzy testy ww. funkcji.
Każdy test wymaga osobnego wykonania (egzekucji) programu.
Pierwszą daną wczytywaną przez program jest liczba – numer testu, który ma być zrealizowany.
Drugą wczytywaną daną jest zarodek (`seed`) generatora liczb pseudolosowych.
Wywoływanie funkcji `srand(seed)` ma na celu uzyskanie powtarzalności otrzymywanych wyników testów.

Test 2.1.1: Wypisanie losowo wygenerowanych 3 liczb z zadanego przedziału $[a, b]$

Test wczytuje granice przedziału i wypisuje trzy wygenerowane liczby w kolejności zgodnej z kolejnością ich generowania.

- **Wejście**
  1 `seed a b`
- **Wyjście**
  Trzy wylosowane liczby całkowite.
- **Przykład:**
  Wejście: 1 100 3 30
  Wyjście: 11 4 10

Test 2.1.2: Losowy wybór permutacji

Test wczytuje liczność zbioru (liczbę elementów zbioru), wywołuje funkcję `rand_permutation()` i wypisuje wylosowaną permutację.

- **Wejście**
  2 `seed n`
- **Wyjście**
  Wylosowana permutacja n liczb całkowitych.
- **Przykład:**
  Wejście: 2 20 10
  Wyjście: 1 0 3 4 6 2 8 9 5 7

Test 2.1.3: Sortowanie elementów tablicy metodą bąbelkową

Test wczytuje liczbę n elementów sortowanej tablicy, wywołuje funkcję generującą permutację `rand_permutation()` oraz funkcję sortującą permutację `bubble_sort()`.

- **Wejście**
  3 `seed n`
- **Wyjście**
  Numer iteracji pętli zewnętrznej (liczony od 1), po której tablica była już uporządkowana, np.:
  dla 0 1 2 3 7 4 5 6 wynik = 1,
  dla 1 2 3 7 4 5 6 0 wynik = 7,
  dla 0 1 2 3 4 5 6 7 wynik = 0.
- **Przykład:**
  Wejście: 3 20 10
  Wyjście: 3

Zadanie 2.2. Stos, kolejka w tablicy z przesunięciami, kolejka z buforem cyklicznym

W programie są zdefiniowane tablice `stack, queue, cbuff`. Ich rozmiary są takie same i równe 10.

2.2.1: Stos

Stos jest realizowany za pomocą tablicy `stack` i zmiennej `top` zdefinowanymi poza blokami funkcji. Szablon programu należy uzupełnić o definicję funkcji obsługujących stos `stack_push(), stack_pop(), stack_state()`.

- Funkcja `stack_push(double x)` kładzie na stosie wartość parametru i zwraca zero, a w przypadku przepełnienia stosu - nie zmienia zawartości stosu i zwraca stałą `INFINITY` (zdefiniowaną w `math.h`).
- Funkcja `stack_pop(void)` zdejmuje ze stosu jeden element i zwraca jego wartość. W przypadku stosu pustego (pustego przed próbą zdjęcia elementu) zwraca stałą `NAN` (też zdefiniowaną w `math.h`).
- Funkcja `stack_state(void)` zwraca liczbę elementów leżących na stosie.

Test 2.2.1 (stosu)
Test pozwala zapisać na stosie dodatnie liczby typu `double`. Wczytuje numer testu oraz ciąg liczb rzeczywistych x reprezentujących operacje na stosie:

- Wpisanie dodatniej liczby x powoduje wywołanie funkcji `stack_push(x)` i w przypadku przepełnienia stosu wypisuje wartość stałej `INFINITY`.
- Wpisanie ujemnej liczby powoduje wywołanie funkcji `stack_pop()` i wypisanie zwracanej przez nią wartości.
- Wpisanie zera powoduje wywołanie funkcji `stack_state()`, wypisanie zwracanej przez nią wartości i zakończenie testu.
- **Przykład:**
  Wejście:
  1
  2. 4. 5. 7. 1. -2. -1. 9. -1. 5. 0.
  Wyjście:
  1.00 7.00 9.00
  4

2.2.2: Kolejka w tablicy z przesunięciami

Obsługa kolejki (typu FIFO) jest realizowana z zastosowaniem tablicy `queue` i zmiennej `in` zdefiniowanymi poza blokami funkcji. Wartością zmiennej `in` jest liczba klientów oczekujących w kolejce. Kolejny pojawiający się klient otrzymuje kolejny numer począwszy od 1. Klient, który zastaje pełną kolejkę, rezygnuje

z oczekiwania, ale zachowuje swój numer (kolejny klient otrzyma następny numer). Numery klientów czekających w kolejce są pamiętane w kolejnych elementach tablicy `queue` w taki sposób, że numer klienta najdłużej czekającego jest pamiętany w `queue[0]`.

Szablon programu należy uzupełnić o definicję funkcji obsługujących kolejkę `queue_push(), queue_pop(), queue_state(), queue_print()`.

- Funkcja `queue_push(int how_many)` powiększa kolejkę o `how_many` klientów. Numer bieżącego klienta jest pamiętany w zmiennej globalnej `curr_nr`. Zwraca 0.0.
  W przypadku, gdy liczba wchodzących do kolejki jest większa niż liczba wolnych miejsc w kolejce, miejsca w kolejce są zajmowane do zapełnienia miejsc, a pozostali "niedoszli klienci" rezygnują (zachowując swoje numery). W takiej sytuacji funkcja zwraca stałą `INFINITY`.

- Funkcja `queue_pop(int how_many)` symuluje wyjście z kolejki (obsługę) `how_many` najdłużej czekających klientów. Funkcja zwraca długość pozostałej kolejki.
  W przypadku gdy `how_many` jest większa od długości kolejki, kolejka jest opróżniana, a funkcja zwraca -1.

- Funkcja `queue_state()` zwraca liczbę czekających w kolejce.

- Funkcja `queue_print()` wypisuje numery czekających klientów (w kolejności wejścia do kolejki).

Test 2.2.2 (kolejki z przesunięciami)

- **Wejście**
  Test wczytuje numer testu oraz ciąg liczb całkowitych reprezentujących operacje na kolejce:
  - Liczba dodatnia jest liczbą klientów dochodzących do kolejki.
  - Liczba ujemna jest liczbą obsłużonych klientów opuszczających kolejkę.
  - Zero powoduje wywołanie funkcji `queue_state()` i wypisanie zwracanej przez nią wartości, wywołanie funkcji `queue_print()` oraz zakończenie testu.

- **Wyjście**
  Wartości stałych:
  - `INFINITY`, gdy wystąpiła próba przepełnienia kolejki,
  - -1 w przypadku próby wyjścia z kolejki klientów, których nie było w kolejce.

  Po wpisaniu na wejściu liczby 0 są wypisywane: liczba czekających klientów oraz ich numery wg kolejności w kolejce.

- **Przykład:**
  Wejście:
  2
  1 3 5 -2 7 -3 -9 3 -1 0
  Wyjście:
  inf -1
  2
  18 19

2.2.3: Kolejka w buforze cyklicznym

Obsługa kolejki (typu FIFO) jest realizowana z zastosowaniem tablicy `cbuff` służącej jako bufor cykliczny i zmiennych `out` i `len` zdefiniowanymi poza blokami funkcji. Wartością zmiennej `len` jest liczba klientów oczekujących w kolejce, a zmiennej `out` – indeks tablicy `cbuff`, w której jest pamiętany numer klienta najdłużej czekającego (o ile długość kolejki `len` > 0).

Szablon programu należy uzupełnić o definicję funkcji obsługujących kolejkę `cbuff_push(), cbuff_pop(), cbuff_state(), cbuff_print()`.

- Funkcja `cbuff_push(int cli_nr)` powiększa kolejkę o jednego klienta o numerze `cli_nr` i zwraca 0.0. W przypadku braku miejsca w kolejce zwraca stałą `INFINITY`.
- Funkcja `cbuff_pop()` symuluje obsługę i wyjście z kolejki najdłużej czekającego klienta. Funkcja zwraca numer klienta wychodzącego z kolejki, a w przypadku, gdy kolejka była pusta, zwraca -1.
- Funkcja `cbuff_state()` zwraca liczbę czekających klientów.
- Funkcja `cbuff_print()` wypisuje numery czekających klientów (wg kolejności w kolejce).

Test 2.2.3 (kolejki w buforze cyklicznym)
Test jest symulacją kolejki z wykorzystaniem bufora cyklicznego. Wczytywane liczby są kodami operacji na kolejce:

1. Liczba dodatnia oznacza przyjście nowego klienta. Pierwszy klient otrzymuje numer 1. Kolejny klient otrzymuje kolejny numer. Klient, który zastaje pełną kolejkę, rezygnuje z oczekiwania, ale zachowuje swój numer (kolejny klient otrzyma następny numer).
   Klient (jego numer) jest umieszczany w kolejce przez wywołanie funkcji `cbuff_push(nr_klienta)`. Funkcja ta zapisuje przesłany numer w elemencie tablicy (bufora) o indeksie `out + len` (z uwzględnieniem „cykliczności” bufora).
2. Liczba ujemna wywołuje funkcję `cbuff_pop()`, która symuluje obsługę i opuszczenie kolejki przez jednego klienta.

3. Zero powoduje wywołanie funkcji `cbuff_state()` i wypisanie zwracanej przez nią wartości, wywołanie funkcji `cbuff_print()` oraz zakończenie testu.

- **Wejście**
  Test wczytuje numer testu oraz ciąg liczb całkowitych reprezentujących operacje na kolejce.

- **Wyjście**
  Pierwsza linia zawiera:

  – numery klientów wychodzących z kolejki,

  – stałą `INFINITY` w przypadku próby przepełnienia,

  – -1 w przypadku próby symulacji wyjścia klienta nieobecnego w kolejce

  w kolejności pojawiania się ww. zdarzeń.
  W drugiej linii wypisywana jest liczba klientów pozostających w kolejce po zakończeniu symulacji, a w trzeciej linii - ich numery.

- **Przykład:**
  Wejście:
  3
  1 1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 -1 1 0
  Wyjście:
  1 2 -1 inf inf 3
  10
  4 5 6 7 8 9 10 11 12 15

Zadanie 2.3. Symulacja gry Wojna

Zadanie polega na napisaniu programu symulującego grę w karty.

Ogólne zasady gry przyjmujemy za Wikipedią `https://pl.wikipedia.org/wiki/Wojna_(gra_karciana)`. Występuje tam pojęcie "wojna" jako "spotkanie się" kart o takim samym poziomie starszeństwa. Dodajmy termin "konflikt" na określenie bardziej elementarnego zdarzenia - spotkania się dwóch kart, przy którym konieczne jest rozstrzygnięcie relacji starszeństwa między nimi.

- **Uwaga:**
  W czasie wojny zachodzą dwa konflikty, tj. spotkanie pierwszych i trzecich kart (drugie karty nie są ze sobą porównywane), każde przedłużenie wojny to dodatkowy konflikt.

  Rozstrzygnięcie jednej wojny może nastąpić po jednym lub wielu konfliktach. Np. Gracz A wykłada karty: 2 5 8 4 K Q, a równocześnie gracz B: 3 5 9 4 3 A. Liczba konfliktów jest równa 4.

- **Wymagania:**
  Karty posiadane przez uczestnika gry (a dokładniej - ich kody) tworzą

kolejkę zapisaną w buforze cyklicznym (kołowym, pierścieniowym) o rozmiarze równym liczbie kart w talii (ewentualnie o 1 większym).
Należy rozważyć możliwość skorzystania z funkcji zdefiniowanych w zadaniu 2.2.

- Dla jednoznaczności otrzymywanych wyników konieczne jest dodanie kilku ograniczeń, które MUSZĄ być w programie symulującym grę uwzględnione.

  1. Liczba graczy = 2, liczba kart = 52, wg starszeństwa: (2,3,4,5,6,7,8,9,10,J,Q,K,A)*4 kolory, (choć dodatkową zaletą programu byłaby jego elastyczność - zadawana liczba kart i kolorów).
  2. Kodowanie kart. Kolory kart (pik, kier, karo, trefl) nie mają znaczenia przy ustalaniu relacji starszeństwa między nimi. Dla zbliżenia symulacji do rzeczywistości, każdej karcie jest przypisany unikalny kod - liczby naturalne z zakresu [0, liczba kart-1]. Dwójki mają kody 0 - 3, trójki 4 - 7,..., asy 48 - 51.
     Wskazówka: Która (pojedyncza) operacja arytmetyczna (lub bitowa) wykonana na wartości kodu pozwala na "wyrównanie starszeństwa" tych samych figur (lub blotek) różnych kolorów?
  3. Algorytmy
     - tasowania kart: Wg algorytmu funkcji `rand_permutation()`,
     - rozdawania kart graczom:
       Gracz A otrzymuje pierwszą połowę potasowanej talii, a gracz B drugą połowę - w tym miejscu jednoznacznie jest określony (rozróżniony) gracz A i B,
     - wstawiania do kolejki kart zdobytych w każdym konflikcie (w tym, na wojnie):
       Po rozstrzygnięciu konfliktu, zwycięzca przenosi karty leżące na stole na koniec swojej kolejki kart w kolejności - najpierw swoje począwszy od pierwszej wyłożonej na stół, a później karty przeciwnika, w tej samej kolejności,

     są zadane i nie należy ich zmieniać - każda zmiana spowoduje niezgodność wyników otrzymanych i przewidywanych w automatycznym ocenianiu.

- **Wersja uproszczona gry**
  Różni się od opisanej wyżej wersji standardowej inną reakcją na spotkanie się dwóch kart o tej samej mocy. W wersji standardowej dochodzi do "wojny", natomiast w wersji uproszczonej każdy z graczy zabiera ze stołu swoją kartę i wstawia ją na koniec swojej kolejki kart. Liczba konfliktów jest powiększana także w przypadku spotkania się dwóch równoważnych kart. Sprawdzarka automatyczna przyznaje 8 punktów za poprawną wersję uproszczoną i 12 punktów za wersję standardową, czyli za program, który realizuje obie wersje (w zależności od kodu wczytywanego jako druga dana wejściowa – patrz poniżej) można uzyskać 20 punktów.

- **Wejście**
  Wartość startowa generatora liczb pseudolosowych seed (liczba naturalna typu int).
  Kod wybieranej wersji: 0 - standardowa, 1 - uproszczona.
  Maksymalna liczba konfliktów. Jeżeli gra nie zakończy się zwycięstwem jednego z graczy po tej liczbie konfliktów, to gra kończy się wynikiem 0.

- **Wyjście**
  W przypadku:

  – Niedokończenia gry (nie jest wyłoniony zwycięzca po rozstrzygnięciu maksymalnej liczby konfliktów):
    * liczba 0
    * liczba kart gracza A
    * liczba kart gracza B.

  – Nierozstrzygnięcia konfliktu lub wojny (do rozstrzygnięcia ostatniego konfliktu lub wojny zabrakło kart jednemu lub obu graczom):
    * liczba 1
    * liczba kart gracza A
    * liczba kart gracza B.

  – Wygranej gracza A:
    * liczba 2
    * liczba konfliktów, do jakich doszło.

  – Wygranej gracza B:
    * liczba 3
    * ciąg kodów kart jakie gracz B miał po zakończeniu gry – ciąg w kolejności od kodu pierwszej karty przeznaczonej do wyłożenia na stół.

  W sytuacji, gdy gra nie jest dokończona albo jest nierozstrzygnięta, do kart należących do gracza dolicza się jego karty, które położył (i leżą) na stole.

- **Przykład**
  Wejście:
  10444 0 100
  Wyjście:
  3
  43 21 13 10 20 8 48 16 33 23 46 25 18 0 41 14 34 2 49 1 37 27 47 39 5 9 28 19 44 36 38 45 30 24 29 22 6 3 50 17 40 12 15 11 51 26 7 42 35 4 32 31

3 Statystyka

3.1 Średnia i wariancja próby losowej

Szablon programu należy uzupełnić o definicję funkcji
`aver_varian(const double tab[], size_t n, double *arith_average, double *variance)`, która oblicza średnią arytmetyczną oraz wariancję zbioru `n` liczb zapisanych w tablicy `tab`. Obliczone wartości zapisuje w pamięci pod adresami przekazanymi w parametrach `arith_average` i `variance`.

Test 3.1

- **Wejście**
  Numer testu, liczba prób n, n wartości zmiennej losowej.
- **Wyjście**
  Wartości średniej arytmetycznej i wariancji.
- **Przykład:**
  Wejście: 1 4
  -1. 0.5 0. 1.5
  Wyjście: 0.250 0.812

3.2 Tablica wyników prób Bernoulliego

Dla przypomnienia:
Próba Bernoulliego to eksperyment losowy z dwoma możliwymi wynikami, np. rzut monetą z wynikami 0 (reszka, porażka), 1 (orzeł, sukces) .
Przyjmujemy, że moneta nie jest symetryczna, tzn. zadajemy, jakie jest prawdopodobieństwo p rezultatu "orzeł" – wyniku 1 (dla monety symetrycznej byłoby równe 0.5).
Symulację takiego eksperymentu należy zrealizować stosując biblioteczny generator liczb pseudolosowych.
Dla powtarzalności wyników programu należy przyjąć, że wynik próby jest równy 1 gdy wylosowana z przedziału $[0, RAND\_MAX]$ liczba jest mniejsza od $p \cdot (RAND\_MAX + 1)$.

Szablon programu należy uzupełnić o definicję funkcji `bernoulli_gen(...)`, która generuje losowo tablicę n prób Bernoulliego. Elementami tej tablicy mają być wyniki prób.

Test 3.2
Test symuluje wykonanie rzutów monetą. Wczytuje liczbę rzutów i założone prawdopodobieństwo wypadnięcia orła (wyniku = 1) oraz wyprowadza wyniki kolejnych prób.

- **Wejście**
  Numer testu, zarodek generatora `seed`, liczba prób n, prawdopodobieństwo p wypadnięcia orła (jedynki).
- **Wyjście**
  Wyniki symulacji n prób.
- **Przykład:**
  Wejście: 2 2 20 0.7
  Wyjście: 0 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1

3.3 Dyskretny rozkład prawdopodobieństwa

Dyskretny rozkład prawdopodobieństwa jest to rozkład prawdopodobieństwa zmiennej losowej, której zbiór możliwych wartości jest przeliczalny. Zdarzeniem losowym w tym zadaniu jest rzut dwoma sześciennymi kostkami do gry w kości. Wartością zmiennej losowej jest suma oczek tych dwóch kostek, czyli liczba z przedziału $[2, 12]$.
Rezultatem tego zadania ma być przybliżony rozkład prawdopodobieństwa tej zmiennej losowej. Przybliżony, bo

1. zamiast rzutu kostkami stosujemy generator liczb pseudolosowych,
2. wykonujemy tylko skończoną liczbę prób.

Szablon programu należy uzupełnić o definicję funkcji `two_dice_probab_distrib(double distrib[], int throws_num)`, która symuluje wykonanie `throws_num` rzutów dwoma kostkami i zapisuje wartości otrzymanego przybliżonego rozkładu prawdopodobieństwa w 11-elementowej tablicy `distrib`. Dla uzyskania powtarzalności wyników, losując liczbę oczek jednej kostki należy tylko raz wywołać funkcję `rand()`.

Test 3.3
Test wczytuje dane, wywołuje funkcję `two_dice_probab_distrib(distrib, throws_num)` i wyprowadza wartości obliczonego rozkładu.

- **Wejście**
  Numer testu, zarodek generatora `seed`, liczba prób `throws_num`.
- **Wyjście**
  Wartości rozkładu prawdopodobieństwa w postaci liczbowej.
- **Przykład:**
  Wejście: 3 20 1000
  Wyjście: 0.024 0.063 0.091 0.095 0.146 0.159 0.146 0.107 0.089 0.056 0.024

3.4 Dysrybuanta (ang. Cumulative Distribution Function)

Szablon programu należy uzupełnić o definicję funkcji
`cum_discret_distrib(double distrib[], size_t n)`,
która na podstawie danego dyskretnego rozkładu prawdopodobieństwa (zapisanego w `n` elementowej tablicy `distrib`) oblicza wartości dystrybuanty dla każdej wartości zmiennej losowej [1]). Obliczone wartości dystrybuanty zapisuje w tablicy `distrib` – na miejscu danych wejściowych.

Test 3.4
Test wywołuje najpierw funkcję `two_dice_probab_distrib(distrib, throws_num)`, która oblicza rozkład prawdopodobieństwa, a następnie wywołuje funkcję `cum_discret_distrib(distrib, n)`, która oblicza wartości dystrybuanty tego rozkładu. Obliczone wartości są wypisywane w postaci liczbowej.

- **Wejście**
  Numer testu, `seed`, liczba prób (rzutów) `throws_num`.
- **Wyjście**
  Wartości dystrybuanty obliczone dla kolejnych wartości zmiennej losowej.
- **Przykład:**
  Wejście: 4 20 1000
  Wyjście: 0.024 0.087 0.178 0.273 0.419 0.578 0.724 0.831 0.920 0.976 1.000

3.5 Histogram

Szablon programu należy uzupełnić o definicję funkcji
`histogram(double tab[], size_t n, int x_start, double y_scale, char mark)`, która w trybie znakowym przedstawia histogram funkcji o `n` wartościach zapisanych w tablicy `tab` o długości `n`. Należy przyjąć założenia:

1. Oś zmiennej niezależnej jest pionowa, skierowana w dół. Oś zmiennej zależnej jest pozioma, skierowana w prawo (nie jest rysowana).
2. Wartości zmiennej niezależnej są kolejnymi liczbami naturalnymi, począwszy od `x_start`. Są one pisane od pierwszej lewej kolumny, w polu o szerokości 2 znaków z wyrównaniem w prawo.
3. W trzeciej kolumnie są spacje, a w czwartej – znaki |, które tworzą oś x.
4. Począwszy od 5. kolumny pisane są znaki `mark`. Liczba znaków jest przeskalowaną i zaokrągloną wartością funkcji. Parametr `y_scale` jest wartością zmiennej zależnej odpowiadającej szerokości jednego znaku na wykresie.
5. Wartości funkcji (liczby nieujemne typu `double`) są wyprowadzane z dokładnością do 3 cyfr po przecinku w każdym wierszu, po jednej spacji na prawo od ostatniego znaku `mark`.

[1] Dla przypomnienia definicji i własności dystrybuanty: `https://en.wikipedia.org/wiki/Cumulative_distribution_function`, w szczególności wykresy w części Properties ilustrujące punkty skokowe i prawostronną ciągłość dystrybuanty dyskretnej.

Test 3.5
Test jest modyfikacją testu 3.3 sprowadzającą się do zmiany sposobu wyprowadzenia wyniku obliczeń - postać liczbowa jest zastąpiona histogramem.
Znak `mark` jest wczytywany ze strumienia wejściowego. Wartości skali `y_scale` jest ustalona w funkcji main() (podana w postaci stałej).

- **Wejście** – Numer testu, `seed`, liczba prób (rzutów) `throws_num` (jak w teście 3.3) oraz – po dowolnej liczbie białych znaków – znak `mark`.
- **Wyjście** – histogram rozkładu prawdopodobieństwa dla rzutu dwoma kostkami
- **Przykład**
  Wejście: 5 20 1000 *
  Wyjście:

```
 2 |***** 0.024
 3 |************* 0.063
 4 |****************** 0.091
 5 |******************* 0.095
 6 |****************************** 0.146
 7 |******************************** 0.159
 8 |****************************** 0.146
 9 |********************* 0.107
10 |****************** 0.089
11 |********** 0.056
12 |***** 0.024
```

Test 3.6
Test 3.6 jest analogiczny do testu 3.5 – jest modyfikacją testu 3.4 sprowadzającą się do zmiany sposobu wyprowadzenia wyniku obliczeń - postać liczbowa jest zastąpiona histogramem.

- **Wejście** – Numer testu, `seed`, liczba prób (rzutów) `throws_num` (jak w teście 3.4) oraz znak `mark`.
- **Wyjście** – histogram dystrybuanty dla rzutu dwoma kostkami.
- **Przykład**
  Wejście: 6 20 1000 #
  Wyjście:

```
 2 |# 0.024
 3 |#### 0.087
 4 |######### 0.178
 5 |############### 0.273
 6 |####################### 0.419
 7 |################################ 0.578
 8 |######################################## 0.724
 9 |############################################### 0.831
10 |################################################### 0.920
11 |###################################################### 0.976
12 |######################################################## 1.000
```

3.7 Monty Hall problem, czyli jak wybierać „drzwi”, aby zwiększyć prawdopodobieństwo wygranej

Paradoks Monty'ego Halla, w przypadku trojga drzwi (bramek) do wyboru, polega na tym, że intuicyjnie przypisujemy równe szanse dwóm sytuacjom — wskazanie wygranej w jednej z dwóch zakrytych ciągle bramek wydaje się równie prawdopodobne jak wskazanie bramki pustej, bo przecież „nic nie wiadomo”. Tymczasem układ jest warunkowany przez początkowy wybór zawodnika i obie sytuacje nie pojawiają się równie często.
Opis problemu: `https://pl.wikipedia.org/wiki/Paradoks_Monty'ego_Halla`.
Szablon programu należy uzupełnić o definicję funkcji
`monty_hall(int *p_switch_wins, int *p_nonswitch_wins, int n)`,
która symuluje `n` rozgrywek. Założenia:

1. W każdej rozgrywce funkcja wywołuje `rand()` dokładnie 2 razy.
2. W pierwszym losowaniu jest wybierany numer drzwi, za którymi jest nagroda.
3. W drugim losowaniu – numer drzwi, które gracz wybiera na początku gry.

Funkcja oblicza (i przekazuje przez adresy przekazane do parametrów `p_switch_wins` i `p_nonswitch_wins`) ile razy w ciągu `n` rozgrywek wygrywał gracz, który po otwarciu jednych drzwi zmieniał pierwotną decyzję i ile razy wygrywał gracz pozostający przy początkowym wyborze.

Test 3.7
Test wczytuje liczbę prób (rozgrywek), wywołuje funkcję `monty_hall(...)` i wypisuje wyniki symulacji.

- **Wejście**
  Numer testu, `seed`, liczba prób (rozgrywek). `n`
- **Wyjście**
  Liczba wygrywających decyzji „zmień wybór” i liczba wygrywających decyzji „nie zmieniaj”.
- **Przykład**
  Wejście: 7 15 1000
  Wyjście: 656 344

4 Operacje na znakach odczytywanych ze stdin

Wskazówka dotycząca strumienia danych wejściowych:
Stała `TEST` jest przełącznikiem między pracą w trybie testowania programu (= 1), a wersją przygotowaną do automatycznej oceny (=0, pomija wyprowadzanie komunikatów dla użytkownika).

W zadaniach tego rozdziału stała ta pozwala przełączać strumień wejściowy programu między wejście standardowe (klawiaturę) `stdin` i plikiem wskazanym przez użytkownika.

- Gdy `TEST` = 1, linie tekstu są wczytywane z klawiatury aż do znaku Ctrl+D (UNIX) albo Ctrl+Z (Windows).
- Gdy `TEST` = 0, ale program nie jest przesyłany do oceny automatycznej, to po wyborze zadania (np. 3) należy wpisać nazwę pliku zawierającego tekst (oraz ewentualne dalsze parametry zadania).

Plik zawierający dane testowe należy utworzyć obok pliku z źródłową wersją pisanego programu (w opcji Edycja, po rozwinięciu menu ikonką + pojawia się ikonka dodawania nowego pliku).

4.1 Zadanie 4.1

4.1.1 Zliczanie linii, słów i znaków

Szablon programu należy uzupełnić o definicję funkcji `wc()`, która czyta tekst ze standardowego wejścia i na tej podstawie zlicza linie, słowa oraz znaki, występujące w tym tekście, podobnie jak komenda `wc` systemu Unix. Słowo to ciąg znaków oddzielony spacją, tabulatorem lub znakiem nowej linii.
Opis komendy `wc`: https://en.wikipedia.org/wiki/Wc_(Unix)

- **Wejście**
  1
  `linie tekstu`
- **Wyjście**
  Liczba linii, słów i znaków w tekście
- **Przykład:**
  Wejście:

  ```
  1
  int main() {
      printf ("Hello\n");
      return 0;
  }
  ```

  Wyjście: 4 8 47

4.1.2 Liczności znaków

Szablon programu należy uzupełnić o definicję funkcji `char_count()`, która czyta tekst ze standardowego wejścia i na tej podstawie zlicza krotności znaków występujących w tym tekście.

Rozpatrujemy znaki należące do przedziału `[FIRST_CHAR, LAST_CHAR-1]`. Powyższe stałe są zdefiniowane w szablonie programu.

Funkcja następnie sortuje liczności znaków malejąco (sortujemy indeksy a nie samą tablicę zliczającą) i zwraca, poprzez parametry `n_char` i `cnt`, `char_no`-ty (co do liczności) znak tekstu oraz liczbę jego wystąpień. W przypadku jednakowej liczności znaki powinny być posortowane alfabetycznie.

- **Wejście**
  2
  `char_no`
  `linie tekstu`
- **Wyjście**
  `char_no`-ty najliczniejszy znak i liczba jego wystąpień
- **Przykład:**
  Wejście:

  ```
  2
  1
  int main() {
      printf ("Hello\n");
      return 0;
  }
  ```

  Wyjście: n 5

4.1.3 Zliczanie digramów

Szablon programu należy uzupełnić o definicję funkcji `digram_count()`, która czyta tekst ze standardowego wejścia i na tej podstawie zlicza krotności digramów znakowych (par znaków) występujących w tym tekście.

Rozpatrujemy znaki należące do przedziału `[FIRST_CHAR, LAST_CHAR-1])`. Powyższe stałe są zdefiniowane w szablonie programu.

Funkcja następnie sortuje liczności digramów malejąco (sortujemy indeksy a nie samą tablicę zliczającą) i zwraca, poprzez parametry `n_char` i `cnt`, `digram_no`-ty (co do liczności) digram tekstu oraz liczbę jego wystąpień. W przypadku jednakowej liczności digramy powinny być posortowane alfabetycznie.

- **Wejście**
  3
  `digram_no`
  `linie tekstu`

- **Wyjście**
  `digram_no`-ty najliczniejszy digram (dwa znaki bez spacji) i liczba jego wystąpień
- **Przykład:**
  Wejście:

```
3
1
int main() {
    printf ("Hello\n");
    return 0;
}
```

  Wyjście: in 3

4.1.4 Zliczanie komentarzy

Szablon programu należy uzupełnić o definicję funkcji `find_comments()`, która czyta ze standardowego wejścia ciąg znaków stanowiący program w języku C. Funkcja zlicza komentarze blokowe (`/* ... */`) i jednoliniowe (`// ...`) w przeczytanym tekście i zwraca uzyskane liczby do funkcji `main()` przy użyciu parametrów.

Zagnieżdżone komentarze nie są liczone, czyli np. następujący fragment kodu:
`/*// tekst*/`
jest uważany za jeden komentarz blokowy.
Można założyć, że wszystkie komentarze blokowe są prawidłowo zamknięte.

- **Wejście**
  4
  `linie tekstu`
- **Wyjście**
  Liczba komentarzy blokowych i liniowych
- **Przykład:**
  Wejście:

```
4
int main() { // comment
    printf ("Hello\n"); /* and another */
    return 0;
    /* and more
    and more ...
    */
}
```

  Wyjście: 2 1

4.2 Zadanie 4.2

4.2.1 Znajdowanie identyfikatorów

Szablon programu należy uzupełnić o definicję funkcji `find_idents()`, która czyta ze standardowego wejścia ciąg znaków stanowiący program w języku C. Funkcja zlicza **unikalne** identyfikatory zawarte w przeczytanym tekście i zwraca uzyskaną liczbę do funkcji `main()`.

Definicja identyfikatora jest taka jak w języku C, czyli jest to ciąg liter i cyfr zaczynający się od litery; znak podkreślnika jest uważany za literę, czyli na przykład identyfikator `_a` jest legalny.

Uwaga 1: W programie wejściowym mogą znajdować się stałe znakowe, napisowe (ciągi znaków ujęte w cudzysłowy) oraz komentarze (jak w zadaniu o komentarzach). Powinny one być pomijane przy liczeniu identyfikatorów.

Uwaga 2: Dla ujednolicenia templatka programu zawiera słowa kluczowe, które należy wyłączyć z listy identyfikatorów.

- **Wejście**
  `linie tekstu`
- **Wyjście**
  liczba unikalnych identyfikatorów zawartych w tekście nie będących słowami kluczowymi
- **Przykład:**
  Wejście:

```
int main() { // comment
    printf ("Hello\n"); /* and another */
    printf ("Greetings\n");
    return 0;
}
```

  Wyjście: 2

5 Operacje na macierzach

5.1 Funkcje „pomocnicze”

W tej części zadania będą definiowane podstawowe funkcje przeznaczone do wykorzystania w bardziej złożonych algorytmach przedstawionych w drugiej części tematu.

5.1.1 Permutacje wierszy tablicy

W zadaniu zastosowane są dwa sposoby implementacji tablicy łańcuchów znakowych:

- tablica wskaźników `char *keywords_ptab[N]`,
- dwuwymiarowa tablica znakowa (dokładniej: tablica tablic znakowych) `char keywords_t2D[N][STRLEN_MAX]`.

Zadanie polega na sortowaniu łańcuchów znakowych zapisanych w tych tablicach bez zmiany ich położenia w pamięci, tj. w czasie sortowania każdy łańcuch pozostaje w pamięci pod niezmienionym adresem. Należy uzupełnić definicje funkcji:

1. `ptab_sort(char *ptab[], size_t n)`. W tablicy `ptab` są zapisane adresy łańcuchów znakowych. Funkcja sortuje elementy tej tablicy (adresy) w kolejności alfabetycznej łańcuchów, na które te elementy wskazują. Do sortowania należy użyć – podobnie jak w zadaniu 4.1 – biblioteczną funkcję `qsort(...)`. Funkcja ta wywołuje funkcję `compar(...)`, którą też należy zdefiniować.
2. `t2D_sort(const char t2D[][STRLEN_MAX], size_t indices[], size_t n)`. W `n` wierszach tablicy `t2D` są zapisane łańcuchy znaków. Stosując algorytm sortowania bąbelkowego, funkcja ma uporządkować elementy tablicy tej w kolejności odpowiadającej odwrotnemu porządkowi alfabetycznemu wskazywanych wierszy[1]. Istotnym warunkiem jest pozostawienie łańcuchów w tym samym miejscu w pamięci (w tych samych wierszach tablicy), w jakim były przed sortowaniem. Zadany porządek łańcuchów ma być zapisany w wektorze permutacji indeksów `indices`.
   W tablicy `indices` mają być zapisane indeksy wierszy tablicy `t2D`.
3. `n_str_copy(char t2D[][STRLEN_MAX],char *ptab[],size_t n)`, która kopiuje łańcuchy wskazywane przez elementy tablicy wskaźników `ptab` do tablicy `t2D`.
   Uwaga: Założenie o pozostawaniu łańcuchów w tym samym miejscu pamięci dotyczy tylko sortowania – w tej funkcji kopiowanie jest dozwolone.
4. `print_ptab(char *ptab[], size_t n)`, która pisze łańcuchy znakowe wskazywane przez `n` pierwszych elementów tablicy `ptab`.
5. `print_t2D_ind(const char (*ptr)[STRLEN_MAX],const size_t *pindices,size_t n)`. Funkcja wyprowadza na ekran `n` łańcuchów znakowych zapisanych w tablicy tablic. Kolejność wypisywanych łańcuchów jest określona tablicą permutacji indeksów `indices`.
   Pierwszy parametr (argument formalny) funkcji jest zmienną typu wskaźnikowego do tablicy o `STRLEN_MAX` elementach typu `char`. W porównaniu z definicją pierwszego parametru funkcji `t2D_sort` (używającą operatora `[][]`), taka definicja tego parametru jest bardziej „naturalna” dla języka C – jawnie pokazuje, że do funkcji jest przekazywany adres pierwszego elementu tablicy (tym elementem jest tablica `STRLEN_MAX` znaków).
   Definicję funkcji należy zapisać wybierając jeden z 4 sposobów: z użyciem dwóch operatorów `[]`, dwóch operatorów dereferencji `*`, dwóch możliwości „mieszanych” (z jednym `[]` i jednym *). Poprawność pozostałych 3 wariantów też należy sprawdzić.

Test 1
Łańcuchy znaków (słowa kluczowe języka C) są definiowane w momencie inicjowania elementów tablicy wskaźników `keywords_ptab[]`. Funkcja `n_str_copy()` kopiuje je do tablicy znakowej `keywords_t2D[]`. Test wywołuje funkcje `ptab_sort()` i `t2D_sort()`, wczytuje liczbę łańcuchów `n` i wypisuje w uporządkowanej kolejności `n` łańcuchów z tablic `ptab` i `t2D`.

- **Wejście**
  Numer testu, liczba wypisywanych łańcuchów n
- **Wyjście**
  n początkowych łańcuchów uporządkowanej tablicy wskaźników
  n początkowych łańcuchów uporządkowanej tablicy tablic znaków

[1] Do określania alfabetycznej kolejności łańcuchów należy korzystać z bibliotecznej funkcji `strcmp(...)` lub `strncmp(...)`.

- **Przykład:**
  Wejście:
  1 3
  Wyjście:
  auto
  break
  case
  while
  volatile
  void

5.1.2 Mnożenie macierzy

Szablon programu należy uzupełnić o definicję funkcji `mac_product(double A[][SIZE], double B[][SIZE], double AB[][SIZE], size_t m, size_t p, size_t n)`. Macierz A o wymiarach $m \times p$ jest zapisana w tablicy `A`, a macierz B o wymiarach $p \times n$ jest zapisana w tablicy `B` (liczba kolumn `SIZE` $\geqslant m, p, n$).Funkcja oblicza iloczyn macierzy $A \cdot B$ i zapisuje go w tablicy `AB`.

Test 2

- **Wejście**
  Numer testu, liczba wierszy i liczba kolumn macierzy A
  elementy macierzy A
  liczba wierszy i liczba kolumn macierzy B
  elementy macierzy B
- **Wyjście**
  elementy macierzy AB
- **Przykład:**
  Wejście:
  2
  2 3
  1 2 3 10 20 30
  3 2
  11 23 1 1.5 -2 0
  Wyjście:
  7.0000 26.0000
  70.0000 260.0000

5.1.3 Triangularyzacja macierzy i obliczanie wyznacznika - wersja uproszczona (bez zamiany wierszy)

Tablice tablic `A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]` są zdefiniowane i wypełniane wczytanymi danymi w segmencie głównym `main`. Rozmiarów tych tablic nie należy zmieniać.

Szablon programu należy uzupełnić o definicję funkcji `gauss_simplified(double A[][SIZE], size_t n)`. Macierz A o wymiarach $n \times n$ jest zapisana w tablicy `A` (liczba kolumn `SIZE` $\geqslant n$). Funkcja przekształca macierz kwadratową A do postaci trójkątnej górnej metodą Gaussa, zapisuje ją w tablicy `A` i zwraca wartość wyznacznika. W przypadku, gdy element na przekątnej głównej jest równy zeru, to triangularyzacja nie jest kontynuowana, a wyznacznik = NAN.

Test 3

- **Wejście**
  Numer testu, liczba wierszy (i kolumn) macierzy A
  elementy macierzy A
- **Wyjście**
  wyznacznik macierzy
  elementy macierzy A
- **Przykład 1:**
  Wejście:
  3
  3
  3 5 7
  1 -3 8
  2 4 -2

Wyjście:
82.0000
3.0000 5.0000 7.0000
0.0000 -4.6667 5.6667
0.0000 0.0000 -5.8571

- **Przykład 2:**
  Wejście:
  3
  3
  1.25 0.125 -2.5
  5.0 0.5 -3.2
  2.5 1.8 0.

  Wyjście:
  nan
  1.2500 0.1250 -2.5000
  0.0000 0.0000 6.8000
  0.0000 1.5500 5.0000

5.2 Rozwiązywanie układu równań liniowych, odwracanie macierzy

5.2.1 Rozwiązywanie układu równań liniowych metodą Gaussa - wersja z rozszerzaną macierzą współczynników

Szablon programu należy uzupełnić o definicję funkcji
`gauss(double A[][SIZE], const double b[], double x[], size_t n, double eps)`, która przekształca macierz kwadratową A zapisaną w tablicy `A` do postaci trójkątnej górnej metodą Gaussa i zwraca wartość wyznacznika. Wiersze macierzy są zamieniane tak, aby wartość bezwzględna elementu głównego była największa. Zamiana wierszy nie jest realizowana poprzez przepisanie wierszy w tablicy, lecz z zastosowaniem wektora permutacji indeksów wierszy. W przypadku, gdy po zamianie wierszy element na przekątnej głównej jest mniejszy od `eps`, to triangularyzacja nie jest dokończana, a wyznacznik przyjmuje wartość 0.
Jeżeli argumenty funkcji `b` i `x` oraz wyznacznik nie są zerowe, to funkcja rozwiązuje układ równań i rozwiązanie zapisuje w tablicy `x`.
Funkcja może zmienić wartości elementów tablicy `A`.
Poprawność funkcji można sprawdzić korzystając z funkcji `mac_vec_product`

Test 4

- **Wejście**
  Numer testu
  liczba wierszy i macierzy `A`
  elementy macierzy `A`
  elementy wektora `b`

- **Wyjście**
  wyznacznik macierzy
  elementy wektora `x`

- **Przykład:**
  Wejście:
  4
  4
  1 -1 2 -1
  2 -2 3 -3
  1 1 1 0
  1 -1 4 3
  -8 -20 -2 4

  Wyjście:
  4.0000
  -7.0000 3.0000 2.0000 2.0000

5.2.2 Odwracanie macierzy kwadratowej metodą Gaussa - Jordana

Szablon programu należy uzupełnić o definicję funkcji
`matrix_inv(double A[][SIZE], double B[][SIZE], size_t n, double eps)`, która wyznacza (i zapamiętuje w tablicy `B`) macierz odwrotną do nieosobliwej macierzy A zapisanej w tablicy `A`. Należy zastosować metodę Gaussa - Jordana z rozszerzaniem macierzy A o macierz jednostkową. Wiersze macierzy rozszerzonej są zamieniane analogicznie do zadania 5.2.1. Funkcja zwraca wyznacznik macierzy A. W przypadku, gdy po zamianie wierszy element na przekątnej głównej jest mniejszy od `eps`, to algorytm odwracania nie jest kończony, i wyprowadzany jest tylko wyznacznik = 0 (układ równań nie jest rozwiązywany).
Funkcja może zmienić wartości elementów tablicy `A`.
Poprawność funkcji można sprawdzić korzystając z funkcji `mac_product()`.

Test 5

- **Wejście**
  Numer testu
  liczba wierszy i macierzy `A`
  elementy macierzy `A`
- **Wyjście**
  wyznacznik macierzy
  elementy macierzy odwrotnej `B`
- **Przykład:**
  Wejście:
  5
  3
  1 2 -1
  2 1 0
  -1 1 2

  Wyjście:
  -9.000
  -0.2222 0.5556 -0.1111
  0.4444 -0.1111 0.2222
  -0.3333 0.3333 0.3333

7 Relacje

Uwagi ogólne

Celem ćwiczenia jest zapoznanie się z operacjami na strukturach i tablicach struktur.

Oprócz zadań opisanych w kolejnych sekcjach, program należy dodatkowo uzupełnic o funkcje:

1. `add_to_relation()`, która dodaje parę liczb do relacji, jeżeli ta para jeszcze w tej relacji nie występuje,
2. `read_relation()`, która czyta liczbę par relacji (n), a następnie n par liczb całkowitych definiujących relację,
3. `print_int_array()`, która wypisuje długość tablicy całkowitej (n), a następnie n wartości tej tablicy.

Zadania

7.1 Własności relacji

Szablon programu należy uzupełnić o definicję funkcji:

1. `is_reflexive()`, która sprawdza, czy relacja jest zwrotna
2. `is_irreflexive()`, która sprawdza, czy relacja jest przeciwzwrotna
3. `is_symmetric()`, która sprawdza, czy relacja jest symetryczna
4. `is_antisymmetric()`, która sprawdza, czy relacja jest antysymetryczna
5. `is_asymmetric()`, która sprawdza, czy relacja jest asymetryczna
6. `is_transitive()`, która sprawdza, czy relacja jest przechodnia

Użyteczny link: `https://en.wikipedia.org/wiki/Homogeneous_relation`.

- **Wejście**
  1
  n (liczba elementów relacji)
  n linii składających się z par liczb naturalnych `(first, second)`
- **Wyjście**
  Sześć wartości 0 lub 1 reprezentujących własności zadanej relacji
- **Przykład:**
  Wejście:

```
1
10
2 3
2 4
3 4
0 1
0 2
0 3
0 4
1 3
1 2
1 4
```

Wyjście:

```
0 1 0 1 1 1
```

7.2 Porządki, wartości minimalne i maksymalne

Szablon programu należy uzupełnić o definicję funkcji:

1. `is_partial_order()`, która sprawdza, czy relacja jest relacją częściowego porządku
2. `is_total_order()`, która sprawdza, czy relacja jest relacją całkowitego porządku
3. `find_max_elements()`, która dla relacji porządku znajduje elementy maksymalne
4. `find_min_elements()`, która dla relacji porządku znajduje elementy minimalne
5. `find_domain()`, która znajduje dziedzinę relacji R (zbiór X, na iloczynie kartezjańskim którego opisana jest relacja, czyli $R \subseteq X \times X$)

Relacja $R \subseteq X \times X$ jest relacją częściowego porządku jeżeli jest zwrotna, antysymetryczna i przechodnia.

Element $g \in X$ jest elementem maksymalnym jeżeli nie istnieje element $x \in X$, taki że $x \neq g$ i gRx. Podobnie element $m \in X$ jest elementem minimalnym jeżeli nie istnieje element $x \in X$ taki, że $x \neq m$ i xRm.

Zbiór X częściowo uporządkowany przez relację R może zawierać kilka elementów maksymalnych / minimalnych.

Relacja $R \subseteq X \times X$ jest spójna jeżeli dla każdych x, $y \in X$ zachodzi xRy lub yRx (lub jedno i drugie).

Jeżeli relacja $R \subseteq X \times X$ jest relacją częściowego porządku oraz jest spójna, to zbiór X jest liniowo (całkowicie) uporządkowany przez R.

Dziedzinę (zbiór X) wyznaczamy jako unikalną tablicę poprzedników i następników par należących do relacji.

Pomocne linki:
`https://en.wikipedia.org/wiki/Partially_ordered_set`.
`https://en.wikipedia.org/wiki/Total_order`

- **Wejście**
  2
  `n` (liczba elementów relacji)
  `n` linii składających się z par liczb naturalnych `(first, second)`

- **Wyjście**
  Dwie wartości 1 lub 0 (relacja jest / nie jest relacją częściowego / całkowitego porządku)
  `d` (liczność domeny relacji)
  `d` liczb całkowitych (domena relacji)
  Jeżeli relacja jest relacją częściowego porządku to dodatkowo program wyprowadza:
  `max` (liczba wartości maksymalnych)
  `max` liczb całkowitych (wartości maksymalne)
  `min` (liczba wartości minimalnych)
  `min` liczb całkowitych (wartości minimalne)

  Uwaga: Wszystkie trzy tablice wyjściowe powinny zawierać unikalne liczby całkowite w porządku rosnącym.

- **Przykład:**
  Wejście:

  ```
  2
  12
  1 4
  1 1
  1 5
  1 6
  2 4
  2 2
  2 6
  3 4
  3 3
  4 4
  6 6
  5 5
  ```

  Wyjście:

  ```
  1 0
  6
  1 2 3 4 5 6
  3
  ```

```
4 5 6
3
1 2 3
```

7.3 Złożenie relacji

Szablon programu należy uzupełnić o definicję funkcji `composition()`, która wyznacza złożenie dwóch zadanych relacji.

Użyteczny link:
`https://en.wikipedia.org/wiki/Composition_of_relations`

- **Wejście**
  3
  `n1`
  relacja `R` – `n1` par liczb całkowitych
  `n2`
  relacja `S` – `n2` par liczb całkowitych

- **Wyjście**

  `n3` – liczba elementów relacji złożonej `S o R`

- **Przykład:**

  Wejście:

  ```
  3
  7
  1 2
  2 3
  3 4
  3 2
  2 5
  1 5
  2 4
  6
  2 4
  1 3
  5 4
  3 5
  3 1
  1 2
  ```

  Wyjście:
  ```
  5
  ```

7 Zastosowanie wskaźników do funkcji w obliczaniu całek

Całka a kwadratura.
Obliczanie całki zastępujemy obliczaniem kwadratury – numerycznego przybliżenia wartości całki Riemanna. Do najbardziej elementarnych metod obliczenia przybliżenia Q całki $\int_a^b f(x)dx$ należą kwadratury (wzory):

1. prostokątów
$$Q_R = (b-a)f(c), \quad c \in [a,b],$$
   - prostokątów w przód (lewostronna), gdy $c = a$,
   - prostokątów wstecz (prawostronna), gdy $c = b$,
   - prostokątów punktu środkowego (centralna), gdy $c = (a+b)/2$.
2. trapezów
$$Q_T = \frac{b-a}{2}[f(a) + f(b)],$$
3. Simpsona
$$Q_s = \frac{b-a}{6}\left[f(a) + 4f(c) + f(b)\right], \quad c = (a+b)/2.$$

W praktyce stosuje się je w wersjach złożonych (zadania 7.1.1 i 7.1.2) lub w algorytmach adaptacyjnych (7.1.3)

7.1 Całki jednokrotne – Kwadratury złożone

Złożona kwadratura wybranego typu polega na podziale przedziału całkowania $[a,b]$ na n równych podprzedziałów o długości $h = (b-a)/n$ i zsumowanie kwadratur prostych tego typu obliczonych dla każdego podprzedziału. Np. złożona kwadratura prostokątów w przód (leftpoint) jest sumą

$$C_{R_{left}} = h\sum_{i=0}^{n-1} f(a+ih).$$

7.1.1 Funkcje z wskaźnikiem na funkcję podcałkową

Szablon programu należy uzupełnić o:

1. definicje funkcji (procedur) obliczających wartości przykładowych funkcji podcałkowych
   - `f_poly(double x)`, która zwraca wartość wielomianu $f(x) = 2x^5 - 4x^4 + 3.5x^2 + 1.35x - 6.25$,
   - `f_rat(double x)`, która zwraca wartość funkcji $f(x) = \frac{1}{(x-0.5)^2+0.01}$,
   - `f_exp(double x)`, która zwraca wartość funkcji $f(x) = 2xe^{-1.5x} - 1$,
   - `f_trig(double x)`, która zwraca wartość funkcji $f(x) = x\mathrm{tg}(x)$
2. definicję nazwy `typedef ...Func1vFp...;` – typu wskaźnikowego do funkcji z jednym parametrem typu `double` i zwracającą wartość typu `double`.
3. definicje funkcji obliczających złożoną kwadraturę dla funkcji `f` z podziałem przedziału całkowania $[a,b]$ na `n` podprzedziałów
   - prostokątów w przód (leftpoint) – `quad_rect_left(...)`,
   - prostokątów wstecz (rightpoint)– `quad_rect_right(...)`,
   - prostokątów punktu środkowego (midpoint) – `quad_rect_mid(...)`,
   - trapezów – `quad_trap(...)`,
   - Simpsona – `quad_simpson(...)`.

   W ostatnich dwóch kwadraturach należy unikać dwukrotnego obliczania wartości funkcji podcałkowej dla tego samego argumentu.
   Każda z tych funkcji ma 4 argumenty:
   (a) wskaźnik do funkcji obliczającej wartość funkcji podcałkowej `f`,
   (b) dolną granicę całkowania `a`,
   (c) górną granicę całkowania `b`,
   (d) liczbę podprzedziałów `n`.
4. definicję nazwy `typedef ...QuadratureFp...` – typu wskaźnikowego do funkcji obliczających wartości kwadratury.

Tablice wskaźników na funkcje
Szablon programu należy uzupełnić o definicję funkcji
`double quad_select(int quad_no,int fun_no,double a,double b,int n)`,
która w przedziale $[a, b]$ oblicza kwadraturę złożoną wskazaną indeksem `quad_no` (z podziałem na `n` podprzedziałow) dla funkcji podcałkowej wskazanej indeksem `fun_no`. Na zewnątrz funkcji `quad_select()` jest definiowana tablica wskaźników do funkcji typu `Func1vFp` oraz tablica typu `QuadratureFp`. Obie tablice są inicjowane wskaźnikami do funkcji zdefiniowanych w punkcie 7.1.1.

Test 1
Wczytuje granice przedziału całkowanie oraz liczbę podprzedziałów i 20 razy wywołuje funkcję `quad_select()`. Dwa pierwsze argumenty tej funkcji są parą iloczynu kartezjańskiego zbioru indeksów tablicy wskaźników do kwadratur `quad_tab` i zbioru indeksów tablicy wskaźników do funkcji podcałkowych `func_tab`.

- **Wejście**
  Nr testu
  granice przedziału całkowania, liczba podprzedziałów
- **Wyjście**
  20 wartości każdej kwadratury dla każdej funkcji
- **Przykład:**
  Wejście:
  1
  0 0.75 25
  Wyjście:
  -3.97887 25.47947 0.14924 -0.48180
  -3.91316 25.77788 0.17020 -0.46719
  -3.94620 25.64102 0.15945 -0.47427
  -3.94602 25.62868 0.15972 -0.47450
  -3.94614 25.63690 0.15954 -0.47434

7.1.2 Algorytm adaptacyjny w wersji rekurencyjnej

W algorytmie jest stosowana jedna, elementarna kwadratura w wersji podstawowej (tzn. nie złożonej). Na każdym etapie obliczeń wyznaczamy przybliżoną wartość S całki z funkcji f w pewnym podprzedziale przedziału $[a, b]$ wg podstawowego wzoru wybranej kwadratury.

Błąd przybliżenia wartości całki kwadraturą jest mały jeżeli długość h podprzedziału, w którym jest obliczana całka, jest mała. Algorytm adaptacyjny ma na celu osiągnięcie wyniku (przybliżonej wartości całki) z błędem bezwzględnym nie większym niż zadany $= \Delta$ skracając długości podprzedziałów tylko tam, gdzie to jest konieczne.

Pierwsze przybliżenie całki jest obliczane dla całego przedziału $[a, b]$. Następnie ten przedział jest dzielony na 2 połowy. Wzór podstawowy wybranej kwadratury jest teraz stosowany osobno dla lewej połówki (od a do $c = (a+b)/2$ i dla prawej (od c do b). Otrzymujemy przybliżone wartości dwóch części obliczanej całki - S_1 i S_2. Jeżeli suma $S_1 + S_2$ różni się od S nie więcej niż o Δ, to uznajemy, że otrzymany wynik jest dostatecznie dokładny i kończymy algorytm. W przeciwnym przypadku stajemy przed dwoma zadaniami - obliczyć całkę na dwóch połówkach (lewej i prawej), każdej z błędem nie większym niż $\Delta/2$. Zauważmy, że są to jakościowo dokładnie dwa takie same zadania, jak zadanie pierwotne (obliczyć całkę z błędem nie większym niż zadany).

Należy tak napisać program, aby liczba obliczeń wartości zadanej funkcji była jak najmniejsza - aby nie obliczać dwukrotnie funkcji dla tego samego argumentu. W tym celu, obliczona na danym poziomie rekurencji wartość kwadratury jest przekazywana przez parametry na kolejny poziom.

W funkcji rekurencyjnej powinna być kontrola poziomu rekursji. Załóżmy, że jeżeli maksymalny zadany poziom rekursji został osiągnięty, a błąd nadal przekracza dopuszczalną granicę `RECURS_LEVEL_MAX` (jest to stała zdefiniowana w programie), to wynikiem obliczeń będzie symbol nieoznaczony NaN.

UWAGA: Ten algorytm nie gwarantuje wyniku w granicach zadanego błędu - możliwe jest przekroczenie zadanego dopuszczalnego błędu (nie trudno podać przykład takiej "złośliwej" funkcji). Dlatego w praktyce obliczeniowej stosuje się dodatkowe zabezpieczenia, które tu - dla uproszczenia zadania - nie są proponowane.

Wartości startowe rekurencji (wartość `S` kwadratury obliczona na całym przedziale $[a, b]$ i początkowy poziom rekurencji) są ustalane we wstępnej procedurze `init_recurs()`.

Szablon programu należy uzupełnić o definicję funkcji inicjującej
`double init_recurs(Func1vFp f,double a,double b,double delta, QuadratureFp quad)`
oraz funkcji wywoływanej rekurencyjnie
`double recurs(Func1vFp f,double a,double b,double S,double delta, QuadratureFp quad,int level)`.

Test 2
Wczytanie danych i wywołanie funkcji `double init_recurs`. Całkowana funkcja: wybrana z tablicy wskaźników do funkcji `func_tab`.
Kwadratura: wybrana z tablicy kwadratur `quad_tab`.

- **Wejście**
  Nr testu
  indeks funkcji podcałkowej, indeks kwadratury
  granice całkowania, dopuszczalny błąd bezwzględny
- **Wyjście**
  Wartość kwadratury
- **Przykład:**
  Wejście:
  2
  1 4
  0 3 0.01
  Wyjście:
  29.04248

7.2 Całka podwójna po powierzchni (*surface integral*)

W tym podrozdziale będą obliczane wartości funkcji dwóch zmiennych. Należy zdefiniować nazwę
`typedef ...Func2vFp...;`
typu wskaźnikowego do funkcji z dwoma parametrami typu `double` zwracającej wartość typu `double`.

7.2.1 Całka po obszarze prostokątnym

Obszar całkowania funkcji $f(x, y)$ można zapisać w postaci

$$R = \{(x, y) : x_1 \leqslant x \leqslant x_2,\ y_1 \leqslant y \leqslant y_2\},$$

Szablon programu należy uzupełnić o definicję funkcji
`double dbl_integr(Func2vFp f, double x1, double x2, int nx, double y1, double y2, int ny)`,
która złożoną metodą prostokątów w przód (leftpoint) oblicza przybliżoną wartość całki. Parametry `nx` i `ny` są liczbami podprzedziałów kwadratur złożonych.

$$V = \int_{y_1}^{y_2} \left(\int_{x_1}^{x_2} f(x, y) dx \right) dy \tag{1}$$

Test 3
Wczytanie danych `x1, x2, nx, y1, y2, ny` i wywołanie funkcji.
`dbl_integr(f,x1,x2,nx,y1,y2,ny)`
Całkowana funkcja `f`: `func2v_2`.

- **Wejście**
  Nr testu
  `x1, x2, nx, y1, y2, ny`
- **Wyjście**
  Wartość kwadratury
- **Przykład:**
  Wejście:
  3
  0 1 100
  0 1 100
  Wyjście:
  1.42662

7.2.2 Całka po obszarze normalnym

Obszar postaci

$$D = \{(x, y) : x_1 \leqslant x \leqslant x_2, g(x) \leqslant y \leqslant h(x)\},$$

gdzie funkcje $g(x)$ i $h(x)$ są ciągłe na odcinku $[x_1, x_2]$, oraz $g(x) < h(x)$ we wnętrzu tego odcinka, nazywamy obszarem normalnym względem osi $0x$.
Użyteczny link:
`https://pl.wikipedia.org/wiki/CaĆka_podwjna`, część: Zamiana na całkę iterowaną.

Wtedy wzór (1) można zapisać w postaci

$$V_n = \int_{x_1}^{x_2} \left(\int_{g(x)}^{h(x)} f(x, y)dy \right) dx$$

Szablon programu należy uzupełnić o definicję funkcji
`double dbl_integr_normal_1(Func2vFp f,double x1,double x2,int nx,double hy,Func1vFp fg,Func1vFp fh)`.
Parametr `hy` jest przybliżoną długością podprzedziału kwadratury złożonej zastosowanej do całkowania wzdłuż zmiennej y. Służy do wyznaczenia liczby podprzdziałów n_y – najmniejszej liczby całkowitej, nie mniejszej od $\frac{h(x_i)-g(x_i)}{h_y}$.

Test 4
Wczytanie danych `x1, x2, nx, hy` i wywołanie funkcji
`dbl_integr_normal_1(f,x1,x2,nx,hy,fg,fh)`
Całkowana funkcja `f`: `func2v_2`.
Funkcje ograniczające obszar całkowania `fg` i `fh`: `lower_bound_2`,`upper_bound_2`.

- **Wejście**
  Nr testu
  `x1, x2, nx, y1, y2, ny`
- **Wyjście**
  Wartość kwadratury
- **Przykład:**
  Wejście:
  4
  0.7 0.9 200
  1e-3
  Wyjście:
  0.14480

7.2.3 Całka po (wielu) obszarach normalnych wewnątrz prostokąta

Rozważmy przypadek obszaru całkowania bardziej ogólnego niż obszar normalny, gdy warunek $g(x) \leqslant h(x)$ nie jest spełniony dla każdego $x \in [a, b]$. Rysunek 1 jest wykresem funkcji $f(x, y)$ całkowanej we wszystkich (dwóch) podobszarach prostokąta

$$R = \{(x, y) : a \leqslant x \leqslant b, \quad c \leqslant y \leqslant d\},$$

w których $g(x) < h(x)$ – obszary te są na rysunku oznaczone kreskowaniem kolorem czarnym. Każdy z nich jest obszarem normalnym względem osi $0x$. Użycie algorytmu z punktu 7.2.2 wymagałoby wyznaczania wszystkich pierwiastków równania $g(x) = h(x)$.

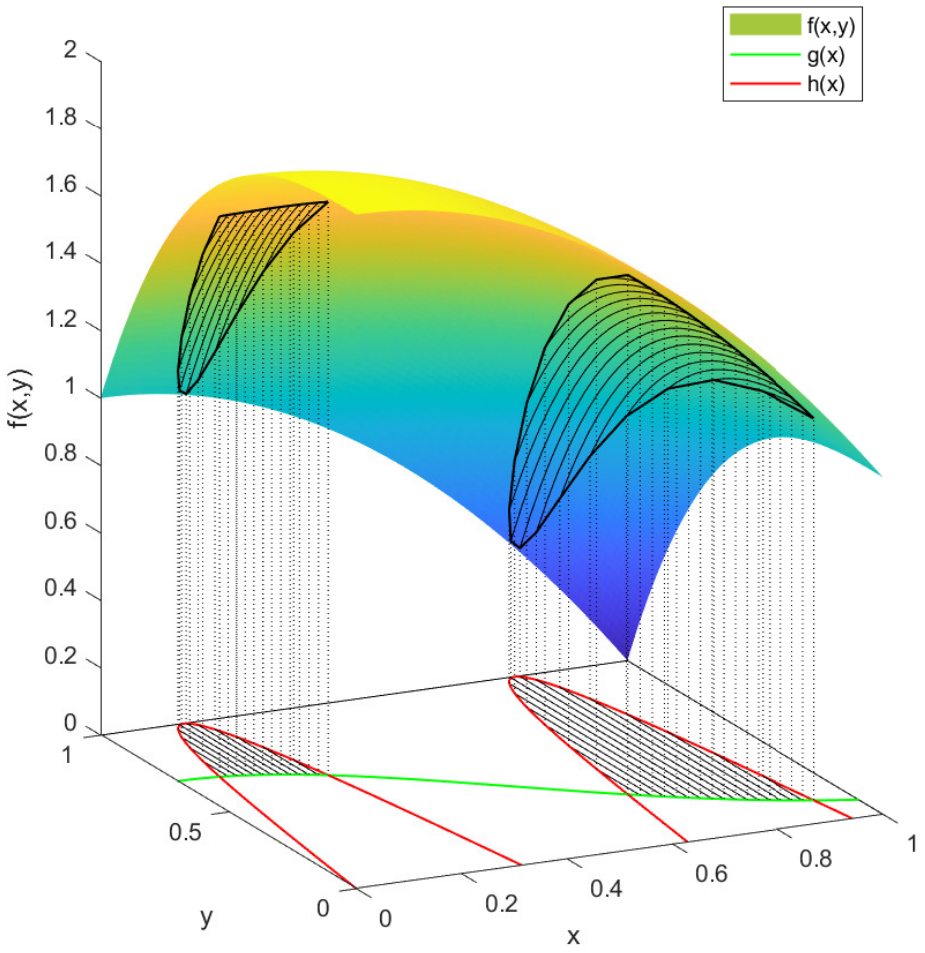
W tym zadaniu należy zastosować prostszy algorytm całkowania – całkowania po obszarze prostokątnym R z predykatem orzekającym, czy dla danej wartości x_i zachodzi nierówność $g(x_i) < y < h(x_i)$. Jeżeli tak, to należy obliczyć całkę (a dokładniej – kwadraturę)

$$Q_i(x_i) \approx \int_{\max(y_1, g(x_i))}^{\min(y_2, h(x_i))} f(x_i, y)dy$$

stosując jedną z kwadratur złożonych.

Szablon programu należy uzupełnić o definicję funkcji
`dbl_integr_normal_n(Func2vFp f, double x1, double x2, int nx, double y1, double y2, int ny, Func1vFp fg, Func1vFp fh))`,
która złożoną metodą prostokątów leftpoint oblicza przybliżoną objętość pod wykresem funkcji `f` nad obszarem normalnym ograniczonym wartościami `x1`, `x2`, i funkcjami $h(x)$ i $g(x)$ oraz prostokątem x_1, y_1, x_2, y_2.

Liczbę podprzedziałów kwadratur $Q_i(x_i)$ należy wyznaczyć podobnie jak w punkcie 7.2.2 przyjmując $h_y = (y_2 - y_1)/n_y$.



Rysunek 1: Przykład zadania całkowania po dwóch obszarach normalnych względem osi $0x$. $f(x,y) = 2 - x^2 - y^3$, $g(x) = 0.7\exp(-2x^2)$, $\quad h(x) = \sin(10x)$.

Test 5
Wczytanie danych `x1, x2, nx, y1, y2, ny` i wywołanie funkcji
`dbl_integr_normal_n(f,x1,x2,nx,y1,y2,ny,fg,fh)`
Całkowana funkcja `f`: `func2v_2`.
Funkcje ograniczające obszar całkowania `fg` i `fh`: `lower_bound_2`,`upper_bound_2`.

- **Wejście**
  Nr testu
  `x1, x2, nx, y1, y2, ny`
- **Wyjście**
  Wartość kwadratury
- **Przykład:**
  Wejście:
  5
  0 1 1000
  0 1 1000
  Wyjście:
  0.21668

7.3 Całki potrójne – z predykatem wyłączającym część obszaru całkowania

Uniwersalny typ funkcji (procedury) obliczającej wartość funkcji n zmiennych.

Procedura obliczająca wartości funkcji n zmiennych wymaga przekazania do niej n wartości zmiennych niezależnych. Aby uniknąć konieczności definiowania kolejnych nazw typów dla kolejnych liczb zmiennych, wartości zmiennych niezależnych będziemy przekazywać poprzez n-elementową tablicę.

Przykładowa funkcja podcałkowa trzech zmiennych jest zdefiniowana w szablonie programu `double func3v(const double v[], int n)`. Nazwa typu wskaźnika do takiej procedury (funkcji) jest w szablonie programu zdefiniowana: `typedef double (*FuncNvFp)(const double*, int);`

Predykat wykluczający dany punkt z obszaru całkowania. Zdefiniowana w szablonie funkcja `int bound3v(const double v[], int n)` jest predykatem zwracającym 1 gdy punkt o współrzędnych zapisanych w tablicy `v` leży wewnątrz obszaru całkowania.

Nazwa typu wskaźnika do procedury – predykatu – zwracającej wartość logiczną warunku określającego, czy zadany punkt w przestrzeni n-wymiarowej należy do zadanego obszaru całkowania, jest zdefiniowana w linii:
`typedef int (*BoundNvFp)(const double*, int)`

Całka potrójna po prostopadłościanie z predykatem boundary akceptującym albo wykluczającym elementarną domenę kwadratury

Szablon programu należy uzupełnić o definicję funkcji
`double trpl_quad_rect(FuncNvFp f, int variable_no, const double variable_lim[][2], const int tn[], BoundNvFp boundary)`,
która oblicza kwadraturę prostokątów wstecz (rightpoint) jako przybliżenie całki potrójnej po prostopadłościanie. Dolne i górne granice przedziałów całkowania wzdłuż kolejnych zmiennych są przekazywane w tablicy `variable_lim`, a liczby podprzedziałów – w tablicy `tn`. Parametr `boundary` jest adresem predykatu. Wartość `NULL` tego parametru oznacza brak predykatu, czyli brak ograniczeń w obszarze całkowania.

Test 6
Wczytanie danych `x1, x2, nx, y1, y2, ny` i wywołanie funkcji
`trpl_quad_rect()`.
Całkowana funkcja: `func3v`.
Predykat: Funkcje ograniczające obszar całkowania `bound3v`.

- **Wejście**
  Nr testu
  `x1, x2, nx`
  `y1, y2, ny`
  `z1, z2, nz`
  0 – gdy nie ma ograniczeń obszaru całkowania,
  1 – gdy istnieje predykat.
- **Wyjście**
  Wartość kwadratury
- **Przykład:**
  Wejście:
  6
  0 1 20
  0 1 20
  0 1 20
  1
  Wyjście:
  0.34824

7.4 Całka n-krotna z predykatem

W szablonie programu są zdefiniowane przykładowe funkcje n zmiennych:
procedura obliczająca wartość funkcji podcałkowej n-wymiarowej
`funcNv(const double v[], int n)`
oraz predykat `int boundNv(const double v[], int n)`.
Liczba zmiennych jest ograniczona zdefiniowaną stałą `N_MAX`.

Całka po hiperprostopadłościanie z predykatem boundary

Szablon programu należy uzupełnić o definicję rekurencyjnej funkcji
`recur_quad_rect_mid(double *psum, FuncNvFp f, int variable_no, double tvariable[], const double variable_lim[][2], const int tn[], int level, BoundNvFp boundary)`,
która oblicza przybliżenie całki z `variable_no` wymiarowej funkcji `f` po hiper-prostopadłościanie określonym ograniczeniami zapisanymi w `variable_lim`. Wzdłuż i-tej zmiennej jest obliczana złożona kwadratura prostokątów punktu środkowego (midpoint) z podziałem na `tn[i]` podprzedziałów.
Parametr `level` można zastąpić zmienną statyczną zdefiniowaną wewnątrz funkcji.
Wartość kwadratury jest zapisywana pod adresem przekazywanym do funkcji przez jej pierwszy parametr.

Test 7

Wczytanie danych `x1, x2, nx, y1, y2, ny` i wywołanie funkcji `recur_quad_rect()`.
Całkowana funkcja: `funcNv`.
Predykat: `boundNv`.

- **Wejście**
  Nr testu
  Krotność całki n
  n linii z przedziałem całkowania i liczbą podprzedziałów
  0 – gdy bez ograniczeń obszaru całkowania,
  1 – gdy istnieje predykat.

- **Wyjście**
  Wartość kwadratury

- **Przykład:**
  Wejście:
  7
  4
  0 1 10
  0 1 10
  0 1 10
  0 1 10
  1
  Wyjście:
  0.98941

8 Sortowanie

Założenie: Nie korzystamy z dynamicznego przydziału pamięci. To zagadnienie będzie omawiane na wykładzie w bliskiej przyszłości. Do tego czasu będziemy stosować „środek zastępczy" – definiowanie tablic o wystarczająco dużych rozmiarach.

8.1 Konstrukcja posortowanej tablicy elementów unikalnych z użyciem zmodyfikowanej funkcji `bsearch`

O zmodyfikowanej funkcji `bsearch`:
Oryginalna, biblioteczna funkcja `bsearch` zwraca adres znalezionego elementu tablicy albo – w przypadku braku szukanego elementu – `NULL`. Funkcja `bsearch` „dociera" do informacji, pod jakim adresem „powinien" być szukany element (ale go tam nie ma). Jednak informacja ta jest tracona. Modyfikacja polega na przekazaniu (do funkcji wywołującej) adresu, pod którym należałoby umieścić szukany element (zachowując uporządkowanie elementów tablicy).

Szablon programu należy uzupełnić o definicję funkcji
`void *bsearch2(const void *key, const void *base, size_t nitems, size_t size,`
`int (*compar)(const void *, const void *), char *result_adr)`. Możliwe są dwa rezultaty szukania:

1. Sukces – funkcja wpisuje pod adres `result_adr` wartość różną od zera oraz zwraca adres znalezionego elementu (jak oryginalna funkcja `bsearch`).
2. Element z kluczem `*key` nie został znaleziony – funkcja wpisuje zero pod adres `result_adr` oraz zwraca adres, pod który należy wpisać nowy element zachowując przyjęty porządek. Funkcja nie sprawdza, czy zwracany adres nie przekracza zakresu pamięci przydzielonej tablicy `base`.

Zadanie to jest przykładem zastosowania funkcji `bsearch2`. Obejmuje wczytanie danych o artykułach spożywczych (cena jednostkowa, ilość, termin ważności[1] `dd.mm.yyyy`, nazwa) i zapisywanie ich w określonym porządku w tablicy struktur typu `Food`. Deklaracja tej struktury, zawierającej dane jednej partii artykułu, jest zapisana w szablonie.

Jeżeli wczytany rekord zawiera dane o artykule, który jest już zapisany w tablicy oraz dane wczytane różnią się (lub nie) od zapisanych tylko wartością w polu `amount`, to nie należy tworzyć dla tego rekordu nowego elementu tablicy, lecz zwiększyć wartość w polu `amount` istniejącej już struktury.

Zapisywanie w tablicy struktur odczytanego ze strumienia wejściowego rekordu zawierającego dane o jednej partii artykułu ma zachowywać kolejność okeśloną relacją porządku. Należy określić taką relację porządkującą elementy tablicy, aby odszukanie elementu, którego wartość pola `amount` ma być powiększona, wymagało tylko jednokrotnego wywołania funkcji `bsearch2`.
Kolejność elementów wg zadanego porządku ma być zachowana także w trakcie wpisywania nowych danych do tablicy.

Szablon programu należy uzupełnić o:

1. Definicję funkcji `bsearch2(...)` wg opisu powyżej.
2. Definicję funkcji `Food *add_record(Food *tab, size_t tab_size, int *np, ComparFp compar, Food *new)`, która wywołuje funkcję `bsearch2(...)` sprawdzającą, czy nowy artykuł (jego dane są zapisane pos adresem `*new`) jest zapisany w tablicy `tab` (o `*np` elementach i rozmiarze `tab_size`). O tym, czy uznać `*new` za nowy decyduje funkcja wskazywana pointerem do funkcji `compar` (typu `ComparFP` – zdefiniowanego w szablonie).
   - Jeżeli `*new` nie jest elementem nowym, to dane zapisane w elemencie tablicy są modyfikowane danymi zapisanymi w `*new` – konkretnie – ilość artykułu znalezionego w tablicy jest powiększana o ilość zapisaną w `*new`. Funkcja zwraca adres modyfikowanego elementu tablicy.
   - Jeżeli `*new` jest elementem nowym, to:
     - Jeżeli rozmiar `tab_size` tablicy `tab` byłby przekroczony wstawieniem nowego elementu, to dodanie elementu nie jest realizowane, a funkcja zwraca `NULL`.
     - W przeciwnym przypadku funkcja `add_record` dodaje we wskazanym miejscu nowy element (z ewentualnym przesunięciem części elementów tablicy), zwiększa liczbę elementów tablicy `*np` i zwraca adres wpisanego elementu.
3. Definicję funkcji wskazywanej pointerem `compar`.
4. Definicję funkcji `int read_stream(Food *p, size_t size, int no, FILE *stream)`, która czyta `no` rekordów (linii) danych ze strumienia wejściowego. Dla każdego rekordu wywołuje funkcję `add_record`.

Test 8.1
Test wczytuje liczbę wprowadzanych linii danych, wywołuje funkcję `read_stream` (dla uproszczenia zadania – nie sygnalizuje próby przekroczenia pojemności tablicy struktur), wczytuje nazwę artykułu i wypisuje wszystkie dane zawarte w strukturach z wskazaną nazwą artykułu (w kolejności: po pierwsze – rosnącej ceny, w drugiej kolejności – "rosnącego" terminu).
W przypadku, gdy stała `TEST` jest równa 0, test wczytuje (po liczbie linii danych) nazwę pliku, z którego dane mają być odczytywane. W

[1] „Termin ważności" należy traktować jako skrótowe określenie np. terminu przydatności do spożycia

- **Wejście**
  1
  liczba linii n
  n linii:nazwa cena ilość dd mm yyyy
  nazwa artykułu

- **Wyjście**
  pamiętane w tablicy dane o artykule o wczytanej nazwie artykułu
  cena ilość, dd mm yyyy
  cena ilość, dd mm yyyy
  ...

- **Przykład W wersji:** `TEST=1`
  Wejście: 1
  6
  kefir 3.50 30 7 6 2023
  ser 7.80 25 15 6 2023
  kefir 3.75 20 7 6 2023
  ser 7.80 12 15 6 2023
  mleko 3.25 44 29 12 2023
  kefir 3.50 22 7 6 2023
  kefir
  Wyjście:
  3.50 52.00 7 6 2023
  3.75 20.00 7 6 2023

- **Przykład W wersji:** `TEST=0`
  Wejście: 1
  6
  foods0.txt
  kefir
  Wyjście:
  3.50 52.00 7 6 2023
  3.75 20.00 7 6 2023

8.2 Sortowanie elementów tablicy struktur

Zadanie polega na posortowaniu biblioteczną funkcją `qsort` tablicy struktur utworzonej w zadaniu 8.1. Relację porządkującą elementy tablicy należy zdefiniować tak, aby przy możliwie małym koszcie obliczeniowym (dla dużej liczby danych) obliczyć wartość towaru, którego termin ważności mija dokładnie po n dniach od założonej daty (termin ważności = a dni + zadana data). Zadana data symuluje tu datę bieżącą.

Sugestia wyboru algorytmu: Posortować (qsort) wg daty, odszukać (bsearch) jeden element - w jego bezpośrednim "sąsiedztwie" są pozostałe z szukaną datą.

Należy zwrócić uwagę na okresy przełomu miesięcy lub lat. Zadanie można sobie ułatwić korzystając z funkcji deklarowanych w pliku nagłówkowym `time.h` standardowej biblioteki.

Szablon programu należy uzupełnić o:

1. Definicję funkcji `int read_stream0(Food *tab, size_t size, int no, FILE *stream)`, która wczytuje ze strumienia `stream`do `tab` (tablicy struktur `no` struktur typu `Food`), `size` jest rozmiarem tablicy) `tab` rekordów.

2. Definicję funkcji
   `float value(Food *food_tab, size_t n, Date curr_date, int delta)`, która oblicza omawianą wyżej wartość artykułów.

Test 8.2
Test wczytuje dane tak, jak w zadaniu 8.1 (liczbę rekordów danych, rekordy danych, zadaną datę i liczbę dni do spradzanej daty ważności - w wersjach dla różnych wartości stałej `TEST`). Następnie wywołuje funkcję `float value`, która oblicza sumę wartości wszystkich artykułów, które tracą ważność za a dni (przykład: jeżeli 5.6.2023 będzie wczytaną datą , to będą poszukiwane artykuły o terminie ważności 10.6.2023).

- **Wejście**
  2
  liczba linii n
  n linii:nazwa cena ilość dd mm yyyy
  data (symulująca datę bieżącą) liczba a dni do szukanej daty ważności.

- **Wyjście**
  Suma wartości artykułów z wskazaną datą ważności

- **Przykład:**
  Wejście:
  2
  6
  kefir 3.50 30 7 6 2023
  ser 7.80 25 15 6 2023
  kefir 3.75 20 7 6 2023
  ser 7.80 12 15 6 2023
  mleko 3.25 44 29 12 2023
  kefir 3.50 22 7 6 2023
  2 6 2023
  5
  Wyjście:
  257.00

8.3 Linia sukcesji do brytjskiego tronu

Przydatne linki o zasadach sukcesji:
`https://www.royal.uk/encyclopedia/succession`
`https://pl.wikipedia.org/wiki/Linia_sukcesji_do_brytyjskiego_tronu`

Pełna lista sukcesji zawiera ponad 5700 osób. Dane o osobach (zapisane w szablonie programu) są ograniczone tylko do potomków Jerzego VI i samego Jerzego VI – przyjmijmy, że jest on pierwszym panującym , poniżej będzie nazywany *first*.

Założenia:

1. Przyjmujemy kolejność sukcesji obowiązującą po modyfikacji w roku 2013.
2. Dane o osobach (unikalne imię, płeć, data urodzenia, imię tego rodzica, po którym dziedziczona jest sukcesja) są pamiętane w tablicy struktur.
3. Kolejność elementów tablicy jest przypadkowa.
4. Imiona są unikalne.
5. *first* nie ma wpisanego imienia jego rodzica. W to miejsce jest wpisana wartość `NULL`.
6. Mimo ograniczenia liczby pretendentów, unikamy prostych algorytmów przeszukiwania całego zbioru danych (ze względu na dużą złożoność obliczeniową).
7. Z uwagi na (tymczasowy) brak możliwości dynamicznego przydziału pamięci, nie tworzymy drzewa genealogicznego.
8. Do zbioru pretendentów są dodane osoby, które już nie pretendują (bo panują, nie żyją albo abdykowali), ale ich dane są niezbędne do ustalenia kolejności sukcesji.

Jeżeli jest możliwość zdefiniowania funkcji, która dla dwóch dowolnych pretendentów wyznacza pierwszeństwo sukcesji na podstawie ich danych (zapisanych w nieuporządkowanym zbiorze), to należy taką funkcję zdefiniować i skorzystać z bibliotecznej funkcji `qsort`. I na tym można zakończyć to zadanie.

W przeciwnym przypadku należy zauważyć, że:

- Wielokrotnie powtarzaną operacją będzie szukanie w tablicy osoby spełniającej pewne warunki i przesunięcie jej danych do innego miejsca w tablicy.
- Czas szukania można skrócić wstępnie sortując elementy tablicy struktur z danymi o osobach.
- Aby zredukować liczbę przesunięć, należy dążyć do grupowania przesuwanych osób (aby przesuwać nie pojedynczy element tablicy, lecz większy, ciągły obszar pamięci).

Algorytm mógłby zawierać następujące etapy:

1. Sortowanie tablicy struktur z użyciem funkcji `qsort`.
   Funkcja porównująca dwa elementy tablicy powinna stosować kryterium takie, aby w posortowanej tablicy sąsiadowały ze sobą osoby, które prawdopodobnie będą blisko w kolejce sukcesji. Takie grupy to np. rodzeństwo (w kolejności wynikającej z zasady progenitury). W kryterium sortowania można też uwzględnić fakt, że pierwszym elementem tablicy ma być *first*.
2. Ustawianie rodzeństwa za ich rodzicem.
   Zaczynając od *first* należy znaleźć jego dzieci i przesunąć je bezpośrednio za *first*. Dalsza procedura jest chyba trywialna.
   Dla przyspieszenia odnajdywania dzieci jednego rodzica należy ten etap poprzedzić utworzeniem „tablicy indeksów”.

3. Tworzenie tablicy indeksów.
   Każdy element „tablicy indeksów" powinien zawierać wskaźnik do imienia rodzica i indeks elementu tablicy osób, w którym są zapisane dane jego dziecka pretendującego w pierwszej kolejności (oraz ewentualnie liczbę dzieci). Jeżeli tablica osób będzie posortowana także według np. alfabetycznej kolejności rodzica, to do wyszukiwania rodzica w tablicy indeksów można użyć funkcji `bsearch` – to kryterium nie jest sprzeczne z wymaganiami wymienionymi w punkcie 1.

4. Usuwanie z kolejki osób, które nie są pretendentami.
   Tu jest dopuszczalne jednokrotne przeglądnięcie danych wszystkich osób zapisanych w tablicy `person_tab`.

Preferowany jest algorytm *in situ* (który nie przepisuje osób z jednej tablicy do innej, lecz do tej samej, ale w innym miejscu). Dopuszczalne jest korzystanie z tymczasowej tablicy wykorzystywanej do chwilowego pamiętania przesuwanych elementów.

Inne propozycje „oszczędnego" algorytmu mile widziane – proszę o nich poinformować prowadzącego zajęcia.

Typ struktury `Person` przewidzianej dla danych o pretendentach oraz typ struktury `Parent` są zdefiniowane w szablonie programu.

Tablica osób `person_tab` jest zdefiniowana i inicjowana danymi w segmencie głównym programu.

Szablon programu należy uzupełnić o definicje funkcji:

- `int fill_indices_tab(Parent *idx_tab, Person *pers_tab, int size)`, która wypełnia „tablicę indeksów" typu `Parent` i zwraca liczbę wpisanych elementów tej tablicy;
- `void persons_shiftings(Person *person_tab, int size, Parent *idx_tab, int no_parents)`, która z wykorzystaniem funkcji `memcpy, memmove` przesuwa kolejne grupy elementów tablicy osób `person_tab`. Modyfikuje wartości indeksów w tablicy `idx_tab`;
- `int cleaning(Person *person_tab,int n)`, która usuwa z tablicy elementy osób niepretendujących (z wykorzystaniem jednej z funkcji `memcpy, memmove`). Funkcja zwraca liczbę pozostałych w tablicy pretendentów;
- `int create_list(Person *person_tab, int n)`, która wywołuje funkcję sortowania tablicy `qsort`, funkcje `fill_indices`, `persons_shiftings` oraz `cleaning`. . Zwraca liczbę pretendentów.

Należy także zdefiniować funkcje porównujące dla funkcji `qsort` i `bsearch`, zdefiniować tablicę indeksów (w takim miejscu, aby jej zasięg był możliwie mały.

Test 8.3

Test wywołuje funkcję `create_list`, wczytuje liczbę n i wypisuje imię pretendenta na n-tej pozycji w kolejce sukcesji.

- **Wejście**
  3 n
- **Wyjście**
  imię pretendenta
- **Przykład:**
  Wejście:
  3 24
  Wyjście
  Lucas

9 Implementacje macierzy / tablic 2D

9.1 Macierz zapisywana jako zlinearyzowana tablica dwuwymiarowa

9.1.1 Macierz prostokątna zapisywana w tablicy definiowanej jako 2D, ale przekazywana do funkcji jak tablica 1D

Szablon programu należy uzupełnić o definicję funkcji `prod_mat()`, która oblicza iloczyn macierzy korzystając z dwóch pomocniczych funkcji `get()` i `set()`, które obliczają adres każdego elementu macierzy na podstawie numerów wiersza i kolumny oraz liczby kolumn.

Informacje o adresach macierzy (dokładniej — początkowych ich elementów) jest przekazywana do każdej z tych funkcji jako parametr typu `int*`.

- **Wejście**

  1
  rows cols (liczba wierszy i kolumn 1. macierzy),
  elementy 1. macierzy,
  rows cols (liczba wierszy i kolumn 2. macierzy),
  elementy 2. macierzy.

- **Wyjście**

  elementy iloczynu macierzy.

- **Przykład**

  Wejście:

```
1
2 3
1 2 3
4 5 6
3 2
10 20
30 40
50 60
```

  Wyjście:

```
220  280
490  640
```

9.2 Tablica 2D o wierszach różnej długości – implementacja za pomocą tablicy wskaźników do początkowych elementów wierszy umieszczonych w ciągłym obszarze pamięci

Wiersze tablicy mają być wczytywane ze strumienia wejściowego – dla uproszczenia – z klawiatury. Założenia dotyczące danych w strumieniu wejściowym:

- Każdy ciąg liczbowy jest w linii (rekordzie) zakończonym znakiem nowej linii.
- W każdej linii są tylko liczby - liczba liczb jest dowolna, ale nie mniejsza niż 1.
- Liczby (w systemie dziesiętnym) są w postaci stałych całkowitych, oddzielone spacjami.

9.2.1 Tablica z wierszami typu numerycznego

Szablon programu należy uzupełnić o definicję funkcji

1. `read_int_lines_cont()`, która wczytuje linie (rekordy) zawierające ciągi liczb (ciągi o różnej liczebności) i zapisuje w postaci numerycznej (nie znakowej) do ciągłego obszaru pamięci. Adresy początkowego elementu każdego wiersza zapisuje w tablicy wskaźników przesyłanej pierwszym parametrem. Funkcja zwraca liczbę wczytanych linii.
2. `write_int_line_cont()`, która wypisuje wybrany wiersz tablicy.

- **Wejście**
  2
  numer wiersza, który ma być wyprowadzony (licząc od 1)
  liczby wiersza 1.
  liczby wiersza 2.
  ...
- **Wyjście**
  elementy wskazanego wiersza tablicy.
- **Przykład**
  Wejście:

  ```
  2
  2
  2 3 1 -5
  1 2 3
  0 -5
  ```

  Wyjście:

  ```
  1 2 3
  ```

9.3 Tablica 2D o wierszach różnej długości – implementacja za pomocą tablicy wskaźników do początkowych elementów wierszy umieszczonych w odrębnych obszarach pamięci

Funkcje korzystają z dynamicznego przydziału pamięci.

9.3.1 Tablica z wierszami typu znakowego

Linie zawierające ciągi znaków ASCII są wczytywane ze ze strumienia wejściowego i zapisywane do obszarów pamięci przydzielanych dynamicznie funkcją `malloc()`.

Szablon programu należy uzupełnić o definicję funkcji

1. `read_char_lines()`, która wczytuje linie (rekordy) zawierające ciągi znaków. Każdy ciąg jest uzupełniany znakiem końca łańcucha i jest zapisywany do przydzielonej pamięci. Adresy początkowego elementu każdego wiersza zapisuje w tablicy wskaźników przesyłanej pierwszym parametrem. Funkcja zwraca liczbę wczytanych linii.
2. `write_char_line()`, która wypisuje wybrany wiersz tablicy.

- **Wejście**

  3
  numer wiersza, który ma być wyprowadzony (licząc od 1)
  znaki wiersza 1.
  znaki wiersza 2.
  ...

- **Wyjście**

  elementy wskazanego wiersza tablicy.

- **Przykład**

  Wejście:

  ```
  3
  2
  To jest wiersz 1,
  a to drugi.
  Trzeciego nie ma.
  ```

  Wyjście:

  ```
  a to drugi.
  ```

9.4 Tablica z wierszami typu numerycznego

W szablonie programu znajduje się deklaracja struktury `line` opisującej pojedynczy wiersz tablicy: adres pierwszego elementu wiersza, liczbę elementów wiersza oraz średnią arytmetyczną tych elementów. Definicja tablicy tych struktur jest zdefiniowana w segmencie głównym.

Szablon programu należy uzupełnić o definicję funkcji

1. `read_int_lines()`, która wczytuje linie (rekordy) zawierające ciągi liczb (ciągi o różnej liczebności), każdy ciąg jest zapisywany w postaci numerycznej (nie znakowej) do obszaru pamięci przydzielanej funkcją `malloc()`. Adresy przydzielanej pamięci, liczbę elementów wiersza oraz średnią arytmetyczną jego elementów zapisuje w tablicy struktur przesyłanej pierwszym parametrem. Funkcja zwraca liczbę wczytanych linii.

2. `sort_by_average()`, która przy użyciu funkcji `qsort()` sortuje wiersze tablicy wg rosnącej średniej elementów.

3. `write_int_line()`, która wypisuje wybrany posortowanej wiersz tablicy.

- **Wejście**

  4
  numer wiersza, (w kolejności rosnącej średniej) który ma być wyprowadzony (licząc od 1)
  liczby wiersza 1
  liczby wiersza 2
  ...

- **Wyjście**

  elementy wskazanego wiersza tablicy
  średnia arytmetyczna elementów tego wiersza

- **Przykład**

  Wejście:

  ```
  4
  2
  1 2 3 4 5
  -1 2
  8
  12 3 1
  ```

  Wyjście:

  ```
  1 2 3 4 5
  3.00
  ```

9.5 Macierze rzadkie: format skompresowanych wierszy (*Compressed sparse row, CSR*)

Użyteczny link: `https://en.wikipedia.org/wiki/Sparse_matrix`

9.5.1 Format CSR

W formacie CSR macierz rzadka $M_{m\times n}$ jest reprezentowana w pamięci przy pomocy trzech wektorów: V, C (o długości N - liczba niezerowych elementów macierzy), oraz R (o długości $m+1$).

Wektory te zawierają odpowiednio:

1. Wartości niezerowych elementów macierzy ułożone wierszami (wektor V).
2. Indeksy kolumn (wektor C, C_i jest indeksem kolumny, w której znajduje się element V_i).
3. Zakresy wierszy (wektor R, R_j jest całkowitą liczbą niezerowych elementów powyżej wiersza j, oznacza to, że zawsze $R_0 = 0$ i $R_m = N$).

Przykład:
Macierz $M_{4\times 4}$ z pięcioma niezerowymi elementami

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 5 & 0 & 7 \end{bmatrix}$$

można zapisać używając 3 tablic:
$V = [\ 5\ 8\ 3\ 5\ 7\ ]$
$C = [\ 0\ 1\ 2\ 1\ 3\ ]$
$R = [\ 0\ 1\ 2\ 3\ 5\ ]$

Aby "rozpakować" i-ty wiersz, definiujemy $b_i = R_i$ i $e_i = R_{i+1}$ a następnie dla wszystkich elementów j, $b_i \leqslant j < e_i$ wstawiamy wartość V_j do kolumny C_j.

Aby rozpakować wiersz $i = 1$ (drugi wiersz), bierzemy $b_1 = R_1 = 1$ i $e_1 = R_2 = 2$. Czyli dla $j = 1$ mamy $V_1 = 8$ i $C_1 = 1$ co oznacza, że w wierszu o indeksie 1 mamy jeden niezerowy element (j przyjmuje tylko jedną wartość) o wartości 8 w kolumnie 1.

Format CSR jest bardzo wygodny do implementacji algorytmu mnożenia macierzy rzadkiej przez wektor x. Ponieważ w mnożeniu uwzględniamy tylko elementy o niezerowych wartościach, algorytm można zapisać w postaci:

for $i \in \{1, 2, \ldots, m-1\}$ **do**
  $y_i = 0$
  for $j \in \{R_i, \ldots, R_{i+1} - 1\}$ **do**
    $y_i = y_i + V_j \times x_{C_j}$
  end for
end for

Szablon programu należy uzupełnić o definicję funkcji:

1. `read_sparse()`, która wczytuje linie zawierające trójki liczb całkowitych. Pierwsze dwie liczby są odpowiednio numerem wiersza i kolumny elementu macierzy, a trzecia stanowi jego wartość. Funkcja zwraca liczbę wczytanych trójek.
2. `make_CSR()`, która na podstawie wczytanych trójek generuje trzy wektory odpowiadające formatowi CSR.
3. `multiply_by_vector()`, która mnoży macierz rzadką w formacie CSR przez zadany wektor x.
4. `write_vector()`, która wypisuje wektor liczb całkowitych.

- **Wejście**

  5
  m n – liczba wierszy i kolumn macierzy
  N – liczba elementów macierzy o niezerowych wartościach
  N trójek wiersz, kolumna, wartość
  i_0 j_0 v_0
  i_1 j_1 v_1
  ...
  n liczb całkowitych - wartości elementów wektora x

- **Wyjście**

  elementy wektora V
  elementy wektora C
  elementy wektora R
  elementy wektora $y = Mx$

- **Przykład**

  Wejście:

  ```
  5
  4 4
  5
  2 2 3
  3 1 5
  0 0 5
  3 3 7
  1 1 8
  1 2 3 4
  ```

  Wyjście:

  ```
  5 8 3 5 7
  0 1 2 1 3
  0 1 2 3 5
  5 16 9 38
  ```

10 Operacje na uogólnionych wektorach

Uwagi ogólne

Celem ćwiczenia jest zaimplementowanie struktury opisującej wektor elementów dowolnego typu oraz funkcji realizujących operacje na takim wektorze. Dobór i nazewnictwo funkcji jest inspirowane klasą `vector` z biblioteki STL C++. Z oczywistych względów realizacja jest zupełnie inna.

W kolejnych punktach na podstawie stworzonego szablonu będziemy tworzyć wektory liczb całkowitych, znaków, oraz struktur `Person`.

Struktura wektor i funkcje ogólne

Struktura `Vector` składa się z:

1. wskaźnika do początku (dynamicznie alokowanej) tablicy przechowującej dane wektora
2. rozmiaru elementu wektora (w bajtach)
3. aktualnej liczby elementów wektora
4. pojemności wektora (rozmiaru aktualnie zaalokowanej tablicy danych)

Szablon programu należy uzupełnić o definicję następujących funkcji:

1. `init_vector()` – alokuje tablicę danych na zadaną pojemność początkową i inicjalizuje pozostałe pola.
2. `reserve()` – realokuje tablicę danych tak, żeby miała co najmniej zadaną pojemność. Jeżeli zadana pojemność nie przekracza aktualnej funkcja nic nie robi.
3. `resize()` – zmienia aktualną liczbę elementów wektora: jeżeli nowy rozmiar jest mniejszy od aktualnego, nadmiarowe elementy są usuwane; jeżeli nowy rozmiar jest większy, wektor jest uzupełniany o odpowiednią liczbę wyzerowanych elementów.
4. `push_back()` – dodaj element na koniec wektora.
5. `insert()` – dodaj element na zadanej pozycji.
6. `clear()` – usuń wszystkie elementy z wektora.
7. `erase()` – usuń element wektora na zadanej pozycji.
8. `erase_value()` – usuń wszystkie elementy wektora o zadanej wartości.
9. `erase_if()` – usuń wszystkie elementy wektora spełniające predykat.
10. `shrink_to_fit()` – dopasuj rozmiar tablicy do aktualnej liczby elementów wektora
11. `print_vector()` – wypisz pojemność wektora i jego elementy

Pomocne uwagi:

1. Do zmiany wielkości tablicy (pojemności wektora) proszę używać funkcji `realloc()`
2. Przy dodawaniu elementów do wektora (funkcje `push_back(), insert()`) jeżeli konieczne jest zwiększenie jego pojemności, pojemność jest zwiększana dwukrotnie i tablica danych jest realokowana.
3. Do kopiowania fragmentów pamięci o zadanym rozmiarze proszę wykorzystać funkcję `memcpy()` (jeżeli obszary nie mają części wspólnej) lub `memmove()` (jeżeli się przecinają).

Ogólna postać danych (do każdego podpunktu):

numer zadania

`n` – liczba komend

`n` linii komend

Każda komenda składa się z litery (kodu komendy) i pozostałych danych (w zależności od typu polecenia).
Lista komend:

1. `p value - push_back(value)`
2. `i index value - insert(index, value)`
3. `e index - erase(index)`
4. `v value - erase_value(value)`
5. `d - erase_if()` – predykat definiowany jest dla konkretnego typu
6. `r new_size - resize(new_size)`
7. `c - clear()`
8. `f - shrink_to_fit()`
9. `s - qsort()`

Wyjściem z każdej sekcji jest pojemność i elementy wektora po wykonaniu wszystkich operacji.

10.1 Wektor liczb całkowitych

Dodatkowo szablon programu należy uzupełnić o funkcje:

1. `read_int()` – czytaj wartość całkowitą do adresu wskazywanego przez `value`
2. `print_int()` – wypisz wartość całkowitą
3. `int_cmp()` – komparator wartości całkowitych (sortowanie malejące)
4. `is_even()` – predykat (zwraca 1 jeżeli liczba jest parzysta).

Przykład:
Wejście:

```
1
14
p 10
p 20
p 5
p 3
p 15
i 0 30
i 4 40
i 7 50
e 4
v 10
e 4
v 20
r 6
f
```

Wyjście:

```
6
30 5 3 50 0 0
```

10.2 Wektor znaków

Dodatkowo szablon programu należy uzupełnić o funkcje:

1. `read_char()` – czytaj wartość typu `char` do adresu wskazywanego przez `value`
2. `print_char()` – wypisz wartość typu `char`
3. `char_cmp()` – komparator wartości znakowych (porządek leksykograficzny)
4. `is_vowel()` – predykat (zwraca 1 jeżeli znak jest samogłoską)

Przykład:
Wejście:

```
2
10
p a
p X
p k
p i
p y
p R
i 1 t
i 3 G
i 5 E
d
```

Wyjście:

```
16
t X G k R
```

10.3 Wektor struktur `Person`

Dodatkowo szablon programu należy uzupełnić o funkcje:

1. `read_person()` – czytaj elementy struktury `Person` do adresu wskazywanego przez `value`
2. `print_person()` – wypisz strukturę `Person`
3. `person_cmp()` – komparator struktur `Person` (malejąco wg wieku, następnie imienia i nazwiska – rosnąco)
4. `is_older_than_25()` – predykat (zwraca 1 jeżeli osoba ma więcej niż 25 lat)

Przykład:
Wejście:

```
3
8
p 23 Dominik Adamczyk
p 27 Natalia Adamiak
p 24 Marcin Chudy
i 1 29 Anna Cichocka
i 4 22 Natalia Deyna
i 0 24 Marcin Bereta
d
s
```

Wyjście:

```
8
24 Marcin Bereta
24 Marcin Chudy
23 Dominik Adamczyk
22 Natalia Deyna
```

11 Implementacja list jednokierunkowych

- **Założenia:**
  - W każdym zadaniu tworzymy listy jednokierunkowe, ogólnego przeznaczenia (patrz – wykład).
  - Nagłówek każdej listy jest typu `List` – zapisany w szablonie programu. W stosunku do przykładu podanego na wykładzie został on rozszerzony o pole wskaźnika na funkcję modyfikującą dane elementu listy
- **Plan ćwiczenia** (wg narastającego stopnia trudności):
  - W elementach listy w zadaniu 11.1 jest używana najprostsza struktura danych – jedna liczba całkowita.
  - Zadanie 11.2 – porównanie czasu wykonywania 3 różnych algorytmów – ma charakter poglądowy i nie wymaga pisania definicji dodatkowych funkcji. To zadanie jest ważne, choć nie będzie oceniane.
  - W zadaniu 11.3 dane, które mają być zapisywane w liście są wyrazami wczytywanymi ze strumienia wejściowego. Dodatkowo, w zadaniu 11.4, dane w każdym elemencie listy są uzupełnione krotnością pamiętanego w nim wyrazu. Dla zmniejszenia nakładu pracy można zastosować strukturę danych obejmującą krotność wyrazów już w zadaniu 11.3.
- W tym tygodniu, z założenia i celowo, nie jest jeszcze przewidziane sortowanie elementów listy. Dlatego w tych zadaniach, w których jest wymagane uporządkowanie, należy w trakcie wykonywania programu wprowadzać dane tak, aby porządek był zachowany.

11.1 Funkcje podstawowe

Szablon programu należy uzupełnić o definicje funkcji (dugi parametr przekazuje do funkcji adres pamięci przydzielonej dla "nowej" danej `a`):

1. `void pushFront(List *list, void *a)` – dodaj element na początek listy;
2. `void pushBack(List *list, void *a)` – dodaj element na koniec listy;
3. `void popFront(List *list)` – usuń pierwszy element listy;
4. `void reverse(List *list)` – odwróć kolejność wszystkich elementów listy w jednorazowym przebiegu pętli po wszystkich elementach (jedyną instrukcją sterującą w tej funkcji jest jedna instrukcja `while()`);
5. `void insertInOrder(List *list, void *a)` – dodaj element do listy (z założenia - uporządkowanej). Jeżeli "nowa" dana jest już zapisana w jednym z elementów listy, to dane tego elementu są modyfikowane funkcją wskazaną w polu `modifyData` nagłówka listy, a pamięć przydzielone danej `a` jest zwalniana. Jeżeli w liście takiego elementu nie ma, to nowy element (z `a`) jest tworzony i dopisany do listy z zachowaniem uporządkowania listy. Przydatną – po niewielkich modyfikacjach – może się tu okazać funkcja `findInsertionPoint()` (poznana na wykładzie 08[1]).

[1] `https://home.agh.edu.pl/~pszwed/wiki/lib/exe/fetch.php?media=08-imperatywne-jezyk-c-dynamiczna-alokacja-pamieci.pdf`

Zadanie 11.1. Podstawowe operacje na elementach listy z danymi liczbowymi

W polu `data` elementu listy jest zapisywany adres danej typu `DataInt` (typ jest zdefiniowany w szablonie programu).

Szablon programu należy uzupełnić o definicje funkcji:

1. `void *create_data_int(int v)` – przydziela pamięć dla danej i zapisuje w niej wartość v, zwraca adres przydzielonej pamięci;
2. `void dump_int(const void *d)`,
3. `void free_int(const void *d)`,
4. `void cmp_int(const void *a, const void *b)`,

Ogólna postać danych (do każdego podpunktu):
numer zadania
`n` – liczba poleceń
`n` linii poleceń

Każde polecenie składa się z litery (kodu polecenia) i pozostałych danych (w zależności od typu polecenia).

Lista poleceń:

1. `f value - pushFront()`
2. `b value - pushBack()`
3. `d - popFront()`
4. `r - reverse()`
5. `i value - insertInOrder()`

Oceniane automatycznie będą dwa zestawy poleceń – bez i z listą uporządkowaną, czyli bez i z poleceniem „i" (funkcją `insertInOrder()`) .

- **Wejście**
  1 liczba poleceń
  kolejne polecenia
- **Wyjście**
  liczby zapisane w kolejnych elementach listy
- **Przykład 1:**

  Wejście:
  1
  4
  b 10
  f 5
  r
  b 3
  Wyjście:
  10 5 3

- **Przykład 2** (z zachowaniem porządku rosnącego):

  Wejście:
  1
  6
  f 5
  b 10
  i 7
  d
  i 13
  i 1
  Wyjście:
  1 7 10 13

Zadanie 11.2. Porównanie efektywności dodawania elementu na końcu listy bez stosowania i z zastosowaniem wskaźnika na ostatni element listy

Należy wykorzystać funkcje napisane w ramach zadania poprzedniego oraz uzupełnić szablon programu o definicję funkcji `pushBack_v0(list,value)`, która – podobnie jak funkcja `pushBack(list,value)` – dopisuje element z liczbą `value` na końcu listy. Różnica polega na tym, że funkcja `pushBack_v0(list,value)` nie korzysta z pola `tail` nagłówka listy, w którym jest pamiętany adres ostatniego elementu listy.

Zadanie polega na utworzeniu listy o zadanej liczbie elementów i wyznaczeniu czasu wykonywania 3 algorytmów, które prowadzą do tego samego rezultatu końcowego – utworzenia listy o tej samej kolejności elementów:

1. z funkcją `pushBack_v0(list,value)` dodającą elementy na końcu listy bez użycia wskaźnika na ostatni element – szablon programu należy uzupełnić o definicję tej funkcji;
2. z funkcją `pushFront(list,value)` tworzącą wszystkie elementy listy oraz funkcją `reverse(list)` odwracającą kolejność elementów listy;
3. z funkcją `pushBack(list,value)` dodającą elementy na końcu listy z wykorzystaniem wskaźnika na ostatni element.

Wyznaczone czasy są mierzone zegarem czasu rzeczywistego, a nie czasem wykonywania przez procesor wskazanego algorytmu. Dlatego pomiar czasu jest obarczony dużym błędem pochodzącym od wliczenia do niego także czasu wykonywania przez procesor innych zadań. Zatem – dla uśrednienia wyników – zadanie należy powtórzyć klika razy dla każdej liczby elementów (np. dla 30 i 30000).

- Wejście:
  2
  liczba elementów listy
- Wyjście:
  Czasy wykonania trzech algorytmów
- Przykład:
  Wejście:

2 3000
Wyjście:
n = 3000. Back bez tail. Czas = 0.018505 s.
n = 3000. Front + revers. Czas = 7.6e-05 s.
n = 3000. Back z tail. Czas = 6.6e-05 s.

Zadanie 11.3. Lista wyrazów wczytanych ze strumienia wejściowego (bez znaków diakrytycznych)

Założenia:

1. Przyjmijmy określenie „wyraz": łańcuch znaków ASCII nie zawierający ograniczników (delimiterów): znaków białych oraz `.,?!:;-`.
2. Kolejność elementów listy ma być zgodna z kolejnością odczytywanych wyrazów.
3. W polu `data` elementu listy jest zapisywany adres danej typu `DataWord` (typ jest zdefiniowany w szablonie programu). Struktura elementu listy zawiera pole (typu wskaźnikowego) dla adresu, pod którym jest pamiętany wyraz.

Szablon programu należy uzupełnić o definicje funkcji:

1. `void *create_data_word(char *string, int counter)` – przydziela pamięć dla łańcucha `string` i struktury typu `DataWord`, do przydzielonej pamięci wpisuje odpowiednie dane, zwraca adres struktury.
2. `void dump_word (const void *d)`
3. `void free_word(void *d)`

- Wejście:
  3
  linie tekstu
  Ctrl-D
- Wyjście:
  odczytane wyrazy (oddzielone spacją, bez innych delimiterów) w kolejności wczytywania[2]
- Przykład:
  Wejście:
  3
  Abc,d; zz EF-gh.
  xxx!
  Ctrl-D
  Wyjście:
  Abc d zz EF gh xxx

[2]Nie należy zwracać uwagi na aspekt użyteczności takiego programu (ten sam wynik można uzyskać bez tworzenia listy) – jest to wstępny etap dla kolejnych zadań.

Zadanie 11.4. Lista wyrazów z tekstu j.w. – dodawanie elementów wg porządku alfabetycznego ze zliczaniem krotności

Zadanie jest analogiczne do poprzedniego. Różnice:

- Struktura danych jest rozszerzona o pole licznika krotności występowania danego wyrazu w tekście.
- Elementy są dodawane do listy tak, aby zachować alfabetyczny porządek wyrazów (wielkość liter – mała czy wielka – nie ma tu znaczenia). Jeżeli dodawany wyraz jest już zapisany w liście, to należy zwiększyć jego licznik krotności.
- Program kończy się wypisaniem w kolejności alfabetycznej wyrazów o zadanej (na wejściu) krotności. Przyjmijmy, że każdy wyraz jest wypisywany małymi literami.

Szablon programu należy uzupełnić o definicje funkcji:

1. `int cmp_word_alphabet(const void *a, const void *b)`
2. `int cmp_word_counter(const void *a, const void *b)`
3. `void modify_word(void *a)` – modyfikacja danej sprowadza się do inkrementacji licznika krotności `counter`
4. `void dumpList_word_if(List *plist, int n)` – Wypisuje dane elementów spełniających warunek równości sprawdzany funkcją wskazywaną w polu compareData nagłówka listy.

- Wejście:
  4
  wybrana krotność `k` wyrazu linie tekstu
- Wyjście:
  wyrazy powtarzające się w tekście `k` razy (małymi literami, w porządku alfabetycznym)
- Przykład:
  Wejście:
  4 2
  Xy, ABC, ab, abc,
  xY, ab - ab.
  Wyjście:
  abc xy

12 Operacje na liście dwukierunkowej tablic z iteratorem

Uwagi ogólne

Celem ćwiczenia jest zaimplementowanie struktury opisującej listę dwukierunkową elementów będących tablicami (alokowanymi dynamicznie) oraz funkcji realizujących operacje na tej liście. Do niektórych operacji będzie wykorzystywana struktura iteratora.

12.1 Dwukierunkowa lista tablic (z iteratorem): zadania do wykonania

Dane wejściowe do programu: część wspólna

W każdym z kolejnych podpunktów dane zawierają:
`to_do` – numer zadania do wykonania
`n` – liczba węzłów
`n` ciągów liczb całkowitych postaci:
`m` – liczba elementów w tablicy węzła,
`m` elementów tablicy rozdzielonych spacją
. . . ewentualne dodatkowe dane

12.1.1 Wstawianie elementów do dwukierunkowej listy tablic

Szablon programu należy uzupełnić o definicję funkcji `push_back()`. Pozwala ona na budowę dwukierunkowej listy tablic poprzez wstawianie na koniec listy kolejnych węzłów zawierających tablice.

- **Wejście**

  1
  wymiary i wartości elementów tablic jak opisano w części ogólnej

- **Wyjście**

  Lista tablic, gdzie `->` oznacza węzeł, a elementy tablicy rozdzielone są spacjami.

- **Przykład:**

  Wejście:

  ```
  1
  3
  3 6 7 9
  2 4 8
  5 9 7 3 5 2
  ```

  Wyjście:

  ```
  -> 6 7 9
  -> 4 8
  -> 9 7 3 5 2
  ```

12.1.2 Iterowanie po strukturze dwukierunkowej listy tablic do przodu

Szablon programu należy uzupełnić o definicje funkcji `skip_forward()` i `get_forward()`, pozwalające poprzez iterowanie po liście tablic do przodu, na wypisanie zawartości wybranych komórek na standardowe wyjście.

- **Wejście**

  2
  wymiary i wartości elementów tablic jak opisano w części ogólnej
  liczba komórek do przejrzenia i numery wybranych komórek

- **Wyjście**

  Wartości zawarte w wybranych komórkach listy tablic (komórki liczone od początku listy, pierwszy element pierwszego węzła ma numer 1).

- **Przykład:**

  Wejście:

```
2
3
3 6 7 9
2 4 8
5 9 7 3 5 2
3 5 4 1
```

  Wyjście:

```
8 4 6
```

12.1.3 Iterowanie po strukturze dwukierunkowej listy tablic do tyłu

Szablon programu należy uzupełnić o definicje funkcji `skip_backward()` i `get_backward()`, pozwalające poprzez iterowanie po liście tablic do tyłu, na wypisanie zawartości wybranych komórek na standardowe wyjście.

- **Wejście**

  3
  wymiary i wartości elementów tablic jak opisano w części ogólnej
  liczba komórek do przejrzenia i numery wybranych komórek

- **Wyjście**

  Wartości zawarte w wybranych komórkach listy tablic (komórki liczone od końca listy, ostatni element ostatniego węzła ma numer 1).

- **Przykład:**

  Wejście:

```
3
3
3 6 7 9
2 4 8
5 9 7 3 5 2
3 5 4 10
```

Wyjście:

```
9 7 6
```

12.1.4 Usuwanie wybranych komórek z listy tablic

Szablon programu należy uzupełnić o definicję funkcji `remove_at()`. Funkcja ta usuwa wskazane komórki dwukierunkowej listy tablic. Jeśli usuwana komórka jest jedyną komórką tablicy w węźle (tablica jednoelementowa) to usuwany jest cały węzeł.

Uwaga: komórka o danym numerze jest wyznaczana w każdym kroku licząc od początku listy, tzn. jeżeli w pierwszym kroku usuniemy komórkę o numerze 1 to w kolejnym komórka następna będzie miała numer 1 (a nie 2 jak przed wykonaniem poprzedniego kroku).

- **Wejście**

  4
  wymiary i wartości elementów tablic jak opisano w części ogólnej
  liczba komórek do usunięcia i numery wybranych komórek

- **Wyjście**

  Lista tablic po usunięciu wskazanych komórek (komórki liczone od przodu) w formacie jak w punkcie 1.

- **Przykład:**

  Wejście:

  ```
  4
  3
  3 6 7 9
  2 4 8
  5 9 7 3 5 2
  3 5 4 1
  ```

  Wyjście:

  ```
  -> 7 9
  -> 9 7 3 5 2
  ```

12.2 Budowa listy na podstawie liczby cyfr elementów tablic

Niech `d(n)` oznacza liczbę cyfr liczby `n`.

Zadanie polega na zbudowaniu listy, której każdy węzeł będzie zawierał posortowaną rosnąco tablicę liczb o tej samej długości (liczbie cyfr).

Szablon programu należy uzupełnić o definicję funkcji `insert_in_order()`. Funkcja ta czyta kolejne liczby całkowite a następnie:

1. Jeżeli w liście znajduje się już węzeł zawierający liczby o długości równej `d(n)` to funkcja dodaje liczbę `n` do tej tablicy zachowując jej rosnące uporządkowanie.

2. W przeciwnym przypadku funkcja dodaje do listy nowy węzeł (tak, by lista była uporządkowana rosnąco względem długości liczb w kolejnych węzłach).

Po dodaniu do listy wszystkich wczytanych liczb, program wypisuje listę.

- **Wejście**

  5
  `n` – liczba liczb do umieszczenia w węzłach listy
  `n` liczb całkowitych (wartości komórek listy)

- **Wyjście**

  Lista tablic po wpisaniu wszystkich elementów z wejścia w formacie jak w punkcie 1. Węzły listy powinny być posortowane rosnąco względem długości (liczby cyfr) liczb, które przechowują. Tablice w węzłach powinny być posortowane rosnąco.

- **Przykład:**

  Wejście:

  ```
  5
  9
  999 14 733 29 22222 334 0 -12 -856
  ```

  Wyjście:

  ```
  -> 0
  -> -12 14 29
  -> -856 334 733 999
  -> 22222
  ```

13 Hash table with separate chaining

Uwagi ogólne

Celem ćwiczenia jest zaimplementowanie struktury opisującej tablicę z haszowaniem elementów dowolnego typu. Do rozwiązywania problemu kolizji zastosujemy metodę łańcuchową. Należy zaimplementować opisane niżej funkcje typowe dla tej struktury danych.

Ponieważ alokowanie pojedynczych elementów typów prymitywnych nie jest rozwiązaniem optymalnym, jako element danych zastosowano unię, składającą się z pól odpowiadających wykorzystywanym typom prymitywnym oraz wskaźnika używanego w przypadku elementu będącego strukturą. W ten sposób typy prymitywne są zawarte bezpośrednio w strukturze elementu a alokacja zachodzi tylko dla typów strukturalnych.

W kolejnych punktach na podstawie stworzonego szablonu będziemy tworzyć tablice liczb całkowitych, znaków, oraz struktur `DataWord` (jak w templatce programu), składających się ze słowa i licznika.

Struktura tablicy i funkcje ogólne

Struktura `hash_table` składa się z:

1. wskaźnika do początku (dynamicznie alokowanej) tablicy przechowującej głowy list (sentinels) elementów o jednakowej wartości funkcji haszującej
2. rozmiaru tablicy mieszającej
3. aktualnej liczby elementów zapisanych w tablicy
4. wskaźników do funkcji odpowiadających za operacje związane z konkretnym typem elementów (wypisywanie elementu, zwalnianie pamięci elementu, wczytanie / utworzenie danych określonego typu, komparator, funkcja haszująca, funkcja modyfikująca dane w przypadku próby wpisania do tablicy istniejącego klucza)

Szablon programu należy uzupełnić o definicję następujących funkcji:

1. `init_ht()` – alokuje tablicę na zadaną początkową długość i inicjalizuje pozostałe pola.
2. `dump_list()` – wypisuje listę elementów związaną z daną wartością funkcji haszującej.
3. `free_element()` – zwalnia pamięć danych i elementu tablicy.
4. `free_table()` – zwalnia pamięć wszystkich elementów tablicy (a także samą tablicę).
5. `insert_element()` – dodaj element do tablicy.
6. `rehash()` – zwiększ dwukrotnie rozmiar tablicy i rozmieść elementy zgodnie z nowymi wartościami funkcji haszującej. **Uwaga**: funkcja pobiera kolejne elementy ze "starej" tablicy i przepina je do "nowej" dodając je na początek właściwej listy

7. `remove_element()` – usuń element o zadanym kluczu

8. `get_element()` – zwróć adres elementu o zadanym kluczu

Uwaga: Jeżeli po dodaniu elementu do tablicy (funkcja `insert_element()`) współczynnik wypełnienia tablicy (stosunek liczby elementów tablicy do jej długości) przekroczy zadaną wartość `MAX_RATE` długość tablicy jest zwiększana dwukrotnie i wołana jest funkcja `rehash()`.

Ogólna postać danych (do podpunktu 1 i 2):
numer zadania
`n` – liczba komend
`index` – indeks tablicy, którego elementy należy wydrukować
`n` linii komend

Każda komenda składa się z litery (kodu komendy) i pozostałych danych (w zależności od typu polecenia).

W przypadku próby dodania istniejącego klucza element nie jest dodawany.

Lista komend:

1. `i value - insert_element(value)`

2. `r value - remove_element(value)`

Wyjściem z każdej sekcji jest długość tablicy i elementy należące do listy o numerze `index`.

13.1 Tablica liczb całkowitych

Dodatkowo szablon programu należy uzupełnić o funkcje:

1. `hash_int()` – oblicz wartość funkcji haszującej dla parametru typu`int`

2. `dump_int()` – wypisz element typu `int`

3. `cmp_int()` – komparator wartości całkowitych

4. `create_int()` – wczytaj wartość całkowitą. Funkcja zwraca unię `data_union` zawierającą wczytaną wartość. Dodatkowo, jeżeli parametr nie jest wskaźnikiem `NULL`, to powinien zawierać adres unii `data_union`, do której zostanie wpisana wczytana liczba.

Przykład:
Wejście:

```
1
10 3
i 12345
i 1334
i 1534
i 3234
i 5234
i 7234
i 8234
```

```
i 9234
r 5234
i 234
```

Wyjście:

```
4
9234 8234 7234
```

13.2 Tablica znaków

Dodatkowo szablon programu należy uzupełnić o funkcje:

1. `hash_char()` – oblicz wartość funkcji haszującej dla parametru typu`char`
2. `dump_char()` – wypisz element typu `char`
3. `cmp_char()` – komparator wartości znakowych
4. `create_char()` – wczytaj znak. Funkcja zwraca unię `data_union` zawierającą wczytaną wartość. Dodatkowo, jeżeli parametr nie jest wskaźnikiem `NULL`, to powinien zawierać adres unii `data_union`, do której zostanie wpisany wczytany znak.

Przykład:
Wejście:

```
2
12 2
i C
i i
i t
i M
r C
i u
i P
i l
i f
i W
r o
i s
```

Wyjście:

```
4
l M t
```

13.3 Tablica struktur `DataWord`

Dodatkowo szablon programu należy uzupełnić o funkcje:

1. `hash_word()` – oblicz wartość funkcji haszującej dla parametru typu `DataWord`
2. `dump_word()` – wypisz element typu `DataWord` (słowo i licznik)

3. `free_word()` – zwolnik pamięć słowa i elementu typu `DataWord`
4. `cmp_word()` – komparator wartości typu `const char*` (słów)
5. `modify_word()` – zwiększ wartość licznika zadanego elementu
6. `create_data_word()` – utwórz i zwróć unię `data_union` ze słowem zadanym parametrem funkcji i licznikiem równym 1
7. `stream_to_ht()` – czytaj linie tekstu, wyodrębnij słowa (zdefiniowane jak w zadaniu z listą jednokierunkową) i dodaj do tablicy po zamianie na małe litery; przy powtarzających się kluczach (słowach) zwiększ licznik słowa (funkcja `modify_data()`).

Wejście

3
`word` – słowo, które należy wypisać
linie tekstu

Wyjście

długość tablicy
słowo (podane na wejściu) i jego liczność

Przykład:

Wejście:

```
3
xyz
abcd xyz ab;XYz qwerty
```

Wyjście:

```
8
xyz 2
```