Первый курс, осенний семестр 2018/19
Практика по алгоритмам #3

Структуры данных, стек

22 сентября

Собрано 22 сентября 2018 г. в 14:49

Содержание

1.1. Изучаем STL

Проведём эксперимет.
Подумаем про аллокацию памяти и её перегрузку.

a) Какой из циклов `for` будет работать быстро/долго?
b) Как ускорить?
c) У какой из структур есть `operator[]`?

1.2. Структуры данных, стек

1. **Очередь с минимумом, часть 2**

```
struct Item { int value, id; };
struct Qmin {
  int q_front, q_back; deque<Item> mins;
  void pop() { if (mins.front().id == q_front++) mins.pop_front(); }
  void push(int x) {
    while (!mins.empty() && mins.back().value >= x) mins.pop_back();
    mins.push_back({x, q_back++});
  }
  int get_min() { return mins.empty() ? INT_MAX : mins.front().value; }
};
```

2. **Амортизация**
   Придумайте потенциал и докажите амортизированное время работы.
   a) Очередь с минимумом с лекции работает за $\mathcal{O}(1)$.
   b) Дек, использующий утроение массива, работает за $\mathcal{O}(1)$.
   c) Вектор с уменьшением памяти в 2 раза при `pop_back` работает за $\mathcal{O}(1)$. Используйте идею с монетками.

3. **Двусвязный список.**
   Нужно уметь перебирать элементы в обоих направлениях,
   но хранить меньше, чем `struct Node { Node *l, *r; int x;}`.

4. **Частичные суммы**
   a) Много раз сделать `+=` на отрезке, в конце один раз вывести массив.
   b) Сперва много раз `+=` на отрезке, затем много раз запрос суммы на отрезке.

5. **Оптимальный подотрезок**
   За линейное время найти подотрезок массива.
   a) С максимальной суммой (два решения).
   b) С максимальной суммой, длины от L до R.
   c) С максимальной суммой, содержащий не менее k различных чисел.

6. **Задачи на стек**
   a) В массиве найти для каждого элемента ближайший $\leqslant$ слева за $\mathcal{O}(n)$.
   b) Дан массив. Найти отрезок $[l, r]$: $(\min_{i \in [l..r]} a_i) \cdot (r - l + 1) \to \max$.
   c) Дана матрица из нулей и единиц. Найти наибольший по площади подпрямоугольник, состоящий только из нулей за $\mathcal{O}(n^2)$.

1.3. Дополнительные задачи

1. **Дек и стеки**
   Придумайте дек с минимумом со всеми операциями за $\mathcal{O}(1)$. Докажите время работы.
2. **Частичные суммы**
   В каждой целой точке x числовой прямой есть $f[x]$, изначально равная нулю.
   Запросы сперва `+=(l,r)`, затем `sum(l,r)`. Координаты запросов целые от 0 до 10^{18}.
3. **Стек и ближайшие**
   a) За один проход со стеком найти ближайшие "$<$" справа и "$\leqslant$" слева.
   b) А теперь **меньший** справа и **меньший** слева.
4. **Правильные скобочные подстроки**
   Дана строка s из скобок, а также число k. Для всех i проверить, что подстрока $s[i..i+k]$ является правильной скобочной последовательностью. $\mathcal{O}(n)$.
   a) Скобки только одного типа.
   b) k типов скобок.
5. **Стек и 3D**
   Дана матрица из нулей и единиц в 3D. Найти наибольший белый подпараллелепипед.

Разбор задач практики

2.1. Изучаем STL

`vector`, `list`, `deque` – структуры данных.
`queue` и `stack` – обёртки над ними.
Дольше всего из трёх структур работает `list<int>`, т.к. на каждый `push` вызывается `operator new`. Раз в 10 дольше.
У `deque` операции `push` чуть быстрее чем у `vector`.
Чтобы ускорить `list`, нужно задать свой аллокатор. Самый простой способ – переопределить `operator new`.
Время работы `queue` и `stack` зависит от того, через что они реализованы. По умолчанию через `deque`.
У дека и вектора есть `operator[]`, у остальных нет.

2.2. Структуры данных, стек

1. **Очередь с минимумом, часть 2**
   `mins.front()` хранит минимум в очереди и его индекс m_1. После него в `mins` лежит минимум среди элементов очереди с индексами (m_1..`q_back`) и его индекс m_2, и так далее. Элеметы `mins` упорядочены по возрастанию и значений, и индексов.
   Делая `pop`, смотрим, не вынули ли индекс минимума. Если вынули, ничего, после него в `mins` новый минимум.
   Делая `push`, обновляем минимум во всё бóльшем хвосте очереди, пока новый элемент не больше последнего минимума.
   Время работы линейно, так как каждый элемет не более одного раза добавлен и удален из `mins`, других операциий нет.

   Потенциалом: $\varphi = |\texttt{mins}|$. $a(\texttt{push}) = t(\texttt{push}) + \Delta\varphi = \mathcal{O}(1)$, сколько вынули, на столько уменьшился φ.
   Заметим, что если явно хранить всю очередь `q`, то можно проверять не индексы, а сами значения. Но данная реализация использует в среднем $\mathcal{O}(\log n)$ памяти на случайных данных.
2. **Амортизация**
   a) **Очередь с минимумом.** $\varphi = |s_{\texttt{push}}|$.
   $a(\texttt{reverse}) = t(\texttt{reverse}) + \Delta\varphi = |s_{\texttt{push}}| + (0 - |s_{\texttt{push}}|) = 0 \Rightarrow a_i = \mathcal{O}(1)$.
   φ всегда неотрицателен.
   b) **Дек.** $\varphi = -\texttt{capacity}$.
   $a(\texttt{rebuild}) = t(\texttt{rebuild}) + \Delta\varphi = \texttt{capacity} + (\texttt{capacity} - 3\texttt{capacity}) \leqslant 0 \ \Rightarrow a_i = \mathcal{O}(1)$.
   $\varphi_0 - \varphi_n \leqslant 3n$.
   c) **Вектор.** Когда уменьшать память в `pop_back` в 2 раза? Когда элементов в 4 раза меньше.
   За `push_back` кладем 2 монетки: одну себе и одну в первую половину. Тогда хватит монет на копирование при увеличении.
   За `pop_back` кладем 1 монетку в первую четверть. Тогда хватит монет на копирование при уменьшении.

3. **Двусвязный список**
   Храним `sum = (size_t)l ^ (size_t)r` вместо `l`, `r`.
   Когда приходим в `v` из `prev`, имеем `next = (Node*)(v->sum ^ (size_t)prev)`.
   Новая структура умеет все, что обычный двусвязный список, кроме вставки и удаления из середины за $\mathcal{O}(1)$.

4. **Частичные суммы**
   a) **Много раз сделать += на отрезке, в конце один раз вывести массив.**
   Прибавление на отрезок – это два прибавления на суффикс. К элементу `a[i]` прибавляется все, что прибавлено к суффиксу, начинающемуся с `j < i`. Итого:
   Прибавление: `a[l] += x, a[r + 1] -= x`
   Восстановление: `for i: a[i] += a[i - 1]`
   b) **Сперва много раз += на отрезке, затем много раз запрос суммы на отрезке.**
   Вычисляем массив, как в предыдущем пункте. Считаем префиксные суммы. Ответ на запрос: `pref[r + 1] - pref[l]`.
   Можно просто два раза подряд вызвать стандартную функцию `partial_sum(a + 1, a + n + 1, a + 1)`, лежит в `<numeric>`.

5. **Оптимальный подотрезок**
   a) **С максимальной суммой.**
   Способ #1.
```
l = 0, sum = 0, answer = 0;
for (r = 0; r < n; r++) {
  if ((sum += a[r]) < 0)
    l = r + 1, sum = 0;
  answer = max(answer, sum);
}
```
   Способ #2.
```
l = 0, r = 0, sum = 0;
while (r <= n)
  sum += a[sum >= 0 ? r++ : l++], answer = max(answer, sum);
```
   Для каждого r ищем отрезок с максимальной суммой, оканчивающийся на r.
   Пусть для $r-1$ оптимален отрезок $[l_{r-1}, r-1]$. Тогда для r либо $l_r = l_{r-1}$, либо $l_r = r$.
   Будь для r оптимально другое начало, оно было бы оптимально и для $r-1$, сумма на этих отрезках отличается на $a[r]$, не зависящее от l.

   Способ #3. Найдем частичные суммы `pref[]`. Сумма на отрезке `[l, r]` равна `pref[r + 1] - pref[l]`, при фиксированном `r` нужно минимизировать `pref[l]`. Перебираем `r` в порядке возрастания. Поддерживаем минимум `pref[l]` по всем пройденным $\texttt{l} \leqslant \texttt{r}$.
   b) **С максимальной суммой, длины от L до R.**
   Теперь нас интересует минимум среди `pref[l]` на отрезке $[\texttt{r} - R, \texttt{r} - L]$. Поддерживаем очередь с минимумом, на каждом шаге один элемент оттуда уходит, и один приходит.
   c) **Содержащий не менее k различных чисел, числа небольшие.**
   Теперь нас интересует минимум среди `pref[l]` на отрезке `[0, s-1]`, где `s` – последняя позиция, что `[s, r]` содержит хотя бы k различных чисел.
   При увеличении `r` позиция `s` только растет, ее можно продолжать двигать со значения с прошлой итерации, тогда суммарно по всем итерациям выйдет $\mathcal{O}(n)$.
   Как проверять, можно ли подвинуть `s` вперед? Поддерживаем массив-счетчик каждого числа.

После `r++` делаем `count[a[r]]++`, перед `s++` делаем `count[a[s]]--`.
Также храним число ненулевых ячеек `count`, меняя его каждый раз, когда только что измененная ячейка начала или перестала быть нулем.
С использованием хеш-таблицы вместо массива можно делать это и для больших чисел.

6. **Задачи на стек**
   a) **Для каждого элемента ближайший меньший слева за $\mathcal{O}(n)$.**
   ```
   for (int i = 0; i < n; ++i) {
     while (!stack.empty() && a[i] <= a[stack.top()]) stack.pop();
     if (!stack.empty()) left[i] = stack.top();
     stack.push(i);
   }
   ```
   В стеке и `left` хранятся индексы элементов.
   Инвариант: для каждого элемента стека перед ним в стеке идет ближайший к нему слева меньший. Когда видим элемент `a[i]`, на вершине стека лежит `a[i - 1]`. Либо он сам меньше `a[i]`, либо нам нужен элемент `a[j] < a[i] <= a[i - 1]` левее, начинаем поиск с ближайшего меньшего к `a[i - 1]`, он как раз дальше по стеку. И так далее.
   b) $(\min_{i\in[l..r]} a_i)\cdot(r-l+1)\to\max$
   Переберем позицию минимума. Пусть a_i будет минимумом на отрезке. Выгодно расширить его как можно сильнее влево и право, то есть до ближайших $< a_i$ слева и справа.
   c) **Найти наибольший подпрямоугольник из нулей за $\mathcal{O}(n^2)$.**
   Найдем для каждой клетки длину столбика $h[i][j]$ только из нулей, идущего вверх из этой клетки (0, если $a[i][j] == 1$, иначе $h[i-1][j]+1$).
   Прямоугольник с нижней границей i, левой l и правой r имеет верхнюю границу не более $\min h[i][l..r]$. Переберем нижнюю границу и минимальный столбик прямоугольника $h[i][j]$. Выгодно расширить по максимуму влево и вправо, то есть до ближайших слева и справа элементов $< h[i][j]$.

2.3. Дополнительные задачи

1. **Дек и стеки**
   Дек через два стека. Если один из стеков кончился, то просто разделим второй пополам на два стека.
   После разделения стека размера $2k$ нам нужно минимум k операций `pop`, чтобы снова нужно было делать разделение. Если класть по монетке за каждый `pop` и `push`, то их хватит на разделение не опустевшего стека.
   Еще можно ввести потенциал: модуль разности размеров стеков.

2. **Частичные суммы, координаты запросов до 10^{18}**
   Отсортируем концы l и $r+1$ всех запросов по координате. Координаты концов запросов l и $r+1$ заменим на их позиции в отсортированном порядке. Этот прием называется «сжатие координат». Операции `+=` и восстановление массива делаются так же.
   Надо учесть, что между элементами `i` и `i + 1` на самом деле целый отрезок `[x[i], x[i + 1])`, у всех его элементов значение равно `value[i]`.
   Т.е. `pref[i + 1] = pref[i] + (x[i + 1] - x[i]) * value[i]` $\Rightarrow$ `pref[i] =` $\sum_{z\in[0,x[i])}$ `f[z]`.
   Заметим, что можно отсортировать все запросы и `+=`, и суммы, тогда мы сразу будем знать сжатые координаты запросов суммы. А можно сжать координаты только для `+=`, тогда нужно будет искать бинпоиском позиции концов запросов суммы.

3. **Стек и ближайшие**

```
for (int i = 0; i < n; ++i) {
  while (!stack.empty() && a[i] < a[stack.top()]) right[stack.pop()] = i;
  if (!stack.empty()) left[i] = stack.top();
  stack.push(i);
}
```

Найдем ближайший меньший слева. Либо ближайший $\leqslant$ уже меньше, либо нас интересует уже посчитанный ближайший $<$ к нему.

4. **Правильные скобочные подстроки**

a) **Один тип скобок.**

Считаем баланс скобок (число открывающих минус число закрывающих) на каждом префиксе. Подстрока – правильная, если в ней поровну открывающих и закрывающих скобок, и у нее нет префикса, на котором закрывающих больше, чем открывающих. То есть $(i..i{+}k]$ – хороший, если:

- `balance[`i`] == balance[`$i{+}k$`]`,
- $\forall j \in (i..i{+}k]$ `balance[`j`] >= balance[`i`]`. То есть, минимиальный баланс на отрезке $(i..i{+}k]$ не меньше `balance[`i`]` (точнее, равен).

Поддерживаем значения баланса на отрезке $(i..i{+}k]$ в очереди с минимумом и проверяем два указанных выше условия.

b) k **типов скобок.**

Запустим алгоритм проверки скобочной последовательности на правильность. Если для какой-то скобки нет парной, запомним это. Так же для каждой парной запомним позицию пары. Ответ на запрос $[l, r]$: проверить, что все скобки на $[l, r]$ имеют пару в $[l, r]$. То есть максимальный номер пары скобки из этого отрезка $\leqslant r$, минимальный $\geqslant l$.

5. **Стек и 3D**

:)