COMP20007 Design of Algorithms

Week 11 Workshop Solutions

Tutorial

1. Counting Sort To perform counting sort we note that the range of possible input characters are a, b, c, d, e, and f. We then loop through the array and tally the frequencies of each character:

```
  a b c d e f
[ 5 2 2 1 0 1 ]
```

We then read these frequencies in order to reconstruct a sorted array. That is, we take 5 `a`s, 2 `b`s *etc.*

```
[a a a a a b b c c d f]
```

2. Radix Sort Start by sorting by the final letter, keeping strings with the same final letter in the original relative order:

```
  abc bab cba ccc bbb aac abb bac bcc cab aba
cba aba | bab bbb abb cab | abc ccc aac bac bcc
```

We then join all of these together and sort by the middle letter:

```
  cba aba bab bbb abb cab abc ccc aac bac bcc
bab cab aac bac | cba aba bbb abb abc | ccc bcc
```

Finally by the first letter:

```
  bab cab aac bac cba aba bbb abb abc ccc bcc
aac aba abb abc | bab bac bbb bcc | cab cba ccc
```

So the final sorted array is

```
aac aba abb abc bab bac bbb bcc cab cba ccc
```

3. Stable Counting Sort To sort tuples by the first element and maintian relative order between tuples with equal first values we must ensure that *counting sort is stable.*

We can use the cumulative coints array based approach introduced in lectures to do this.

Alternatively, rather than storing frequencies in a frequency array, we have an array of linked lists with each object appended to the linked list corresponding to the value we're sorting by. The original array can be reconstructed by sequentially removing the head of the linked list and inserting into the new reconstructed array.

Both methods will require $\Theta(n + k)$ additional space, where k is the number of distinct values we're counting and n is the size of the input array.

4. Horspool's Algorithm For that pattern we calculate the shifts: $S[G] = 3, S[O] = 2, S[R] = 1, S[x] = 4$ for all other letters x. So the first shift (when the string's O fails to match E) is 2 positions, bringing E under the string's I. The next shift is 4, which will take us beyond the end of the string, so the algorithm halts (after just two comparisons), reporting failure.

5. Horspool's Algorithm Continued

(a) The pattern's last 1 will be compared against every single 0 in the text (except of course the first four), since the skip will be 1. So 999,996 comparisons.

(b) Here we will will make two comparisons between shifts, and each shift is of length 2. So the answer is again 999,996 comparisons.

(c) For the last pattern, the skip is 4. So we will make 249,999 comparisons.

6. Horspool's Worst-Case Time Complexity Consider the case where the text contains n zeros, and the pattern contains a 1 in the first position, followed by $m-1$ zeros. In this case each skip will be just a single position, and between skips, m comparisons will be made.

Skips will occur until the pattern is starting at index $n-m$. Since we start at index 0 and end at index $n-m$ we do m comparisons $n-m+1$ times, giving $m(n-m+1) \in O(nm)$ comparisons.

We can see that if our pattern contains a 1 followed by $n/2$ zeros this time complexity becomes $O(n^2)$.

7. (Revision) Recurrence Relations

(a) $T(1) = 1$
$T(n) = T(n/2) + 1$

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= (T(n/4) + 1) + 1 \\ &= (T(n/8) + 1) + 1 + 1 \\ &\vdots \\ &= T(n/2^k) + k \\ &\vdots && \text{let } k = \log_2(n) \\ &= T(n/2^{\log_2(n)}) + \log_2(n) \\ &= T(n/n) + \log_2(n) \\ &= T(1) + \log_2(n) \\ &= 1 + \log_2(n) \end{aligned}$$

So $T(n) = 1 = \log_2(n) \in \Theta(\log n)$.

(b) $T(0) = 0$
$T(n) = T(n-1) + \frac{n}{5}$

$$\begin{aligned} T(n) &= T(n-1) + \frac{n}{5} \\ &= T(n-2) + \frac{n-1}{5} + \frac{n}{5} \\ &= T(n-3) + \frac{n-2}{5} + \frac{n-1}{5} + \frac{n}{5} \\ &\vdots \\ &= T(n-k) + \frac{n-(k-1)}{5} + \cdots + \frac{n-1}{5} + \frac{n}{5} \\ &\vdots && \text{let } k = n \\ &= T(n-n) + \frac{n-(n-1)}{5} + \cdots \frac{n-1}{5} + \frac{n}{5} \\ &= 0 + \frac{1}{5} + \cdots \frac{n-1}{5} + \frac{n}{5} \\ &= \frac{1}{5}\left(1 + \cdots + (n-1) + n\right) \\ &= \frac{n(n+1)}{10} \end{aligned}$$

So $T(n) = n(n+1)/10 \in \Theta(n^2)$.

8. (Optional) Karp-Rabin Hashing First we must compute the hash of the pattern $P =$"CAB":

$$\begin{aligned} h(\text{“CAB”}) &= a^2 \cdot \text{chr(C)} + a \cdot \text{chr(A)} + \text{chr(B)} \mod m \\ &= 4^2 \cdot 2 + 4 \cdot 0 + 1 \mod 11 \\ &= 33 \mod 11 \\ &= 0 \end{aligned}$$

As discussed in the question we can compute $h(\text{“CAD”})$ like so:

$$\begin{aligned} h(\text{“CAD”}) &= a^2 \cdot \text{chr(C)} + a \cdot \text{chr(A)} + \text{chr(D)} \mod m \\ &= 4^2 \cdot 2 + 4 \cdot 0 + 3 \mod 11 \\ &= 35 \mod 11 \\ &= 2 \end{aligned}$$

Then we can compute each successive substring of 3 characters like so:

$$\begin{aligned} h(\text{“ADA”}) &= 4\left(h(\text{“CAD”}) - 4^2 \cdot 2\right) + 0 \mod m \\ &= 4(2 - 32) + 0 \mod 11 \\ &= 4(-30) + 0 \mod 11 \\ &= 4(-30 \mod 11) \mod 11 \\ &= 4(3) \mod 11 \\ &= 12 \mod 11 \\ &= 1 \end{aligned}$$

From $h(\text{“ADA”})$ we can compute $h(\text{“DAC”})$ like so:

$$\begin{aligned} h(\text{“DAC”}) &= 4\left(h(\text{“ADA”}) - 4^2 \cdot 0\right) + 2 \mod m \\ &= 4(1) + 2 \mod 11 \\ &= 6 \mod 11 \\ &= 6 \end{aligned}$$

Completing this for the rest of the substrings with 3 characters in T we get:

$$\begin{aligned} h(T[0 \ldots 2]) &= h(\text{“CAD”}) = 2 \\ h(T[1 \ldots 3]) &= h(\text{“ADA”}) = 1 \\ h(T[2 \ldots 4]) &= h(\text{“DAC”}) = 6 \\ h(T[3 \ldots 5]) &= h(\text{“ACA”}) = 8 \\ h(T[4 \ldots 6]) &= h(\text{“CAB”}) = 0 \end{aligned}$$

Notice that $h(T[4 \ldots 6]) = 0$ and $h(P) = 0$. Therefore, $T[0 \ldots 2]$ *may* be equal to P.

Why can we not be sure that $T[0 \ldots 2] = P$, as we may get a **collision** with our hash function. The hash function may give the same result for two different substrings, for instance $h(\text{“DBCD”})$ also equals 0. As a result we must check manually that the strings match. If we choose a large m then these collisions become increasingly less likely, and thus the additional computation complexity required by checking “false positives” (*i.e.*, when the hash function values are the same but the strings differ) becomes negligible.

How about the time complexity of this algorithm? Initially computing $h(S)$ for a string S of n characters takes $O(n)$ time. So computing $h(P)$ takes $O(|P|)$ time.

Note that there are $|T| + 1 - |P|$ substrings of length $|P|$ in T. The first of which takes $O(|P|)$ time to hash. However, due to the ability to compute hashes incrementally in $O(1)$ time the remaining $|T| - |P|$ substrings only take $O(1)$ time each. Thus the total time complexity of computing the hashes for all substrings becomes:

$$|P| + (|T| - |P|) = O(|T|).$$

Taking into account the time complexity of computing the hash of P we get a total time complexity of the Karp-Rabin string search algorithm of $O(|T| + |P|)$.