

# CSE 101: Efficient Iterative Algorithms

April 9, 2004

The next few lectures are on how to make a reasonably fast iterative algorithm faster. Some techniques to use:

- Restructuring: performing computations in a different order, to allow reuse of partial results
- Pre-processing: putting the input into a format that makes repeated steps faster
- Data structures (linked lists, stacks, heaps, trees, arrays): allow us to store and reuse previously computed values

## Example: View lengths

**Problem:** Informal statement: Given a sequence of buildings, for each building  $i$ , find the building  $j$  that obstructs building  $i$ 's view in the direction of increasing indices.

**Instance:**  $A[1..n]$  array of non-negative integers.

**Output:**  $\text{View}[1..n]$  such that  $\text{View}[i]$  is either  $\min\{j \mid j > i, A[j] \geq A[i]\}$ , or  $n + 1$  if no such  $j$  exists.

Naive algorithm:

```
1   for i = 1..n
2       j <- i + 1
3       while A[j] <= A[i] and j <= n
4           j <- j + 1
5       View[i] = j
```

This looks like  $O(n^2)$ , since it has two nested  $n$ -loops. Is it  $\Theta(n^2)$ ? Yes, it is, when the buildings are ordered in order of decreasing height.

How can we do better? Try to think about reusing work you've already done. In this case, let's restate the problem: building  $j$  is *visible* at position  $i$  if  $\forall k \in (i, j), A[k] \leq A[j]$ , i.e. if no building between  $i$  and  $j$  is taller than  $j$  (and  $A[j] > A[i]$ ). From this, we can note that for positions  $j_1 < \dots < j_q$  visible at  $i$ ,  $A[j_1] < A[j_2] < \dots < A[j_q]$ .

Let's look at what's visible in an example using this representation. This will allow us to better see the structure of the problem:

Position	Height	Visible
6	5	7
5	7	7,6
4	8	7,6,5
3	5	7,6,4
2	3	7,3
1	6	7,3,2
0	10	7,3,2,1

Note that the buildings are always added and removed in stack order. A stack maintains a changing ordered set of objects, and supports the following operations:

1. Push: Add an object to the end of the list
2. Pop: Delete the last item on the list
3. Top: Returns the last item on the list, without changing the list

There are many ways to implement a stack so that each operation is  $O(1)$ , e.g., as an array if we know the size, or as a linked list.

This suggests that we use this *data structure* to improve our algorithm:

```

1  Vis : stack of building numbers
2  View[n] = n + 1
3  push(n, Vis)
4  for i = n - 1 .. 1
5      while A[j] > A[top(Vis)]
6          pop Vis
7      if Vis not empty
8          View[i] <- top(Vis)
9      else
10         View[i] <- n + 1
11     push(i, Vis)

```

**Running time:** Using a linked-list implementation of stacks, stack operations are  $O(1)$ . Stack memory allocation in (1) takes  $O(n)$ . We can never push more than  $n$  items, since we never push the same index twice, and we can't pop more than we push. Therefore the total time for *all* pushes and pops is  $O(n)$ . The rest of the lines in the loop are  $O(1)$ , so the total time for the algorithm is  $O(n)$ .

**Summary:** How did we speed this up?

1. *Restructure* our computation to avoid repeated work, by doing subproblems first;
2. Use a *data structure* to store the results of this formerly-repeated work. How do we choose a data structure? Ask what information we want to

keep track of, how we access it, and how the structure changes. Then choose a standard structure that performs these operations as quickly as possible.

### Example: sorting

```
1   copied : array of boolean F
2   A, B : array of number
3   for i = 1..n
4       for j = 1..n
5           if A[j] < min and copied[j] = F
6               min <- A[j]
7               position <- j
8           B[i] <- min
9   copied[position] <- T
```

This is  $O(n^2)$ , which seems wasteful. To restate the approach in terms of a data structure, we need a structure  $S = \{A[i] | 1 \leq i \leq n\}$  supporting operations  $\text{min}(S)$  and  $\text{delete\_min}(S)$ . An unordered array, which we use above, is one such structure, but not a very efficient one; a more efficient alternative is a binary heap.