

Answers to the Final Exam

Given May 7, 2001.

1. We want to search the “update” graph. In this graph, x_1 would point to x_2 because x_1 occurs on the right side of the x_2 equation. This update graph is the reverse (or “transpose”) of the dependency graph. In the dependency graph, there is an edge from v to w if w appears on the right side of the v equation. We need an edge that points from w to u in that case. the following pseudocode fragment constructs the reverse graph.

```
for ( v a variable ) create v.ul; // an empty list that will be the
                                // edge list for the update graph
for ( v a variable ) {
    for ( w in v.dl ) add v to w.ul;
}
```

To tell whether u and v have disjoint update lists we mark every variable in the update list for u and then check whether any of these variables are in the v update list. To find the u update list, we mark every variable (vertex) in the graph as **UNVISITED**. We do a DFS (BFS would also work here) starting from u marking vertices **U-VISITED**. Then we do the DFS starting from v marking vertices **V-VISITED**. If any of these vertices is **U-VISITED**, there is an overlap in the update lists.

```
// The procedure that visits a vertex in the u update list.
```

```
U_visit(x) {
    mark x U-VISITED;
    for ( y in x.ul ) if ( y UNVISITED ) U_visit(y);
}
```

```
// The procedure that visits a vertex in the V update list.
```

```
V_visit(x) {
    mark x V-VISITED;
    for ( y in x.ul ) {
        if ( y U-VISITED ) disjoint = FALSE;
        if ( y UNVISITED ) V_visit(y);
    }
}
```

```

// The main program.

upDis ( u, v ) {
    for ( x a variable ) mark x UNVISITED;
    disjoint = TRUE;
    U_visit(u);
    V_visit(v);
    return disjoint;
}

```

The work for this is proportional to the number of variables and the sum of the lengths of the dependency lists (i.e. the number of edges). Since you at least have to compute the reverse graph, which involves examining all the vertices and edges, you cannot do much better than this.

```

2. A.    heapOrder = TRUE;
        for (j=2, .., n) {
            if (j is even) k = j/2;    // k is the parent of j.
            else          k = (j-1)/2;
            if ( a[k] > a[j] ) heapOrder = FALSE;
        }

B.      // Here is a procedure that exchanges a[k] with a child or its parent
        // if it's out of order. It returns the index of the new location,
        // which is k if the number does not need to move.

        int heapSwap(a, k, n) {

            j = k; // j is the new location, which might be k

                // Compute the parent if there is one
            if (k > 1 ) {
                if ( k is even ) p = k/2;
                else p = (k-1)/2;
                if ( a[p] > a[k] ) {
                    swap a[p] and a[k];
                    j = p;
                    return j
                }
            }

            // Find the children that might be there

```

```

    lch = 2*k;
    if (lch > n) return j; // there are no children
    rch = lch + 1;
    if (rch > n) { // there is just one child.
        possibly swap a[k] with a[lch];
    }
    else
        possibly swap a[k] with the smaller of a[lch] and a[rch]

// Here is the main delete routine.

a[k] = a[n];
n--;
while ( k!= (j = heapSwap(a, k, n) ) k = j;

```

3. It would be easier to write this in C/C++ than Java because Java cannot pass a sub array as an array. I will write assuming the Java restriction. The algorithm works with an interval of uncertainty with $a \leq k < b$ at all times. We want to find a set of numbers. At all times all the numbers in $a[j]$ for $j < a$ are definitely in the set, while all the numbers $a[j]$ for $j \geq b$ are definitely out. Also, all the numbers in $a[j]$ for $j < a$ are not larger than any number in $a[j]$ for $j \geq a$ and all the numbers in $a[j]$ for $j > b$ are not less than any number in $a[j]$ for $j \leq b$. This is the situation produced by quick select. At any stage, X will be the sum of the numbers $a[j]$ for $j < a$. The quick select step will divide the $a[j]$ for $a \leq j < b$ using a pivot chosen at random from these numbers. This pivot will end up in $a[c]$. If the sum of the numbers between a and c is small, then all of them are in the set we want so the new interval is (c, b) , otherwise it is (a, c) .
4. **A.** Use BFS, which is the search tree of minimum depth.
- B.** Sort the numbers using quicksort. Radix sort would be slower because there are so many digits. Then compare (by subtraction) neighbors.
- C.** Use a hash table. All the operations listed take $O(1)$ time.
5. The maximum height comes from the minimum branching factor and the minimum height comes from the maximum branching factor. To find the minimum height, take branching factor $64 = 2^6$. The total number of keys in the tree is 256 million = $2^8 \cdot 2^{20} = 2^{28}$. Then

$$\begin{aligned}
 \text{root has } 64 &= 2^6 \text{ keys.} \\
 \text{height 1 has } 64^2 &= 2^{12} \text{ keys.} \\
 \text{height 2 has } 64^3 &= 2^{18} \text{ keys.} \\
 \text{height 3 has } 64^4 &= 2^{24} \text{ keys.} \\
 \text{height 4 has } 64^5 &= 2^{30} \text{ keys.}
 \end{aligned}$$

Thus, height 3 is not enough but height 4 is plenty. The maximum height is 4. For the minimum height, take the maximum branching factor, which is $128 = 2^7$. Then the number of keys at height h is

$$128^{h-1} = 2^{7 \cdot (h-1)} \geq 256 \text{ million} = 2^{28}$$

gives $h = 3$. Conclusion: the maximum possible height is 4 and the min is 3, if you count a tree with just a root as height 0.

6. A. For large n , $\lg(n)$ is much smaller than $5n$ and n^4 is much larger than n^2 . Therefore we neglect the smaller terms and have $W(n) \approx 5n \cdot \lg(n^4) = 5n \cdot 4 \lg(n) = 20n \cdot \lg(n)$. Thus $W(n) = \Theta(n \lg(n))$.

B. We had two problems of this kind for homework, one in the triangle problem and one in the Strassen algorithm. Since T is an increasing function of n , we can figure out its order of magnitude if we get it for $n = 4^k$. We choose $n = 4^k$ because then $\lceil n/4 \rceil = n/4 = 4^{k-1}$, which makes the recurrence relation simpler. If we say $U(k) = T(4^k)$, then the recurrence relation is

$$U(k) = 3U(k-1) + 4^{k^2} = 3U(k-1) + 4^{2^k} = 3U(k-1) + 16^k .$$

We can unroll this a few times to get

$$\begin{aligned} U(k) &= 3 \left(3U(k-2) + 16^{k-1} \right) + 16^k \\ &= 3 \cdot 3U(k-2) + 3 \cdot 16^{k-1} + 16^k \\ &= 3 \cdot 3 \left(3U(k-3) + 16^{k-2} \right) + 3 \cdot 16^{k-1} + 16^k \\ &= 3^3 U(k-3) + 3^2 16^{k-2} + 3 \cdot 16^{k-1} + 16^k \end{aligned}$$

and eventually

$$U(k) = 16^k + 3 \cdot 16^{k-1} + 3^2 \cdot 16^{k-2} + \dots + 16 \cdot 3^{k-1} + 3^k .$$

The bottom sum is a geometric series which we can sum using the general method for geometric series:

$$\begin{aligned} U(k) &= 3^k \left((16/3)^k + (16/3)^{k-1} + \dots + 1 \right) \\ &= 3^k \cdot \frac{(16/3)^{k+1} - 1}{(16/3) - 1} \\ &= \frac{16^{k+1} - 3^{k+1}}{16 - 3} \\ &\approx \frac{16}{12} 16^k \\ &= \frac{4}{3} n^2 = \Theta(n^2) . \end{aligned}$$

In the next to last line we used the fact that for large k , 16^k is much larger than 3^k . In the last line we used again the fact that $n = 4^k$ so $n^2 = 16^k$. This is a very hard problem. I put it on the exam because I had said that some of the hard homework problems would appear on the exam.

7. **A.** FALSE. We could have $W(n) = n^2$ for n even and $W(n) = 10 \cdot n^2$ for n odd. This is $\Theta(n^2)$ but if n is odd then $2n$ is even so $W(2n) = (2n)^2 = 4n^2 < 10n^2 = W(n)$.
- B.** FALSE. The expected number of collisions is on the order of $n^2/(\text{table size})$. I don't remember the exact formula but this was a homework question.
- C.** TRUE. A rotation just changes a few pointers around the tree element
- D.** FALSE. This is a version of the traveling salesman problem. It is very important in an algorithms class not to think that someone knows a great algorithm for any problem.