CS125 : Introduction to Computer Science

Lecture Notes #33
Selection Sort Analysis

Lecture 33 : Selection Sort Analysis

Consider our selection sort code:

```
public static int findMinimum(int[] arr, int lo, int hi) // lo <= hi
{
   if (lo == hi)
      return lo;
   else
   {
      int indexOfMinOfRest = findMinimum(arr, lo + 1, hi);
      if (arr[lo] <= arr[indexOfMinOfRest])
         return lo;
      else
         return indexOfMinOfRest;
   }
}

public static void swap(int[] arr, int i, int j)
{
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}

public static void selectionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)
   {
      int indexOfMin = findMinimum(arr, lo, hi);
      swap(arr, lo, indexOfMin);
      selectionSort(arr, lo + 1, hi);
   }
}
```

If we imagine calling `selectionSort(...)` from `main(...)`, on a subarray of size `n`, we would end up with `n` total calls to `selectionSort`. This is because every time we hit the recursive case of `selectionSort`, we will make another `selectionSort` call, on a subarray one smaller in size than before. If our subarray is initially size `n` when we make our first call to `selectionSort`, then since each recursive call decreases the subarray size by one, we will need to make `n-1` more calls before the subarray size has reached `1`. And when the subarray size reaches `1`, that is the base case and so there would not be any additional calls to `selectionSort`. Thus, we have our first call (the one from `main(...)`, plus the `n-1` more calls to get down to a subarray of size `1` – making `n` calls total to `selectionSort`.

So, what work gets done by `selectionSort` as we go through those `n` calls? Well:

- You will sometimes need to evaluate the base-case condition to see if you are in a recursive case or a base case, i.e. `lo < hi`
- You will sometimes need to make a call to `findMinimum`
- You will sometimes need to perform a swap
- You will sometimes need to add `1` to the parameter `lo` in preparation for a recursive call (i.e. you'll need to evaluate the recursive call arguments that need evaluating)
- You will sometimes start a new method, which means creating a new method notecard and copying the argument values onto that notecard. Along with that work, is the eventual destruction of that notecard when that method call has completed. We are not speaking here of all the work that goes on while you are *on* that notecard, nor are we speaking of the work that gets done on elsewhere whenever you leave that notecard temporarily. We are only speaking here, of the work involved in creating the notecard, and some time later, the work involved in permanently destroying that notecard and returning to the previous method call.

So, how often does each of those things happen?

- Each of the `n` calls to `selectionSort(...)` must check once to see if you are at the base case or the recursive case; you'll be at the recursive case `n-1` of thoses times, and at the base case, one of those times. But you evaluate the `lo < hi` condition within `selectionSort(...)`, `n` different times in all.
- You will need to make a call to `findMinimum(...)` every time you are in the recursive case, so you'll make `n-1` calls to `findMinimum(...)` from `selectionSort(...)`.
- You will need to make a call to `swap(...)` every time you are in the recursive case, so you'll make `n-1` calls to `swap(...)` from `selectionSort(...)`.
- You will need to add `1` to the parameter `lo` in preparation for a recursive call, every time you are in the recursive case, so you'll add `1` to `lo`, `n-1` separate times from within `selectionSort(...)`.
- And as we've already explained, you will start – and later return from – `n` separate calls to `selectionSort(...)`.

So, you evaluate the base-case condition and make/return from a method call, `n` times, and you call `findMinimum(...)`, call `swap(...)`, and add `1` to `lo`, `n-1` times.

Finally, what is the order of growth of the time the computer needs to do each one of those steps once, as the array grows bigger and bigger?

- The base-case condition check: evaluating a "less than" comparison of two integers is a simple machine instruction and thus is constant time; if the array becomes ten times as large, it doesn't change the time needed to compare two integers to each other once. We will call this constant, c_{base}.
- Let's ignore `findMinimum(...)` for just a moment; we'll deal with that on its own after we figure everything else out.
- A call to `swap(...)` consists of the overhead to start and return from one method, plus four array accesses and three assignments. An array access is constant time, as we've already discussed. So is an assignment. And so is method call overhead. So, the whole package adds up to constant time for one swap operation. We will call this constant, c_{swap}.
- Adding `1` to `lo` is a simple machine operation, i.e. a single addition implemented by a single addition instruction. This does not take longer as the array grows larger, so it is constant time. We will call this constant $c_{addition}$.
- The method call set-up/destroy overhead will not depend on the size of the array, since no matter how large the array is, we are only passing a single reference to the array, as an argument. So, this cost will also be constant time. We will call this constant, $c_{methodcall}$.

And that leaves us with:

$$(c_{base} * n) + (c_{swap} * (n-1)) + (c_{addition} * (n-1)) + (c_{methodcall} * n)$$

which, after running some algebra, becomes:

$$(c_{base} + c_{swap} + c_{addition} + c_{methodcall}) * n + (c_{swap} + c_{addition}) * -1$$

If we condense our constants by creating a new constant c_W to be the sum of four of these constants:

$$c_W = c_{base} + c_{swap} + c_{addition} + c_{methodcall}$$

and another new constant c_X to be the sum of two of those terms:

$$c_X = c_{swap} + c_{addition}$$

then our algebra becomes

$$c_W * n - c_X$$

i.e., it is a linear function.

And that brings us to `findMinimum(...)`. What work gets done by `findMinimum(...)` on a subarray of size `n`? well, there will be `n-1` recursive calls and `1` base case. That said:

- Checking the `lo == hi` condition will happen in every one of the `n` method calls, and it takes constant time. Let's call this constant, $c_{equality-comp}$.
- Every one of the `n-1` recursive cases needs to add `1` to `lo`. A single addition, runs in constant time. We'll call that $c_{add-in-find-min}$.
- Every one of the `n-1` recursive cases will need to perform an assignment to write the value of the recursive acll into a variable. A single asignment, runs in constant time. We'll call that c_{assign}.
- Every one of the `n-1` recursive cases will need to compare `arr[lo]` to `arr[indexOfMinOfRest]`. That is two arrays accesses and a comparison; together, that work is still constant time, since having more array cells won't change the time it takes to read two array cells and perform one comparison. Each array access is constant time, and the comparison is constant time, and three constants added together still result in a constant. We'll call that constant $c_{min-compare}$.
- We are actually returning a value in every case of this method. The time to evaluate the (very simple) expression inside the `return` statement, and to run the `return` statement, is constant; having more array cells won't make it take longer to run one `return` statement. We'll call this constant c_{return}.
- We have to start, and eventually, return from `n` different `findMinimum(...)` calls. Each such call/return cost is a constant, as we have already discussed. We'll call this constant $c_{find-min-call}$.

And that gives us:

$$(c_{equality-comp} * n) + (c_{add-in-find-min} * (n-1)) + (c_{assign} * (n-1)) + (c_{min-compare} * (n-1)) + (c_{return} * n) + (c_{find-min-call} * n)$$

which, after running some algebra, becomes:

$$(c_{equality-comp} + c_{add-in-find-min} + c_{assign} + c_{min-compare} + c_{return} + c_{find-min-call}) * n + (c_{add-in-find-min} + c_{assign} + c_{min-compare}) * -1$$

If we condense our constants by creating a new constant c_Y to be th e sum of these six of these constants:

$$c_Y = c_{equality-comp} + c_{add-in-find-min} + c_{assign} + c_{min-compare} + c_{return} + c_{find-min-call}$$

and another new constant c_Z to be the sum of three of those terms:

$$c_Z = c_{add-in-find-min} + c_{assign} + c_{min-compare}$$

then our algebra becomes

$$c_Y * n - c_Z$$

i.e., it is a linear function.

Now, the one problem here is, even though `findMinimum(...)` is called once in each of the recursive-case method calls of `selectionSort(...)`, the "`n`" that the `findMinimum(...)` call runs on is different each time. So to get a true assessment of the situation, we need to list the `findMinimum(...)` cost of each step of `selectionSort(...)`, and then add those costs together:

selection sort step #	running time of findMinimum(...) in this step
1	cY * n - cZ
2	cY * (n-1) - cZ
3	cY * (n-2) - cZ
4	cY * (n-3) - cZ
5	cY * (n-4) - cZ
6	cY * (n-5) - cZ
.	.
.	.
.	.
k	cY * (n-k+1) - cZ
.	.
.	.
.	.
n-4	cY * 5 - cZ
n-3	cY * 4 - cZ
n-2	cY * 3 - cZ
n-1	cY * 2 - cZ

If you add up the second column, that's the work for `findMinimum(...)` over the lifetime of the `selectionSort(...)` algorithm. And that sum is just:

```
cY * (sum of numbers from 2 through n) - (n-1)*cZ
```

Now, it turns out that:

```
n + (n-1) + (n-2) + (n-3) + ... + 4 + 3 + 2 + 1 == n(n+1)/2
```

(You'll probably prove this in CS173 using mathematical induction.) So, with substitution, this becomes:

$$c_Y * ((n * (n+1)/2) - 1) - (n-1) * c_Z$$

which if simplified, becomes:

$$((c_Y/2) * n^2) + (((c_Y/2) - c_Z) * n) + c_Z$$

and that's a quadratic function. Even if we add the earlier work, we get:

$$((c_Y/2) * n^2) + (((c_Y/2) - c_Z + c_W) * n) + (c_Z - c_X)$$

which is still quadratic.

A quick way to summarize this is:

- All the work with the exception of `findMinimum(...)` is constant per step, and thus linear total.
- The `findMinimum(...)` result is linear per step, and thus when you add the `findMinimum(...)` costs for each step together, you see that it is quadratic.
- A linear plus a quadratic is a quadratic, i.e. the non-`findMinimum(...)` costs, plus the `findMinimum(...)` costs, add up to a quadratic function.

Normally you can just reason through the summary like that; you don't need to always indicate every little constant the way we did here. But it's useful to have gone through at least one very detailed analysis, just so you see all the constants and how they would add up.

What we just did was the worst-case analysis, but note that it is also the average-case, and even the best-case analysis – since `findMinimum(...)` doesn't run any faster when the minimum is the first cell of the array, `selectionSort(...)` won't run any faster even if you pass a perfectly-sorted array to `selectionSort(...)`. So, there is no difference between any of the cases. All of them have running times whose orders of growth are quadratic.