ЗКШ, весна 2016/17
Конспект лекции по динамике

Собрано 3 марта 2017 г. в 21:42

Содержание

Лекция #1: Оптимизации к динамике

3 марта 2017

Цель лекции – дойти до популярных оптимизаций “convex hull trick”, “divide & conquer” и т.д.
Для разминки мы сперва разберём пару простых примеров.
В большинстве динамик я буду описывать только переход, считая базу очевидной.

1.1. Умный пересчёт

1.1.1. Числа Фибоначчи

Все вы умеете считать числа Фибоначчи $f_n = f_{n-1} + f_{n-2}$.
Обобщёнными k-числами Фибоначчи назовём $f_n = f_{n-1} + f_{n-2} + \cdots + f_{n-k}$.
Если считать “в лоб”, получится время $\mathcal{O}(nk)$:

```
for (int i = k; i <= n; i++)
  f[i] = 0;
  for (int j = 1; j <= k; j++)
    f[i] += f[i - j];
```

Чтобы получить $\mathcal{O}(n)$, можно заметить “`f[i] = сумма на отрезке`”,
а сумму на отрезке мы умеем считать за $\mathcal{O}(1)$.

1.1.2. Максимальная по сумме подпоследовательность

Задача: найти подпоследовательность с началом в 1, концом в n, у которой расстояние между соседними элементами не более k, из таких выбрать подпосл-ть с максимальной суммой.

Пусть $f[i]$ – max подпосл-ть с концом в i. Решение “в лоб” будет работать за $\mathcal{O}(nk)$:

```
for (int i = k; i <= n; i++)
  f[i] = INT_MIN;
  for (int j = 1; j <= k; j++)
    f[i] = max(f[i], f[i - j] + a[i]);
```

Чтобы получить $\mathcal{O}(n)$, можно заметить “`f[i] = минимум на очереди`”,
а минимум на очереди мы умеем считать за $\mathcal{O}(1)$.

1.1.3. Обобщим

Чтобы получить линейное решение, мы сперва выписываем хоть какую-нибудь динамику, затем пытаемся её оптимизировать. Стандартная оптимизация: увидеть функцию на отрезке.
В обоих примерах мы записывали “динамику назад”. Иногда, чтобы получилось соптимизировать, нужно записать “динамику вперёд”.

1.1.4. Динамика ли?

Задача: даны n точек на прямой, найти k отрезков минимальной суммарной длины, покрывающие все точки.

Можно пытаться решать задачу так: $f[i, j]$ – минимальная стоимость покрыть левые i точек j отрезками. Получится решение за $\mathcal{O}(n^2k)$, можно даже его соптимизировать до $\mathcal{O}(nk)$... Вот только на самом деле задача решается жадно: отсортируем точки, возьмём $n - 1$ расстояний между соседними и удалим $k-1$ максимальных из них. Увидеть, что задача имеет жадное решение, – высокое искусство. При решении задач важно себе всегда задавать вопрос “динамика или жадность”. К сожалению, этот вопрос выходит за рамки нашей лекции.

1.1.5. Два указателя

Задача: даны n точек с весами w_i на прямой, для каждого отрезка точек $[L, R]$ найти $m[L, R]$ минимизирующий сумму взвешенных расстояний на $[L..R]$ до точки m: $\sum_i w_i |x_i - m| \to \min$.

Решение “в лоб” будет работать за $\mathcal{O}(n^4)$: для каждого отрезка перебрали m, для каждого m посчитали сумму. Чтобы получить $\mathcal{O}(n^3)$, достаточно раскрыть скобки и разделить $\sum_i w_i|x_i - m|$ на 4 частичные суммы, каждая считается за $\mathcal{O}(1)$.

Интересная часть задачи – выбор оптимального m. Здесь нам поможет метод двух указателей:

```
for (int l = 0; l < n; l++)
  int m = l;
  for (int r = l; r < n; r++)
    while (m + 1 < r && F(l, r, m + 1) < F(l, r, m))
      m++;
```

Здесь F вычисляется за $\mathcal{O}(1)$ через частичные суммы, получили решение за $\mathcal{O}(n^2)$. На самом деле точка m обладает свойством “самая левая, что сумма весов слева $\geqslant$ суммы весов справа”. Из этого факта следует корректность применения двух указателей к данной задаче.

Вопрос: “дана произвольная динамика, работает ли метод двух указателей?”. Общий способ проверки мне не известен, но опыт подсказывает, что обычно быстрее всего написать две функции (в лоб и с двумя указателями) и “постресcить их на рандоме”.

1.1.6. Convex Hull Trick (CHT): сведение

Задача: даны n точек на прямой, найти k отрезков, покрывающие все точки, минимизировать *сумму квадратов* длин отрезков.

Решение “в лоб”: $f[j, i]$ – минимальная стоимость покрыть левые i точек j отрезками. nk состояний, из каждого n переходов, итого $\mathcal{O}(n^2 k)$.

```
// точки имеют номера 1..n, используя 0 отрезков, можем покрыть 0 точек
f[0] = {0, INF, INF, ...};
for (int j = 0; j < k; j++)
  for (int i = 0; i <= n; i++)
    f[j+1][i] = INF;
    for (int r = 1; r <= i; r++)
      relax(f[j+1][i], f[j][r-1] + cost(r, i)) // relax делает 'min=', как '+='
answer = f[k][n]; // пусть k <= n
```

Чтобы применить CHT нужно расписать функцию `cost` и увидеть линейную функцию:
`f[j][r-1] + cost(r, i) = f[j][r-1] + (r-i)`^{2} `= f[j][r-1] + r`^{2} `- 2ri + i`^{2}
Мы минимизируем эту величину для данного i. Отбросим i^2, он для всех r один и тот же. Оставшееся сгруппируем: `(f[j][r-1] + r`^{2}`) - 2ri = B`$_r$ `+ A`$_r$`·i` – линейная от i функция.
Замечание: в некоторых динамиках нам прямо в условии дают готовые линейные функции.

Теперь получили задачу, решение которой и называется “Convex Hull Trick”:
поочерёдно добавлять линейные функции и искать максимум по всем добавленным в точке i.

```
// точки имеют номера 1..n, используя 0 отрезков, можем покрыть 0 точек
f[0] = {0, INF, INF, ...};
for (int j = 0; j < k; j++)
  CHT.init();
  for (int i = 0; i <= n; i++)
    f[j+1][i] = CHT.getMin(i) + i*i; // мы его отбросили при минимизации, теперь вернули
    int r = i + 1; // для всех следующих i мы будем просматривать такое r
    CHT.addLinear(f[j][r-1] + r*r, -2*r);
answer = f[k][n]; // пусть k <= n
```

1.1.7. Convex Hull Trick (CHT): реализация

Здесь должна быть картинка: **TODO** (пока представьте себе параболу ветвями вверх).

Казалось бы в общем случае нам нужно пересекать произвольные полуплоскости, добавлять новые, что не очень приятно. К счастью обычно задача гораздо проще.
Будем решать её в такой формулировке: есть прямые вида $y = k_i x + b_i$, все $k_i > 0$.
Прямые уже отсортированы в порядке возрастания k_i. Нужно добавлять прямые *в отсортированном порядке* и уметь считать $get(x) = max_i(k_i x + b_i)$ для $x \geqslant 0$.
При этом все k_i, b_i, x целочисленны. Добавим фиктивную прямую $(k_0 = -\infty, b_0 = -\infty)$.

```
struct CHT {
  vector<line> l;
  vector<points> p;
  void init() {
    l = {line(-INF, -INF)};
    p = {};
  }
  void addLine( int k, int b ) {
    line new_line(k, b)
    while (!p.empty() && p.back() is under new_line)
      p.pop_back(), l.pop_back();
    p.push_back(l.back() intersect with new_line) // вещественные числа!
    l.push_back(new line)
  }
  int getMax( int x ) {
    // i - первая точка пересечения правее x
    int i = lower_bound(p.begin(), p.end(), Point(x, -INF));
    return l[i].value(x); // значение прямой l[i] в точке x
  }
};
```

При желании можно $\mathbb{R}$ числа заменить на рациональные и все вычисления провести без погрешности. Суммарное время работы всех `addLine` – $\mathcal{O}(n)$, каждый `getMax` работает за $\mathcal{O}(\log n)$.
Иногда x-ы возрастают, тогда можно применить метод двух указателей и получить суммарное время всех `getMax` и `addLine` $\mathcal{O}(n+m)$, где m – число запросов.

1.2. Рюкзак

1.2.1. Обычный рюкзак с линией памяти

Задача: даны n предметов с положительными целыми размерами (весами) a_i и рюкзак размера W, выбрать подмножество предметов $i_1, i_2, \dots i_k$: $f = \sum a_{i_j} \leqslant W$ и среди таких $f \to \max$.

Есть три стандартных решения такой задачи – перебор за $\mathcal{O}(2^n)$, meet in the middle за $\mathcal{O}(2^{n/2}n)$ и динамика за $\mathcal{O}(nW)$. Мы сейчас сосредоточимся именно на третьем. Для экономии времени опустим вопрос восстановления ответа $\{i_1, \dots, i_k\}$, будем искать f.

```
vector<bool> f(W + 1); // инициализируется нулями
f[0] = 1;
for (int i = 0; i < n; i++)
  for (int x = W - a[i]; x >= 0; x--)
    f[x + a[i]] |= f[x];
```

1.2.2. Добавим bitset

`unsigned long long` можно рассматривать как массив из 64 бит, операции “`|=`”, “`<<`” происходят за один такт. `bitset<n>` – аналогичный объект на n битах, все операции происходят за $\frac{n}{64}$.

```
bitset<W+1> f; // инициализируется нулями, W - обязательно константа
f[0] = 1;
for (int i = 0; i < n; i++)
  f |= f << a[i];
```

И код укоротился, и время улучшилось. При желании к этому способому можно добавить восстановление ответа без потерь во времени.

1.2.3. Рюкзак на отрезке

Более крутая задача: запрос “можно ли отрезком предметов `[L..R]` набрать вес ровно `X`”.

Можно пойти лобовым путём: модифицируем стандартную динамику, добавим параметр `L`: `f[X,L]` – можно ли набрать `X` предметами индексами не менее `L`.
Тогда к моменту `R` мы для каждой пары $\langle$`L`, `X`$\rangle$ знаем ответ.

В нашей динамике есть некая не оптимальность: мы храним 0 или 1. Эта не оптимальность даёт возможность сделать следующую оптимизацию. Давайте после обработки `R` первых предметов хранить `L[X]` – максимальное число такое, что предметами `[L[X]..R]` можно набрать `X`.

```
vector<int> f(W + 1, -1);
for (int R = 0; R < n; R++) {
  f[0] = R; // всегда можем набрать 0
  for (int x = W - a[i]; x >= 0; x--)
    relax(f[x + a[i]], f[x]);
  // Здесь можно отвечать на все запросы с правым концом R
  get(L, X) = (f[X] >= L);
}
```

1.2.4. Мостостроение: тренируемся грамотно выбирать состояние

Задача: даны a брёвен длины x, b брёвен длины y, число l, хотим построить мост из l не обязательно равных рядов брёвен, вопрос – какой максимальной ширины будет самое узкое место моста? $a, x, b, y, l \leqslant n$.

Решение за $\mathcal{O}(n^5)$: `f[a,b,l]` – ответ на задачу, переход – перебрать, сколько брёвен первого и второго типа будем брать для очередного ряда.

Оптимизация #1: бинпоиск по ответу (обозначим ответ x). Сделаем бинпоиск, получим динамику `l[a,b]` – сколько максимум рядов ширины x можно положить. При переходе теперь, зафиксировав число брёвен первого типа i, можем однозначно посчитать минимальное число брёвен второго типа: $\lceil \frac{l-ai}{x} \rceil$. Итого $\mathcal{O}(n^3 \log \texttt{ANS})$.

Оптимизация #2: "измельчение перехода" + "хранить сразу пару чисел в динамике".
Мы по-прежнему делаем бинпоиск, и отталкиваемся от динамики из предыдущего пункта. За один переход будет класть ровно одно бревно в текущий ряд. Тогда перехода всего два: положить бревно первого типа, положить бревно второго типа. Итого динамика следующая: $\langle$`l,z`$\rangle$`[a,b]`, где z – длина последнего ряда, l – количество рядов. $\mathcal{O}(n^2 \log \texttt{ANS})$.

1.3. Пересчёт "по слоям" в квадратичных динамиках

1.3.1. Формулировка задачи, наивное решение

Задача: даны n точек на прямой, выбрать какие-то $k : i_1, i_2, \dots, i_k$, минимизировать сумму по точкам взвешенных расстояний до ближайшей выбранной: $\sum_j \left[w_j \min_t |x_j - x_{p_t}|\right] \to \min$.

Сразу замечаем, что задача на самом деле равносильна разбиению n точек на k отрезков, а в каждом отрезке $[L, R]$ выборе точки $m[L, R]$, как мы делали в разд. 1.1.5.

Сейчас предположим, что у нас уже посчитаны массивы $m[L, R]$ – оптимальная точка и $cost[L, R]$ – сумма расстояний внутри отрезка $[L, R]$. В разд. 1.1.5 мы научились это делать за $\mathcal{O}(n^2)$. Как теперь разбить на отрезки? Решение "в лоб" даёт время $\mathcal{O}(n^2 k)$:

```
1 // f[i][j] - стоимость разбить точки [0,j) на i отрезков
2 f[0] = {0, INF, INF, ...};
3 for (int i = 1; i <= k; i++)
4   f[i][0] = 0; // можем бесплатно покрыть 0 точек
5   for (int j = 1; j <= n; j++)
6     f[i][j] = INF;
7     // p - левый конец последнего отрезка
8     for (int p = 0; p < j; p++)
9       relax(f[i][j], f[i-1][p] + cost[p][j-1])
```

Обозначим `p`, на котором достигается минимум в строках 7 и 8, за `p[i][j]`. Следующие оптимизации будут основаны на том, что можно быстро искать оптимальное `p[i][j]`.
Зная `p[i][j]`, можно посчитать `f[i][j] = f[i-1][p[i][j]] + cost[p[i][j]][j-1]`.

1.3.2. Оптимизация Кнута

Утверждение без док-ва: `p[i-1][j]` $\leqslant$ `p[i][j]` $\leqslant$ `p[i][j+1]`.

Оптимизация Кнута заключается в том, что `p`-шки мы вычисляем в таком порядке, что при

подсчёте `p[i][j]` уже посчитаны `p[i-1][j]` и `p[i][j+1]`: $i\uparrow$, $j\downarrow$.
Тогда `p[i][j]` можно перебирать не в $[0, j)$, а в $[$`p[i-1][j]`,`p[i][j+1]`$]$.

Докажем, что время работы полученного алгоритма $\mathcal{O}(n^2)$.
$\texttt{Time} = \sum_{i,j}\langle$время на подсчёт $\texttt{p[i][j]}\rangle = \sum_{i,j}\left[\texttt{p[i][j+1]} - \texttt{p[i-1][j]} + 1\right] =$
$nk + \sum_{i,j}\left[\texttt{p[i][j+1]} - \texttt{p[i-1][j]}\right] \leqslant (n+k)n$, так как не сократятся только $n + k$ слагаемых со знаком плюс и каждое из них не более n.

1.3.3. Разделяй и властвуй

Воспользуемся почти такой же посылкой, как и в предыдущей оптимизации.
Кстати, на самом деле посылка для Кнута следует из этой.
Утверждение без док-ва: `p[i][j]` $\leqslant$ `p[i][j+1]`.

Пусть уже посчитаны слой динамики `p[i-1]` и слой `f[i-1]`. Хотим за $\mathcal{O}(n \log n)$ насчитать все `p[i]`, будем это делать рекурсивной функцией, которая считает "в лоб" `p`-шку от средней точки и запускается от левой и правой половины, используя посчитанное значение, как ограничение:

```
// считаем p[i][l..r], предполагая, что все они должны лежать в [pl..pr]
void go( int l, int r, int pl, int pr ) {
  if (l > r) return; // пустой отрезок
  int m = (l + r) / 2, tmp;
  f[i][m] = INF;
  for (int p = pl; p <= min(pr, m - 1); p++)
    if (f[i][m] > (tmp = f[i-1][p] + cost[p][j-1]))
      f[i][m] = tmp, p[i][m] = p;
  go(l, m - 1, pl, p[i][m]);
  go(m + 1, r, p[i][m], pr);
}
```

Докажем, что время работы $\mathcal{O}(n \log n)$: глубина рекурсии $\log n$, на каждом уровне рекурсии отрезки $[\texttt{pl}_i..\texttt{pr}_i]$ обладают замечательным свойством $\texttt{pr}_i = \texttt{pl}_{i+1}$.

1.3.4. Сравнение

Сравним изученные оптимизации.
Насколько хорошо они справляются с задачей "по данным `cost[l,r]` посчитать `p[i,j]`"?

- Convex Hull Trick не помогает решить данную задачу
- Divide & Conquer даёт время $\mathcal{O}(nk \log n)$.
- Метод Кнута даёт время $\mathcal{O}(n^2)$.

Ясно, что при $k \leqslant \frac{n}{\log n}$ нужно использовать Divide & Conquer.

1.4. P.S.

Чтобы получить более подробные объяснения можно было сходить на лекцию вживую, можно пообщаться с присутствовавшими, а можно написать мне на почту `burunduk30@gmail.com`

Удачи!