cs124 Asst 4 Suggested Solutions

abhi shelat

March 16, 1997

Problem 1 *Cities*

The insight here is to compute the *cumulative* distances between the first city and the ith city. Then any query can be satisfied by computing $|d[i] - d[j]|$. The preprocessing takes $O(n)$ and queries are clearly $O(1)$.

Distance-Preprocess(A, n)
(1) **for** $i = 1$ **to** n
(2) $A[i] \leftarrow A[i-1] + A[i]$

Query$(A, city_1, city_2)$
(1) **return** $|A[city_1] - A[city_2]|$

Problem 2 *20-bit words*

There are two basic approaches. First, you can allocated a 2^{20} bit array (128K bytes of memory), and scan through the file once. For every word that you read, mark the corresponding ith bit in your array. After the first pass, begin at the top of your bit array, and return the index of the first unmarked bit. Complexity is n time $O(n)$ space.

It is also possible to solve this in $O(1)$ space and $2n$ time by using a partitioning scheme. Partition on the first bit of each word and determine which side has fewer elements; discard the other side. Suppose wlog that it is the 0-side. This implies there is a word that begins with a 0 that is not in the file. Set our top bit to 0, and now continue the same process with the second bits of the remaining words. This solution gives the simple recurrence

$$T(n) = T(\frac{n}{2}) + n$$

which turns out to be $T(n) = 2n$ - remarkable!

Problem 3 *Sorting in 5 compares*

Take the decision tree model: There are 5! permutations on five objects which implies you need $\lceil(\lg 5!)\rceil = 7$ compares. Your humble TF is wrong!

To actually sort using 7 compares, consider a generic array of 5 elements:

and start by comparing 2 pairs, (a, b) and (c, d). We can order these two pairs using only two comparisons (in the drawing, vertical edges refer to the "greater than" relationship):

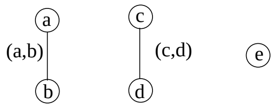

Figure 1: Information after 2 compares

Now we recursively sort the winners from this tournement. In this case, since there are only two winners, this takes one comparison. All of a sudden, we have a list of three sorted elements.

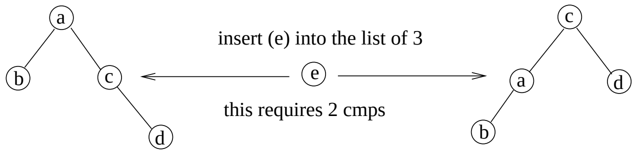

Figure 2: (a) The two possible situations after 3 compares (b) We can insert e into the list of 3 sorted elements using only 2 compares

Now notice that we can insert the element e into the list of 3 sorted elements using only 2 compares. Hence, after 5 comparisons, we have created a list of 4 sorted elements. The diagram below shows the four possible results after the insertion, the other four cases arising from the second situation above are symmetrical.

For the last element, we already know that it is less than one of the four elements in our 4-element list. This leaves a sublist of at most 3 elements, and once again, hence the last element can be inserted into the sublist with at most 2 more comparisons for a total of 7!. The trick was to insert e before

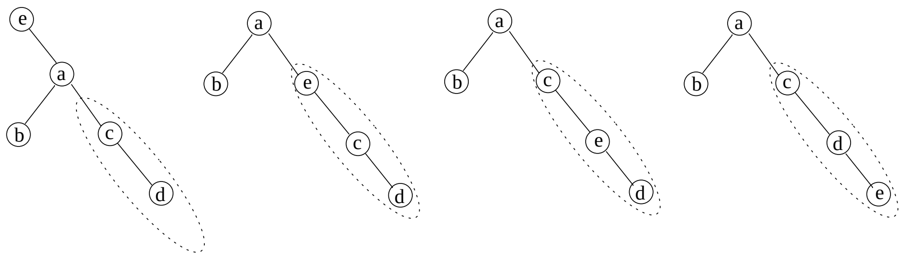

Figure 3: (a) After insertion of e. The sublist for b is circled.

inserting the loser from the first round of comparisons. This process can be generalized for larger n as well.

Problem 4 *Bookie*

Just use the partition algorithm. We are guarenteed that each person who owes us is exchanged no more than once.

```
BOOKIE(A, n)
    i ← −1
    j ← n + 1
    while i < j
        while i < n and A[i] < 0
            i ← i + 1
        while j > 0 and A[j] ≥ 0
            j ← j − 1
        SWAP(A[i], A[j])
    return j
```

Problem 5 *Avg value*

The idea here is to notice the structure of the sum. Instead of computing the sum of all the values of $f(p_i)$, we compute the sum of the values in the *columns* of $f(p_i)$.

For $n = 3$, notice

$$
\begin{aligned}
f(p_1) &= |1-1| + |2-2| + |3-3| \\
f(p_2) &= |1-1| + |3-2| + |2-3| \\
f(p_3) &= |2-1| + |1-2| + |3-3| \\
f(p_4) &= |2-1| + |3-2| + |1-3| \\
f(p_5) &= |3-1| + |1-2| + |2-3| \\
f(p_6) &= |3-1| + |2-2| + |1-3|
\end{aligned}
$$

Notice that in column j, the variable a_j will take on each of the values in $\{1, \ldots n\}$ exactly $(n-1)!$ times. But we must also subtract j from each of the values in column j. For $n = 5$, we would get $0, 1, 2, 3, 4$ each occuring $4!$ times in column 1, and $1, 0, 1, 2, 3$ each occuring $4!$ times in column 2, etc. Here is a picture:

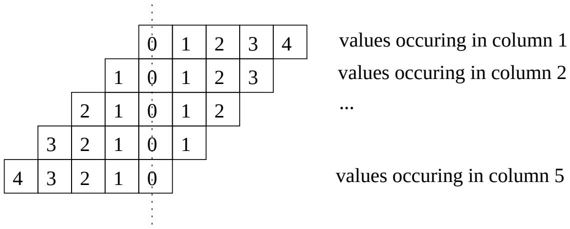

Figure 4:

The symmetry allows us to sum one half of the parallelogram and multiply by two. The math boils down to

$$
\begin{aligned}
2(n-1)! \sum_{j0}^{n-1} \sum_{i0}^{j} i &= 2(n-1)! \sum_{j0}^{n-1} \frac{(j)(j+1)}{2} \\
&= (n-1)! \left(\sum_{j0}^{n-1} j^2 + \sum_{j0}^{n} j \right) \\
&= (n-1)! \left(\frac{(n-1)(n)(2n-1)}{6} + \frac{(n-1)(n)}{2} \right) \\
&= (n-1)! \left(\frac{(n-1)(n)(2)(n+1)}{6} \right) = (n-1)! \left(\frac{(n-1)(n)(n+1)}{3} \right)
\end{aligned}
$$

Since there are $n!$ total permutations on n objects, we must divide the above by $n!$ giving us the answer

$$\frac{(n^2-1)}{3}$$

as the average value of f.

Problem 6 *Hashing*

The values turn out to be $700, 318, 936, 554, 172$.

Problem 7 *Quadratic Probing*

The first part is simple. Write the probing sequence down (let $c = h'(k)$): $c, c+1, c+3, c+6, c+10, \ldots$. Notice the that ith probe turns out to be the sum of the first i numbers. Using the old formula:

$$\sum_{i0}^{n} i = \frac{1}{2}(n(n+1)) == \frac{1}{2}i + \frac{1}{2}i^2$$

which imply $c_1 = c_2 = \frac{1}{2}$.

The second part is much more tricky. The solution here was inspired by several people in the class– much nicer than my own.

Suppose not, i.e, that after m probing attempts, there is one spot that has not been probed. By the pigeon hole principle, this implies that another spot was hit at least twice- without loss of generality (wlog), assume that the ith and jth probe, where $0 < i < j < m$, probed the same spot. Let $k = j - i$. Given the definition of the probing sequence, this implies that

$$\begin{aligned}
&(1+2+\cdots+i) \equiv (1+2+\cdots+j) \mod m \\
\Longrightarrow\ &(1/2)(j^2+j) - (1/2)(i^2+i) \equiv 0 \mod m \\
\Longrightarrow\ &(j^2-i^2) + (j-i) \equiv 0 \mod 2m \\
\Longrightarrow\ &(j+i)(j-i) + (j-i) \equiv 0 \mod 2m \\
\Longrightarrow\ &(j+i+1)(j-i) \equiv 0 \mod 2m
\end{aligned}$$

This means that either $(j+i+1)|2m$ or $(j-i)|2m$. By assumption, we have that $i < j < m$ which implies $j \leq m-1$, or $j+i+1 \leq 2m-1$ but it is impossible for a number smaller than $2m$ to be divisible by $2m$. So the second condition must be true. Again, $0 < j - i \leq m-1$ which gives the same contradiction.

Problem 8 *Bit-Reversal Binary*

Many people misunderstood this problem. The bit-reversed permutation acts on elements in the array whose *indicies* are bit-wise reversals of each other. For example, if $k = 4$, then we need to swap $A[2]$ with $A[4]$ since the binary representation of $2 = 0010 \leftarrow 4 = 0100$ are reversals of each other. Note however, that if we just use the obvious for loop, than we could end up swapping $A[4]$ with $A[2]$ again, resulting in the wrong permutation.

BIT-REVERSED-PERMUTATION(A, k)
  $n \leftarrow 2^k$
  for $i = 0$ **to** n
    $j \leftarrow rev_k(i)$
    if $i > j$ **then** SWAP$(A[i], A[j])$

This algorithm will take $O(kn)$. The loop will execute n times, and each call of rev_k will take time k. The comparison and swap are a constant number of operations.

In part (b), we can use the insight from the bit-increment algorithm to write a bit-reversed-increment algorithm which produces $rev_k(rev_k(a) + 1)$ in amortized $O(1)$ time. This allows the entire loop to run in $O(n)$ time since each loop-iteration takes only a constant number of operations.

FAST-BIT-REVERSED-PERMUTATION(A, k)
  $j \leftarrow 0$
  for $i = 0$ **to** n
    $j \leftarrow$ BIT-REVERSE-INCREMENT(j)
    if $i > j$ **then** SWAP$(A[i], A[j])$

Notice that the bit-reversed increment operation is equivalent to the bit-increment operation, with the exception that the reversed-increment operation occurs from left to right, as opposed to the normal right-left propogation of carries. We just "reverse" the bit INCREMENT() algorithm presented on page 359 of CLR.

BIT-REVERSED-INCREMENT(a)
  $i \leftarrow length[A]$
  while $i > 0$ and $A[i] = 1$
    $A[i] \leftarrow 0$
    $i \leftarrow i + 1$
  if $i > 0$ **then** $A[i] \leftarrow 1$

For part (c), there are really two answers. If the machine does not support shifting by arbitrary amounts, then we cannot use the above algorithm as it is to achieve an $O(1)$ solution. However, there are several tricks that we can play. The first is to use a little extra space ($O(k)$) to store a table of k words where the ith word has only the ith bit set. During the loop, the arbitrary shifts can now be performed by lookup. The second idea is to store 2^{k-1} as a constant, and use it to initialize a temporary variable at the beginning of the loop. Every iteration can now right-shift the temporary variable to get the correct vector.