CPSC 320 Sample Midterm 2
March 2009

[12] 1. Skip Lists

[4] a. When we augment a skip list to support efficient Order-Statistics queries, we store a `count` value with each pointer of the skip list. What does `count` represent, and how is it used by `SkipListSelect`?

Solution : This value is the difference in rank (position) in the skip list between the node that contains the pointer, and the node the pointer points to.

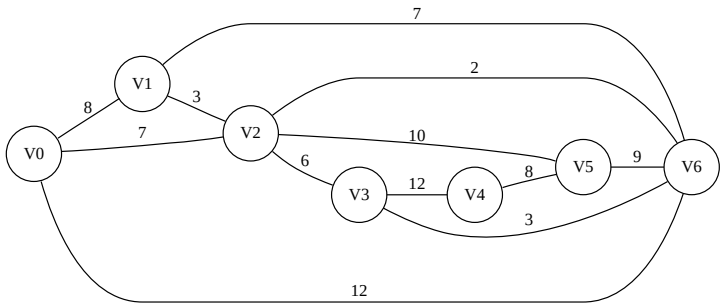
[4] b. How does algorithm `UpdatedSkipListInsert` use the `Rank` array filled by algorithm `UpdatedSkipListSearch` to recompute the `count` values associated with the pointers stored in the new node?

Solution : It uses the `Rank` array to compute the ranks of the new node (that is, `Rank[1] + 1`), and the rank of the node the pointer points to (that is, `Rank[lev] + count(Pointer[lev], lev)`). The `count` value we want will be the difference of these two ranks.

[4] c. When we analyzed the expected running time of algorithm `SkipListSearch` we looked at the path taken by the algorithm through the skip list "backwards". That is, we analyzed the expected number of left moves of the reverse path (from the node we are searching for to the head of the skip list) on each given level, instead of analyzing the expected number of right moves on each level. Why?

Solution : Because when going from right to left, we know the probability of going up (p) and the probability of going left ($1 - p$). These allow us to compute the expected number of left moves on level i. When we go from left to right, on the other hand, we do not know the probabilities of going down or right.

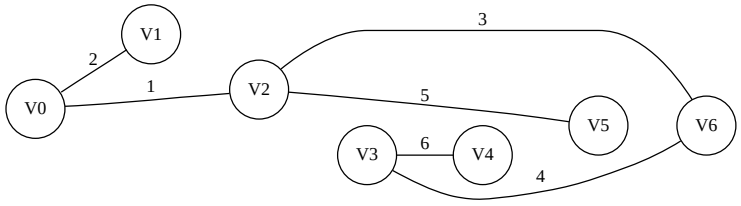
[8] 2. Show the tree constructed by Dijkstra's algorithm for the graph in the following figure, assuming that the first vertex added to the tree is vertex $V0$. Label each edge of your tree by a number that indicates the order in which the edges were added to the tree (so the first edge added will be labeled "1", the second edge added will be labeled "2", etc). Show the cost associated with each node at each stage of the algorithm.



Solution : Here is the status of the heap as the algorithm progresses:

$V0$	0						
$V1$	$+\infty$	8_{V0}	8_{V0}				
$V2$	$+\infty$	7_{V0}					
$V3$	$+\infty$	$+\infty$	13_{V2}	13_{V2}	12_{V6}		
$V4$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	24_{V3}	24_{V3}
$V5$	$+\infty$	$+\infty$	17_{V2}	17_{V2}	17_{V2}	17_{V2}	
$V6$	$+\infty$	12_{V0}	9_{V2}	9_{V2}			

and here is the resulting tree:



[8] 3. After attending tutorials and learning that algorithm `DeterministicSelect` also works if we use groups of 7 elements, but not if we use groups of 3 elements, a student decides to implement a version of the algorithm in which he makes $n/\lfloor\sqrt{n}\rfloor$ groups of $\lfloor\sqrt{n}\rfloor$ elements each. Instead of finding the median of each group by sorting the group, he decides to find it by using algorithm `DeterministicSelect`.

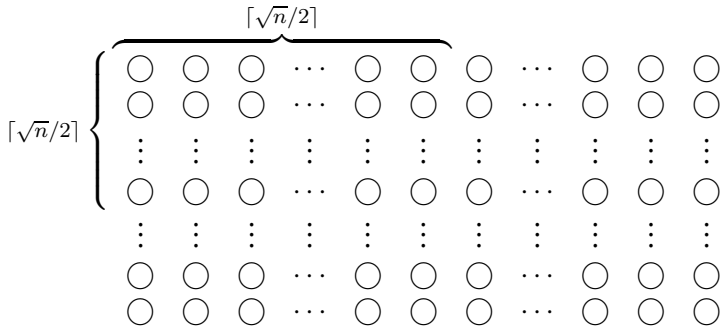
Write a recurrence relation describing the running time of this student's version of algorithm `DeterministicSelect`. You **must** explain where each term of your recurrence relation comes from. You may ignore floors and ceilings.

Solution : Let $T(n)$ represent the worst-case running time of the student's algorithm when it needs to find the i^{th} order statistics in a subarray with n elements. The student's algorithm needs to:

- Find the median of each group of $\lfloor\sqrt{n}\rfloor$ elements; this takes time in $\Theta(\sqrt{n}T(\sqrt{n}))$ — remember that we will ignore floors and ceilings.
- Find the median of the $\sqrt{n}$ elements — this takes another $T(\sqrt{n})$ time.
- Run partition using the median of the medians as pivot — this takes $\Theta(n)$ time.

In order to determine the last term, we need bounds on the number of elements of the array that are $\leq$ the pivot. Once again, we ignore the floors and ceilings, and assume that we have

exactly $\sqrt{n}$ groups of $\sqrt{n}$ elements:



There will be at least $n/4$ elements that are $\leq$ to the pivot, and so at most $3n/4$ elements are $\geq$ that pivot. Similarly (looking at the bottom-right corner of the figure) there will be at least $n/4$ elements that are $\geq$ to the pivot, and so at most $3n/4$ elements are $\leq$ that pivot. Therefore

- The algorithm recurses on at most $3n/4$ elements, which gives us a term equal to $T(3n/4)$.

Assuming we treat the cases where $n < 4$ as base cases, we thus get the recurrence

$$T(n) = \begin{cases} T(3n/4) + (\sqrt{n}+1)T(\sqrt{n}) + \Theta(n) & \text{if } n \geq 4 \\ \Theta(1) & \text{if } n < 4 \end{cases}$$

[12] 4. Consider the following problem:

> You are in charge of hiring security guards for a number of Olympic events. You have been given a list of events' starting and ending times, and need to hire as few guards as possible (they are expensive) so that every event is covered by one guard in its entirety.

The problem can be modeled as a graph-coloring problem: each vertex (event) is an interval on the X-axis (starting time, ending time), an edge joins two vertices if the corresponding events (intervals) overlap, and you want to "color" each vertex using the name of a specific guard you are hiring.

[9] a. Describe a greedy algorithm that is guaranteed to give you the minimum number of guards (colors). Hint: consider the endpoints of the intervals one at a time, in a specific order.

Solution : Throughout the execution of the algorithm, we will maintain a list of guards that have been hired, and are available at the current time.

- First we sort the starting and ending time of the events in increasing order, in such a way that the end time of an event that finishes at time x is listed before the starting time of an event that starts at time x.

- Then we consider the starting and ending times x of events in order:
  - If x is the ending time for an event e, then we add the guard assigned to e back to our available list.
  - If x is the starting time for an event e, then
    - If the list of available guards is empty, we hire a new guard and add him/her to the list.
    - We pick an arbitrary guard from the list and assign it to event e.

[3] b. What is the running time of your algorithm as a function of the number of events that need a guard?

Solution : Sorting the starting and ending times of the events take $O(n \log n)$ time, were n is the number of events. The list of available guards can be kept as a stack, so each time x is handled in constant time afterwards. This gives us a total running time in $O(n \log n)$.

[2] c. **Bonus**: Prove the correctness of your algorithm.

Solution : Suppose that we use k guards in total. We added the k^{th} guard because there were k events that all overlapped (that is, all $k-1$ other guards were still busy, and we needed a guard to cover the next event). We can not cover k events that overlap with fewer than k guards, which means that our algorithm found the smallest number of guards needed.