Chapter 1

Two Fibonaccis

$fib(n)$:
if $n = 0$ or $n = 1$ **then**
return n
else
return $fib(n-1) + fib(n-2)$
end if

We can state a recurrence for this algorithm:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + O(1) \\ &\geq fib(n-1) + fib(n-2) \\ &= fib(n) \end{aligned}$$

$fib2(n)$:
if $n = 0$ **then**
return 0
end if
create array $f[0..n]$
$f[0] \leftarrow 0, f[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
$f[i] \leftarrow f[i-1] + f[i-2]$
end for
return $f[n]$

Addition of two numbers in the preceding algorithm takes constant time until the values exceed the maximum value that can be stored in a word. After which, we need to consider how values of arbitrary length are added.

Chapter 2

Integer Multiplication

Let us consider an integer X which is composed of X_L which are the leftmost bits of X, and X_R which are the rightmost bits of X.

$$X = X_L|X_R$$

We can multiply integers X, Y as follows:

$$\begin{aligned} XY &= (2^{n/2}X_L + X_R)(2^{n/2}Y_L + Y_R) \\ &= 2^n X_L Y_L + 2^{n/2}X_L Y_R + 2^{n/2}X_R Y_L + X_R Y_R \\ &= 2^n X_L Y_L + 2^{n/2}(X_L Y_R + X_R Y_L) + X_R Y_R \end{aligned}$$

Which gives the recurrence

$$\begin{aligned} T(n) &= 4T(n/2) + O(n) \\ &\leq 4T(n/2) + cn \\ &\leq 4(4T(n/4) + cn/2) + cn \\ &\leq 4(4(4T(n/8) + cn/4) + cn/2) + cn \\ &\leq 64T(n/8) + cn(1 + 2 + 4) \\ &... \\ &\leq 4^i T(n/2^i) + cn(1 + 2 + ... + 2^{i-1}) \end{aligned}$$

Where i is the number of times we can divide n by 2, or $log_2 n$.

$$\begin{aligned} T(n) &\leq 4^{log_2 n}T(n/2^{log_2 n}) + cn(1 + 2 + ... + 2^{log_2 n - 1}) \\ &\leq n^{log_2 4}T(n/n^{log_2 2}) + cn\sum_{i=0}^{log_2 n - 1} 2^i \\ &\leq n^2 T(1) + cn2^{log_2 n} \\ &\leq n^2 T(1) + cnn^{log_2 2} \\ &\leq n^2 T(1) + cn^2 \\ &\leq n^2(T(1) + c) \\ &\leq n^2(O(1) + c) \\ &\leq O(n^2) \end{aligned}$$

Can we do better? Yes.

We need: $X_L Y_L, X_R Y_R$, and $X_L Y_R + X_R Y_L$

Observe:

$$\begin{aligned} &(X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R \\ &= X_L Y_L + X_R Y_L + X_L Y_R + X_R Y_R - X_L Y_L - X_R Y_R \\ &= X_R Y_L + X_L Y_R \end{aligned}$$

Since we must compute $X_L Y_L$ and $X_R Y_R$ anyway, this saves us an entire multiplication. Reducing our recurrence from $T(n) = 4T(n/2)+O(n)$ to $T(n) = 3T(n/2)+O(n)$.

We can solve this new recurrence as follows:

$$
\begin{aligned}
T(n) &= 3T(n/2) + O(n) \\
&\leq 3T(n/2) + cn \\
&\leq 3(3T(n/4) + cn/2) + cn \\
&\leq 3^i T(n/2^i) + cn(1 + 3/2 + ... + (3/2)^{i-1}) \\
&\leq 3^{log_2 n} T(n/2^{log_2 n}) + cn(1 + 2 + ... + (3/2)^{log_2 n - 1}) \\
&\leq n^{log_2 3} T(1) + cn \sum_{i=0}^{log_2 n - 1} (3/2)^i \\
&\leq n^{log_2 3} T(1) + cn(3/2)^{log_2 n} \\
&\leq n^{log_2 3} T(1) + cn n^{log_2 (3/2)} \\
&\leq n^{log_2 3} T(1) + cn n^{log_2 3 - log_2 2} \\
&\leq n^{log_2 3} T(1) + cn n^{log_2 3 - 1} \\
&\leq n^{log_2 3} T(1) + cn n^{log_2 3} n^{-1} \\
&\leq n^{log_2 3} (T(1) + cn/n) \\
&\leq n^{log_2 3} (T(1) + c) \\
&\leq O(n^{log_2 3})
\end{aligned}
$$

Chapter 3

Solving Recurrences

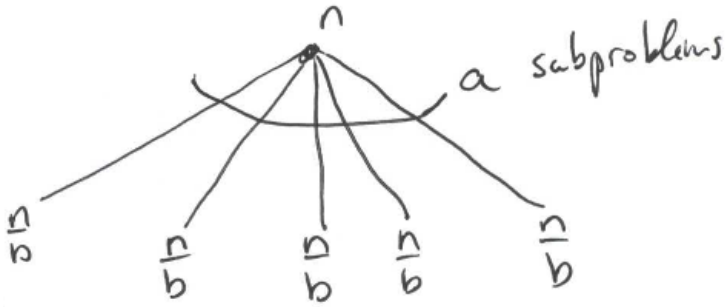
3.1 The Master Theorem

If $T(n) = aT(\frac{n}{b}) + O(n^d)$, then

$$T(n) = \begin{cases} O(n^d) & \text{if} d > log_b a \\ O(n^d logn) & \text{if} d = log_b a \\ O(n^{log_b a}) & \text{if} d < log_b a \end{cases}$$

3.2 Recursion Tree

We can reason about a recurrence of the form: $T(n) = aT(\frac{n}{b}) + f(n)$ where $a \geq 0, b > 0$ with the following recursion tree:



This tree has the following properties:

1. The number of nodes at level i: a^i
2. Work done at each node of level i: $f(\frac{n}{b^i})$
3. Number of levels: $log_b n$
4. Number of leaves: $n^{log_b a}$

We can use this information to solve the recurrence:

$$\begin{aligned} T(n) &= \sum_{i=0}^{log_b n} (\#\text{ nodes at level } i)(\text{work done at level } i) \\ &= \sum_{i=0}^{log_b n} a^i f(\frac{n}{b^i}) \end{aligned}$$

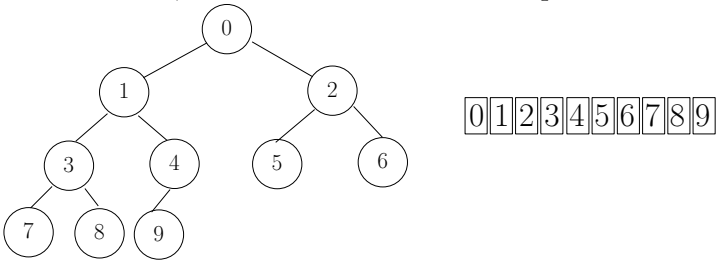
Chapter 4

Heaps

For the purposes of generality, instead of referring to elements that are "greater than" or "less than" others, we will simply say that they are "better than" or "worse than" others. For any particular ordering, the "best" element is desired first. A Heap is a binary tree that satisfies the following properties:

- The root of a heap is better than its two children (the heap property)
- The children of the root are also heaps
- A heap is a complete binary tree (only the last level may not be full, and all elements in the last level are on the left)

A heap differs from most binary trees in that it provides no particular ordering of the elements, but rather guarantees that the best element is at the root. Further, while heaps are usually discussed and defined as binary trees, they need not be implemented as such. A heap can in fact be implemented as an array with no performance reduction. To do so, we can implement a simple indexing. If an element is at the ith index in the array, it's left and right child are at the $2i + 1$th and $2i + 2$th indexes respectively. By placing the root at the 0th index, all others follow. This is depicted below.



Heaps are commonly used to implement priority queues, because they do the minimum amount of work required to keep track of the best element.

4.1 How to Implement a Heap

A heap must support the following operations

- $best()$: returns the best element in the heap
- $pop - best()$: removes the best element in the heap
- $insert(x)$: inserts x into the heap

From the definition of a heap, we know the we can easily implement $best()$ by returning the root of the heap, which should take $O(1)$ time. However the other operations are less obvious.

To $insert(x)$ recall that a heap must be complete, therefore, if a new element is added, it must be added to the left-most available space in the last row of the heap. However, the heap property may now be violated. If the new element x is worse than its parent y, the heap property is satisfied and we may stop. However if it is not satisfied we may swap x with y and recurse on x's new position. This works because we know that y is better than all of x's children, because the heap property was satisfied before x was added. Further, because x is better than y, it is also better than all of y's children. However x may still be better than its new parent, so we must recurse. If x is the new best element, it will eventually reach the root. Because heaps are complete, this operation will take $O(\log n)$ time, as this is the height of the heap.

To $pop - best()$, we may simply remove the root, however this completely destroys the entire heap. Instead, we will swap the root with the bottom-left-most element, y. Now removing the best element leaves us with a still complete tree. However the heap property has likely been violated once more. If y is better than both its children, then we may stop. However, if not, we shall swap y with its largest child and recurse on $y's$ new position. Because the element we swap with y is better than both y and the other child, the heap property has been satisfied for this sub-heap. However, the heap property may still be violated for $y's$ new sub-heap, so we must do this again, until y is the root of a valid heap. Once

more, this operation requires $O(\log n)$ time, as it must at worse traverse the entire height of the tree.

Therefore, a heap may support $best()$ in $O(1)$ time, $insert(x)$ in $O(\log n)$ time, and $pop - best()$ in $O(\log n)$ time.

4.2 Building a Heap

Now that we can support all the operations that a heap must implement, it would be nice to be able to actually construct one given a list of n elements. A naive approach is to simply call $insert(x)$ on every element in the list. However, since $insert(x)$ requires $O(\log n)$ time, this will require $O(n \log n)$ time. These seems pretty bad, considering one can find the best element in a list by brute force in $O(n)$ time. Can we achieve a construction time comparable to the brute force time? Instead of building the heap top down with $insert$, we can build it from the bottom up. Remark that a single element is a valid heap. If we were to try to build from the bottom up, we could first take the last $n/2$ elements in the list. All of these elements are their own valid and complete heaps, and we therefore do not need to do anything to them. To add the next $n/4$ elements, we simply perform the procedure we did in $pop-best$ to fix the fact that the new root might be violating the heap property, knowing that all the elements below it are valid heaps. By repeating this process until we reach the first element in the list, we will have created a valid heap on n elements.

Because we are doing very little work for the majority of the elements, we end up doing only $O(n)$ work over all, which is optimal, as this is the amount of time required to find the best element.

4.3 Heapsort

Another nice property of a heap is that once one has been implemented, it provides a very simple procedure for sort elements. A simple algorithm to do this to construct a heap on the list and then simply return $best$ and then call $pop - best$ over and over until there are no more elements in the heap. In fact, since our algorithm for building the heap is in-place and takes $O(n)$ time, and our remove method leaves the best element at the end of the array, by simply building a heap on the input array and calling $pop-best$ n times, we will be left with a reverse sorted array in $O(n \log n)$ time. This algorithm is particularly excellent because it requires no extra space, runs deterministically, and is worst-case optimal.

Chapter 5

Selection

Consider the following problem: given an array A of n elements, output the i-th smallest element of A.

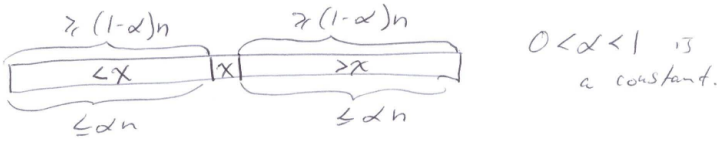
As a simple first solution, we can sort A and then return $A[i]$. Since sorting takes $O(nlogn)$ time and returning $A[i]$ takes $O(1)$ time, this solution takes $O(nlogn) + O(1) = O(nlogn)$ time.

But it should be easy to see that we can do better in specific cases like $i = 1$ or $i = n$. Simply iterate once over the array and store the minimum ($i = 1$) or maximum ($i = n$) value. Since looking at a particular element of the array takes $O(1)$ time, and we look at all n elements, this takes total $n \cdot O(1) = O(n)$ time.

We can also tell that this is optimal, because we know that to determine the i-th element, we need to look at all n elements in the array, so we have a lower bound of $\Omega(n)$ time.

But is this possible in general, for any value of i? Yes.

Suppose in linear time we can find element x such that



x is somewhere around the middle of the array, and is preceded only by elements smaller than x, and followed only by elements larger than x. We also know that there are $\geq (1-\alpha)n$ and $\leq \alpha n$ elements both before and after x.

We can calculate this x as follows:

1. Split A into groups of 5. There will be $\frac{n}{5}$ of these groups.
2. Compute the median m_j of each group M_j for $1 \leq j \leq \frac{n}{5}$.
3. Compute the median x of $m_1, m_2, ..., m_{n/5}$.

It should be clear that step 1 takes constant time, step 2 takes constant time for each group of constant size and $O(n)$ time total for all $\frac{n}{5}$ groups, and step 3 takes $T(n/5)$ time.

Claim 5.1. This x has the properties we needed above.

Proof. We know that $\frac{1}{2}$ of m_j are smaller than x, and since there are $\frac{n}{5} m_j$s, we know $\frac{n}{10}$ of m_j are $\leq x$.

So for each m_j where $m_j \leq x$

- there are 3 elements that are $\leq m_j$
- so 3 (or more) elements are $\leq x$

□

Now we must put x into its position in the array using partitioning. As a side note, partitioning is used in quicksort.

1. Find x, put it at the end
2. Partition elements around x
3. Put x into its proper position

We now have an x that satisfies the properties we needed, and it is properly located at position q in A. We are left with 3 cases:

1. If $i = q$: x is the i-th element of A.
2. If $i < q$: recurse on the subarray which is $< x$
3. If $i > q$: recurse on the subarray which is $> x$, and $i \leftarrow i - q$

This last step gives a recurrence of $T\left(\frac{7n}{10}\right)$ in the worst case because at least $\frac{3n}{10}$ elements in A are smaller than x.

5.1 Analysis

We now have the following recurrence:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

Claim 5.2. $T(n) \leq cn$

Proof.

$$\begin{aligned} T(n) &= dn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \\ &\leq dn + c\frac{n}{5} + c\frac{7n}{10} \\ &= dn + \frac{9}{10}cn \\ &= cn\left(\frac{d}{c} + \frac{9}{10}\right) && \leq cn \end{aligned}$$

As long as

$$\begin{aligned} \frac{d}{c} + \frac{9}{10} &\leq 1 \\ \frac{d}{c} &\leq \frac{1}{10} \\ 10d &\leq c \end{aligned}$$

□

5.2 What is special about 5?

First of all, we need an odd number for there to be a median. Secondly, notice:

$$\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$$

Dividing into groups of 3 doesn't work because:

$$T(n) = O(n) + T(n/3) + T(2n/3) = \Theta(nlogn)$$

Dividing into groups of 7 actually does work because:

$$T(n) = O(n) + T(n/7) + T(5n/7) = \Theta(n)$$

Chapter 6

Union-Find

Consider the following problem. Given n disjoint sets of 1 element each, perform n unions and then m queries for what set a given element is in. We will call these two operations $union(A, B)$ and $find(x)$.

6.1 Approach 1: Linked Lists

Our first approach to this problem is to describe our sets as linked lists. We know we can combine linked lists quite quickly, so this seems ideal for *union*. All we must do is have the head of B point to the tail of A, which can be done in constant time. However, linked lists are not particularly well suited for $find$. To resolve this, at every node we shall store a *backpointer* to the linked list the node is part of. This allows us to perform $find$ in constant time. However, now *union* needs some extra work. When we call $union(A, B)$, we will now walk through B and fix all of its back pointers to point to A. This will take time linear in the size of B.

Given n sets, what is the worst possible way to *union* them? Well, since the run time *union* is linear in the size of the second list, adversarial we can do is to *union* every individual element with the current unioned set. For instance, $union(D, union(C, union(A, B)))$.... Clearly this will require $O(\sum_{i=1}^{n-1} i)$, which is $O(n^2)$. If we then perform m $find$s, all of which take $O(1)$ time, we will have performed n *union*s and m $find$s in $O(n^2 + m)$ time.

6.2 Approach 2: Better Linked Lists

Somehow our first approach was naive, which allowed us to "game" the system to create a very bad result for the *union*s. To get a better result, we will make a slight modification to our *union* algorithm. Instead of blindly attaching B to the end of A, we will attach the smaller set to the larger. This fixes our adversarial approach, but is it actually better?

Consider how frequently we need to change the back pointers on an individual node. At first, it is part of a set of size one, and will have to change its pointer when unioned to a set of size ≥ 1, placing it in a set of size ≥ 2. The next time it will be changed is when it is unioned to a set of size ≥ 2, then ≥ 4 and so on. The last time will be when it is unioned to a set of size $\geq n/2$ after which we can not find another set of large enough size to change it again. Therefore each back pointer needs to be changed $O(\log n)$ times at worst. Since there are n back pointers, this new approach only require $O(n \log n)$ time to perform the unions. Since the $find$ operation is not effected, our approach now performs n *union*s and m $find$s in $O(n \log n + m)$ time.

6.3 Approach 3: Trees

Having to fix back pointers is still fairly wasteful, what if we didn't have to? Instead of implementing our sets as linked lists, we can instead use trees. Each set will be a node with either the name of the set, or a pointer to its parent. When we perform $union(A, B)$, we will simply replace the name of the shorter set with a pointer to the head of the taller set. Since we are just changing a pointer, this will only take $O(1)$ time. Our $find$ algorithm, however, will now have to walk from the node all the way to the root of the tree it is in to find out what list it is in. By a similar argument from the previous section, the height of our tree will only ever be $\log n$. Therefore this approach can support n *union*s and m $find$s in $O(n + m \log n)$ time.

6.4 Approach 4: Path Compression

Our *union* algorithm is optimal using the tree approach, but it has made our $find$ algorithm suffer. To fix this, we make a simple observation. Since our find algorithm must already walk through several nodes in the tree, once we get to the root we can, without worsening the time, relabel all of their pointers to point directly to the root. This approach is called *pathcompression*. This will make subsequent queries on these elements and their children substantially faster. The analysis of this algorithm is beyond the scope of this class, but evidently it can support n $union$s and m $find$s in $O(n + m\alpha(n))$ time. Where α is the inverse Ackermann function. Although $\alpha(n)$ tends towards infinity as n does, for any "practical" n it is at most 4. It turns out that this is in fact optimal for the union-find problem.

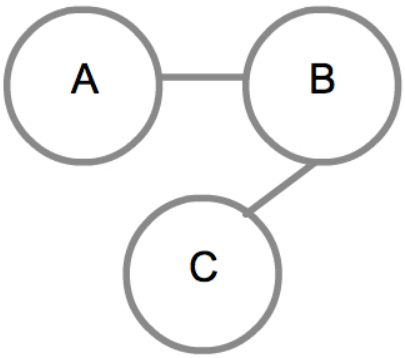
Chapter 7

Graphs

A *graph* is a set of *vertices* V and *edges* E. An edge is a pair of vertices which are said to be connected. For example:

$$V = \{A, B, C\}$$
$$E = \{(A, B), (B, C)\}$$

Which can be represented more visually:

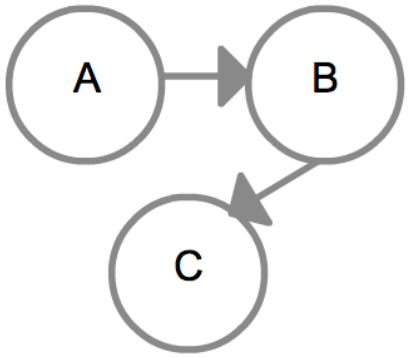


Note that the vertices are represented by labeled circles and the edges are represented by lines connecting vertices to one another.

The number of edges connecting a vertex v is called the degree of v or $deg(v)$. In the above example:

$$deg(A) = deg(C) = 1$$
$$deg(B) = 2$$

Sometimes an edge is directional, meaning (A, B) connects A to B, but not B to A. We say such an edge is incoming on B and outgoing on A. This is represented visually by an edge with an arrow at one end, indicating the direction:



Generally when we say graph, we mean undirected graph and will specify directed graph or *digraph*.

In a digraph, the number of incoming edges of v is the in-degree or $deg^-(v)$. Similarly, the number of outgoing edges is the out-degree or $deg^+(v)$.

In the above example:

$$deg^+(A) = deg^+(B) = 1$$
$$deg^-(B) = deg^-(C) = 1$$
$$deg^+(C) = deg^-(A) = 0$$

A *simple graph* is a graph which contains no edges from any vertex v to itself (v, v), called loops. A simple graph also contains no multi-edges which connect more than two vertices.

A *path* is a sequence of edges from vertex u to vertex v.

Two vertices are said to be *connected* if there exists a path between them.

Two vertices are said to be *adjacent* if they are connected by a path consisting of a single edge. A graph is said to be connected if for any two vertices, there exists a path between them.

A *complete* graph is one in which every vertex is adjacent to every other vertex.

A *subgraph* is a graph consisting of a subset of the vertices and edges in another graph.

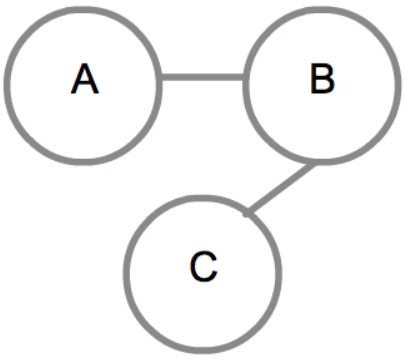
A *connected component* is a connected subgraph which does not disconnect any adjacent vertices. It should be easy to see that a connected graph has exactly one connected component.

7.1 Representation

7.1.1 Adjacency List

7.1.2 Adjacency Matrix

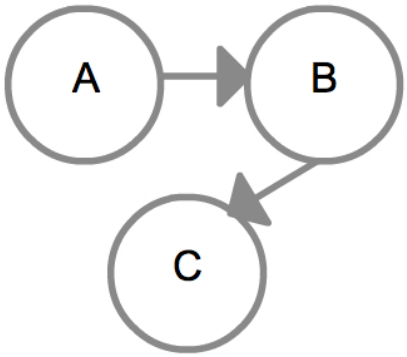
Another representation of a graph is as a square matrix with n rows and columns where the element at row i and column j is 1 if there is an edge between v_i and v_j and 0 otherwise. For example:



can be represented

$$\begin{bmatrix} & a & b & c \\ a & 0 & 1 & 0 \\ b & 1 & 0 & 1 \\ c & 0 & 1 & 0 \end{bmatrix}$$

As for directed graphs, row i column j is 1 if there exists an edge (v_i, v_j). For example:



can be represented

$$\begin{bmatrix} & a & b & c \\ a & 0 & 1 & 0 \\ b & 0 & 0 & 1 \\ c & 0 & 0 & 0 \end{bmatrix}$$

Chapter 8

Depth-First Search