We saw the algorithm for BFS last week:

\[ \text{BFS}(G, s) \]

1. \( X \leftarrow \{s\} \)
2. \( F \leftarrow \{s\} \)
3. \( i \leftarrow 0 \)
4. \( \text{while } F \neq \emptyset \)
5. \( \text{do } F' \leftarrow N_G(F) \setminus X \)
6. \( \text{label all vertices in } F' \text{ as } i + 1 \)
7. \( X \leftarrow X \cup F' \)
8. \( F \leftarrow F' \)
9. \( i \leftarrow i + 1 \)

The important operations in the above algorithm is

1. calculating \( N_G(F) = \bigcup_{u \in F} N_G(u) \),
2. calculating \( N_G(F) \setminus X \)
3. calculating \( X \leftarrow X \cup F' \).

How we calculate these depends on the data structure we use to keep the adjacency information of the graph.

**Using Sets:** If we keep the neighbors of each vertex using a set data structure, then we can simply use the difference and union operators to do the last two operations. For the first operation, we can do a reduction on the union operation since union of sets is an associative operation.

**Using Arrays:** We can also simply use sorted adjacency arrays and use merge operations to do the union and get sorted merged arrays. You then have to run a remove duplicates algorithm that uses scan. For the difference, again, one can use a modified merge where you mark each node as coming from either \( N_G(F) \) or \( X \) and then run a version of remove duplicates.
1 Analysis

The number of iterations of the while loop in the iterative code is the diameter of $G$, namely $d$. Now we just need to compute the cost of each iteration. Again, recall, that the important operations are:

1. calculating $N_G(F) = \bigcup_{u \in F} N_G(u)$,
2. calculating $N_G(F) \setminus X$
3. calculating $X \leftarrow X \cup F'$.

In order to calculate the cost of computing $N_G(F)$ we can use the following lemma.

1.1 Reduction lemma

So far we have only analyzed reduce when each operation is a constant time operation. For breadth first search, we need a more general cost for reduction, since computation of $N_G(F)$ is basically a reduction on the union operation. Consider a combine operator $f$ where the cost of combining $x$ and $y$ depends on some measure, generally size, of $x$ and $y$. Say work of the combine is $W(f(x, y))$ and the span is $S(f(x, y))$.

**Lemma 1** For any combine function $f$ and a monotone size measure $s$, if for any $x, y$,

1. $s(f(x, y)) \leq s(x) + s(y)$ and
2. $W(f(x, y)) \leq c_f (s(x) + s(y))$ for some universal constant $c_f$ depending on the function $f$,

and say $(\text{reduce } f \ S)$ is reduction based on operator $f$ of the set $S$. Then, we have

$$W(\text{reduce } f \ S) = O\left(\log |S| \sum_{x \in S} s(x)\right).$$

Using the reduction lemma, we can calculate the cost computing $N_G(F)$. The cost of union of two nodes $v_1$ and $v_2$ is $|N(v_2)| \log(|N(v_1)| + |N(v_2)|)$. Note that this function is always $O(|N(v_1)| + |N(v_2)|)$. Therefore, the conditions of the above lemma are satisfied. Therefore, the cost of merging the entire frontier is

$$W = O(|\log |F| \sum_{v \in F} |N(v)||) = O(|N_G(F)| \log |F|)$$

2
1.2 Without using the reduction lemma

Just to give you some intuition about the reduction lemma, let us analyze the reduction operation from scratch using arrays. We can simply merge the adjacency arrays for all elements in \( F \) pairwise in a tree (just as we do a reduction). Cost at every level of the tree is \( O(|N_G(F)| \log |F|) \) and the number of levels is \( \lg |F| \), giving us the total cost of \( |N_G(F)| \|N| \log |F| \).

1.3 Analysis of Breadth First Search

As we analyzed above, the work of the reduction in each iteration is \( O(|N_G(F)| \log |F|) \). Similarly, the cost of calculating \( N_G(F) \setminus X \) is also \( O(\min\{|X|, |N_G(F)|\} \log(\frac{|X| + |N_G(F)|}{\max(|X|, |N_G(F)|)})) \) and the cost of calculating \( X \leftarrow X \cup F' \) is \( O(\min\{|X|, |F'|\} \log(\frac{|X| + |F'|}{\max(|X|, |F'|)})) \). The second two quantities are also \( O(|N_G(F)| \log n) \), since \( F' \subseteq N_G(F) \).

The span of each iteration is \( O(|F| \log \max_{u \in F} |N_G(u)|) \) for the reduction. Therefore, we can upper bound it by \( \lg^2 n \).

Naive Analysis  We can upper bound \( N_G(F) \) as \( m \) in each iteration giving us a per-iteration work of \( m \log n \) and the total work \( O(dm \log n) \). The total span is \( O(d \lg^2 n) \).

Tighter Analysis of Work  We can do a tighter analysis of work by noting that each node is in exactly one frontier. Therefore, we can add up the work over the diameter.

\[
W = O \left( \sum_{i=0}^{d} \log n \sum_{v \in F_i} |N(v)| \right)
\]

\[
= O \left( \log n \sum_{i=0}^{d} \sum_{v \in F_i} |N(v)| \right)
\]

\[
= O \left( (\log n) |E| \right)
\]

\[
= O(m \log n)
\]

Now we increased the work a little bit — sequential BFS has the work of \( O(m) \). One can do better, but you need concurrent writes.

2 Binary Search Trees (BSTs)

Search trees are tree-based data structures that can be used to store and search for items that satisfy a total order. There are many types of search trees designed for a wide variety of purposes. Probably, the most common use is to implement sets and tables (dictionaries, mappings). What's
common among search trees is that they store a collection of elements in a tree structure and use values (most often called keys) in the internal nodes to navigate in search of these elements. A **binary search tree (BST)** is a search tree in which every node in the tree has at most two children.

If search trees are kept “balanced” in some way, then they can usually be used to get good bounds on the work and span for accessing and updating them. We refer to such trees as **balanced search trees**. If trees are never updated but only used for searching, then balancing is easy—it needs only be done once—but what makes balanced trees interesting is their ability to efficiently maintain balance even when updated. To allow for efficient updates, balanced search trees do not require that the trees be strictly balanced, but rather that they are approximately balanced in some way. You should convince yourself that it would be impossible to maintain a perfectly balanced tree while allowing efficient (e.g. $O(\log n)$) updates.

There are dozens of balanced search trees that have been suggested over the years, dating back at least to AVL trees in 1962. The trees mostly differ in how they maintain approximate balance. Most trees either try to maintain height balance (the two children of a node are about the same height) or weight balance (the two children of a node are about the same size, i.e., the number of elements). Here we list a few balanced trees:

1. **AVL trees.** Binary search trees in which the two children of each node differ in height by at most 1.

2. **Red-Black trees.** Binary search trees with a somewhat looser height balance criteria — no path is more than twice as long as any path.

3. **2–3 and 2–3–4 trees.** Trees with perfect height balance but the nodes can have different number of children so they might not be weight balanced. These are isomorphic to red-black trees by grouping each black node with its red children, if any.

4. **B-trees.** A generalization of 2–3–4 trees that allow for a large branching factor, sometimes up to 1000s of children. Due to their large branching factor, they are well-suited for storing data on disks.

5. **Splay trees.** Binary search trees that are only balanced in the amortized sense (i.e. on average across multiple operations).

6. **Weight balanced trees.** Trees in which the children all have the same size. These are most typically binary, but can also have other branching factors.

7. **Treaps.** A binary search tree that uses random priorities associated with every element to keep balance.

8. **Random search trees.** A variant on treaps in which priorities are not used, but random decisions are made with probabilities based on tree sizes.
9. **Skip trees.** A randomized search tree in which nodes are promoted to higher levels based on flipping coins. These are related to skip lists, which are not technically trees but are also used as a search structure.

Traditional treatments of binary search trees concentrate on three operations: search, insert and delete. Out of these, search is naturally parallel since any number of searches can proceed in parallel with no conflicts\(^1\) However, insert and delete are inherently sequential, as normally described. For this reason, in later lectures, we’ll discuss union and difference, which are more general operations that are useful for parallel updates and of which insert and delete are just a special case.

## 3 Treaps

A Treap (tree + heap) is a randomized BST that maintains balance in a probabilistic way. In the next couple lectures, we will show that with high probability, a treap with \(n\) keys will have depth \(O(\log n)\). In a Treap, a random priority is assigned to every key. In practice, this is often done by performing a hash on a key, or by assigning a different random number to each key. We will assume that each priority is unique although it is possible to remove this assumption.

The nodes in a treap must satisfy two properties:

- **BST Property.** Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).
- **Heap Property.** The associated priority satisfy the heap property. The (max) heap property requires for every node that the priority at a node is greater than the priority of its two children.

Consider the following key-value pairs:

\[(a,3), (b,9), (c, 2), (e,6), (f, 5)\]

These elements would be placed in the following treap.

\[\text{\includegraphics[width=0.5\textwidth]{treap.png}}\]

\(^1\)In splay trees and other self-adjusting trees, this is not true since a searches can modify the tree.
Theorem 2 For any set $S$ of key-value pairs, there is exactly one treap $T$ containing the key-value pairs in $S$ which satisfies the treap properties.

Proof. The key $k$ with the highest priority in $S$ must be the root node, since otherwise the tree would not be in heap order. Then, to satisfy the property that the treap is ordered with respect to the nodes’ keys, all keys in $S$ less than $k$ must be in the left subtree, and all keys greater than $k$ must be in the right subtree. Inductively, the two subtrees of $k$ must be constructed in the same manner.

Now, if you are given a set of elements with keys and priorities, how would you build a treap out of them? Take the element with the largest priority, make that the root. Then partition around that root based on keys to find the elements in the right and left subtrees and recurse. What does this remind you of?

This procedure is exactly the procedure you do for quicksort. There is a straightforward relationship between the analysis of quicksort and the height of a treap.

4 A Different Analysis of Randomized Quicksort

We now turn to analyzing quicksort. As mentioned earlier, the motivation for analyzing this now is that this analysis is a nice segue into the Treap analysis. The argument here is essentially the same as the analysis we needed to show that the expected depth of a node in a Treap is $O(\log n)$.

For the analysis, we’ll consider a completely equivalent algorithm which will be slightly easier to analyze. Before the start of the algorithm, we’ll pick for each element a random priority uniformly at random from the real interval $[0, 1]$. At each step, instead of picking a pivot randomly, we’ll instead pick the key with the highest priority (sound familiar?). Notice that once the priorities are decided, the algorithm is completely deterministic; you should convince yourself that the two presentations of the algorithm are fully equivalent (modulo the technical details about how we might store the priority values).

We’re interested in counting how many comparisons \texttt{QUICKSORT} makes. This immediately bounds the work for the algorithm because this is where the bulk of work is done. That is, if we let

$$X_n = \text{# of comparisons \texttt{QUICKSORT} makes on input of size } n,$$

our goal is to find an upper bound on $E[X_n]$ for any input sequence $S$. For this, we’ll consider the final sorted order\footnote{Formally, there’s a permutation $\pi: \{1, \ldots, n\} \to \{1, \ldots, n\}$ between the positions of $S$ and $T$.} of the keys $T = \texttt{SORT}(S)$. In this terminology, we’ll also denote by $p_i$ the priority we chose for the element $T_i$.

We’ll derive an expression for $X_n$ by breaking it up into a bunch of random variables and bound them. Consider two positions $i, j \in \{1, \ldots, n\}$ in the sequence $T$. We use the random indicator
variables $A_{ij}$ to indicate whether we compare the elements $T_i$ and $T_j$ during the algorithm—i.e., the variable will take on the value 1 if they are compared and 0 otherwise.

Looking closely at the algorithm, we have that if some two elements are compared, one of them has to be a pivot in that call. So, then the other element will be either in the left partition or the right partition, but the pivot won’t be the part of any partition. Therefore, once an element is a pivot, we never compare it to anything ever again. This gives the following observation:

**Observation 1** In the quicksort algorithm, if some two elements are compared in a QUICKSORT call, they will never be compared again in other call.

Therefore, with these random variables, we can express the total comparison count $X_n$ as follows:

$$X_n \leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} A_{i,j}$$

By linearity of expectation, we have $E[X_n] \leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[A_{i,j}]$. Furthermore, since each $A_{i,j}$ is an indicator random variable, $E[A_{i,j}] = Pr\{A_{i,j} = 1\}$. Our task therefore comes down to computing the probability that $T_i$ and $T_j$ are compared (i.e., $Pr\{A_{i,j} = 1\}$) and working out the sum.

**Computing the probability** $Pr\{A_{i,j} = 1\}$. The crux of the matter is in describing the event $A_{i,j} = 1$ in terms of a simple event that we have a handle on. Before we prove any concrete result, let’s take a closer look at the quicksort algorithm to gather some intuitions. Notice that the top level takes as its pivot $p$ the element with highest priority. Then, it splits the sequence into two parts, one with keys larger than $p$ and the other with keys smaller than $p$. For each of these parts, we run QUICKSORT recursively; therefore, inside it, the algorithm will pick the highest priority element as the pivot, which is then used to split the sequence further. With this view, the following observation is not hard to see:

**Claim 3** For $i < j$, $T_i$ and $T_j$ are compared if and only if $p_i$ or $p_j$ has the highest priority among \{$p_i, p_{i+1}, \ldots, p_j$\}.

**Proof.** We’ll show this by contradiction. Assume there is a key $T_k$, $i < k < j$ with a higher priority between them. In any collection of keys that include $T_i$ and $T_j$, $T_k$ will become a pivot before either of them. Since $T_k$ “sits” between $T_i$ and $T_j$ (i.e., $T_i \leq T_k \leq T_j$), it will separate $T_i$ and $T_j$ into different buckets, so they are never compared.

Therefore, for $T_i$ and $T_j$ to be compared, $p_i$ or $p_j$ has to be bigger than all the priorities in between. Since there are $j - i + 1$ possible keys in between (including both $i$ and $j$) and each has equal probability of being the highest, the probability that either $i$ or $j$ is the greatest is $\frac{2}{(j-i+1)}$. 


Therefore,

\[
E[A_{i,j}] = \Pr \{A_{i,j} = 1\} = \Pr \{p_i \text{ or } p_j \text{ is the maximum among } \{p_i, \ldots, p_j\}\} = \frac{2}{j - i + 1}.
\]

Hence, the expected number of comparisons made is

\[
E[X_n] \leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[A_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} (k = j - i) < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} < \sum_{i=1}^{n-1} 2H_n = O(n \log n)
\]