Lecture Summary

In this lecture, we took a closer look at Linear Search and Binary Search, and analyzed their running times. We briefly touched upon Interpolation Search. Then, we saw how to keep an array and a linked list sorted, and the benefits (or lack thereof) we would get from doing so.

1 Searching in a list

Returning to the beginning of the semester, let’s think in a bit more detail about the problem of finding one particular element in a list. Suppose that the list is stored in an object of our datatype List. We wish to find whether a particular item is in the list, and if so, where. Let’s say that the List is implemented such that access to any element just takes time $\Theta(1)$.

The first approach is simply a linear search with a for loop. Start from the beginning of the list, run until the end (or until you find the element), and try every index in the list. The time for this is $O(n)$ for a list of length $n$, as we have to do at most a constant amount of work for each element. It is also $\Omega(n)$ in the worst case, as we may actually have to go through all of the elements to realize the element is not in the list. Even if the element we are looking for is just in the second half, we still have to go through half of the list, which still gives us $\Omega(n)$. So in total, this algorithm is $\Theta(n)$.

That doesn’t look particularly good, but if the items are unordered in the list, there is not much else we can do. However, if the items in the list (say, integers, or perhaps strings) are sorted, then we can do much better with Binary Search. The pseudo-code, which most students have probably seen before, is as follows:

```java
BinarySearch (Element x) {
  Check the median of remaining elements.
  If it equals x, we found it and are done.
  Otherwise:
    If x is smaller than the median element,
      Binary search in the left half.
    Else
      Binary search in the right half.
}
```

Next, we want to analyze the running time of binary search somewhat carefully. How many steps does it take? Every time we check at the median, the remaining array is at most half as big as the current array. (Sometimes, we may get lucky and find the element right away, but we’re looking for an upper bound on the worst case here.) If we’re already quite experienced in analyzing running times of algorithms, we can identify the fact that we have one recursive call on an array half the size as a clear sign that the running time will be $O(\log n)$.

However, most of us are not that experienced yet, so we will try to derive the exact formula more carefully. Let $T(n)$ be the worst-case running time of binary search on an array of size $n$. Then, we can express the following recurrence relation for the running time of Binary Search: $T(n) \leq 1 + T(n/2), T(1) = 1$. The explanation is that we always spend one step (or a constant number of steps) for looking at the median element, and unless we are lucky and found the element, we recursively have to check another array whose size is at most $n/2$. That takes $T(n/2)$, by definition of $T$. 

(To be more precise, we should really replace $T(n/2)$ by $[T((n - 1)/2)]$ to make sure that all numbers are integers. However, experience with running time analysis tells us that rounding up or down virtually never is the crucial part of a running time analysis, and if leaving out ceilings or floors makes the calculations easier, it’s usually worth it.)

The inequality $T(n) \leq 1 + T(n/2), T(1) = 1$ captures the running time of Binary Search quite precisely, but it does not actually give us a formula that we can easily use to “get a feel” for the running time. So we would like a closed-form solution, i.e., one without recursion or sums.

To arrive there, let’s think about how often we need to test the median element. Each time we do, the array size is divided by 2 (unless we were lucky and already found the element, but remember that we are doing worst-case analysis here). So the remaining array size after $k$ iterations is $\frac{n}{2^k}$. The recursion stops when the array size is (at most) 1, which means that we stop when $\frac{n}{2^k} \leq 1$. Solving this for $k$ gives us that we are guaranteed to stop as soon as $k > \log_2(n)$. (In fact, in this class, unless otherwise specified, all logarithms are base 2, so we will leave out the base from now on.) Since each iteration does a constant amount of work, we come up with our guess that $T(n) = \Theta(\log n)$. To make the proof of this guess work, we need to be a little more careful: for instance, for $n = 1$, the result would be $\Theta(0)$, which says that it takes no time. So we amend our guess to $T(n) \leq \log(n) + 1$.

A guess is nice, in particular if corroborated by good reasoning. But let’s actually prove this formally. Any time you prove anything about an algorithm that involves either (1) recursion, or (2) a loop (for or while), it is a good bet that somewhere in your proof, you will want to use induction. This case is no exception: we will use induction on $n$, the array size.

Our base case is $n = 1$: Here, $T(1)$ is constant (since our array has size 1), which we’ll just treat as 1. The right-hand size is $\log(1) + 1 = 0 + 1 = 1$. So the base case works.

For the induction step, we normally do induction from $n$ to $n + 1$. But notice that here, we are expressing $T(n)$ in terms of $T(\frac{n-1}{2})$, rather than in terms of $T(n-1)$. So our “regular” induction approach doesn’t work. The simplest way around it is to use strong induction instead.

The strong Induction Hypothesis states that for all $m < n : T(m) \leq 1 + \log(m)$. We have to prove that $T(n) \leq 1 + \log(n)$, and get to use the strong induction hypothesis for any numbers $m < n$. What do we know about $T(n)$? The one thing we know is the recurrence relation $T(n) \leq 1 + T(\lceil \frac{n-1}{2} \rceil)$. We can check that regardless of whether $n$ is even or odd, $\lceil \frac{n-1}{2} \rceil < n$. Therefore, we are actually allowed to apply the induction hypothesis to $\lceil \frac{n-1}{2} \rceil$, which gives us that $T(\lceil \frac{n-1}{2} \rceil) \leq 1 + \log(\lceil \frac{n-1}{2} \rceil)$. Using rules for logarithms, we can rewrite the right-hand side as $\log(2 \cdot \lceil \frac{n-1}{2} \rceil) \leq \log(n)$. Plugging that in gives us that $T(n) \leq 1 + \log(n)$, as we wanted. This finishes the induction step and thus the proof.

We could also use induction to prove that Binary Search is actually correct, in particular, that it always finds any element that is actually in the list. We didn’t get to it in class because of time constraints, but the interested student is encouraged to work through it anyway.

### 1.1 Aside: Interpolation Search

If we go back to our initial example for searching in a phone book, we might notice that as humans, we don’t exactly perform Binary Search. For instance, if we are looking for a Mr. or Ms. Algorithm, we will probably not start in the middle of the phone book. Rather, we know the range of entries to expect, and that suggests that the entry is likely close to the beginning, so we look immediately close to the start. If we don’t find it on the page — for instance because we found Mr. Boole instead, we will probably go about 1/3 of the way from the beginning to the current page.

The algorithm we’d be executing is called Interpolation Search. The way it works is as follows: we need to assume that there is some sense in which the entries can be translated into integers, so we can do basic arithmetic on them. Suppose that we are left with an array $a[\ell] \ldots a[r]$ with left and right endpoints $\ell, r$; and we are looking for an entry $x$. The range of entries is $a[\ell] \ldots a[r]$, so if things were evenly spaced, we

---

1For students sufficiently well-versed in induction proofs, it is worth noting that the “clean” way to do induction here would be to do induction on $\lceil \log(n) \rceil$ instead of $n$. We could write $k = \lceil \log(n) \rceil$. Then, our induction step would actually be from $k$ to $k + 1$, and regular induction would suffice. But if we don’t want to perform this variable substitution, strong induction is another correct way to do what we want, though perhaps a slightly bigger hammer.
would expect \( x \) to be about a fraction \( \frac{x-a[\ell]}{a[r]-a[\ell]} \) into the array. Therefore, we would next search at position \( m = \ell + (r - \ell) \cdot \frac{x-a[\ell]}{a[r]-a[\ell]} \). Then, we’d recurse just like for Binary Search.

In a sense, Interpolation Search zooms in much faster on the part of the array where \( x \) is likely located. In fact, one can prove that if the elements of the array are “roughly evenly spread” (in a precise mathematical sense which is beyond this lecture, though not particularly difficult), then Interpolation Search finds \( x \) in \( O(\log(\log n)) \) steps, which is much faster. But notice that this only works when the elements are evenly spread. It is not very difficult to construct somewhat weird inputs in which Interpolation Search does no better than Linear Search, in that it will scan the array from left to right. Those instances contain clusters of lots of people whose names start with the same letters. We won’t include it here, but it may amuse you to work out an input like that by hand.

2 Keeping a List sorted

In order to reap all those nice benefits of a sorted List (Binary Search, and maybe even Interpolation Search), we have to make sure that it remains sorted all the time. Previously, our List type had functions

\[
\begin{align*}
&\text{void set (int pos, const T & data);} \\
&T \text{ get (int pos);} \\
&\text{void insert (int pos, const T & data);} \\
&\text{void remove (int pos);} \\
\end{align*}
\]

Nothing needs to change for get, of course. And for remove, it also makes sense to just remove the item at position pos — that will not cause the List to become unsorted.

The function set really does not make sense to have. If we choose to overwrite an element, then how do we guarantee that the result is sorted? We should just not be allowed to overwrite.

Similarly, it doesn’t really make sense for us to be allowed to insert an element at a position of our choice. Instead, we should have a function \text{insert-sorted (const T & data)}, which will find the right place to insert the element by itself.

Next, we want to see how long it takes to perform the sorted insertions and subsequent lookups and searches.

1. If we use an array, then we can use Binary Search to find the place where data should go — this takes \( O(\log n) \). But after that, we have to shift/copy data to make room for the insertion, and in the worst case, this will take \( \Theta(n) \), for instance, every time the element needs to be inserted in the first half of the array. Thus, the time for inserting is \( \Theta(n + \log n) = \Theta(n) \).

In return, the get function runs in \( O(1) \), so we can now run Binary Search in time \( O(\log n) \) any time we are looking for an element. It’s a tradeoff — slower insertion vs. faster searching — but it may be worth it to keep the array sorted. In particular, it is worth it if we have many (fast) lookups of elements, and fewer (expensive) insertions. Of course, soon enough, we’ll learn about data structures that are fast for both.

An alternative would be to insert a bunch of elements first (without keeping the list sorted), and only later run a sorting algorithm on the heretofore unsorted list. As we will learn in a few weeks, an array can be sorted in \( \Theta(n \log n) \); we wouldn’t want to do that too often, but if there are stretches of time when we don’t need the array to be sorted, this may be a worthwhile alternative.

2. Using a Linked List, inserting an element in the right position only takes \( \Theta(1) \), once we know the position. But finding the right position takes \( \Theta(n) \). The reason is that the get function is really slow on linked lists because we need to scan linearly: to read position \( i \) takes \( \Theta(i) \), as we learned a few lectures ago. So linear search is actually the better choice to find the right position, and sorted insertion takes \( \Theta(n) \), just like for arrays.

What do we get in return? Absolutely nothing! Because the get function takes \( \Theta(n) \), if we were to implement Binary Search on top of a linked list implementation of a Sorted List, it would take \( \Theta(n) \)
for each step of the Binary Search, which would give a total of $\Theta(n \log n)$. In other words, Binary Search is much slower than Linear Search, and there is absolutely no point in implementing it on liked lists. For that reason, there is also pretty much no reason to keep a linked list in sorted order (unless it’s really important for printing it in order).

3 A brief introduction to graphs

In the next lecture, we will learn more about graphs, but for now, we want to learn about what we can model with them, and what they are at a high level.

A **graph** is a set of “individuals” and their “pairwise relationships”. The individuals are called **nodes** or **vertices** (singular: **vertex**), and sometimes also **points** (mostly by mathematicians). The relationships are called **edges** (or also **arcs** if they are directed), and sometimes also **lines**.

Many things in the real world can be very naturally represented by graphs. Examples include:

- Computer networks (including the Internet)
- Social networks (such as friendship, enemies, job supervision, family ties, sexual contacts, crushes, . . .). Notice that some of these relationships (such as friendship, enemies, or sexual contacts) are by definition undirected, i.e., automatically reciprocated, while others (job supervision, crushes) are not necessarily so.
- Road systems or other transportation networks (train routes, flights between cities, . . .). Again, notice that some edges go both ways (2-way streets), while others don’t.
- Financial networks, such as investments between companies or financial obligations, overlap among boards of directors, and others.
- Supervision between different branches/entities of government.
- The WWW and other hypertext.
- Many more, including physical proximity, matches played between sports teams, symbiotic or predator behavior between species.

We typically represent graphs pictorially by drawing little circles (or points, or rectangles) for nodes, and lines for edges. When edges are not reciprocated, then we use arrows to indicate the direction. The distinction is also called **directed** vs. **undirected** edges. Directed edges are also called **arcs**.

Before getting too carried away and wanting to model everything as graphs, we should keep in mind that graphs only capture **pairwise** relationships. Graphs will not distinguish between 3 people who frequently all hang out together, and 3 people who frequently hang out in pairs, in all combinations of 2 people. There is a notion of **hypergraphs** which allows for **hyperedges** that contain more than two people, but its theory is much less clean and illuminating. In this class, we only focus on graphs.