Notes on Introduction to Algorithms

comaecliptic1997

March 2019

1 Goals

$$\begin{aligned} \text{Heap} &\Rightarrow \text{Amortized Analysis (exercises left to do)} \\ &\Rightarrow \text{Dynamic Table} \\ &\Rightarrow \text{Disjoint Sets} \\ &\Rightarrow \text{Hash Tables} \\ &\Rightarrow \text{Sort and Order Statistics} \\ &\Rightarrow \text{Fibonacci Heaps} \end{aligned}$$

2 Basic

2.1 Formula

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n e^{\alpha_n},\ \frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$$

$$\lceil \log n \rceil ! = \Theta(n^{\log \log n})$$

3 Probabilistic Analysis

3.1 Indicator random variable

We call $I\{A\}$ an indicator random variable. Important property: $E[I\{A\}] = P(A)$. For example: the expected number of inversion in a random permuted array? Let $X_{i,j}$ be the indicator random variable indicating that (i, j) is an inversion pair or not, then the answer is

$$E[X] = E[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{i,j}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Pr(X_{i,j} = 1) = \sum \frac{1}{2} = \frac{n(n-1)}{4}$$

3.2 Randomly permuted array

Associate randomly a priority ranging from 1 to n^3, then sort. The chance that the associated priorities are unique is

$$\begin{aligned} \frac{\binom{n^3}{n} n!}{(n^3)^n} &= (1 - \frac{1}{n^3})(1 - \frac{2}{n^3})...(1 - \frac{n-1}{n^3}) \\ &\geq (1 - \frac{n-1}{n^3})^n \\ &\geq 1 - \frac{1}{n} \end{aligned}$$

To show that the array generated is indeed in random order, we show that a given permutation occurs with probability exactly $\frac{1}{n}$. First, consider a permutation that $A[i]$ receives the i-th smallest priority. Let E_i be the event that $A[i]$ receives the i-th smallest priority, then

$$\begin{aligned} Pr(E_1E_2...E_n) &= Pr(E_1)Pr(E_2|E_1)...Pr(E_n|E_1E_2...E_{n-1}) \\ &= \frac{1}{n}\frac{1}{n-1}...\frac{1}{2}\frac{1}{1} \\ &= \frac{1}{n!} \end{aligned}$$

A similar argument can be easily extended to handle any permutation of priorities.

A better method: from head to tail, swap the current element with a randomly chosen one behind it. Loop invariant: prior to the i-th iteration, the chance that a specific $i-1$ permutation occurs in the previous subarray is $\frac{1}{\binom{n}{i-1}(i-1)!}$

For the random sample problem, namely choosing randomly an m elements set out of an n elements set. We can simply use the above procedure and take the first m elements. In this way, however, we make too many calls to the $RANDOM$ procedure. A better way relies on a recursive procedure $SAMPLE(n, m)$ which returns a sample set out of the given array $A[1..n]$. In each call, it firstly recursively calls itself $S = SAMPLE(n-1, m-1)$ to get an $m-1$ elements set. Then it chooses randomly from 1 to n a number i. If $i \in S$, then return $S \cup \{n\}$, else return $S \cup \{i\}$. Analysis: for a given m elements set S, if $n \notin S$, for a specific element $j \in S$, let E_j be the event that j is randomly chosen in the procedure, let F_j be the event that the recursive call returns $S - \{j\}$. Then

$$Pr(E_jF_j) = Pr(F_j)Pr(E_j|F_j) = \frac{1}{\binom{n-1}{m-1}} \times \frac{1}{n}$$

There's m such combinations, therefore the probability is

$$\frac{1}{\binom{n-1}{m-1}} \times \frac{1}{n} \times m = \frac{1}{\binom{n}{m}}$$

If $n \in S$, the only possibility is that the recursive call returns $S - \{n\}$, and either n is randomly chosen, or some element in $S - \{n\}$ is chosen. Therefore the probability is

$$\frac{1}{\binom{n-1}{m-1}} \times (\frac{1}{n} + \frac{m-1}{n}) = \frac{1}{\binom{n}{m}}$$

3.3 Further probability analysis

3.3.1 Trick in designing event

In the birthday paradox, let A_i be the event that the i-th person has a different birthday from others, then $B_n = \cap_{i=1}^{n} A_i$ is the desired event, with

$$Pr(B_k) = Pr(B_{k-1})Pr(A_k|B_{k-1})$$

3.3.2 Balls and bins model

How many balls must we toss until every bin contains at least one balls? Let's call ths toss of a ball into an empty bin a hit. We then divide the n tosses into stages. The i-th stage consists of the tosses after the $(i-1)$-th toss until the i-th toss. In the i-th stage, there's $b-i+1$ empty bins, therefore the probability of obtaining a hit is $\frac{b-i+1}{b}$. Let n_i denote the number of tosses in the i-th stage, then this random variable has

a geometric distribution and hence $E[n_i] = \frac{b}{b-i+1}$. Therefore the expected amount of tosses is

$$\begin{aligned} E[n] &= E[\sum_{i=1}^{b} n_i] \\ &= \sum_{i=1}^{b} E[n_i] \\ &= \sum_{i=1}^{b} \frac{b}{b-i+1} \\ &= \sum_{i=1}^{b} b\frac{1}{i} \\ &= b(\ln b + O(1)) \end{aligned}$$

What is the expected amount of empty bins (out of n bins) after n tosses? We can use indicator to solve this problem. Let Y_i be the indicator random variable indicating whether the i-th bin is empty, then the answer is $E[X] = E[\sum_{i=1}^{n} Y] = n(1-\frac{1}{n})^n \approx \frac{n}{e}$

3.3.3 Streak

Suppose you flip a fair coin n times, what is the longest streak of consecutive heads that you expect to see? The answer is $\Theta(\ln n)$.
Firstly we prove the expected length is $O(\log n)$. Let A_{ik} be the event that a streak of heads of length at least k begins with the i-th coin flip, then $Pr(A_{ik}) = \frac{1}{2^k}$. For $k = 2\lceil \log n \rceil$,

$$\begin{aligned} Pr(A_{2,2\lceil \log n \rceil}) &= \frac{1}{2^{2\lceil \log n \rceil}} \\ &\leq \frac{1}{2^{2\log n}} \\ &= \frac{1}{n^2} \end{aligned}$$

Therefore the probability that a streak of heads of length at least $2\lceil \log n \rceil$ begins at anywhere is

$$\begin{aligned} Pr(\cup_{i=1}^{n-2\lceil \log n \rceil+1} A_{i,2\lceil \log n \rceil}) &\leq \sum_{i=1}^{n-2\lceil \log n \rceil+1} \frac{1}{n^2} \\ &\leq \sum_{i=1}^{n} \frac{1}{n^2} \\ &= \frac{1}{n} \end{aligned}$$

Now let L_j denote the event that the longest streak of heads has length exactly j, then (note that L_j are exclusive)

$$\begin{aligned}
E[L] &= \sum_{j=0}^{n} jPr(L_j) \\
&= \sum_{j=0}^{2\lceil \log n \rceil - 1} jPr(L_j) + \sum_{j=2\lceil \log n \rceil}^{n} jPr(L_j) \\
&< (2\lceil \log n \rceil) \sum_{j=0}^{2\lceil \log n \rceil - 1} Pr(L_j) + n \sum_{j=2\lceil \log n \rceil}^{n} Pr(L_j) \\
&\leq 2\lceil \log n \rceil + n \cdot \frac{1}{n} \\
&= O(\log n)
\end{aligned}$$

Next we prove that the expected length is $\Omega(\log n)$. To prove this bound, we look for streaks of length s by partitioning the n flips into approximately $\frac{n}{s}$ groups of s flips each. If we choose $s = \lfloor \frac{\log n}{2} \rfloor$, we can show that it is likely that at least one of these groups comes up all heads. We now partition the n coin flips into at least $\lfloor \frac{n}{\lfloor \frac{\log n}{2} \rfloor} \rfloor$ groups of $\lfloor \frac{\log n}{2} \rfloor$ consecutive flips, and we bound the probability that no group comes up all heads. From the previous analysis we have

$$\begin{aligned}
Pr(A_{i,\lfloor \frac{\log n}{2} \rfloor}) &= \frac{1}{2^{\lfloor \frac{\log n}{2} \rfloor}} \\
&\geq \frac{1}{\sqrt{n}}
\end{aligned}$$

Therefore the probability that every one of these groups fails to bbe a streak of length $\lfloor \frac{\log n}{2} \rfloor$ is at most

$$\begin{aligned}
(1 - \frac{1}{\sqrt{n}})^{\lfloor \frac{n}{\lfloor \frac{\log n}{2} \rfloor} \rfloor} &\leq (1 - \frac{1}{\sqrt{n}})^{\frac{2n}{\log n} - 1} \\
&\leq e^{-(2n/\log n - 1)/\sqrt{n}} \\
&= O(e^{-\log n}) \\
&= O(\frac{1}{n})
\end{aligned}$$

Here we use the fact that $1 + x \leq e^x$. Therefore

$$\begin{aligned}
E[L] &= \sum_{j=0}^{n} jPr(L_j) \\
&= \sum_{j=0}^{\lfloor \frac{\log n}{2} \rfloor} jPr(L_j) + \sum_{j=\lfloor \frac{\log n}{2} \rfloor + 1}^{n} jPr(L_j) \\
&\geq \sum_{j=\lfloor \frac{\log n}{2} \rfloor + 1}^{n} jPr(L_j) \\
&\geq \lfloor \frac{\log n}{2} \rfloor \sum_{j=\lfloor \frac{\log n}{2} \rfloor + 1}^{n} Pr(L_j) \\
&\geq \lfloor \frac{\log n}{2} \rfloor (1 - O(\frac{1}{n})) \\
&= \Omega(\log n)
\end{aligned}$$

How can we estimate that it is $\Theta(\log n)$? We can use indicator random variable to calculate the expectations of a n flip with at least k consecutive heads. Let $X_{i,j}$ denote the indicator variable of $A_{i,k}$, Then

$$\begin{aligned} E[X] &= E[\sum_{i=1}^{n-k+1} X_{i,k}] \\ &= \sum i = 1^{n-k+1} Pr(A_{i,k}) \\ &= \sum i = 1^{n-k+1} \frac{1}{2^k} \\ &= \frac{n-k+1}{2^k} \end{aligned}$$

By letting $k = c \log n$ where c is a positive number, we have $E[X] = \Theta(\frac{1}{n^{c-1}})$. So if $c < 1$, there're many streaks in a n-flip.

4 Amortized Analysis

In an **amortized analysis**, we average the time required to perform a sequence of data-structure operations over all the operations performed. There are three most common techniques, namely **aggregate analysis**, in which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations, the average cost per operation is then $\frac{T(n)}{n}$. **Accounting method**, in which we determine an amortized cost of each operation. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit". Later in the sequence, the credit pays for operations that are charged less than they actually cost. **Potential method**, which is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole. Two examples to be discussed: One is a stack with the additional operation MULTIPOP, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation INCREMENT.

4.1 Aggregate analysis

For the stack operations, in a sequence of n POP, PUSH, MULTIPOP operations, we have a worst case $O(n^2)$ running time. Using aggregate analysis, we observe that the number of POPs equals to the number of PUSHs, which equals to at most n, therefore the entire cost is $O(n)$, rendering each operation costs averagely $O(1)$.

For the binary counter increments, in a sequence of n INCREMENT operations, the worst case runnning time is $O(nk)$, where k is the number of bits (flip all bits). Using an aggregate analysis, we notice that $A[0]$ flips every time, $A[1]$ flips every two INCREMENT operations, therefore totally $\lfloor \frac{n}{2} \rfloor$ times, and we conclude that $A[i]$ flips totally $\lfloor \frac{n}{2^i} \rfloor$ times. Therefore the running time is $\sum_i \lfloor \frac{n}{2^i} \rfloor \leq \sum_{i \geq 0} \frac{n}{2^i} = 2n$.

4.2 The accounting method

For each operation we assign an amortized cost. When an operations'a amortized cost exceeds its actual cost, we assgin the difference to specific objects in the data structure as credit. Credit can help pay for later operations whose amortized cost is less than their actual cost. If we denote the actual cost of the ith operation by c_i and the amortized cost by $\hat{c}_i$ then we require

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

For all sequences of n operations. That is to say, the total credit associated with the data structure must be nonnegative at all times.

In the satck operatiosns example, we assign assign the following amortized costs:

PUSH	2
POP	0
MULTIPOP	0

Every time we push an element into the stack, the amortized cost charges one more as a credit. When we pop an element, we take away the prepaid credit.
In the incrementing a binary counter example, we charge an amortized cost of 2 for setting a bit to 1. It cost us one actual cost to set the bit, and one credit for later use. At any point in time, every 1 in the counter has a credit on it, thus we can charge nothing to reset a bit to 0. The INCREMENT operation sets at most one bit, therefore for n such operations, the total amortized cost is $O(n)$. Therefore average cost is $O(1)$.
Another example: we perform a sequence of stack operations on a stack whose size never exceeds k. Meanwhile after every k operations, we make a copy of the entire stack for backup purposes. We assign the amortized cost for each PUSH operation 2 units. For other operations we assign 0 unit. In this way, at any time, every element in the stack has one credit, therefore we can charge nothing while popping or copying. Actually this is a solution to an arbitrary stack with usual operations upon it, if, when copying, we only take those elements in the stack into account. If we need to copy the entire k slots stack, then we need to assign 2 units for the POP operation.
One final example, suppose we wish not only to increment a counter but also to reset it to zero. Counting the time to examine or modify a bit as $\Theta(1)$, for a sequence of n INCREMENT and RESET operations, the running time is $O(n)$. (Note that we keep a pointer to the high-order 1, every operation will modify the pointer for at most one time, therefore the time for manipulating the pointer is $O(n)$). We assign 3 units for setting a bit to 1, 0 unit for setting a bit to 0 (therefore 3 units for the INCREMENT operation). Additionally, we assign 0 unit for the RESET operation. Whenever we try to reset, each lower 0 bit has at least 1 credits, which compensates for the examine operation, each lower 1 bit has at least 2 credits, which compensates for the examine and then set operation. Therefore we're free to charge nothing while resetting.

4.3 Potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the potential method represents the prepaid work as "potential energy", which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure. The potential method works as follows. We will perform n operations, starting with an initial data structure D_0. For each $i = 1, 2, ..., n$, we let c_i be the actual cost of the ith operation and D_i be the data structure taht results after applying the ith operation to data strcture D_{i-1}. A potential function Φ maps each data structure D_i to a real number, which is the potential associated with this data structure. The amortized cost $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

. The amortized cost is therefore

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

Therefore it is required that we define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$. In practice, we often require that $\Phi(D_i) \geq \Phi(D_0)$ for all i. *Overcharges increase potential, while undercharges release potential.*
For the stack operations example, we define the potential function as the number of objects in the stack. Therefore $\Phi(D_0) = 0$ since initially there's no object in the stack. And since the number of objects stays non-negtive, we have $\Phi(D_i) \geq 0$. Now the amortized cost of each operation can be computed. PUSH will cost 2 units, POP will cost 0 unit, MULTIPOP will cost $k - k = 0$ unit. Therefore the amortized cost for each operation is $O(1)$.
In the incrementing a binary counter case, we define the potential function to be the number of 1's. Then $\Phi(D_0) = 0$, and obviously $\Phi(D_i) \geq 0$. For each INCREMENT operation, if there's no overflow, and we reset t bits, then the cost is $(1 + b) - (b - 1) = 2$. If there's an overflow, then the cost is $b - b = 0$. Therefore the amortized cost is $O(1)$.

Exercise 17.3-3. (Hint, amortize the cost of extracting to inserting). The potential function is defined as $\Phi(\mathcal{H}) = \sum_{i=1}^{k} \log k$, where k is the number of element in the heap $\mathcal{H}$.

Exercise 17.3-4. With the analysis above, using the previous potential function, the amortized cost of PUSH is 2, and that of POP and MULTIPOP is 0. Therefore

$$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n) \\ &\leq 2n + s_0 - s_n\end{aligned}$$

Exercise 17.3-5. Similar with above.

$$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n) \\ &\leq 2n + b - \Phi(D_n) \\ &\leq 2n + b\end{aligned}$$

Therefore the total cost is $O(n)$ if $n = \Omega(b)$.

Exercise 17.3-6. We index the two stacks by $\mathcal{I}$ and $\mathcal{O}$. The two operations Enqueue and Dequeue are implemented as the following:

1. Enqueue : push the elements into $\mathcal{I}$.
2. Dequeue : if $\mathcal{O}$ is empty, then pop all the elements in $\mathcal{I}$ and push them into $\mathcal{O}$, then pop the top one. Otherwise, pop the top one in $\mathcal{O}$.

And we define the potential function as $\Phi(\mathcal{I}, \mathcal{O}) = 2 \cdot \#\mathcal{I}$.

Exercise 17.3-7. Don't know how to do yet.

4.4 Dynamic Table

4.4.1 Basic

A table with dynamic expansion and contraction. Property: amortized $O(1)$ cost to insert or delete. Load factor $\alpha(T)$ has a constant lower bound. T is the data structure "table".

4.4.2 Expansion

If insert only, a common heuristic is to allocates a new table with twice as many slots as the old one. In this case the load factor will be never less than $\frac{1}{2}$. As for implementation, we maintain a *num* and *size* field for this data structure, representing number of objects, number of slots, respectively. When triggering an insert operation, if the size is 0, then allocates a table with 1 slot. If num equals to size, then allocate a new table with $2 \cdot \text{size}$ many slots. After the above action, insert the new object into the table.

It's easy to use account method to show that the amortized cost for a sequence of n insertion is $3n$. Each insertion pays 3, one for insertion, one for later removal, yet another one for later removal of its correspondence in the first half of this table. Or we can use potential function method. The potential function for the table T is $\Phi(T) = 2 \cdot \text{num} - \text{size}$. Property of this potential: $\Phi(T) = 0$ immediately after each expansion.

4.4.3 Expansion and Contraction

If we contract the size of table to its half upon a deletion where the num equals to a half of the size, the load factor will be kept larger than $\frac{1}{2}$, but the amortized cost raises to $\Theta(n^2)$. Consider the scenario that after $\frac{n}{2}$ insertions, the size jumps back and forth between $\frac{n}{2}$ and n. Therefore, we must allow the load factor to be less than $\frac{1}{2}$.

The potential function is hereby

$$\Phi(T) = \begin{cases} 2 \cdot \text{num} - \text{size} & \alpha(T) \text{ is greater or equals to } \frac{1}{2} \\ \text{size}/2 - \text{num} & \alpha(T) \text{ is less than } \frac{1}{2} \end{cases}$$

Note that when the load factor is reaching the limit ($\frac{1}{2}$ and $\frac{1}{4}$), the potential is exactly num.

5 Advanced Data Structures

5.1 Heap

5.1.1 Heap Data Structure

A complete tree with number of nodes n has height $\lfloor \log n \rfloor$. We can argue this as follows: a perfect tree (all nodes have either 0 or 2 children and is complete) of height k has $1 + 2 + 2^2 + ... + 2^k = 2^{k+1} - 1$ nodes. We write n into $2^k + l$, where $0 \leq l < 2^k$, and argue by induction. In an array representation, the root is in index 1. Then the left child of node i is $2i$. We write i into $2^k + l$, where $0 \leq l < 2^k$. Then i has depth k. The index of a node denotes the number of nodes before (including) it. The number of nodes before (and including) this depth is $2^{k+1} - 1$. The number of nodes at depth k before i is l. Therefore the index of the left child is $2^{k+1} - 1 + 2l + 1 = 2i$. Heap property: $A[\text{Parent}(i)] \leq A[i]$.

5.1.2 Building a Heap

Given an array $A[1...n]$, then $A[(\lfloor n/2 \rfloor) + 1...n]$ are all the leaves of it. This is too easy, since the last nonleaf node must be the parent of n. Also, the number of nodes of height h is at most $\lceil \frac{n}{2^{h+1}} \rceil$. This can be proved by reverse induction on the height. Suppose $h = \lfloor \log n \rfloor$, then $0 < \frac{n}{2^{\lfloor \log n \rfloor}} \leq \frac{n}{2^{\log n}} = 1$, we have $\lceil \frac{n}{2^{\lfloor \log n \rfloor}} \rceil = 1$. Since the number of nodes of height h is at most twice of that of $h + 1$, therefore we have $\leq \lceil \frac{n}{2^{h+2}} \rceil \times 2 = \lceil \frac{n}{2^{h+1}} \rceil$. By heapify down from index $\lfloor n/2 \rfloor$ down to 1, the running time is

$$\begin{aligned} \sum_{h=0}^{\lfloor n/2 \rfloor} O(h) \lceil \frac{n}{2^{h+1}} \rceil &= O(\sum_{h=0}^{\lfloor n/2 \rfloor} h \frac{n}{2^h}) \\ &= O(n) \end{aligned}$$

Exercise 6-3 Young Tableau. .

c when the $(1, 1)$-th entry was extracted away, the second least element of this tableau must be picked to fill in its place. And it must be either $(2, 1)$-th entry or $(1, 2)$-th entry. Furthermore we notice a property: when an minimum element is extracted, every entry of this tableau will be at least as large as the original one. Therefore we can recursively solve the sub-problem. The recurrence relation is given as

$$T(n + m) = T(n + m - 1) + O(1)$$

f start from the left-bottom corner. Suppose we're looking for k. If the key k' in the entry has $k' < k$, then we go right, if $k' > k$ then go up, if equals then we stop. We claim that either we find the desired element or we walk off the tableau. The reasoning is easy, too. Suppose the desired element is indeed in this tableau. We notice that we cannot walk to the right-bottom or the left-top region of this element. Take right-bottom as an example. If we ever walk to this region, this means that we have a step where we're under this element but then we move right. Which means this element is larger than some element right below it, which is a contradiction. Since we have to walk anyway, we must hence finally reach this element. In this argument, we also prove that we can always reach the closest one.

5.2 Red-Black Trees

5.2.1 Properties

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either red or black. Leaves contain no attribute. Red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.

4. If a node is red, then both its children are black.

5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

We call the number of black nodes on any simple paht from, but not including, a node x down to a leaf the black-height of the node. We denote $\text{bh}(x)$ to be this function of node. The black-height of a red-black tree is defined to be the black-height of its root.

Lemma A. red-black tree with n internal nodes has height at most $2\log(n+1)$.

We start by showing that the sub-tree rooted at node x contains at least $2^{\text{bh}(x)}-1$ nodes. And we prove this by induction on the height of this tree. If the tree is of height 0, then x must be itself a leaf. Therefore it contains $2^0-1=0$ internal nodes. For the inductive step, suppose the tree has positive height and is an internal node with two children l and r. It's clear that $\text{bh}(l) \geq \text{bh}(x)-1$ and $\text{bh}(r) \geq \text{bh}(x)-1$. Therefore by inductive hypothesis we've proved the desired result. On the other hand, by property 4, at least half of the nodes in a simple path from x, but not included, to the leaf must be black, we conclude that $\text{bh}(x) \geq h/2$, therefore $n \geq 2^{h/2}-1$, and hence $h \leq 2\log(n+1)$ □

5.3 Data Structures for Disjoint Sets

5.3.1 Definition and Basic Operations

A **disjoint-set data structure** maintains a collection $\{S_1, S_2, ...S_k\}$ of disjoint dynamic sets, with each set a representation associated. There're three operation: MAKE-SET(x) that creates a new set whose only member is x. UNION(x, y) unites the dynamic sets that contain x, and y into a new set. FIND-SET(x) returns a pointer to the representative of the set containing x.

5.3.2 Disjoint-set forests

Two main heuristics: **union by rank**, for each node, we maintain a rank, which is an upper bound on the height of the node. **path compression**, make each node on the find path point directly to the root. Consider a sequence of m MAKE-SET, UNION.FIND-SET operations, where n of which are MAKE-SET operations (which means n elements), with union by rank heuristic alone, the running time is $O(m \log n)$.

6 String Match

6.1 String matching with finite automata

A finite automaton M induces a function ϕ, called the *final-state function*, from Σ^* to Q. $\phi(\epsilon) = q_0$, $\phi(wa) = \delta(\phi(w), a)$

For a given pattern $P[1..m]$, we construct a string-matching automaton in a preprocessing step before using it to search the text string. We first define an auxiliary function σ, called the *suffix function* corresponding to P. The funtion σ maps Σ^* to $\{0, 1, ..., m\}$ such that $\sigma(x)$ is the length off the longest prefix of P that is also a suffix of x : $\sigma(x) = \max\{k : P_k \sqsupset x\}$. Notice that since $P_0 = \epsilon$ is a suffix of any string, this function is well-defined. For a pattern P, $\sigma(x) = m$ if and only if $P \sqsupset x$. And we have $x \sqsupset y$ implies $\sigma(x) \leq \sigma(y)$.

We deffine the string-matching automaton that corresponds to a given pattern $P[1..m]$ as follows:

. The state set Q is $\{0, 1, ..., m\}$. The start state q_0 is state 0, and state m is the only accepting state.

. The transition function δ is defined by the following equation, for any state q and character a, $\delta(q, a) = \sigma(P_q a)$

The purpose of this design is that when the state machine goes to state q, then q is the length of the maximum prefix of P that matches the current text. That is to say, the automaton maintains

$$\phi(T_i) = \sigma(T_i)$$

There're two cases to consider for $T[i+1] = a$. The first one is when $a = P[q+1]$, the character continues to match. In this case, $\delta(q, a) = q + 1$. The second one is when the character doesn't continue to match. In this case we must find a smaller prefix of P that is also a suffix of T_i.

What we need to prove is that, when the state machine goes to state i, the corresponding state is $\sigma(T_i)$. Since $\sigma(T_i) = m$ if and only if $P \sqsupset T_i$, ...

Lemma Suffix-function Inequality. For any string x and character a, we have $\sigma(xa) \leq \sigma(x) + 1$

Lemma Suffix-function recursion. For any string x and character a, if q $= \sigma(x)$, then $\sigma(xa) = \sigma(P_q a)$

Proof. Firstly, $\sigma(x) = q \Rightarrow P_q \sqsupset x \Rightarrow P_q a \sqsupset xa \Rightarrow \sigma(P_q a) \leq \sigma(xa)$.

Secondly, let $l = \sigma(xa)$, then $l \leq \sigma(x) + 1 = q + 1$, together with $P_l \sqsupset xa$, we have $P_l \sqsupset P_q a$. Therefore, $\sigma(xa) = l = \sigma(P_l) \leq \sigma(P_q a)$.

Hence, $\sigma(xa) = \sigma(P_q a)$ □

Theorem Main. $\sigma(T_i) = \phi(T_i)$

Proof. By induction on i. It's trivial when $i = 0$. And

$$\begin{aligned}
\phi(T_{i+1}) &= \delta(\phi(T_i), T[i+1]) \\
&= \sigma(P_{\phi(T_i)} T[i+1]) \\
&= \sigma(P_{\sigma(T_i)} T[i+1]) \\
&= \sigma(T_i T[i+1]) \\
&= \sigma(T_{i+1})
\end{aligned}$$

□

Now what we need to do is to calculate the transition function. But this design will be revealed until next subsection (KMP algorithm).

6.2 The Knuth-Morris-Pratt algorithm

This algorithm avoids computing the whole transition function δ, but using an array π to efficiently (amortized) compute δ on the fly. $\pi[q]$ contains the information we need to calculate $\delta(q, a)$ independently of a.

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. Typically, we are interested in the following question: Given that pattern characters $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s' > s$ such that for some $k < q$, $P[1..k] = T[s'+1..s'+k]$, where $s' + k = s + q$? In other words, knowing that $P_q \sqsupset T_{s+q}$, we want the longest proper prefix P_k of P_q that is also a suffix of T_{s+q}. From the equation, we have $s' = s + (q - k)$. In the best case, when $k = 0$, we immediately rule out shift $s + 1, ..s + q - 1$. In any case, at the new shift s' we don't need to compare the first k characters of P with the corresponding characters of T, since the above equation guarantees that they are equal. q is the currently matched number of characters. We store, for each q, the value k characters that also match. We can formalize it as follows:

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

, that is to say, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q.

The precomputing takes time $\Theta(m)$, where m is the length of the pattern. Firstly we notice that $k < q$ upon entry of the for-loop. Secondly we observe that for all $k < q$ upon entry of the for-loop, $\pi[k] < k$. Since k increase at most $m - 1$ times, the while loop takes time all togather at most $m - 1$ times. Therefore, the running time of preprocessing is $\Theta(m)$.

Lemma Prefix-function iteration lemma. let $\pi^*[q] = \{\pi[q], \pi^{(2)}[q], .., \pi^{(t)}[q]\}$. The iteration stops upon $\pi^{(t)}[q] = 0$. Then $\{k : k < q \text{ and } P_k \sqsupset P_q\} = \pi^*[q]$

Proof. "$\supseteq$": Trivial.

"$\subseteq$": Pick $j < q$ such that $P_j \sqsupset P_q$. If $j = \pi[q]$ then we're done. Otherwise, let $j' = \min\{i \in \pi^*[q] : i > j\}$. Then $P_j \sqsupset P_{j'}$. This implies that $\pi[j'] = j$, otherwise j' won't be the minimum one.

□

Lemma .. If $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^*[q-1]$

Proof. Notice that in this case, q can not be 0 or 1. The proof is trivial. □

In order to prove the correctness of the prefix-computation algorithm, we define

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ and } P_k \sqsupset P_{q-1} \text{ and } P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ and } P_{k+1} \sqsupset P_q\} \end{aligned}$$

Corollary .. Let P be a pattern off length m, and let pi be the prefix function of P, then for $q = 2, ..m$,

$$\pi[q] = \begin{cases} 0 & \text{if} \quad E_{q-1} = \emptyset \\ 1 + \max\{k \in E_{q-1}\} & \text{if} \quad E_{q-1} \neq \emptyset \end{cases}$$

Proof. First, we notice that if $\pi[q] > 0$, then $\pi[q] - 1 \in E_{q-1}$. Since if $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^*[q-1]$, and $P[\pi[q]] = P[q]$, hence $\pi[q] - 1 \in E_{q-1}$

If $E_{q-1} = \emptyset$, then $\pi[q] = 0$ by the above argument. If $E_{q-1} \neq \emptyset$, let $l = \max\{k \in E_{q-1}\}$, we must have $\pi[q] - 1 \leq l \Rightarrow \pi[q] \leq l + 1$. But by definition, $P_{l+1} \sqsupset P_q \Rightarrow \pi[q] \geq l + 1$. Therefore, $\pi[q] = l + 1$. □

Now the correctness of the precomputation step is clear. The value of $\pi[q-1]$ is stored upon entry of the inner while-loop. And during the while-loop, the algorithm iterates through $\pi^*[q-1]$ in decreasing order to find $\max E_{q-1}$.

As for the correctness of the whole algorithm, we reduce it to the correctness of finite automaton by proving that the index q matches the state upon entry of the main for-loop. This is done by induction on the loop. Upon the first entry, $q = 0$, which matches the initial state. Suppose upon the ith entry, $q' = \sigma(T_{i-1})$.

If $\sigma(T_i) = 0$, then $P_0 = \epsilon$ is the only prefix of P that is also a suffix of T. In the while loop, the algorithm iterates $q \in \pi^*[q']$, but fails to find a q such that $P[q+1] = T[i]$. And q finally falls to 0, and therefore, $q = T_i$.

If $\sigma(T_i) = q' + 1$, then $P[q'+1] = T[i]$, and the while-loop abort mid-way, and q is updated by increment. Hence $q = q' + 1 = \sigma(T_i)$.

If $0 < \sigma(T_i) \leq q'$, then the while-loop execute at least once, and terminates at some $q < q', q \in \pi^*[q']$, where $P[q+1] = T[i]$. Therefore, we have $q+1 = \sigma(P_{q'}T[i])$. Since $\sigma(T_{i-1}) = q'$, we have $q+1 = \sigma(P_{q'}T[i]) = \sigma(T_{i-1}T[i]) = \sigma(T_i)$, this ends up the proof.