CS125 : Introduction to Computer Science

Lecture Notes #35
Exponentiation and Searching

Lecture 35 : Exponentiation and Searching

A second and third algorithm for exponentiation

The subproblem we chose for our first exponentiation algorithm – reducing the exponent by one and keeping the base the same – is not the *only* subproblem that works for the exponentiation problem. Consider what would happen if we still kept the base the same, but instead of subtracting `1` from the exponent, we divided it by `2`.

For example, if we are trying to calculate 2^{20}, how does knowing the value of 2^{10} help us? And the answer is:

$$2^{20} = 2^{10} * 2^{10}$$

So, let's try this:

$$x = 2^{10}$$
$$2^{20} = x * x$$

That is, instead of doing 20 multiplications to calculate 2^{20}, let's just do 10 multiplications to get 2^{10}, and then square the "result so far" to get 2^{20}. That's only 11 multiplications total! In fact, we could get away with even fewer multiplications, since we could calculate 2^{10} in this manner as well. 2^{10} is $2^5 * 2^5$, so we could perform 5 multiplications to obtain 2^5, and then square it (a 6th multplication) to get 2^{10}, and then square *that* result (a 7th multiplication) to get 2^{20}. In the implementations we were just discussing, we would have needed 15 multiplications to go from 2^5 to 2^{20}, but by squaring 2^5, and then squaring that square, we can go from 2^5 to 2^{20} in just *two* multiplications, which is much faster than using 15 multiplications to get there.

In our first exponentiation algorithm, our important recursive idea was:

$$base^{exp} = base * base^{exp-1}$$

now, we will instead rely on the following recursive idea instead:

$$base^{exp} = base^{exp/2} * base^{exp/2}$$

Now, this doesn't quite work for exponents that are odd numbers, if for no other reason than, mathematically, $exp/2$ is not an integer if exp is odd, and we are trying to stay with integer exponents. But our solution to this problem can be seen by considering the calculation of 2^{21}. We already know that $2^{20} = 2^{10} * 2^{10}$, and we only need to multiply 2^{20} by 2 to get 2^{21}. So, that gives us:

$$2^{21} = 2 * 2^{10} * 2^{10}$$

If we wanted to calcualte 2^{22}, that would just be multiplying by 2 one more time:

$$2^{22} = 2 * 2 * 2^{10} * 2^{10}$$

but that's just the equivalent of:

$$2^{22} = 2^{11} * 2^{11}$$

So, as long as we have an even exponent, we can just use the

$$base^{exp} = base^{exp/2} * base^{exp/2}$$

pattern, and for an odd exponent, we'll use this pattern instead:

$$base^{exp} = base * base^{(exp-1)/2} * base^{(exp-1)/2}$$

That is, we have the following algorithm as a second algorithm for calculating exponentiation, where we express $base^{exp}$ in terms of $base^{exp/2}$

```
// second algorithm for exponentiation
                  ______  1                    if exp > 0
                  |
                  |
        exp       |           exp/2          exp/2
   base      -----|----- base        *  base
                  |
                  |                            if exp > 0 and exp is even
                  |
                  |
                  |                   (exp-1)/2       (exp-1)/2
                  |_____ base  *   base          * base

                                               if exp > 0 and exp is odd
```

This is a fundamentally different way of performing the exponentiation than the previous algorithm was. However, we will still perform the same number of multiplications in the end. What we might note, though, is that in both recursive cases, we are performing the same recursive calculation twice. If we save the result the first time, we can just use it again instead of recalculating it from scratch. And this will give us a third algorithm for computing exponentiation:

```
// third algorithm for exponentiation
                  ______  1                    if exp > 0
                  |
                  |
        exp       |                             exp/2
   base      -----|-----  x * x, where x = base      , if exp > 0 and exp is even
                  |
                  |
                  |
                  |                                      (exp-1)/2
                  |      base *  x * x, where x = base         , if exp > 0 and
                  |______                                           exp is odd
```

If we actually implement the above in the way the formula implies – by calculating your recursive call once and then squaring the result – then this third algorithm will use fewer multiplications

than the previous algorithms did. The three code results we looked at for the first algorithm, were all implementations of the same algorithm, and thus did the same calculation work, just organized in different ways (some of those organizations needing many method calls, and others not needing those calls). But this is not simply a different way to code the same work; this is a completely different computational process in the first place. There is a difference between coding the same algorithm in a slightly different way, and a completely different algorithm entirely. And that is why all three code examples for algorithm 1 all performed the same number of comparisons and the same number of multiplications and the same number of subtractions, but we are already talking above about how we can save so many multplications using this approach, and we haven't even written any code yet!

Saving intermediate results is sometimes helpful, because some recursive algorithms can repeat subproblems and we'd rather not have to perform the same calculation many times. (This saving of intermediate results is called *dynamic programming*; you will explore the concept of dynamic programming much more in a later course.)

For example, consider the algorithm above that was labelled our "second exponentiation algorithm"; we can code it as follows:

```
public static int pow(int base, int exp)  // base >= 1, exp >= 0
{
   if (exp == 0)
      return 1;
   else  // exp > 0
   {
      if (exp % 2 == 0)
         return pow(base, exp/2) * pow(base, exp/2);
      else // (exp % 2 == 1)
         return pow(base, (exp - 1)/2) * pow(base, (exp - 1)/2) * base;
   }
}
```

Note that, if `exp` is odd, `(exp - 1) / 2` is equal to `(exp / 2)`, leading to the following alteration to the above code:

```
public static int pow(int base, int exp)  // base >= 1, exp >= 0
{
   if (exp == 0)
      return 1;
   else  // exp > 0
   {
      if (exp % 2 == 0)
         return pow(base, exp/2) * pow(base, exp/2);
      else // (exp % 2 == 1)
         return pow(base, exp/2) * pow(base, exp/2) * base;
   }
}
```

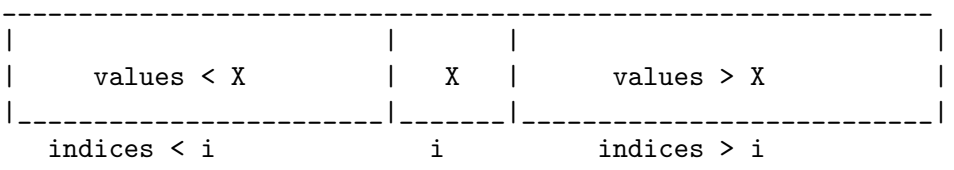
In both those cases, we are calling `pow` twice with the same arguments, thus repeating work. If we instead write code for our third exponentiation algorithm, we get the following:

```
public static int pow(int base, int exp)   // base >= 1, exp >= 0
{
   if (exp == 0)
      return 1;
   else
   {
      int intermediateResult = pow(base, exp/2);
      if (exp % 2 == 0)
         return intermediateResult * intermediateResult;
      else
         return intermediateResult * intermediateResult * base;
   }
}
```

By saving the value of our recursive call, we avoided having to run that entire recursive calculation a second time.

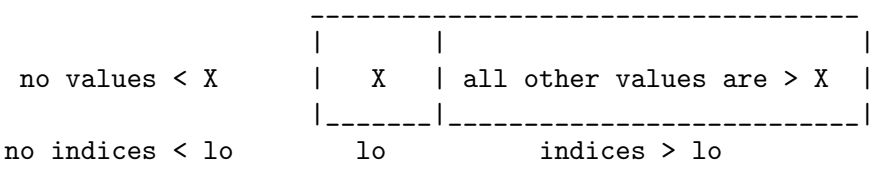
If our linear search is unsuccessful, we will end up having searched every cell in the range `lo` through `hi` – there is no other way to be sure every cell is unequal to the search key. However, as we said back in lecture `1`, if we have access to additional information about our collection or the items inside it, we can sometimes use that additional information to rule out certain items without having to inspect them.

So, we will consider the problem of having a very particular piece of extra information – knowledge that the array is *sorted*. That is, the items in our array cells increase as we move from `arr[lo]` to `arr[hi]` (or if we will allow duplicate items, then the items in the array are *non-decreasing* as we move from `arr[lo]` to `arr[hi]`). Knowing that the array is sorted allows us to improve our search algorithm tremendously. How? Well, pick some value in the array to examine. Obviously, if that value turns out to be our key, then we are done! But if it is not our key, then we only need to search one side of the value, or the other. For example, suppose the value `X`, at index `i`, is the initial value we inspect:

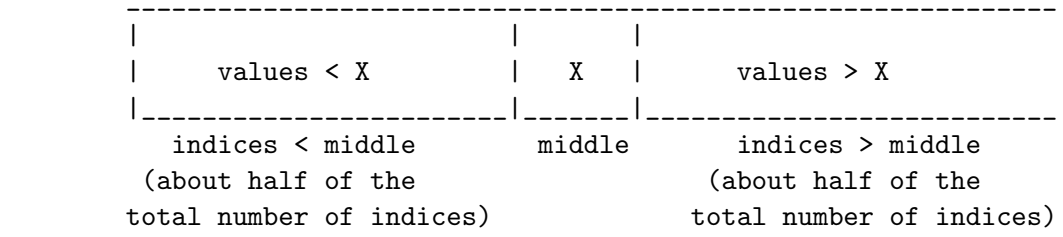


Everything less than `X` will be to the left of `X`, and everything greater than `X` will be be to the right of `X`. So, if the item we are looking for is less than `X`, we will *only* need to search the subarray to the left of `X`; nothing to the right of `X` is less than `X`, so the entire subarray to the right of `X` can be ignored. And we can use a similar reasoning if what we are looking for is greater than `X` – in that case, nothing in the subarray to the left of `X` can be greater than `X`, so the subarray to the left of `X` can be ignored entirely.

Now, if `X` were the first value in the array, then if we're really lucky, we are looking for either `X` itself, or if not, we are looking for an item less than `X`, so that the next subarray we search is of size `0` (since if `X` is the first value in the array, there is nothing to its left). But if in that case – where `X` is the first value in the array – if we are unlucky, we need to find a value larger than `X`, and in that case, we can only rule out the cell containing `X` (once we inspect that cell) but still have the entire rest of the array to the right of `X` to search:



So, inspecting the first cell of the array first is not the nicest decision. Sure, sometimes we finish our search quickly, but other times, we have the entire rest of the array to inspect, and that wouldn't be any better than linear search, and so we're not really taking advantage of our knowledge that the array is sorted. It would be nicer to inspect the *middle* cell of the array first – that way, whether we are searching for a value less than `X` or greater than `X`, we'll be able to eliminate about half the array from needing to be searched:



So, our subproblem could be to search on half the array. We could check the *middle* value of the array, rather than the first value, and if the middle value was not our search key, then we *either* recursively search the subarray to the left of the middle value, *or* we recursively search the subarray to the right of the middle value. We won't need to make both recursive calls, only one. This is because the search key – assuming it's not equal to the middle value – can only be less than that middle element *or* greater than the middle value. It cannot be both less than *and* greater than the middle element; that makes no sense.

We'll start with the same parameters we had for linear search, and the same return type:

```
public static int binarySearch(int[] arr, int key, int lo, int hi)
```

As with linear search, if `lo > hi`, then you have no values to explore and should return `-1`. Otherwise, find the middle index of the sorted array and call it `mid`. The middle index will be the average of the lowest and highest indices, since the indices are all consecutive. If you have an odd number of cells in the range `lo...hi`, then `(lo + hi)/2` would be an integer, and an existing index. If you have an even number of cells in the range `lo...hi`, then `(lo + hi)/2` will end up truncating a division. For example, if you had the indices `0` through `9`:

```
 0  1   2   3   4   5   6   7   8   9
             -----
```

then `4` and `5` are the cells in the middle. There is no one clear middle cell, since we have an even number of cells; mathematically, `(lo + hi)/2` gives us `4.5`, and we could take either `4` or `5` as the middle cell. But since, in our actual code, `(lo + hi)/2` will be truncated to `4`, then the index `4` will be the easiest choice for the middle index. That is, when we have an even number of cells in our subarray, we'll take the left of the two middle cells, as the middle cell we use in our algorithm.

Call the middle index `mid`. If what you are searching for is at that index, return the index. Otherwise, if what you are searching for is less than that value, you recursively search the subarray to the left of the middle index. Since we're looking to the left, the low index is still `lo`, but the high index is now the highest index that is still to the left of `mid` – namely, the index `mid - 1`. So, if what we are searching for is less than the value at `mid`, then we recursively search the subarray with index range `(lo...mid - 1)`. Likewise, if what we are searching for is greater than the value at `mid`, then we want to search the subarray to the right of `mid`. In that case, the low index of our new subarray will be the smallest index that is still to the right of `mid` – namely, `mid + 1` – and therefore the subarray we search recursively has the index range `(mid + 1...hi)`.

For example, suppose we are searching for `71` in the array below:

```
          ---------------------------------------
arr[i]    3   11  31  46  59  67  71  78  93  94
  -------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
lo == 0, hi == 9
```

Our `mid` index is `4`, and `71` is greater than `arr[4] == 59`, so we recursively search the subarray `5...9`:

```
                              ------------------
arr[i]    3   11  31  46  59  67  71  78  93  94
  -------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
lo == 5, hi == 9
```

Now, `mid == (5 + 9)/2 == 7`, and `71` is less than `arr[7] == 78`, so we recursively search the subarray `5...6`:

```
                              ------
arr[i]    3   11  31  46  59  67  71  78  93  94
  -------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
lo == 5, hi == 6
```

Now, `mid == (5 + 6)/2 == 5`, and `71` is greater than `arr[5] == 67`, so we recursively search the subarray `6...6`:

```
                                  --
arr[i]    3   11  31  46  59  67  71  78  93  94
  -------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
lo == 6, hi == 6
```

Now, `mid == (6 + 6)/2 == 6`, and `71` is equal to `arr[6]`, so we have found our value and we return `6`, the index where that value is located.

If our search turns out to be unsuccessful, we end up shrinking our subarray down to size `0`. For example, if we were searching for `72` instead of `71`, the first three of the four steps above would be the same; `72` is greater than `59`, less than `78`, and greater than `67`. But in the fourth step, instead of having found our value, as we did above, we instead have a search key greater than the current middle value:

```
                                    --
arr[i]    3   11  31  46  59  67  71  78  93  94
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
lo == 6, hi == 6

mid == (6 + 6)/2 == 6, and 72 is greater than arr[6] == 71
```

So, we'd search the range `mid + 1...hi` recursively. Except, `mid + 1 == 7`, so now we are searching the range `7...6` and we have `lo > hi`:

```
                                      |     <----- subarray of size 0
                                      |
arr[i]    3   11  31  46  59  67  71  78  93  94
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
lo == 7, hi == 6
```

And as previously stated, in that case we would return `-1`, indicating an unsuccessful search.

Certainly, each of our two search algorithms finds some values faster – linear search works better if our value is at the first index, for example, and binary search works better if our value is at the middle index. The catch is that we *don't know* where the value is ahead of time, so we can't really select an algorithm based on where the value is. But one thing we *can* do is to select an algorithm based on the maximum number of comparisons we might need to make before we know our return value for certain. And, in that case, binary search wins out (assuming we can use it to begin with...i.e. assuming the array is sorted) – linear search potentially searches every cell in the array, whereas binary search is continually discarding half the remaining array without even having to inspect the cells in that half of the array. Since there are so many cells that binary search won't need to look at, the maximum number of cells it *will* need to look at will be much less than the maximum number of cells that linear search will need to look at. We'll quantify this difference in a moment, but for now we can at least note that binary search, at worst, inpects significantly fewer cells than the worst possible case of linear search.

Our code that implements this algorithm appears below:

```
public static int binarySearch(int[] arr, int key, int lo, int hi)
{
   int returnVal;
   if (lo > hi)
      returnVal = -1;
   else  // lo <= hi
   {
      int mid = (lo + hi)/2;
      if (arr[mid] == key)
         returnVal = mid;
      else if (key < arr[mid])
        returnVal = binarySearch(arr, key, lo, mid - 1);
      else   // key > arr[mid]
         returnVal = binarySearch(arr, key, mid + 1, hi);
   }
   return returnVal;
}
```

Of course, we'd probably have a wrapper method around that as well, as we did for `linearSearch`:

```
public static int binarySearch(int[] arr, int key)
{
   return binarySearch(arr, key, 0, arr.length-1);
}
```

In both of these algorithms, the recursive call operates on a collection that is half the size of the original collection. The exponentiation algorithm divides the exponent in half each time, and the binary search algorithm recursively searches on half the remaining array each time.

For exponentiation, it takes constant time to go through the call once, and likewise, for binary search, it takes constant time to go through the call once. So, the question is, how many recursive calls are started? If you divide the value in half each time, how many times can you do that before the problem can no longer be divided in half?

The answer is, the number of times you can divide `n` in half before you reach `1` is $\log_2 n$. So, if you have constant time per step, and $\log_2 n$ steps, then your algorithm will take $\log_2 n$ time, that is, logarithmic time.

The first exponentiation algorithm's running time grew linearly with the change in exponent; this new exponentiation algorithm's running time grows logarithmically with the change in exponent. Similarly, the running time for linear search grew linearly with the change in the array size, but for binary search, the running time grows logarithmically with the change in the array size. So we've managed to improve the order of growth of the running time, of both exponentiation, and the searching of a sorted array.