Werbung (ADS)

Aufgabe Gegeben ein $m \times n$ Grid an Zahlen a, finde die maximale Summe in einem Ball mit Radius k, in Manhattan-Metrik.

Allgemein kann man hier jede mögliche Position durchprobieren (in $\mathcal{O}(n \cdot m)$). Die Frage ist wie man effizient die Summe an einer gegebenen Position berechnet:

Subtask 1 Man kann für jedes Feld naive prüfen, ob es von der aktuellen Position sichtbar ist:

```
for (int i = 0; i < m; i++) // Jede Position (i, j)
   for (int j = 0; j < n; j++) {
      int score = 0;
      for (int b = 0; b < m; b++)  // Jedes Feld (b, c)
         for (int c = 0; c < n; c++)
            if (abs(i-b) + abs(j-c) < k)
               score += a[b][c];
      ans = max(ans, score);
   }
```

So eine Lösung hat Laufzeit $\mathcal{O}((n \cdot m)^2)$ und genügt nur für Subtask 1. Man kann die Grenzen der inneren Schleifen anpassen, sodass nur zwischen $i-k$ und $i+k$ bzw. $j-k$ und $j+k$ iteriert wird. Bei hinreichend großem k bringt das allerdings nichts, sodass die Lösung trotzdem zu langsam ist.

Subtask 2 Man kann erkennen, dass ein Ball in Manhatten-Metrik einem um 45 Grad gedrehtem Quadrat entspricht. Für jeden y Abstand von der aktuellen Position, gibt es also genau ein Intervall an x Positionen, die enthalten sind. Wie im Tutorial bei der Registrierung erklärt, lassen sich Interallsummen effizient mit der Präfixsumme berechnen. Man kann also eine Präfixsumme pro Zeile vorberechnen und dann den Prozess in $\mathcal{O}(n \cdot m^2)$ simulieren (bzw. $\mathcal{O}(n^2 \cdot m)$, was keinen Unterschied macht, da beide gleich groß sein können).

Subtask 3 Es gibt mehrere Möglichkeiten die Berechnung auf $\mathcal{O}(n \cdot m)$ zu optimieren. Im Endeffekt möchte man eine 2D-Präfixsumme, also eine Präfixsumme von Präfixsummen. Da die x Intervalle für fixes y variieren, ist das abfragen nicht ganz einfach. Deshalb kann man stattdessen Präfixsummen über die Diagonalen aufbauen. Alternativ kann man das Array um 45 Grad drehen und mit $\sqrt{2}$ skalieren, z.B. durch die Transformation $(i, j) \mapsto (i+j, i-j)$ und einem entsprechendem Shift. Tendenziell ist es für die Implementierung hilfreich, wenn man das gegebene Array mit hinreichend vielen nullen padded. Dann müssen keine Spezialfälle betrachtet werden, falls das Quadrat nicht ganz im Array enthalten ist.

Die Bergsteiger (BERG)

Aufgabe Gegeben ein Array a aus n Zahlen, zähle die Anzahl an Paaren $(i, j), i \neq j$, sodass $a[i] = a[j]$ und kein Wert im Intervall $[i, j]$ (bzw. $[j, i]$) größer als $a[i]$ ist.

Wir zählen Paare mit $i < j$, das eigentlich Ergebnis ist das doppelte davon. Zudem kann die Eigenschaft eines Paares als $max(a[i...j]) = a[i] = a[j]$ umformuliert werden.

Subtask 1 Hier kann jedes Paar fixiert werden ($\mathcal{O}(n^2)$ Möglichkeiten) und für jedes in $\mathcal{O}(n)$ geprüft werden, ob ein größerer Wert dazwischen vorkommt. Insgesamt $\mathcal{O}(n^3)$.

Subtsak 2 Anstatt in jedem Intervall das Maximum neu zu berechnen, fixieren wir den linken Startpunkt. Das rechte Ende wird schrittweise erhöht und abgebrochen, sobald man was zu großes sieht:

```
for(int i = 0; i < n; i++)
   for(int j = i+1; j < n; j++)
      if (a[i] < a[j])
         break;
      else if (a[i] == a[j])
         ans++;
```

Subtask 3 Wir können es uns nicht mehr leisten durch alle Paare zu iterieren. Gruppieren wir die Berge nach Höhe, sodass wir z.B. wissen, dass Höhe h an Positionen $p_1 < p_2 < \ldots < p_l$ vorkommt. Wenn wir den linken Startpunkt als p_i fixieren, dann gibt es ein k_i, sodass alle Paare (p_i, p_j) mit $j \leq k_i$ funktionieren, aber (p_i, p_j) mit $j > k_i$ nicht mehr. Die Antwort ist die Summe dieser $k_i - i$, für jeden linken Startpunkt. Um bei fixem linken Startpunkt dieses k_i zu berechnen, kann man in der entsprechenden Liste an Bergen mit selber Höhe binär suchen. Mit einer Datenstruktur wie einem Segmenttree oder einer Sparsetable kann man dabei effizient das Maximum im Intervall (RMQ) berechnen. Das hat eine Laufzeit von $\mathcal{O}(n \cdot \log^2 n)$ bzw. $\mathcal{O}(n \cdot \log n)$. Alternativ kann man erkennen, dass für dieselbe Höhe h mit aufsteigendem Index i das entsprechende k_i nur größere wird. So kann man die binäre suche sparen und die k_i jeweils solange erhöhen wie möglich. Dieses erhöhen ist amortisiert konstant, sodass die Gesamtlaufzeit (wegen dem RMQ) $\mathcal{O}(n \cdot \log n)$ ist. Mit einer linearen RMQ Datenstruktur wäre es sogar linear, durch den großen konstanten Faktor bei unseren Experimenten aber zu langsam.

Subtask 4 Ähnlich wie bei Subtask 3 erkennt man, dass es sich immer um Intervalle derselben Höhe handelt. Anstatt jede Höhe für sich zu behandeln, speichern wir sie kombiniert. Wir bearbeiten die Höhen von links nach rechts. Wenn wir eine Höhe h sehen, heißt das, dass alle bisherigen Höhen $h' < h$ nicht mehr relevant sind. Wenn wir für jede Höhe einen Zähler speicheren, müssen wir alle jene Höhen zurücksetzten. Der Zähle für unsere aktuelle Höhe gibt dann an, wie viele Indizes wir davor noch "sehen". Danach kann unser eigener Zähler erhöht werden, um den aktuellen Index mitzuzählen. Das ist allerdings quadratisch, wenn wir alle Höhen davor beim Zurücksetzten durchgehen. Wir halten wir auf einem Stack die vorherigen Höhen in absteigender Reihenfolge aufrechterhalten, die noch sichtbar sind. Davon wird immer Suffix entfernen, der zu klein geworden ist. Das sind nur amortisiert Konstant viele und liefert eine Laufzeit von $\mathcal{O}(n)$ mit gutem konstanten Faktor.

Zentralmatura (MATURA)

Aufgabe Platziere n Lehrer auf einem $n \times n$ Grid, sodass jeder Lehrer in seinem bevorzugtem Rechteck steht und sich in jeder Zeile und Spalte genau ein Lehrer befindet.

Subtask 1 Jede Art von Bruteforce, z.B. rekursives Backtracking wie beim 8 Damen Problem, genügen hier.

Subtask 2 Für jeden Lehrer steht bereits die Zeile fest. Das Problem reduziert sich also darauf, dass wir für jeden Lehrer in Intervall gegeben haben. Daraus sollen wir jeweils einen Punkt auswählen, sodass jede Position genau zu einem Lehrer gehört. Die Standardmethode dafür ist bipartites Graph Matching, wobei z.B. links die Lehrer und rechts die Positionen sind. Das könnte für diesen Subtask noch schnell genug sein, ist aber nicht die angedachte Lösung. Da jeder Lehrer wir für jeden Lehrer ein Intervall und keine beliebige Menge an Positionen gegeben haben, funktioniert folgender Greedy-Algorithmus:
Wir bearbeiten die Positionen von links nach rechts. Von allen Lehrer, die mit Position 1 zufrieden währen, welchen sollen wir auswählen? Da die Lehrer sich in diesem Fall nur durch ihr rechtes Intervallende unterscheiden, sollten wir jenen mit minimalem Endindex nehmen. Denn dieser Lehrer hat im Vergleich zu den anderen am wenigsten Optionen. Allgemein kann man an jeder Positionen von allen noch nicht verwendeten Lehrern den auswählen, der mit der aktuellen Positionen zufrieden ist und dessen Intervall am frühsten wieder aufhört. Falls es keinen Lehrer gibt, gibt es keine Lösung. Wenn man das naiv implementiert und an jeder Position alle anderen Lehrer durchgeht, erhält man eine Laufzeit von $\mathcal{O}(n^2)$.

Subtask 3 Hier dürfen wir an jeder Position nicht mehr alle Lehrer durchgehen. Wir müssen eine Menge verwalten, die alle nicht verwendeten Lehrer enthält, die mit der aktuellen Position zufrieden währen. Unter diesen möchten wir den Lehrer mit minimalem rechten Index abfragen. Dafür bietet sich ein Heap an. Der kann in $\mathcal{O}(\log n)$ das Minimum einer Menge finden und entfernen, sowie in $\mathcal{O}(\log n)$ ein Element einfügen.
An jedem Index kommen einige Lehrer hinzu, die beginnend ab der aktuellen Position zufrieden währen. Diese findet man effizient, wenn man die Lehrer am Anfang nach Startindex gruppiert. Falls die Datenstruktur danach leer ist, gibt es keine Lösung. Anderenfalls können wir den Lehrer mit kleinstem Endindex effizient abfragen und entfernen. Ist dessen Endindex genau die aktuelle Position, muss zusätzlich geprüft werden, ob es andere Lehrer mit diesem Endindex gibt. Denn dann gibt es genauso keine Lösung, da wir diesen Lehrer keine Position zugewiesen haben. Das kann geprüft werden, indem man sich das nächste Element im Heap anschaut. Insgesamt ist die Laufzeit $\mathcal{O}(n \log n)$.
Tatsächlich lässt sich diese Berechnung auf ähnliche weiße mit einer DSU optimieren und in $\mathcal{O}(n\alpha(n))$ lösen. Uns ist allerdings keine lineare Lösung bekannt.

Subtask 4 Jetzt ist uns die Zeile nicht mehr bekannt. Für jeden Lehrer kennen wir a_i, b_i, c_i und d_i (wie in der Angabe) und möchte seine Position p_i, q_i bestimmen, sodass er im Rechteck steht. Formal heißt das, dass $a_i \leq p_i \leq c_i$ und $b_i \leq q_i \leq d_i$ gelten muss. Anders ausgedrückt sind also p_i und q_i unabhängig. Wir können also einfach zweimal das eindimensionale Problem aus Subtask 3 lösen, einmal für die Zeilen und einmal für die Spalten.

Sitzdesign (MVV)

Aufgabe Gegeben einen String aus den Zeichen R, G und B der Länge n. Betrachte wiederholt benachbarte Zeichen und erstelle daraus einen eins kürzeren String. Falls zwei benachbarte Zeichen dieselben sind, ist das neue Zeichen dasselbe. Anderenfalls ist es das Fehlende.

Subtask 1 Es genügt die Vorschrift aus der Angabe zu simulieren. Das hat eine Laufzeit von $\mathcal{O}(n^2)$. Wenn der String in `s` gespeichert ist, ist nach Ausführen dieses Codes das Ergebnis `s[0]`.

```
for (n--; n; n--)
   for (int i = 0; i < n; i++)
      if (s[i] != s[i+1])
         s[i] = 'R'+'G'+'B'-s[i]-s[i+1];
```

Subtask 2 Man kann Ergebnisse für kleinere Dreiecke vorberechnen. Genauer gibt es für fixes k genau 3^k Farbstrings der Länge k. Für diese kann man naive in $\mathcal{O}(3^k \cdot k^2)$ die finale Farbe vorberechnen. Damit kann man die eigentliche Simulation auf $\mathcal{O}(\frac{n^2}{k})$ beschleunigen, da man jeweils k Zeilen überspringen kann. Wenn man $k = 13$ setzt sind diese beiden Terme etwa gleich und die Laufzeit ist gut genug für Subtask 2. Allgemein kann man hier k z.B. auf $\log n$ setzten und erhält eine Laufzeit von $\mathcal{O}(n \cdot \log^2(n) + \frac{n^2}{\log n}) = \mathcal{O}(\frac{n^2}{\log n})$.

Subtask 3 Substituiere $R = 0$, $G = 1$ und $B = 2$. Wir wollen uns eine Formel zum Kombinieren von zwei Farbei überlegen. Man kann erkennen, dass die Summe aller drei Farben beim Kombinieren immer ein vielfaches von 3 ist. In der Tat, wenn man zweimal dieselbe Farbe kombiniert, ist das Ergebnis dieselbe Farbe, also 3 mal der ursprüngliche Wert. Anderenfalls ist die Summe $0 + 1 + 2 = 3$. Anders gesagt gilt, dass $a + b + c \equiv 0 \mod 3$, wobei a und b die ursprünglichen Farben sind und c den Wert der neuen Farbe angibt. D.h. $c \equiv -(a + b) \mod 3$. Weil $-1 \equiv 2 \mod 3$ gilt also $c \equiv 2 \cdot (a + b) \mod 3$. Unsere Transformation ist also linear. Dadurch können wir den Faktor von 2 vorerst ignorieren und das Ergebnis dann mit 2^{n-1} multiplizieren (also im Endeffekt, falls n gerade, ist mit 2 multiplizieren). Dann hängt jeder Wert nur noch der Summe der beiden Werte darüber ab. Es ist leicht zu sehen, dass jeder Wert in der letzten Zelle so oft gezählt wird, wie es Wegen nach unten zur Spitze des Dreiecks gibt. Nachdem dieses Dreieck praktisch dem Pascalschem Dreieck entspricht, sind das die Binomialkoeffizienten. Insgesamt gilt also, dass $2^{n-1} \cdot \sum_{k=0}^{n-1} \binom{n-1}{k} \cdot a_k \mod 3$ der Wert der finalen Farbe ist. a_k gibt hier den ursprünglichen Farbstring, als 0, 1 oder 2, an.
Zum Berechnen der Binomialkoeffizienten gibt es mehrere Möglichkeiten. Man kann Lucas's Theorem anwenden und die Basis 3 Darstellung von n und k betrachten. Das hat $\mathcal{O}(n \cdot \log n)$ Laufzeit. Genauso kann man die Fakultäten vorberechnen. Damit man das Ergebnis mod 3 bekommt und keine großen Zahlen benötigt, muss man dabei die 3er Faktoren extra behandeln. Das lässt sich in $\mathcal{O}(n)$ implementieren.
Alternativ kommt man auch ohne Binomialkoeffizienten aus. Als Korollar der Formel erkennt man, dass wenn n eine Dreierpotenz ist, das Ergebnis nur die Summe der äußeren beiden Werte ist. Deshalb kann man jeweils die größte Dreierpotenz betrachten, die maximal n groß ist und damit so viele Zeilen in der Simulation überspringen. Wegen der geometrischen Reihe hat das eine Laufzeit von $\mathcal{O}(n)$.