Studio 09-D

Sorting and memoization

Jia Xiaodong

22 Oct 2018

Loops

Definition

A for loop is made as such:

```
for (expr a; expr b; expr c) {
    statements
}
```

Definition

A while loop is made as such:

```
while (expression) {
    statements
}
```


Brief recap.

Something useful

Loops model iterative processes.

Consider

```
function findmin(arr) {
    let a = Infinity;
    for (let i = 0; i < length(arr); i = i + 1) {
        if (arr[i] < a) {
           a = arr[i];
        }
    }
    return a;
}
```

Loops are a good way of doing something over and over again. This is a very natural concept for humans. Consider this function to find the smallest element in an array. We simply go through the array one by one and keep track of the smallest element we have seen so far.

Checking for existence

Consider

```
function find(arr, x) {
    for (let i = 0; i < length(arr); i = i + 1) {
        if (arr[i] === x) {
            return i;
        }
    }
    return -1;
}
```

Questions to ask:

- How fast is this?
- Can we do better?

Let's look at this function to find something in an array. Again it is easy to model this as an iterative process. We go through the array one by one and compare it with what we are looking for.

How fast is this? Certainly it is in linear time.

Can we do better? No. If the list is random, then in the worst case, we must check everything. Perhaps sometimes we might be lucky and stop earlier, but on average you can imagine the number of checks would be $n/2$ times so it will still be linear.

However we have seen binary search trees before in the assignments and we know that if a list is sorted the search is much faster than linear. This is because the extra information given by the ordering guides our search.

Sorting

Easy! Consider this:

Algorithm

Input: `arr` to sort.

1. If `arr` not empty, do `m = findMin(arr)`, remove `m` from `arr` (or set it to ∞).
2. Append `m` to an array `res`, return to step 1.
3. Return `res`.

- How fast is this?
- Can we improve?

Let us try to sort things. This is one of the simplest kind of sorts. We just keep taking the smallest element out and appending it to another array. If you recall your lectures this looks vaguely like insertion sort but it is perhaps even simpler than that.

How fast does this run? Quadratic time. See if you can reason why.

Better sorts

Merge sort:

- Split arrays into two
- Sort both halves recursively
- Merge both halves in linear time
- Merge at each level takes $O(n)$ time. There are $\lg n$ levels of splits. Total: $O(n \lg n)$ time.

So the more naive approaches aren't cutting it. Enter merge sort. With the power of wishful thinking we can now sort things, fast.

With this we get from quadratic to linearithmic time. Can we go any faster? It turns out that the answer is no. There is a proof for this relying on information theory which is quite elegant. You can find it on Wikipedia. In essence, we start off from a state of complete ignorance and decide at every comparison the order of the elements we have seen. So the algorithm kind of generates a decision tree. Each node is a possible ordering of the elements. This decision tree is balanced, which roughly means that the tree branches are all about the same length (there isn't a super long branch sticking out). The worst case runtime is then the length of the longest branch, which turns out to be $n \lg n$.

Aside

I want to get the median element of an array.

- Brute force method: `findMin` $\frac{n}{2}$ times?
  - How fast is this?
- Sort the array and take the $\frac{n}{2}$-th element?
  - How fast is this?
- Can we go faster?

We have seen this before.

Select-Kth

```
function pivot(arr)...
function partition(arr, p)...
function select(k, arr) {
    p = pivot(arr);
    L, R = partition(arr, p);
    if (length(L) === k - 1)
        return arr[p];
    else if (length(L) > k - 1)
        return select(L, k);
    else if (length(L) < K - 1)
        return select(R, k - length(L) - 1);
}
```

Quicksort

```
function pivot(arr)...
function partition(arr, p)...
function quicksort(arr, lo, hi) {
    if (lo < hi) {
        const pivot = partition(A, lo, hi);
        quicksort(arr, lo, pivot - 1);
        quicksort(arr, pivot + 1, hi);
    }
}
```

There is a sort method very similar to what we have just seen known as quicksort. The idea is quite simple — we pick a pivot and put the smaller things on one side and the larger things on the other side. Then we use some wishful thinking and sort both sides. We now end up with a sorted list.

Memoization

Memoization:

- Put things down on a memo pad.
- *Referentially transparent* [1].

[1] Usually you won't have to worry about this.

Memoization is a simple technique that can result in huge time savings. All you have to do is to find a signature for your recursive calls, and save them into some array. Then in the future you can refer back to the array instead of recalculating everything.

A recursive function can only be memoized if it is referentially transparent, or if it is pure. Intuitively this means the function does not change the state of anything else. A function that reads and modifies an array that is not local to it is therefore not referentially transparent. You can see why this cannot be memoized — the function is not guaranteed to do the same thing every time it is run.

This is more of a note, though. In this course it will be quite clear when you can and when you cannot memoize something.

S10 Q1

Draw the environment during the evaluation of the following:

```
function swap(A, i, j) {
    let temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
function reverse_array(A) {
    const len = array_length(A);
    const half_len = math_floor(len / 2);
    let i = 0;
    while (i < half_len) {
        const j = len - 1 - i;
        swap(A, i, j);
        i = i + 1;
    }
}
const arr = [1, 2, 3, 4, 5];
reverse_array(arr);
arr;
```

Skipped

This question is skipped. Answers will be shared somewhere else.

S10 Q2

```
function bubblesort_array(A) {
    const len = array_length(A);
    for (let i = len - 1; i >= 1; i = i - 1) {
        for (let j = 0; j < i; j = j + 1) {
            if (A[j] > A[j + 1]) {
                const temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            } else { }
        }
    }
}
```

What is the time complexity for this function?

$O(n^2)$.

The two loops each go through the array (almost) once. The if block takes constant time to finish. So in total it runs in quadratic time.

S10 Q2

Write `bubblesort_list` that works on lists instead of arrays.

```
function bubblesort_array(A) {
    const len = array_length(A);
    for (let i = len - 1; i >= 1; i = i - 1) {
        for (let j = 0; j < i; j = j + 1) {
            // swap if larger
            if (A[j] > A[j + 1]) {
                const temp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = temp;
            } else { }
        }
    }
}
```

```
function bubblesort_list(L) {
    const len = length(L);
    for (let i = len - 1; i >= 1; i = i - 1) {
        let p = L;
        for (let j = 0; j < i; j = j + 1) {
            if (head(p) > head(tail(p))) {
                const temp = head(p);
                set_head(p, head(tail(p)));
                set_head(tail(p), temp);
            } else { }
            p = tail(p);
        }
    }
}
```


It is important to know how bubble sort works before we go in. Bubble sort works by "bubbling" the big items to the end of the list. On every pass, the largest bubble will rise to become the last element of the list. Then we repeat the procedure again, ignoring the last element because it is already in the correct position.

So what do we have to change here? The outer loop can stay the same. Its purpose is to keep track of the number of times we have bubbled something up. It does not do anything to the list itself so we don't need to touch it. The inner loop has controls over the swapping so we will have to modify that. The simplest way would be to traverse the list left to right by calling `tail`. To swap, we would just use `set_head` on the contents of the two list cells.

S10 Q3

```
function coin_change(amount, kinds_of_coins) {
    if (amount === 0) {
        return 1;
    } else if (amount < 0 || kinds_of_coins === 0) {
        return 0;
    } else {
        return coin_change(amount, kinds_of_coins - 1)
            + coin_change(amount - first_denomination(kinds_of_coins), kinds_of_coins);
    }
}
function first_denomination(kinds_of_coins) {
    return [undefined, 5, 10, 20, 50, 100][kinds_of_coins];
}
```

Can this be memoized?

Can this be memoized? Yes, it certainly can be. However a better question is: should this be memoized? Yes. This is because if you try an example run you will meet many duplicate computations. Whenever this occurs we should consider memoization to cut down on computing time.

S10 Q3

Implement the memoized version, and give its space and time complexities.

```
function mcc(n, k) {
    if (read(n, k) !== undefined) {
        return read(n, k);
    } else {
        const result = n === 0
            ? 1
            : n < 0 || k === 0
                ? 0
                : mcc(n, k - 1)
                    + mcc(n - first_denomination(k), k);
        write(n, k, result);
        return result;
    }
}
```


Here is an implementation. It is quite a simple change.

This implementation is not complete. There are some extra checks you have to perform that have been left out. For example, you need to handle the case of accessing `undefined` entries in your memo, which will otherwise crash your program. Also `write` will need some checks to prevent writing to invalid locations. However those are fairly simple, and left for you to fill in.