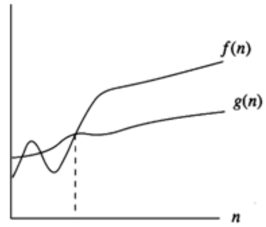
Worksheet 2 – More Asymptotic Notation

Big Ω (Asymptotic Lower Bound): Big O Flipped

Idea. Want a *function* $g(n)$ such that for

- *big* enough n,
- $0 \leq b \cdot g(n) \leq f(n)$
- where b is a *constant*.



$f(n) = \Omega(g(n))$

"Big Omega." Let $g \in \mathcal{F}$. $\Omega(g)$ is the *set* of functions $f \in \mathcal{F}$ such that

$$\exists b \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \rightarrow f(n) \geq b \cdot g(n) \geq 0$$

To show that an algorithm has a tight bound we need to prove that the worst case complexity is bounded from above ("big Oh") and from below ("big Omega"). We think of $T(n) \in \Omega(g(n))$ as the algorithms complexity $T(n)$ *growing* at least as fast as $g(n)$.

Consider INSERTION SORT again:

```
def IS (A):
    i = 1
    while (i < len(A))
        t = A[i];
        j = i;
        while (j > 0 AND A[j-1] > t)
            A[j] = A[j-1];
            j = j-1;
        A[j] = t;
        i = i+1;
```

If we think about an input A that *forces* `IS(A)` to take as many steps as possible, then we are *proving* a lower bound on the number of steps `IS` must take *in the worst case*.

Q. What is an example of a *bad* input to `IS`?

A.

If we assume each line takes *at least* 1 step, then we can determine a lower bound on the number of steps `IS` must take on A of length n. Fill in the table to the right to determine the minimum number of steps that A forces `IS` to make.

i	A[0..i] after outer loop	inner loop steps
1	n-2,n-1	loops 1 time, so $\geq 1 \cdot 3$+1 for exit
2		
3		
$\vdots$		
k		

Q. $T_{IS}(n)$ is at least how big? I.e., in the worst case, at least how many steps must `IS(A)` take?

Worksheet 3 – Augmented AVL Trees

Augment the tree Take 2

Q: How can we *augment the nodes* of AVL trees so that we can perform all our *queries* efficiently?

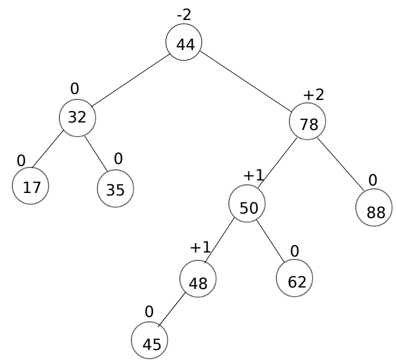
Q: What *property of subtrees* could help us with questions about *rank* ?

A.

Q: How is this related to '*rank*'?

A.

Relative Rank



Q: Now with respect to the *left subtree* rooted at x, what is the *relative* `RANK(x)` ?

A.

For example, add size fields to each node and then calculate the rank of 62.

So the *rank* of a node is related to the *size of the subtrees* rooted at neighbouring nodes.

Computing `RANK(k)` : Given *key* k, do a

- `SEARCH(k)` keeping track of the *rank of the current node.*
- Each time you go *down a level* you must:
  →
- Think of this as the "*relative*" *rank* of the key to the left of the subtree you are exploring.

Computing Rank as we Search

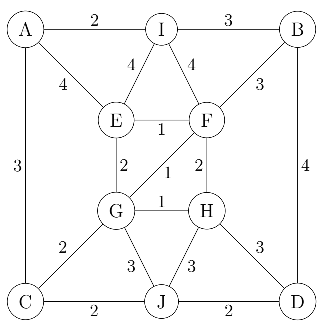
Worksheet – MST

Recall Kuskal's Algorithm:

Find an *MST* by repeatedly adding the *least weight edge* that does *not induce* a *cycle*.

1. At first, each vertex is its own small cluster (tree/set in textbook).
2. Find an edge of minimum weight, use it to merge two clusters into one.
3. Do it again…
4. In general, find an edge of minimum weight that crosses two clusters; merge them into one.

Perform Kruskal's algorithm on the following graph:



Proof of Correctness of Kruskal's

Proof by Contradiction.

Let O be an optimal minimum spanning tree.

- Since O is connected, there must exist a *unique path* p from u to v and an edge e' on p that is not in K.
- Since K did not select e' (but had the option to), $w(e') \geq w_i$.

  Case 1. $w(e') = w_i$.

  Case 2. $w(e') > w_i$.

Complexity of Kruskals

```
Kruskal(E, V)
    S := new container() for chosen edges
    PQ := min priority queue of edges and weights
    for each vertex v:
        v.cluster := {v}
    while not PQ.is_empty():
        {u,v} = PQ.extract_min():
        if u.cluster ≠ v.cluster:
            S.add({u,v})
            union(u.cluster, v.cluster)
    return S
```