Big O and Simple Data Structures

Oliver-Matis Lill

February 27, 2018
Recall that \( f(x) \in O(g(x)) \) means that:

\[
|f(x)| \leq M|g(x)| \text{ for all } x > x_0 \text{ where } x_0 \in \mathbb{R}, M \in \mathbb{R}^+
\]

To prove \( f(x) \in O(g(x)) \) we just need to find constants \( x_0 \) and \( M \) where the above holds.

To prove \( f(x) \notin O(g(x)) \) we can show that for each \( M \) and \( x_0 \) there exists some \( x > x_0 \), such that \( |f(x)| > M|g(x)| \).
Example

Let’s show that:

\[ 2x^3 + 5x^2 + 4x + 9 \in O(x^3) \]

1. Note that \( x^3 > x^2 > x > 1 \) for all \( x > 1 \). Pick \( x_0 = 1 \)

2. Let’s pick \( M = 2 + 5 + 4 + 9 \), then we have
   \[ M|g(x)| = 2x^3 + 5x^3 + 4x^3 + 9x^3 \]

3. By combining (1) and (2) we see that \( |f(x)| \leq M|g(x)| \) for \( x > x_0 \) given our picks of \( x_0 \) and \( M \)
Example

Let’s show that:

\[ 2x^3 + 5x^2 + 4x + 9 \notin O(x^2) \]

1. Note that \( f(x) > x^3 \) for all \( x > 1 \)
2. Suppose by contradiction that there exist \( x_0 \) and \( M \) satisfying the condition
3. Pick \( x = cM \), where \( c \) is a positive integer such that \( cM > x_0 \)
4. Combining (1) and (3) we get that
   \[ |f(x)| > c^3 M^3 \geq c^2 M^3 = M|g(x)|, \] which contradicts (2)
Problem

Given that:

1. \( f_1(n) \in O(g_1(n)) \)
2. \( f_2(n) \in O(g_2(n)) \)
3. \( g_1(n) \in O(g_2(n)) \)

Prove that:

\[ f_1(n) + f_2(n) \in O(g_2(n)) \]
Recall that an array stores its data consecutively:

```
int a[3][4];
```

Pointer is a variable that points to a memory location.

You can allocate variable amount of memory on runtime and use pointers to refer to it:

```
int *p=malloc(6*sizeof(int));
```

```
int *q=malloc(3*sizeof(int));
```
## Exercise

Look at two implementations of a 2-D array:

```c
// Implementation A
int **a = malloc(8*sizeof(int*));
for(int i = 0; i < 8; i++) {
    a[i] = malloc(11 * sizeof(int));
}
```

```c
// Implementation B
int a[8][11];
```

1. How would memory be arranged in implementation A?
2. In A, how many memory locations would you have to read to get the value of some element a[i][j]? How many in B?
3. What are the advantages and disadvantages of A and B?
Lists

- A list is a group of objects connected into a "chain" using pointers. It might look like this:

- It’s advantage is that it allows you to insert to and delete from anywhere in $O(1)$.

- On the flip side the elements are dispersed in memory, eliminating vectorization and cache locality opportunities and making it much slower than an array in practice.
Example

```c
void insert(struct List* list, struct Node* pos, int toAdd) {
    struct Node* cur = malloc(sizeof(struct Node));
    cur->prev = pos; cur->value = toAdd;
    if(cur->prev == 0) {
        cur->next = list->first;
        list->first = cur;
    }
    else {
        cur->next = cur->prev->next;
        cur->prev->next = cur;
    }
    if(cur->next == 0)
        list->last = cur;
    else
        cur->next->prev = cur;
}
```
Lists

Exercises

- What are the differences between a list that allows traversal in only one direction and another that allows both directions?
- How does a list of 32-bit integers look like in memory in a 32-bit program? What is the memory overhead?
- Suppose we want to get the i-th element. How many operations do we have to do asymptotically (in Big O)? How does it compare to an array?
- Vectors are basically resizeable arrays
- Vectors have a capacity. They reserve some extra memory where new elements are added

```
append(4)  (3, 7) → (3, 7, 4)  Size = 3  Capacity = 5
```

- Once capacity is exceeded the following is done:
  1. A new memory block with (2x) larger capacity is allocated
  2. The old data is copied over into the new block
  3. The old block is deallocated

```
append(7)  (3, 7, 4, 6) → (3, 7, 4, 6, 7)
```
**Vectors**

- **O(1) append operation** (insertion to the last position)
- The advantage is that the data is consecutive, so in practice it's about as fast as an array with the benefit of resizeability.
- The disadvantage is that it can have significant memory overhead (up to 3x during reallocation), however usually it doesn't matter.
- Note that internally vector is just some metadata with a pointer to the data. What’s interesting is how the metadata is used to facilitate resizeability.
Example

```c
void append(struct Vector* vector, int toAdd) {
    if(vector->size == vector->capacity) {
        int* newData = malloc(2*vector->capacity*sizeof(int));
        memcpy(newData, vector->data, vector->size * sizeof(int));
        free(vector->data);
        vector->data = newData;
        vector->capacity *= 2;
    }
    vector->data[vector->size++] = toAdd;
}
```
Vectors

Exercises

- Suppose we create pointers that point to elements of a vector. What are the risks of doing that?
- Suppose we fill a vector by appending $m$ times. How many operations do we have to do asymptotically (in Big O)?
- Suppose we create a vector of vectors of integers (vector<vector<int>>). Suppose that:
  1. We add $m$ integers in total to subvectors
  2. All subvectors will end up non-empty
  3. The subvectors can be added at any time

What is the worst case complexity (Big O of the number of operations) if in the top-level vector reallocation we:
  1. Move only the metadata of the lower-level vectors
  2. Move the whole contents of the lower-level vectors
Deques

Deques are resizeable sequences like vectors, but internally they behave differently.

Deques store elements in (reasonably large) chunks and when capacity is exceeded they simply add a new chunk.

The data is more consecutive than in a list, but less than in a vector.

As a result in practice it’s much faster than a list, but somewhat slower than a vector.

The main benefit over a vector is that no reallocation is performed, so pointers to its data don’t get invalidated.