Алгоритмы и структуры данных

22 мая 2015 г.

Часть I
Геометрические алгоритмы

1 §30. Выпуклая оболочка в $2D$

В данном параграфе будет рассмотрена задача нахождения выпуклой оболочки некоторого множества точек на плоскости.

Определение 1.1. Выпуклой оболочкой множества точек X в некотором аффинном пространстве $\mathcal{A}$ называется минимальное по включению множество точек $ConvexHull(X) \supseteq X$. Из курса линейной алгебры известно, что данное множество существует для любого ограниченного множества X.

Можно заметить, что если множество X является конечным, то множество $ConvexHull(X)$ является многоугольником. Нашей задачей является нахождение выпуклой оболочки конечного множества X, состоящего из конечного числа точек (обозначим через $n = |X|$), в частности, нахождение вершин множества $\partial(ConvexHull(X))$ и их последующее упорядочивание в порядке следования в полученном многоугольнике.

Напомним основные операции, применимые к векторам в пространстве $\mathbb{R}^3$:

Определение 1.2. Скалярным произведением векторов $\overrightarrow{u}$ и $\overrightarrow{v}$ в пространстве $\mathbb{R}^3$ называется число $(\overrightarrow{u}, \overrightarrow{v}) = u_x v_x + u_y v_y + u_z v_z$.

Определение 1.3. Векторным произведением векторов $\overrightarrow{u}$ и $\overrightarrow{v}$ в пространстве $\mathbb{R}^3$ называется вектор $[\overrightarrow{u}, \overrightarrow{v}] = det \begin{pmatrix} i & j & k \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{pmatrix}$ направление вектора (в геометрическом определении) определяется по направлению наименьшего угла поворота от вектора $\overrightarrow{u}$ к вектору $\overrightarrow{v}$.

Утверждение 1.1. $[\overrightarrow{u}, \overrightarrow{v}] \perp \overrightarrow{u}$, $[\overrightarrow{u}, \overrightarrow{v}] \perp \overrightarrow{v}$.

Утверждение 1.2. $[\overrightarrow{u}, \overrightarrow{v}] = |\overrightarrow{u}| \cdot |\overrightarrow{v}| \cdot |sin\alpha|$

Рассмотрим некоторые алгоритмы, решающие задачу нахождения выпуклой оболочки точек в двумерном пространстве.

Алгоритм 1. *Алгоритм Джарвиса (алгоритм "заворачивания подарка") -* ***Jarvis, 1973.***
Ход алгоритма:

1. Выберем самую нижнюю точку из исходного множества точек P, а среди самых нижних точек — самую левую (т.е. наименьшую по координате Ox). Обозначим полученную точку через c_0.
2. Выбираем точку с наименьшим углом поворота относительно предыдущего отрезка. Для сравнения точек по полярному углу воспользуемся знаком векторного произведения. При равенстве углов поворота возьмем вектор, имеющий большую длину. Обозначим полученную точку (на k-ой итерации шага (2)) через c_k. Повторяем шаг (2) до тех пор, пока не будет выполнено равенство $c_l = c_0$ для некоторого $l \in \mathbb{N}$.

Время работы данного алгоритма — $O(nh)$, где n— количество точек в исходном множестве X, а h — количество вершин в многоугольнике $\partial ConvexHull(X)$.

Как видно, сложность алгоритма Джарвиса может достигать оценки $O(n^2)$, поэтому приведем другой алгоритм построения выпуклой оболочки множества точек X.

2 §31. Выпуклая оболочка в $3D$

В данном параграфе будут рассмотрены алгоритмы построения выпуклой оболочки множества точек в $\mathbb{R}^3$. Чтобы найти выпуклую оболочку конечного количества точек, необходимо и достаточно построить выпуклый многогранник, содержащий данное множество точек.

Замечание 2.1. Будем считать, что никакие четыре точки не лежат в одной плоскости (иначе данную грань можно триангули-ровать любой треугольник).

Алгоритм 2. *Построение выпуклой оболочки методом полного перебора.*

Перебираем все точки множества $p_1,\ p_2,\ p_3$. Проверяем, является ли грань внешней: $\overrightarrow{n} = \overrightarrow{p_1p_2} \times \overrightarrow{p_1p_3}$ (проверим, что грань торчит наружу). Такой алгоритм имеет сложность $O(n^4)$.

Замечание 2.2. Заметим, что данный алгоритм обобщается на случай $\mathbb{R}^k$ и будет работать за $O(n^{k+1}k^3)$. Чтобы проверить ориентацию, надо будет проверить знак определителя $det \begin{pmatrix} \overrightarrow{pp_1} \\ \overrightarrow{pp_2} \\ \overrightarrow{pp_3} \\ \dots \\ pp_{k-1} \end{pmatrix}$.

Принадлежность точки плоскости можно проверить следующим образом: $det \begin{pmatrix} x-x_1 & y-y_1 & z-z_1 \\ x_2-x_1 & y_2-y_1 & z_2-z_1 \\ x_3-x_1 & y_3-y_1 & z_3-z1 \end{pmatrix} = 0$. Тогда уравнение плоскости имеет вид $Ax + By + Cz + D = 0$, где $\overrightarrow{n} = \begin{pmatrix} A \\ B \\ C \end{pmatrix}$ — вектор нормали. Таким образом, надо проверить знак скалярного произведения. Сложность алгоритма умножается на k^3 из-за вычисления определителя матрицы.

Алгоритм 3. *Алгоритм построения методом заворачивания подарка.*

Ход алгоритма:

1. Находим точку p_0 с минимальной координатой по оси z. Данный шаг выполняется за $O(n)$.
2. Находим точку p_1, для которой выполняется следующее условие: $\angle(\overrightarrow{p_0p_1}, \overrightarrow{Oz})$ максимален. Данный шаг выполняется за $O(n)$.
3. Нахождение первой грани: рассмотрим множество треугольников Δpp_0p_1, среди них необходимо выбрать такую точку p, что все остальные точки лежат в одной стороне относительно плоскости p_0p_1p. Выполняется аналогично алгоритму Джарвиса в двумерном случае (или вычисляется максимальный угол раскрытия или наибольший угол между осью Oz и $\overrightarrow{n} = \overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}$). Данный шаг выполняется за $O(n^2)$.
4. Поиск грани от некоторого "незакрытого" ребра $\overrightarrow{p_kp_{k+1}}$: ищем ту точку, для которой угол между нормалями граней минимальна. Но существует проблема: мы могли закрыть некоторые смежные грани: закрываем от 1 до 3 граней. Главное — записать информацию об открытости/закрытости ребер. В конце данной итерации надо открыть новые ребра. Далее будет доказано, что количество открытых ребер и количество граней равняется $O(n)$.

В итоге, из того, что количество итераций шага (4) равно $O(n)$, и алгоритм 31.2 работает за $O(n^2)$. Осталось доказать следующее утверждение:

Утверждение 2.1. Граф выпуклой оболочки является планарным. При этом количество ребер в выпуклой оболочке равняется $O(n)$.

Доказательство. Доказывается непосредственным применением формулы Эйлера для планарного графа. □

Алгоритм 4. *Алгоритм построения выпуклой оболочки методом "divide and conquer"*

Разобьем вертикальными плоскостями множества точек на группы из 5-7 точек (в лексикографическом порядке). Строим выпуклую оболочку точек каждой группы тривиальным алгоритмом. Заметим, что полученные выпуклые многогранники не пересекаются.

Покажем, как можно объединить две выпуклые оболочки за объём времени, равный $O(n)$. Обозначим через P, Q — исходные многранники. На идейном уровне: если сделать проекцию многогранников на ось Oxy, то получим два непересекающихся многоугольника. Построим для проекций объединение выпуклых оболочек за $O(n)$ времени. Запомним "крайнее

ребро" при объединении (обозначим данное ребро через p_1q_1). Воспользуемся аналогом алгоритма "заворачивания подарка". Заметим, что ребро $p_1q_1 \in \partial(ConvexHull(P \cup Q))$. Относительно данного ребра будем заворачивать выпуклую оболочку точек. Рассмотрим такую точку x, что все точки лежат по одну сторону от плоскости p_1q_1x (можно считать наименьший угол между нормалью к плоскости p_1q_1x и осью Oy). Единственной проблемой является выбор следующей грани. Далее необходимо выбирать множество точек таким образом, чтобы "параллельно" идти по множествам точек ∂P и ∂Q. Покажем, как решить данную проблему. Действительно, можно минимизировать угол между нормалями. В конце алгоритма осталось выкинуть точки $Inn(ConvexHull(P \cup Q))$.

В качестве структуры для хранения граней можно использовать граф. В данном графе можно выкидывать лишние ребра. При этом можно еще выкидывать соответствующие точки из проекций. При помощи DFS можно пометить обход, по которым обходит объединение.

В итоге, все итерации объединения будут работать за $O(n)$. Действительно, по каждое ребро мы рассматриваем не более 2 раза (а количество ребер, как было доказано в ***утверждении 1****, равняется $O(n)$). Таким образом, суммарное время работы алгоритма составляет $O(n(logn))$.*

Упражнение 2.1. Придумать улучшение данного алгоритма для многомерного случая.

3 §32. Алгоритмы вычислительной геометрии

Алгоритм 5. *Алгоритм поиска ближайших точек в $2D$.*

На плоскости надо найти пару точек с наименьшим расстоянием. Будем использовать метод "разделяй и властвуй". Для этого разделим множество точек на две части вертикальной прямой (обозначим их через A_1 и A_2). В каждой из них найдем минимальную пару точек на расстояниях h_1 и h_2 соответственно. Обозначим через $h = min(h_1, h_2)$. Тогда новое минимальное расстояние может быть в полосе $B = \{p_i : \ |p_i.x - x| \leq h\}$. Тогда будем искать пару в B за $O(n)$. Рассмотрим множество $C_i = \{p_j : \ p_j \in B, \ p_i.y - h \leq p_j.y < p_i.y\}$. Тогда утверждается, что $|C_i| \leq 7$. Если отсортировать все точки по координате y, то тогда можно выбирать все точки $p_i \in B$ (можно данную вещь прокрутить через очередь точек, находящихся в множестве B, и лежащих на расстоянии не более h).

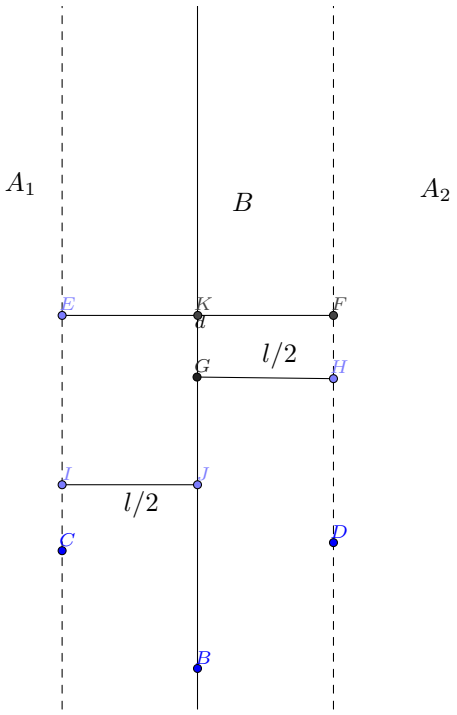
В итоге, алгоритм будет работать за время, равное $O(n \log n)$.

Алгоритм 6. *Алгоритм поиска треугольника с наименьшим периметром выполняется аналогичным образом.*

Отсортируем все точки по x-координате.

1. Разделяем множества точек на множества A_1 и A_2, в каждом их которых рекурсивно вызывается алгоритм. Аналогичным образом отсортируем все точки по координате y.

2. Обозначим периметры минимальных треугольников через l_1 и l_2 соответственно, $l = min(l_1, \ l_2)$. Далее обозначим полосу B шириной l (этого хватит, ибо наибольшая сторона треугольника тогда не более $l/2$). Далее будем пойти по блоку высоты $l/2$. Утверждается, что в таком блоке ограниченное количество точек (на самом деле, не более 16)



Алгоритм 7. *Поиск диаметра точек на плоскости.*

Определение 3.1. Опорной прямой к фигуре Ф называется прямая l, параллельная, касательная к фигуре Ф, такая что Ф лежит в одной полуплоскости относительно Ф.

Утверждение 3.1. Пусть L_1 и L_2 — параллельные опорные прямые в точках A_1 и A_2, при этом расстояние между L_1 и L_2 максимально. Тогда $A_1A_2 \perp L_1$, $A_1A_2 \perp L_2$

Доказательство. Предположим противное. Тогда рассмотрим прямые $L_1', L_2' \perp A_1A_2$. Тогда доведем их до опорных и получим противоречие. □

Утверждение 3.2. Диаметр фигуры — максимальное расстояние между парами опорными прямыми.

Доказательство. Обозначим диаметр через AB, построим для этого прямые L_1 и L_2, перпендикулярные AB и доведем их до опорных прямых. Тогда расстояние не уменьшится □

Теперь можно заняться построением алгоритма (методом вращающих калиперов)

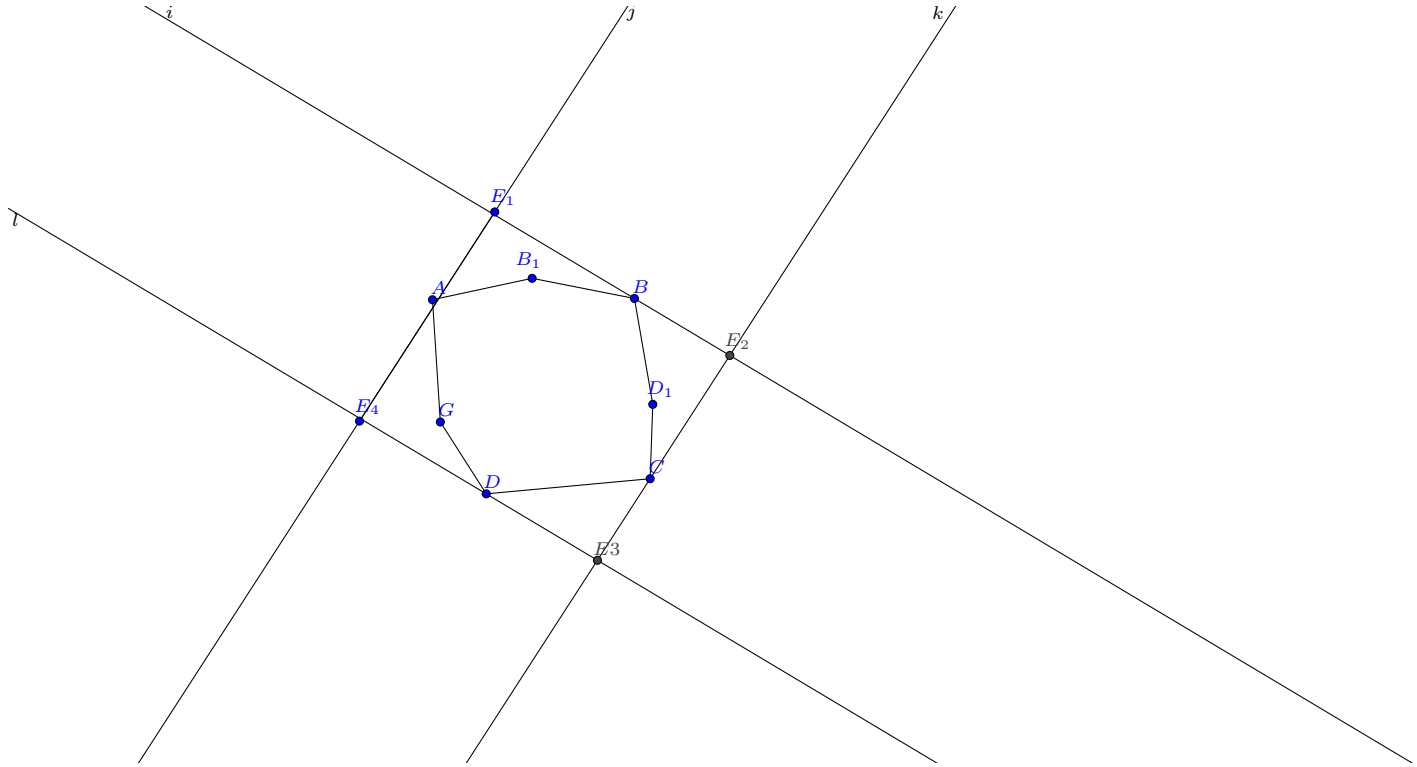
Перебираем пары перпендикулярных опорных прямых: обозначим прямые через L_1 и L_2 на угол, равный $min(\alpha, \beta)$, и ищем ма $max(A_i, B_i)$ по таким точкам. В итоге, будет $O(n)$ вращений в время работы — $O(n)$.

Алгоритм 8. *Поиск накрывающего прямоугольника минимального периметра.*

Рассмотрим на данном чертеже сумму $P_1 = AE_1 + E_1B = AB \cdot (\sin(\alpha_1) + \cos(\alpha_1)) = AB \cdot (\sin\alpha_1 + \sin(\alpha_1 + \frac{\pi}{2})) = AB \cdot \sqrt{2} \cdot \sin(\alpha_1 + \frac{\pi}{4})$ *— выпуклая функция на отрезке* $[0, \frac{\pi}{2}]$. *Заметим, что сумма* $P_1 + P_2 + P_3 + P_4$ *— выпуклая функция, где* $\alpha_i = \alpha + c_i$, α *— угол поворота прямоугольника. Тогда* $\min(P_1 + P_2 + P_3 + P_4)$ *достигается на границе диапазона* α.

Научимся выполнять алгоритм при помощи данной идеи. Будем поворачивать наш прямоугольник на минимально возможный угол (при этом сохраняется инвариант: одна из сторон многоугольника лежит на границе прямоугольника). При пересчете можно учитывать только один угол.

В начале алгоритма возьмем крайние точки (самая нижняя, самая левая, самая верхняя, самая правая). Алгоритм заканчивает свою работу, когда уже будет встречена исходная точка или произойдет суммарный оборот на $90°$.



Алгоритм 9. *Поиск накрывающего прямоугольника минимальной площади*

Будем использовать схожую идею, что и в предыдущем алгоритме. Вычислим площадь "опорного" прямоугольника: $S = AC \cdot \cos\gamma_1 \cdot BD \cdot \cos\gamma_2 = \frac{1}{2}AC \cdot BD \cdot (\cos(\gamma_1 - \gamma_2) + \cos(\gamma_1 + \gamma_2)) = \frac{1}{2}AC \cdot BD \cdot (cos\varphi + cos\psi)$, *где* $\varphi = const$, $pi \leq \psi = c_1 + c_2 + 2\alpha \leq \pi$. *Таким образом,* S *— выпуклая функция, и можно построить аналогичный алгоритм.*

Алгоритм 10. *Обход граней плоского графа.*

Даны координаты вершин и список ребер. Найдем все грани плоского графа.

Для начала докажем следующую теорему:

Теорема 3.1. *(Эйлер) Для планарного графа* G *выполнено соотношение:* $v + f - e = 2$, *где* v *— количество вершин графа* G, f *— количество ребер графа* G, e *— количество ребер графа* G.

Доказательство. Доказывается тривиальной индукцией с применением леммы Жордана. □

Ход алгоритма:

1. Сортируем все ребра в каждой вершине по полярному углу (по теореме Эйлера данный шаг выполняется за $O(v \log v)$ операций).

2. Выполняем поиск очередной грани: если мы прошли по ребру $\overrightarrow{ab}$, то ищем следующее ребро из данной вершины b после ребра ba. При этом будем обратное ребро ba класть в очередь. Таким образом, если начать строить очередную грань по ребру из очереди, то обход граней будет идти в порядке возрастания удаленной от первой грани.

Заметим, что данный алгоритм позволяет на сфере найти наикратчайший путь по ребрам грани. Но на плоскости существует внешняя грань, которая не позволяет уже утверждать данное.

Алгоритм 11. *Поиск внешней грани плоского графа.*

Чтобы найти внешнюю грань, необходимо выполнить данные действия:

1. Найти крайнюю точку множества (в нашем случае, нижнюю левую точку).

2. Берем ребро с минимальным полярным углом и повторяем данный шаг до тех пор, не прошли ли мы в исходную точку.

Алгоритм 12. *Проверка, является ли грань внешней?*

Можно предоставить два способа, судя по исходным идеям:

1. Вычислить сумму углов у грани с n вершинами: если сумма равна $\pi(n-2)$, то грань является внутренней, если сумма равна $\pi(n+2)$, то грань является внешней.

2. Вычислить ориентированную площадь данного многоугольника по следующей формуле: $\sum(p_{i+1}.x - p_i.x) * \left(\frac{p_{i+1}.y + p_i.y}{2}\right)$, и по знаку данной суммы можно ответить на поставленный вопрос.

4 §33. Сумма Минковского

Определение 4.1. Суммой Минковского двух фигур E и F называется множество $G = E \oplus F = \{p + r \mid p \in E, r \in F\}$.

Пример 4.1. Сумма двух точек на плосости есть точка.

Пример 4.2. Сумма точки и фигуры F есть фигура F.

Пример 4.3. Сумма двух отрезков есть параллелограмм.

Пример 4.4. Сумма круга и квадрата с центром в одной точке есть закругленный прямоугольник.

Теорема 4.1. *Обозначим через E, F — выпуклые многольники с m и nвершинами соответственно. Тогда $G = E \oplus F$ является выпуклым многоугольником с не более, чем $m + n$ вершинами.*

Доказательство. Рассмотрим некоторое направление $\overrightarrow{e}$ и посмотрим крайнюю точку множества G — она будет складываться из крайней точки E и крайней точки F. Поэтому можно посмотреть на нормаль к ребру множества E и показать, что для любой точки $K \in F$ будет являться крайним ребром (в обратную сторону все работает аналогично). Осталось обработать случай параллельных ребер: тогда их сумма будет крайним ребром. □

Из доказательства данной теоремы можно придумать следующий:

Алгоритм 13. *Алгоритм построения суммы Минковского двух выпуклых многоугольников.*
Ход алгоритма:

1. Берем крайние левые точки в множествах E и F. Обозначим их через A_1 и B_1 соответственно. Строим точку $C_1 = A_1 + B_1$.

2. Среди следующих отрезков A_1A_2 и B_1B_2 выбираем отрезок с наименьшим углом (предположим, что выбран отрезок A_1A_2). Тогда надо добавить данный отрезок к C, и переместить точку $A_1 \to A_2$ (если прямые параллельны, то надо продвинуть сумму точек).

Данный алгоритм работает за $O(N)$, где N — количество вершин в E и F.

Алгоритм 14. *Определить, пересекаются ли два выпуклых многоугольника?*

Алгоритм очень прост: отразим один относительно начала координат, и если сумма Минковского содержит точку 0. *(Проверку можно осуществлять, к примеру, таким образом: векторы $\overrightarrow{p_ip_{i+1}}$ и $\overrightarrow{p_i0}$ всегда образуют правый поворот или всегда образуют левый поворот).*

Проблема 4.1. Есть диван и робот, которые имеют форму выпуклого многоугольника. Определить ГМТ точек центра робота R, где он может оказаться.

Заметим, что ГМТ ограничивает некоторый многоугольник, который сильно похож на сумму Минковского D и R (где через D обозначается диван). Что происходит, когда робот пересекает диван? Тогда для вектора $\overrightarrow{x}$ выполнено соотношение: $R + \overrightarrow{x} \cap D \neq \varnothing$. Значит, существуют такие $\overrightarrow{r} \in R$, $\overrightarrow{d} \in D$, такие что $\overrightarrow{r} + \overrightarrow{x} = \overrightarrow{d}$, что равносильно $\overrightarrow{x} = \overrightarrow{d} - \overrightarrow{r}$. В итоге, $X = D \oplus (-R)$.

5 §34. Метод сканирующей прямой

Проблема 5.1. Задача "Кассы". Для каждой кассы известно время открытия и время закрытия касс. Надо найти промежуток времени, в который кассы работают одновременно. Таким образом, необходимо найти отрезок, в который работает хотя бы одна касса.

Можно рассматривать только началы и концы касс, и считать количество работающих в данный момент количество касс. Значит, надо найти периоды времени, в которые количество работающих касс равно количеству касс.

На задаче, рассмотренной выше, можно сформулировать идею метода сканирующей прямой: есть некоторый набор событий, на каждое из которых можно реагировать и решать определенные задачи.

Рассмотрим некоторые задачи вычислительной геометрии, которые решаются при помощи данного метода.

Проблема 5.2. Поиск наибольшего креста. На плоскости расположены вертикальные и горизонтальные отрезки. Найти 2 пересекающихся отрезка с максимальной суммой длин.

Обрабатываем события:

1. Начало горизонтального отрезка.
2. Конец горизонтального отрезка (в первом и втором случаях надо изменить состояние структуры данных).
3. Появление вертикального отрезка (среди всех открытых горизонтальных отрезков надо найти пересекающимися с ними и ищем самый длинный из них, чтобы попытаться обновить ответ). Заметим, что открытые горизонтальные отрезки, пересекающиеся с данным, имеют последовательный диапазон. Для этого можно использовать дерево отрезков (надо хранить отображение из горизонтального отрезка в индекс в дереве отрезков).

Таким образом, каждая операция будет обрабатываться за $O(\log N)$ времени.

Замечание 5.1. Можно в качестве структуры для решения данной задачи можно использовать декартово дерево (надо хранить y-координату, max на отрезке и приоритете в каждом узле). Можно еще использовать *splay*-дерево.

Проблема 5.3. Подсчет площади объединения прямоугольников на плоскоти.

Рассмотрим следующие события:

1. Начало прямоугольника.
2. Конец прямоугольника.

Как производится обработка события? Надо добавлять площадь объединения отрезков при переходе от одного события к другому. Для того, чтобы найти объединение отрезков, можно воспользоваться сканирующей прямой. Занумеруем все начала и концы. В дереве отрезков будем хранить количество минимумов на отрезке и сам минимум (и нам отстанется проссумировать количество минимумов, если этот минимум равен 0, можно воспользоваться отложенными операциями).

Проблема 5.4. Поиск набора пересекающихся отрезков на плоскости (надо проверять два соседних отрезка на порядок их изменения, и считать текущую позицию изменения отрезков).

6 §35. $K-d$ дерево

K-d дерево разбивает множество точек на вертикальные и горизонтальные прямоугольники.

Алгоритм 15. *Построение K-d дерева.*

Разобьем множество точек (если их больше 1) в поддереве вертикальной или горизонтальной прямой по медиане (чередуем горизонтальные и вертикальные направления) (слева — не больше медианы, справа — больше). В узлах дерева будут храниться проведенные прямые, а в листьях будут храниться точки (когда размер поддерева равен 1).

Заметим, что количество листьев равно $O(n)$, а глубина дерева $O(\log n)$. Таким образом, размер дерева равен $O(n)$, а построение данного алгоритма будет происходить за $O(n \log n)$, если медиану искать за $O(n)$.

Проблема 6.1. Отвечать на запросы вида нахождения всех точек из прямоугольника R на плоскости.

```
def search(v, R):
    if v == Leaf:
```

```
    //check and give result
if v.left ∩ R ≠ ∅:
    search(v.left, R)
if v.right∩R ≠ ∅:
    search(v.right, R)
```

Обозначим через $Q(n)$ число регионов в поддереве из nвершин, которые могут пересекать прямую r(построенная по ребру многоугольника). Так как мы делим на следующем ходу в другом направлении, то получим, что $Q(n) = 2 + 2 * Q(\frac{n}{4})$, (ибо с каждой стороны могут пересекать прямую r). Из этого следует, что $Q(n) = O(\sqrt{n})$. В k-мерном случае, получаем $Q(n) = 2^{k-1} + 2^{k-1}Q(\frac{n}{2^k})$, из чего можно получить оценку $Q(n) = O(n^{1-\frac{1}{k}})$.

Проблема 6.2. Поиск ближайшей точки к данной на плоскости

Будем использовать dfs, который начинает обход из лучшего поддерева. Когда доходим до листа, то обновляем минимальное расстояние. При этом, если понятно, что расстояние заведомо не обновляется, то можно не заходить в ту часть дерева.

Алгоритм работает в среднем хорошо, но есть анти-тест, в котором все точки находятся на одинаковом расстоянии от центра.

7 §36. Триангуляция Делоне

Определение 7.1. Обозначим через V — некоторое множество точек на плоскости. Триангуляцией Делоне называется такая триангуляция плоскости, что для любого треугольника из триангуляции его описанный круг содержит только вершины треугольника.

Для построения триангуляции будем добавлять по одной точке. Будем считать, что никакие 4 точки не лежат на одной окружности, тогда триангуляция будет единственной.

Утверждение 7.1. (Критерий Делоне) Ребро AB принадлежит триангуляции Делоне, согда существует окружность с хордой AB, внутренность которой не содержит других точек триангуляции.

Доказательство. Необходимость. Если ребро принадлежит триангуляции, то тогда существует треугольник, который содержит данное ребро, и малым шевелением окружности получим требуемое.

Достаточность. Если не существует такой окружности, то тогда существует ребро CD, которое отделяет точки A и B. Тогда можно понять, что сумма углов четырехугольника $ABCD$ меньше 360°. □

Определение 7.2. Ребро AB называется хорошим в некоторой триангуляции, если 2 смежных с ней треугольника подчиняются условию Делоне

Утверждение 7.2. Все ребра в триангуляции Tr хорошие, согда Tr является триангуляцией Делоне.

Доказательство. Достаточность очевидна.

Необходимость. Предположим, что существует треугольник ABC и точка E, что E лежит в описанном круге треугольника ABC. При этом треугольник $ABE \notin Tr$, ибо тогда ребро AB не является хорошим. Тогда если точка D образует с AB треугольник из триангуляции, то она лежит вне описанного круга треугольника ABC. Тогда $\angle BEA > \angle BDA$, так как точка E лежит внутри, а точка D — снаружи. Значит точка E принадлежит описанному кругу треугольника ABD. Поэтому $\angle BEA < \angle BED$, и этот угол будет возрастать, что и приводит к противоречию. □

Алгоритм 16. *Построение триангуляции Делоне.*

Будем строить триангуляцию итеративным образом. Рассмотрим три варианта:

1. $v_i \in \triangle ABC$. Ребра v_iA, v_iB, v_iC будут хорошими по критерию Делоне. Действительно, построим окружности, содержащие только ребра v_iA, v_iB и v_iC. Поэтому добавим их в триангуляцию, а с ребрами AB, BC и AC поступим следующим образом: если новый круг содержит точку, смежную с труегольником ABC, то произведем операцию $flip$: замену ребра в новом четырехугольнике на противоположные. Далее надо проверить новые противоположные ребра AX и BX. Суммарное количество ребер будет линейно, но позже будет доказано, что количество $flip$-ов будет линейно.

2. $v_i \in AB$. Обозначим смежные треугольники через ABD и ABC. Тогда ребра Av_i, Bv_i, v_iC и v_iD — хорошие по критерию Делоне. Аналогичным образом можно будет запускать $flip$ для ребер AB, BC, CD, AD.

3. v_i лежит вне всех треугольников — рассмотрим огромный треугольник, а затем выкидываем лишние ребра.

Алгоритм 17. *Построение минимального остовного дерева на плоскости.*

Утверждение 7.3. Самое легкое ребро разреза S и $V \backslash S$ принадлежит триангуляции Делоне.

Доказательство. Предположим противное. Тогда рассмотрим окружность с диаметром данного ребра. По критерию Делоне, существует точка внутри окружности, и найдется меньшее расстояние в разрезе. □

Для существования триангуляции Делоне необходима проекция плоскости на параболоид (можно почитать в литературе).

В алгоритме Воронова в среднем на одно добавление точки приходиться около 3 выполнений операции $flip$. Так как $K - d$ дерево работает в среднем за $O(\log n)$, то триангуляция Делоне (при локализации треугольника), работает за $O(N \log N)$ (в худшем случае эта вещь работает за $O(N^2)$).

Алгоритм 18. *Построение триангуляции Делоне методом "разделяй и властвуй"*

Разбиваем множество пополам вертикальной прямой (если точек больше 12, иначе действуем руками). Далее их надо объединить. Для этого мы смотрим самый нижний отрезок, соединяющий две части (он будет стартовым отрезком). Строить новый отрезок будем следующим образом: надуваем окружность, до содержания одной из точек выпуклой оболочки (то есть, если AB — текущий отрезок, то надо выбрать такую точку C, которая находится выше отрезка AB, при этом $\angle ACB$ минимален). При этом возможны два случая: точка C лежит на границе одного из двух выпуклых многоугольников (тогда просто добавим этот отрезок и перейдем далее). Оказывается, что CA принадлежит триангуляции Делоне, ибо окружность, описанная около треугольника ABC, не содержит других точек, поэтому можно воспользоваться критерием Делоне. Осталось только удалить все ребра, которые пересекают отрезки AC и CB, ибо при соединении данные точки будут использовании. Шаг заканчивается в том случае, когда отрезок оказался верхним.

Шаг алгоритма работает за $O(N)$, ибо каждое ребро триангуляции Делоне (которая является планарным графом) будет просмотрено не более двух раз. В итоге, алгоритм работает за $O(N \log N)$ детерминированным образом.

8 §37. Диаграмма Вороного

Задача: для каждой точки на плоскости (множество точек конечно) надо понять, какая точка из множества является ближайшей.

Определение 8.1. Диаграммой Вороного называется разбиение конечного множества X на ячейки X_i таким образом, что $X_i = \{y \,|\, \rho(x_i, y) \to min\}$.

Существует тривиальный алгоритм за $O(N^4)$, который просматривает серединные перпендикуляры любой пары точек, а затем их пересекает.

Утверждение 8.1. Ребро AB принадлежит триангуляции Делоне тогда и только тогда, когда ячейки $C(A)$ и $C(B)$ диаграммы Воронова имеют общую границу (хотя бы точку).

Доказательство. Заметим, что если рассмотреть пересечение ячеек диаграммы Воронова, то эта точка будет центром окружности многоугольника, которые создают данные ячейки.

Достаточность. Существует точка на серединном перпендикуляре, окружность которого содержит только необходимые точки. Следовательно, по критерию Делоне, ребро AB будет принадлежать триангуляции Делоне.

Необходимость. По критерию Делоне строим окружность с хордой AB, не содержащей других точек. Тогда её центр принадлежит границе ячеек $C(A)$ и $C(B)$. □

Алгоритм 19. *Построение диаграммы Вороного (алгоритм Форчуне)*

Будем использовать метод сканирующей прямой. Рассмотрим для каждой из точек ГМТ, таких что расстояния до прямой и до точки совпадают (как известно, данное ГМТ является параболой).

Рассмотрим несколько случаев событий:

1. *Появление новой точки (пересечение серединных перпендикуляров будет лежать правее — есть проблема, нельзя сразу обработать). Поэтому надо просмотреть события, которые могут быть связаны со смежными серединными перпендикулярами (изначально добавится пересечение серединных перпендикуляров O треугольника ABC, где A — добавленная точка), и если они идут раньше, то необходимо удалить событие с точкой O.*

2. *Пересечение серединных перпендикуляров (аннигиляция параболы). В итоге, пары смежных к данной ячейке пересекаются. Надо добавить новый серединный перпендикуляр двух "встреченных" точек, открытый в необходимую сторону, и добавляем в пересечение ожидаемых ребят (которые были соседними со встреченными вершинами).*

3. *Пересечение хотя бы трех серединных перпендикуляров — добавление всех новых серединных перпендикуляров во встрече.*

Время работы алгоритма состоит из сортировки точек $O(N \log N)$, и поиск локализации событий при хранении в сете выполняется в сумме за $O(N \log N)$.

Часть II
Параллельные алгоритмы

9 §38. Потоки

Изначально процессоры уже затачивались на многозадачности, при этом выполнялся процесс *task scheduling,* в котором происходило переключение контекста (было некоторое время на смену). Таким образом, уже было писать многопоточные программы. При наличии нескольких ядер на машине, можно уменьшить накладные расходы (можно перераспределять процессы). В Windows есть 7 приоритетов процессов и 7 приоритетов потоков (в сумме их 35).

У процессов ресурсы не зависимы, а у потоков есть одно общее виртуальное адресное пространство (в 32-битных системах под код и данные отдается ровно 3 Гб информации, остальные используются на адрес ядра). В начале есть участок из 64 кб, отведенный на нулевой указатель (в начале участка ядра есть аналогичное пространство). В $x86$ используется 4 Тб виртуального адресного пространства (при записи связывается с реальным адресным пространством).

Если на машине есть одно ядро, то не надо синхронизировать доступ (raise-condition не будет). У каждого потока есть свой стек. По умолчанию, размер стека в Windows равен 1 Мб. Перед распараллеливанием можно узнать информацию о возможности распаллеливания процессов (количество ядер и возможное количество запускаемых потоков).

Многопоточность необходима для:

- Ускорения работы программы
- Разделения задач (отделение от интерфейса, к примеру). Можно реализовать механизм call-back-ов (прогресс о выполнении потока).

Рассмотрим *API* для использования в *STL* C++11 . В 11-м стандарте появился заголовочный файл *thread*. Функция инициализации потока равнозначна точке входа (вызываются конструкторы статических полей класса перед main, после вызова — вызываются деструктора). Поток завершает свою работу, когда функция потока закончит свою работу, либо когда функция *main* завершит работу, либо когда вызываются *terminate* процессы (искусственное прерывание потоков). К примеру, если есть unhandled exception, то тот вызывает terminate.

```
#include <thread>
void doSmth();
int main() {

    std::thread t(doSmth); //инициализация потока
    t.join(); // ожидание завершения выполнения функции потока

}
```

Как можно заметить, что класс *thread* не является шаблонным, но его конструктор является мощным шаблоном (можно передать функтор, который имеет оператор круглые скобки).

Как это выглядеть?

```
class BackgroundTask {
public:
    void operator()() {
        ...
        doSmth...
    }
}
BackgroundTask task;
std::thread t(task);
std::thread t2(BackGroundTask());
std::thread t3([] {

    doSmth...; // лябмбды видят локальные параметры (но появляется невалидность при создании нового потока)
    // но, скорее всего, можно передать параметры по значению
});
```

Существуют две возможности управления потоком: join и detach (отвязать объект thread от потока). Если был вызван деструктор *thread*, а поток не завершил работу и приаттачен к сущности std::thread, то вызывается std::terminate.

```
bool thread::joinable(); // возвращает true, если поток жив и присоединен
```

Напишем плохой код:

```
struct Functor {
    int& data;
    Functor(int& _data): data(_data) {}
    void operator() () {
        for (int i = 0; i < 1000000; ++i) {
            data = i;
        }
    }
}
void Oops() {
    int localData = 0;
    Functor f(localData);
    std::thread t(f);
    t.detach();
}
```

В данном коде есть проблема: порча стека, меняя данные в нем или можно получить access violation.

```
void notOops() {
    int localData = 0;
    Functor f(localData);
    std::thread t(f);
    try {
        doSmth(); // без обработки поток получает terminate
    } catch (...) {
        t.join();
        throw ...;
    }
    t.join();
```

```
}
```

Лучше всего писать так (идеология RAII — Resouce Acquisiton Is Initializing, гарантия освобождения ресурсов в деструкторе)

```
class ThreadGuard {
public:
    ThreadGuard(std::thread& _t): t(_t)
    ~ThreadGuard() {
        if (t.joinable()) {
            t.join();
        }
    }
private:
    std::thread& t;
    ThreadGuard(const ThreadGuard& other) = delete; //запрет конструктора копирования
    ThreadGuard& operator =(const ThreadGuard& other) = delete; // запрет присваивания,
}
void notOopsNeo() {
    int localData = 0;
    Functor f(localData);
    std::thread t(f);
    ThreadGuard g(t);
    doSmth();
}
```

В поток можно передавать параметры, они передаются по значению.

```
template<class F, class ... Args>
std::thread::thread(F, Args...)
```

Напишем пример передачи параметров в thread (с косяком):

```
void func(const string& s) {


}
void oopsK(int param) {
    char* buffer = new char[1024];
    sprintf(buffer, "%i", param);
    std::thread t(func, buffer); // buffer будет скопирован как указатель, а преобразование к string
    //6удет выполняться в контексте другого потока
    t.detach();
    delete[] buffer; // плохо, ибо поток он передаст указатель на разрушенную область памяти
}
```

Именно поэтому многопоточные программы сложно отлаживать (во-первых, контроль ресурсов, во-вторых, слежение для блокировки). Лучше написать так:

```
std::thread t(func, std::string(buffer));
```

Далее рассмотрим Interlock-functions (atomic functions), который используется для счетчика ссылок.

Информация о потоках разрешается при помощи функции std::hardware_concurrency() — статическая функция, возвращающая действительное число ядер. Потоки можно идентифицировать (

```
std::thread::id myId = t.get_id()
```

). Также в Windows можно использовать следующие функции: id потока (unsigned int), HANDLE потока (объект ядра, с которыми работают Windows-функции, можно распознать объект ядра по 32 битам информации).

10 §39. Разделение данных между потоками

Рассмотрим способы разделения данных между потоками. Если данные доступны только для чтения, то их разделять не надо. Если данные доступны для записи, то могут возникнуть проблемы.

К примеру, рассмотрим операцию удаления из двусвязного списка: 1) надо сначала найти удалить элемент x; 2) затем надо перенаправить указатели; 3) выполнить delete. В итоге, если второй поток желает работать в середине удаления элемента из списка, то возникнут проблемы (список становится невалидным).

Возникает понятие *"race condition"* (гонка за ресурсами). Как можно бороться с race condition:

1. Защищенный доступ данных.
2. Изменить структуру таким образом, чтобы ее изменение состоит из набора независимых изменений (Lock-free programming)
3. Транзакционная работа: есть набор изменений, а затем происходит либо commit, либо rollback (если был конфликт со стороны другого потока) на все операции.

Мы будем выполнять блокировки. Разобьем кусок на блоки и пометим его таким образом, чтобы его мог выполнять только один поток. Надо создать mutex (mutually exclusive).

```
mutex.lock()
// some code
mutex.unlock()
// есть mutex.try_lock(), который пытается зайти
```

Этот механизм использует стандартные объект ядра.

В WinAPI есть CRITICAL_SECTION и методы EnterCriticalSection, TryCriticalSection и LeaveCriticalSection (в одном процессе), также можно создать мютекс:

```
HANDLE CreateMutex(..., name)
HANDLE OpenMutex(..., name)
//можно синхронизировать разные процессы
```

Критическая секция — это пара из mutex и количество локов (если локов 0, то идет выполнение, иначе — режим ожидания). Если unlock не происходит быстро, то влетающий поток может уйти в спячку надолго. Чтобы такого не возникало, использовался EnterCritSectionWithSpinCount (атомарный count).

Чтобы выполнять проверку на то, чтобы выполнить unlock, существует std::lock_guard:

```
std::list<int> myList;
std::mutex myMutex;
void addToList(int value) {

    std::lock_guard<std::mutex> guard(myMutex);
    myList.push_back(value);

}
//Нельзя возвращать всякого рода ссылки на стек!!
```

Напишем потоконезависимый стек:

```
class thread_safe_stack {
public:

    bool empty() const;
    size_t size() const;
    T pop(); // not good interface
    void pop();
    void push(const T& e);

private:

    std::stack<T> _internalStack;
    std::mutex m;
```

```
}
```

Но реализация ломается на таком коде:

```
thread_safe_stack<int> s;
if (!s.empty()) {
    const int value = s.top();
    s.pop();
    doSmth(value);
}
```

Проблема состоит в том, что top-ы сработают одновременно, а удаляться будет другой элемент. Аналогично нельзя возвращать pop по значению.

Напишем исправленный вариант:

```
std::shared_ptr<T> pop() {
    std::lock_guard<std::mutex> lock(m);
    if (internal_stack.empty()) throw empty_stack.exception();
    std::shared_ptr<T> res(std::make_shared<T> (internal_stack.top()));
    internal_stack.pop();
    return res;
```

Перейдем к рассмотрению взаимных блокировок (Dead Lock):

```
// Thread-1
mutexA.lock()
mutexB.lock()
//Thread-2
mutexB.lock()
mutexA.lock()
```

Несколько советов:

1. Блокировать сначала mutexA, а затем только блокировать mutexB
2. Объединить всесозможные блокировки в один класс и выстраивать их иерархию.

11 §40. Атомарные типы

В прошлом параграфе была проблема с неатомарностью метода pop() в стеке. Поэтому воспользуемся atomic типами. В STL C++ 11 они представлены в виде std::atomic

```
template<class T>
std::atomic<T>
//some methods
```

Для пользовательских типов определены следующие методы: operator =, is_lock_free(), store(), load(), operator T, exchange, compare_exchange. А для примитивных типов есть методы fetch_add, fetch_sub, fetch_add(for booleans with and), fetch_or, fetch_xor. В WinAPI есть аналогичные interlocked функции: _InterlockedIncrement, _InterlockedDecrement, ... (работают с shared_ptr).

К примеру,

```
if(_InterlockedDecrement(refCount) == 0) {
    delete object;
}
```

Здесь операция декремента работает корректно с shared_ptr.

_InterlockedBittestandSet используется в spin-блокировках:

```
while(_interlockedBittestandSet(flag, false, true)) {
    //doSmth
}из
//unlock
_InterlockedExchange(flag, false);
```

На очень быстрых действиях лучше использовать interlock-функции, потому что не происходит связи с mutex, поэтому работает быстрее. std::atomic в Windows реализован при помощи спин-блокировок. С помощью _InterlockedCachedCompareExchange можно реализовать любые арифметические операции.

12 §41. Спецификации мютексов

Как было видно, std::mutex не может просто взять и по призыву приостановить действие lock-а. Поэтому для этого используется std::timed_mutex:

```
class std::timed_mutex {
    lock();
    bool try_lock();
    t<> bool try_lock_for(const chrono::duration<...>& duration); //lock for this time
    t<> bool try_lock_until(const chrono::point_of_time<...>& p);
}
```

Иногда надо делать так, чтобы при попытке записи происходила блокировка. Для этого используется shared mutex. Что хотят добавить в стандарт C++:

```
std::shared_timed_mutex {
    lock
    try_хи
    try_lock_for
    try_lock_until
    //shared-block
    lock_shared()
    try_lock_shared
    ..._for
    ..._until
}
```

В реализации используются семафоры при помощи event-ов.

Пример 12.1. DNS-cache

```
class DNSCache {
    std::map<std::string, dns_entry> entries;
    mutable boost::shared_mutex m;
public:
    dns_entry find(const std::string& domain) const {
        boost::shared_lock<boost::shared_mutex> lock(m) // analogue of shared guard
        auto it = entries.find(domain);
        return (it == entries.end()) ? dns_entry() : it->second;
    }
    void update_entry(const std::string& domain, const dns_entry& d) {
```

```
        std::lock_guard<boost::shared_mutex> g(m);
        entries[domain] = d;
    }
}
```

Предоставим еще один способ работы с блокировками заключается в том, что все синхронизационные объекты получает свою иерархию (присваивается уровень иерархии для каждого mutex при помощи топологической сортировки). Но надо гарантировать во время выполнения, что никакой mutex с меньшим уровнем не заблокируется перед исходным lock-ом. Поэтому надо написать иерархический класс, который будет контролировать это дело:

```
class HierarchicalMutex {
    std::mutex;
    static thread_local<int> current_thread_level = MAX_INT;
    const int mutex_level;
    //lock состоит из 3 частей:
    // 1: проверка того, что mutex_level < current_thread_level);
    // 2: m.lock();
    // 3: prev_level = current_thread_level;
    // 4: current_thread_level = mutex_level;
    //
    // unlock, разблокировка происходит по транзитивному замыканию
}
```

Операционная система представляет API для работы с общим хранением потоков (причем их число ограничено, порядка 1000). Есть некоторые методы, которыми можно пользоваться: TLSAlloc, TLSFree, TLSSetValue, TLSGetValue(index, data) — WinAPI, и thread_local в STL C++11.

13 Events

Рассмотрим механизм обработки событий. Как это делать при помощи mutex-ов:

```
bool flag;
std::mutex m;
void wait_for_flag() {
    std::unique_lock<std::mutex> l(m);
    while (!flag) {
        l.unlock();
        std::this_thread::sleep_for(std::chrono::multiseconds(100));
        l.lock();
    }
}
```

В данном случае проходят очень много времени. Есть в Windows самосбрасывающиеся и несамосбрасывающиеся события.

```
#include <condition_variable>
std::mutex m;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
void data_preparation_thread() {
    while(hasData()) {
        const data_chuk data = prepare();
        std::lock_guard<std::mutex> l(m);
        data_queue.push(data);
        data_cond.notify_one();
```

```
    }
}
void data_processing_thread() {
    while (true) {
        std::unique_lock<mutex> l(m);
        data_cond.wait(l, []{return !data_queue.empty()});
        data_chunk data = data_queue.foront();
        data_queue.pop();
        l.unlock();
        process(data);
        if (is_last_chunk(data)) {
           return;
        }
    }
}
```

При этом все хорошо, ибо notify_one предупреждаем ровно один объект и извлекаем ровно один объект. Если использовать notifyAll, то проснутся все потоки, и информация только достанет один элемент, а остальные не получат данные, и тогда информация будет получать false на wait, в итоге получится бесконечный цикл на блокировке. Notify_all полезно слать тогда, когда надо завершить приложений.

Напишем обработчик заданий, который не зависит от методов синхронизации:

```
#include <mutex>
#include <condition_variable>
#include <memory>
template <class T>
class ThreadSafeQueue {
public:
    ThreadSafeQueue();
    void push(T value);
    bool try_pop(T& value);
    shared_ptr<T> try_pop();
    void wait_and_pop(T& value);
    shared_ptr<T> wait_and_pop();
private:
    std::mutex m;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
}
template<class T>
void ThreadSafeQueue::push(T newValue) {
    std::lock_guard<...> l(m);
    data_queue.push(newValue);
    data_cond.notify_one();
}
template<class T>
shared_ptr<T> ThreadSafeQueue::wait_and_pop() {
    std::unique_lock<...> l(m);
    data_cond.wait(l, [this] {return !data.push.empty();]});
    std::shared_ptr<T> result = std::make_shared(data_queue.front());
    data_queue.pop();
    return res;
}
```

14 Future, packaged tasks, promise

Теперь хотим получить результат выполнения задания, ибо у нас есть объект для отправки заданий. Можно аналогично завести очередь результатов, но тогда нам надо будет ждать либо всегда, либо периодически. Поэтому будем использовать future.

```
#include <future>
std::future <T>
std::shared_future <T>
int find_answer(params...);
int main() {
    std::future<int> answer = std::async(find_answer,params...);
    //...
    std::cout << answer.get();
}
```

Если первый параметр — это функция, то тогда второй, третий аргументы и т.д. — параметры функции. Если первый параметр — std::launch, то тогда далее передается либо std::launch_differed – выполнение на текущем потоке или std::launch::async – на отдельном потоке. По default — то тогда отдается уже приоритет реализации.

Можно организовывать выполнение заданий через выполнение packaged task (более гибкое выполнение), его можно перемещать и вызывать много раз (внутри есть ссылка на вызываемую функцию). Иногда лучше, чтобы тот участок, запрашивающий данное дело, не должно знать информацию о задаче. Это можно сделать при помощи класса promise.

Полагаем, что есть один поток, знающий о соединении с удаленной машиной и работает с пакетами.

```
struct data_packet;
struct outgoing_thread;
class Connection {
    bool has_incoming();
    bool has_outgoing();
    data_packet incoming();
    std::promise<payload_type> get_promise(int dataId);
    outgoing_packet top_of_outgoing();
    void send(payload);
};
void process_connections(connections) {
    while(!done) {
        for(connection in connections) {
            if (connection->has_outgoing_data()) {
              data_packet data = connection()->outgoing();
              data.promise.set_value(connection->send(data.payload));
            }
        }
    }
}
```

Данная вещь хороша для отсылки сервера информации (надо сделать такую-то задачу), outgoing_data — знает, что надо отправить на исполнение задачу.

Перейдем к обработке исключений:

```
future<double> f = async(square_root, -1);
double y = f.get(); //async сохраняет исключение, и оно пробрасывается через get
double square_root(double x) {
    if (x < 0) {
```

```
        throw std::out_of_raGnge(“x<0”);
    }
    return sqrt(x);
}
```

С packaged task все аналогично. Но в promise все идет немного не так:

```
promise<double> p;
try {
    p.set_value(square_root(data param));
} catch (std::exception e) {

    p.set_exception(std::current_exception() --- std::exception_ptr); //задерживает exception и future разб
    // и get бросит исключение в запрашивающем потоке
}
```

Перейдем к вопросу ожидания с нескольких потоков. Для этого есть класс std::shared_future::shared_future<T>(std::future<T>& x), и можно теперь дожидаться копию объекта, полученного из future. И тогда get() получит управление. Как сделать так, чтобы остановить все потоки? Можно передавать по future, чтобы получить на get то, что надо остановить потоки.

Рассмотрим ThreadPool. Если мы хотим ждать все потоки, то надо прервать все через некоторое время. Хочется выполнять пул потоков, которые надо выполнять и сообщать результаты.

Поэтому есть функции submit, которое добавляет задание в очереди. Рабочий класс:

```
void worker_thread() {
    while(!done) {
        std::future<void()> task;
        if work_queue.try_pop(task)) {
            task();
        } else {

            std::this_thread::yield();//по идее, надо бы не ждать, а реализоывать все дело через event’ы.
        }
    }
}
```