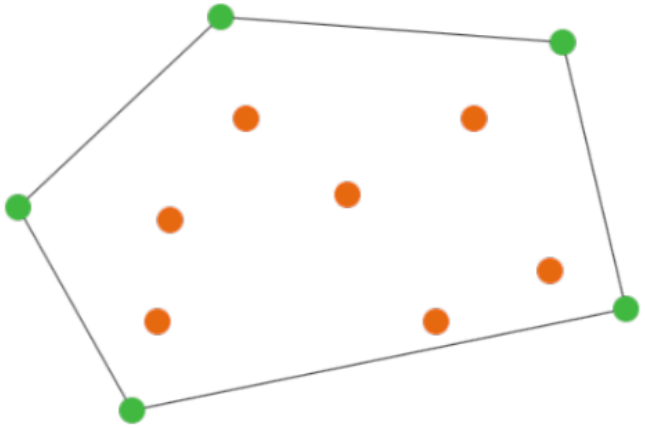
Rozpatrywany problem:

Zadanie 2

Napisz w języku C++ program, który spośród zbioru punktów na przestrzeni dwuwymiarowej znajdzie podzbiór takich punktów, które otaczają wszystkie inne punkty.

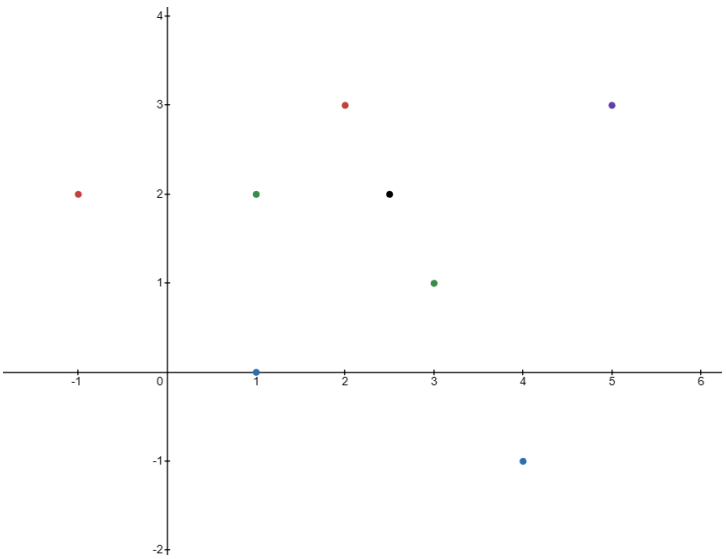


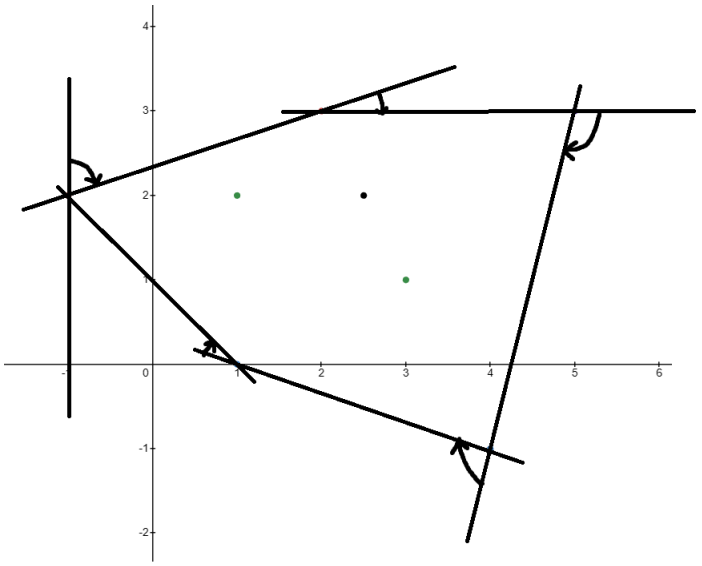
Rysunek 1: Przykład problemu

1.Idea algorytmu

W programie zaimplementowałem algorytm tzw. zawijania prezentu. Opiera się on na dwóch spostrzeżeniach:

- Łatwo zauważyć, że **skrajne** punkty zbioru **na pewno** będą należały do otoczki (nie da się nie wykorzystując punktu z minimalną/maksymalną współrzędną x/y poprowadzić otoczki która by go zawierała)
- Jeżeli chcemy znaleźć następny punkt otoczki wystarczy znaleźć punkt o **najmniejszym** kącie względem krawędzi tworzonej przez poprzednie dwa punkty (zakładamy że dla pierwszego punktu jest to prosta równoległa do osi OY). Wtedy wszystkie pozostałe punkty będą **po tej samej stronie** nowo powstałej krawędzi.





Operację tą można przeprowadzić zgodnie lub przeciwnie do ruchu wskazówek zegara i zaczynając w dowolnym skrajnym punkcie zbioru. To co robimy jest porównywalne do **owijania taśmy** wokół naszego zestawu punktów, stąd algorytm bierze swoją nazwę.

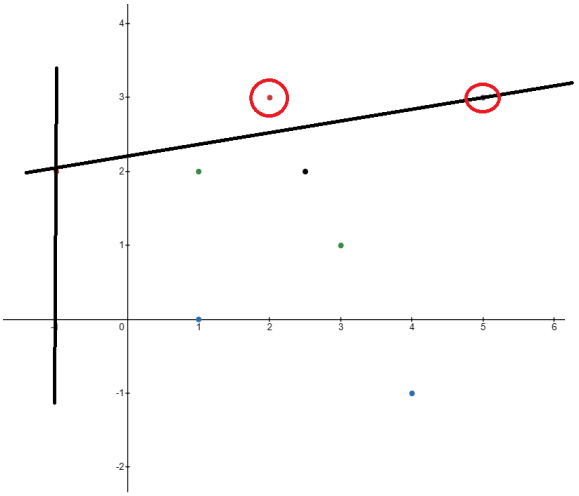
2.Implementacja

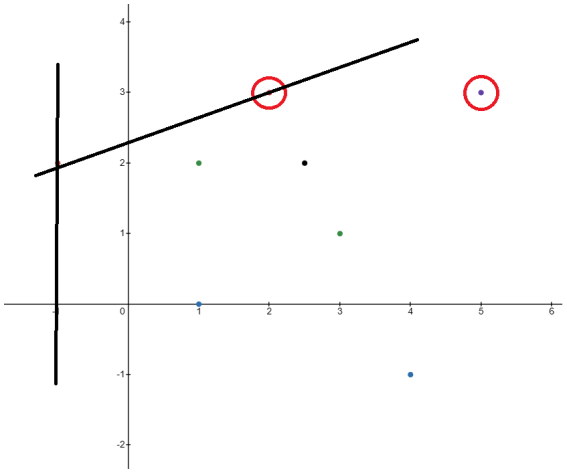
Mój program składa się z 4 funkcji:

- program - odpowiedzialna za wczytywanie danych i wypisanie wyniku
- find_hull - funkcja przyjmująca listę punktów i zwracająca listę punktów w otoczce (po angielsku zestaw punktów, którego szukamy to “convex hull”)
- find_next_point - znajduje następny punkt otoczki
- comp - komparator zwracający dla poprzedniego punktu i dwóch kandydatów który z kandydatów jest dla nas bardziej pożądany

Funkcja find_hull wykonuje schemat przedstawiony w 1. Zaczyna wyszukiwanie od punktu z **najmniejszą współrzędną x** po czym szuka kolejnych punktów do momentu zamknięcia się otoczki.

Funkcja find_next_point pamiętając poprzedni punkt (będziemy nazywać go **“a”**) na początku wybiera dowolny punkt (będziemy nazywać go **“b”**) a następnie dla każdego punktu (będziemy nazywać je **“c”**) sprawdza czy jest on położony po **lewej stronie** wektora ab. Jeśli tak jest to zastąpienie b punktem c **zmniejszy** kąt między poprzednią a następną krawędzią. Po sprawdzeniu każdego punktu otrzymamy ten, który daje najmniejszy kąt.

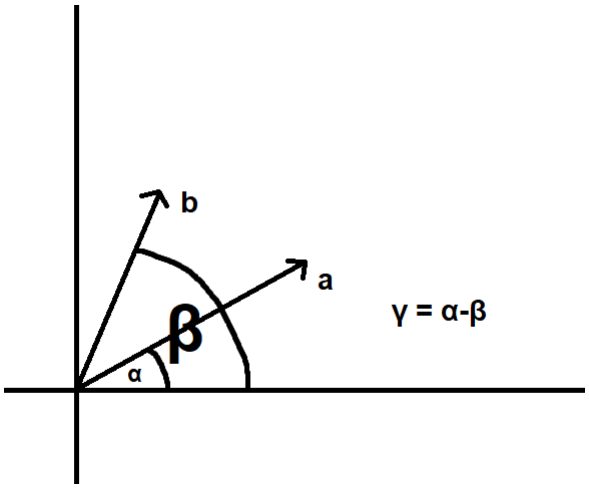




Funkcja comp w celu sprawdzenia czy dany punkt leży po lewej stronie danego wektora wykorzystuje jedną z własności **iloczynu wektorowego**:

$$|\vec{a}| \cdot |\vec{b}| \cdot \sin\gamma = a_y b_x - a_x b_y$$

Gdzie γ to różnica kąta między a i osią OX i kąta między b i osią OX.



Widać, że sinus kąta γ tak samo jak sam kąt będzie **dodatni** gdy wektor b jest **“pod”** wektorem a oraz **ujemny** gdy wektor b jest **“nad”** wektorem a.

Wiedząc to możemy łatwo sprawdzić czy punkt c jest po lewej stronie wektora ab konstruując wektor **ac** i sprawdzając znak sinusa kąta pomiędzy nimi.

Warto zauważyć, że do sprawdzenia znaku sinusa wystarczy nam iloczyn $a_y b_x$ - $a_x b_y$, ponieważ długości a i b zawsze będą **dodatnie**.

3.Ważniejsze przypadki brzegowe

- **Ilość punktów równa 1 lub 2**: w przypadku braku znalezienia kolejnego punktu otoczki funkcja find_next_point zwraca poprzedni punkt co jest sygnałem dla funkcji find_hull do zakończenia działania programu. Program zwróci jedyną możliwą w tym wypadku otoczkę, czyli taką składającą się z wszystkich punktów.
- **3 lub więcej punktów w jednej linii**: jeżeli iloczyn $a_y b_x$ - $a_x b_y$ będzie równy 0 musimy wybrać bliższy z punktów, który jednocześnie nie należy już do otoczki. Jest to zaimplementowane w funkcji comp przy użyciu warunku “bardziej od b opłaca się wziąć c jeżeli: (c jest bliżej a niż b i c nie jest jeszcze w otoczce) lub (b jest już w otoczce i c nie jest jeszcze w otoczce)”

4.Złożoność i ewentualne optymalizacje

Algorytm dla każdego punktu w otoczce sprawdza wszystkie pozostałe, więc złożoność wynosi **O(nh)** gdzie n - liczba punktów na płaszczyźnie, h - liczba punktów w otoczce. W pesymistycznym przypadku O(nh) będzie równe **$O(n^2)$**.

Jeżeli mamy zamiar operować na ilości punktów rzędu 10000 lub większej bardziej praktyczne byłoby napisanie przykładowo tzw algortytmu mergehull, który rekurencyjnie dzieli zbiór punktów na 2 mniejsze, wyznacza otoczkę dla zbiorów 3, 4 i 5 elementowych a następnie łączy mniejsze rozwiązania w całość. W taki sposób można zmniejszyć złożoność do poziomu **O(nlogn)**, jednak dla normalnych zestawów danych O(nh) w języku C++ będzie zadowalające.