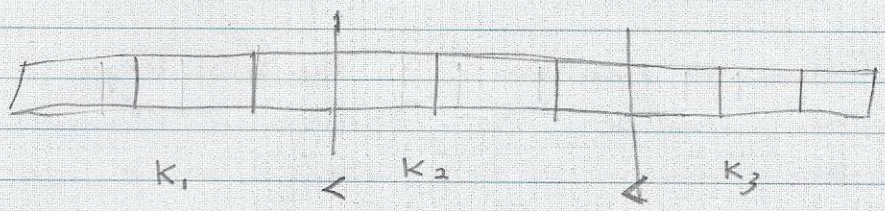
Kelsey Neis – neis@umn.edu
Section 001
Homework 3, Due Oct. 29

~~8.1-4; 9.2-3; 9.3-3,5; 21.3-1,2,3,4; 15.2-2,4~~

8.1-4



Say $n = 9$ and $k = 3$, giving 3 subsequences

decision tree for k_1: $k_1!$ leaves
decision tree for k_2: $k_2!$ leaves
decision tree for k_3: $k_3!$ leaves
} $n/k \cdot k!$ leaves total

$$\frac{n}{k} \cdot k! \leq \frac{n}{k} 2^h$$

$$k! \leq 2^h$$

$\log k! \leq h$ → one subsequence

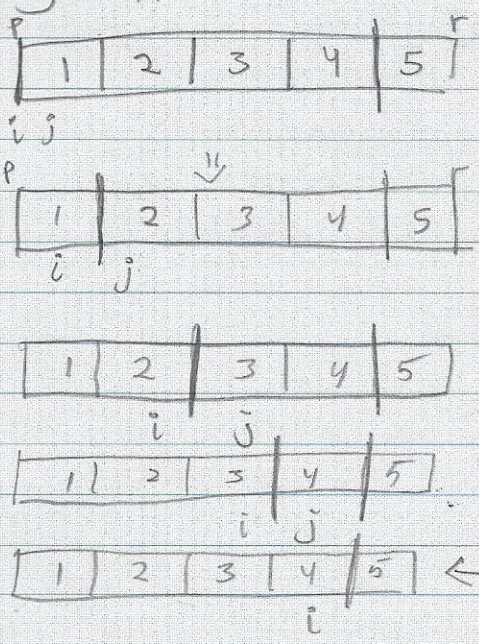
$\frac{n}{k} \cdot k \log k \leq h$

$= n \log k \leq h \Rightarrow \Omega(n \log k)$

9.2-3 RandomizedSelect(A, p, r, i)

```
while true
    q = RandomizedPartition(A, p, r)
    k = q - p + 1
    if p == r
        return A[p]
    if i == k
        return A[q]
    else if i < k
        p = q
        r = q - 1
    else
        p = q + 1
        i = i - k
```

9.3 3) Since the time to run partition the first time is $\Theta(n)$, the worst case would be if every subsequent recursion took one less than the previous recursion: $n+(n-1)+(n-2)+\ldots+(n-n)$. This would only happen if the set were already sorted:



The next recursive call to Quicksort would be called on $(A, p, r-1)$, the following on $(A, p, r-2)$ and so on.

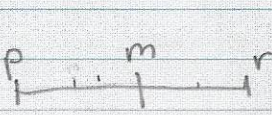
So, the worst case is

$$\sum_{i=0}^{n} n-i = \log(n!) < \log(n^n) = n\log n$$

5) - linear time 'mean' function - given k int, find kth smallest element

- Select(A, p, r, k)

1. Find median m
2. Partition set on m
3. If $m-p == k$, return $A[m-p]$
4. If $m-p > k$, Select(A, p, m)
5. If $m-p < k$, Select(A, m, r)



cn
cn
c
$\Big\rangle \Theta(n)$

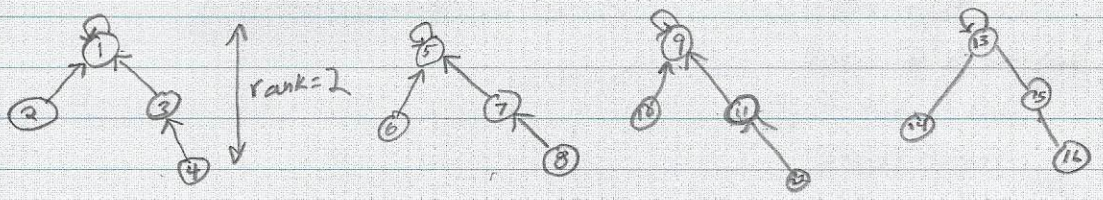
$T(n) = T(n/2) + \Theta(n) = \Theta(n)$

21.3

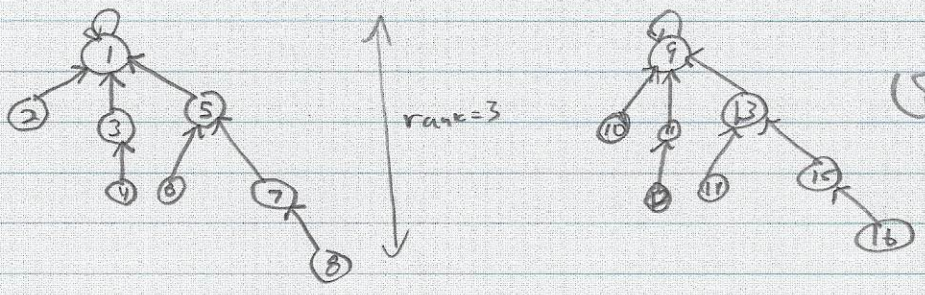
1) disjoint-set forest w/ union by rank & path compression

(3-4)

(5-6)

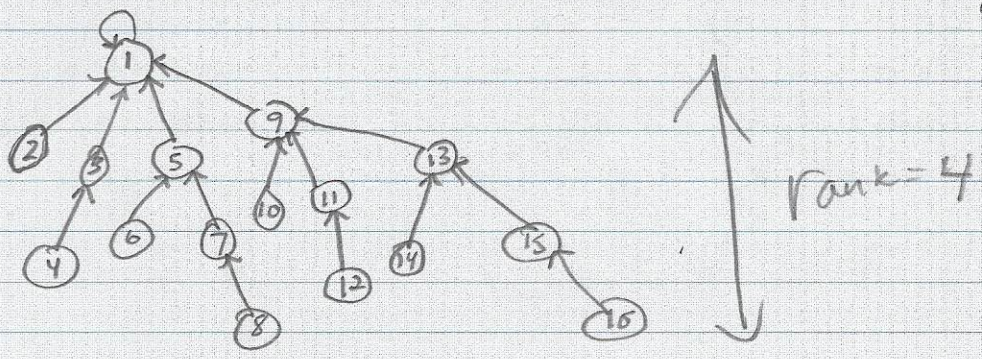


(7)



(8) Findset on X_{11} yields X_9, but does not change the structure, since X_9 is its direct parent

(9)



(Since the disjoint sets all have equal ranks when unioned, X_j is always appended to X_i and rank increases each time)

(10) FindSet(X_2) = X_1 (no change to structure)

(11) FindSet(X_9) = X_1 (no change to structure, since X_1 is X_9's direct parent)

2) FindSet(x)

```
root = x
while root ≠ root.p
    root = root.p
while x ≠ root
    next_x = x.p
    x.p = root
    x = next_x
```

get root (assuming assignment of root to root.p makes a new root.p)

assign each x's parent to root

3)

MakeSet(1) → {1} cost: 1
MakeSet(2) → {2} cost: 1
MakeSet(3) → {3} cost: 1
MakeSet(4) → {4} cost: 1
Union(1, 2) → {1, 2} cost: findset(1) + findset(2) + Link = 3
Union(2, 3) → {1, 2, 3} cost: findset(2) = 2 + findset(3) + Link = 4
Union(2, 4) → {1, 2, 3, 4} cost: findset(2) = 2 + findset(4) + link = 4
FindSet(1) → 1 : cost: 1

Total: 16 = 8log4 = nlogn

4) If each node stored an array of its children, print could traverse to the root and recursively print the children until none remained. It would cost the same amount as FindSet plus the number of nodes in the array. It would look something like this:

```
print(x)
    print x // as in print to console
    if x.children
        for i in x.children
            print(x.children[i]) // recursive call
```

The call to Union would add the child to the parent node when linking the trees together, so would not greatly impact its cost.

15.2

2) Matrix-chain-multiply(A, s, i, j)

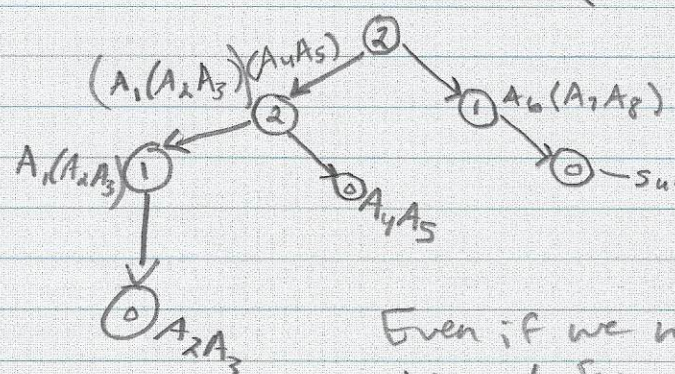
```
1  if A.length == 1      // if there is only one matrix, return it
2      return A
3  if i == (j-1)         // if i and j are adjacent, perform mult.
4      A_i = Matrix-Multiply(A_i, A_j)
5      remove A_j from A          // Keep only the product of A_i and A_j
6      return
7  Matrix-chain-Multiply(A, s, i, s[i,j])
8  Matrix-chain-Multiply(A, s, i+1, j - s[i,j] + i)
```

The right hand side i and j values are adjusted assuming that once the left-hand side (line 7) is finished recursing, it will be reduced down to one matrix. So, the right hand recursive call starts at i+1 and ends at $j - (s[i,j] - i)$, where $s[i,j] - i$ is the size of the left side.

4) A dependency graph for matrix chain multiplication of n matrices would have n-1 vertices, because there are n-1 values of k. Each vertex would have an edge pointing to each of its immediate subproblems (at least 0, at most 2). The number of edges would be n-2, since each vertex has exactly one vertex whose edge is pointing to it (i.e. parent), except for the root, which represents the last multiplication.

Example:

optimal parenthesization: $((A_1(A_2A_3))(A_4A_5))(A_6(A_7A_8))$ (last mult.)



n = 8
vertices = 7
edges = 6

Even if we moved the vertices around for an alternate paren., the number of vertices and edges would not change.