CS 240: Data Structure and Data Management

Charles Shen

Spring 2016, University of Waterloo

Formulas, run time, and more.

Feel free to email feedback to me at echen902@gmail.com.

Contents

1 Order Notation

1.1 Big O

O-Notation (bound from above; worst case):

$$f(n) \in O(g(n))$$

if there exists constants $c, n_0 > 0$ such that

$$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$$

1.2 Big Omega

Ω-Notation (bound from below; best case):

$$f(n) \in \Omega(g(n))$$

if there exists constants $c, n_0 > 0$ such that

$$0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0$$

1.3 Theta Bound

θ-Notation (tight bound):

$$f(n) \in \theta(g(n))$$

if there exists constants $c_1, c_2, n_0 > 0$ such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$

1.4 Little o

o-Notation (bound from above for all):

$$f(n) \in o(g(n))$$

if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that

$$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$$

1.5 Little omega

ω-Notation (bound from below for all):

$$f(n) \in o(g(n))$$

if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that

$$0 \leq cg(n) \leq cf(n) \quad \forall n \geq n_0$$

1.6 Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$.
Suppose that

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty \end{cases}$$

1.7 Relationships Between Order Notations

$$\begin{aligned} f(n) \in \theta(g(n)) &\iff g(n) \in \theta(f(n)) \\ f(n) \in O(g(n)) &\iff g(n) \in \Omega(f(n)) \\ f(n) \in o(g(n)) &\iff g(n) \in \omega(f(n)) \end{aligned}$$

$$\begin{aligned} f(n) \in \theta(g(n)) &\iff f(n) \in O(g(n)) \text{ and } f(n)\Omega(g(n)) \\ f(n) \in o(g(n)) &\iff f(n) \in O(g(n)) \\ f(n) \in o(g(n)) &\iff f(n) \notin \Omega(g(n)) \\ f(n) \in \omega(g(n)) &\iff f(n) \in \Omega(g(n)) \\ f(n) \in \omega(g(n)) &\iff f(n) \notin O(g(n)) \end{aligned}$$

1.8 Algebra of Order Notation

Transitivity: If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$.

"Maximum" Rules: Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$, then:

$$O(f(n)+g(n)) \iff O(max\{f(n), g(n)\})$$
$$\theta(f(n)+g(n)) \iff \theta(max\{f(n), g(n)\})$$
$$\Omega(f(n)+g(n)) \iff \Omega(max\{f(n), g(n)\})$$

Prove $h \in O(f+g) \iff h \in O(max(f,g))$
Suppose $h \in O(f+g)$, then $\exists c > 0, n_0 > 0$ such that

$$h \leq c(f+g) \quad \forall n \geq n_0$$
$$h \leq cf + cg$$

Without loss of generality, let $max(f,g) = f$, then $f \geq g$
Then

$$h \leq cf + cg \leq cf + cf$$
$$h \leq cf + cg \leq 2cf$$
$$h \leq cf + cg \leq 2c[max(f,g)] \quad \forall n \geq n_0$$

Let $c_1 = 2c$, then

$$h \leq c_2 max(f,g)$$

So

$$h \leq c(f+g) \leq c_2 max(f,g) \quad \forall n \geq n_0$$

Thus

$$h \in O(max(f,g)) \quad \Box$$

2 Summation Formulas

Arithmetic:

$$\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \theta(n^2) \text{ for } d \neq 0$$

Geometric:

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} & \in \theta(r^n) \text{ if } r > 1 \\ na & \in \theta(n) \text{ if } r = 1 \\ a\frac{1-r^n}{1-r} & \in \theta(1) \text{ if } 0 < r < 1 \end{cases}$$

Harmonic:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \in \theta(\log n)$$

More:

1) $\displaystyle\sum_{i=1}^{n} ir^i = \frac{nr^{n+1}}{r-1} - \frac{r^{n+1}-r}{(r-1)^2}$

2) $\displaystyle\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi}{6}$

3) for $k \geq 0, \displaystyle\sum_{i=1}^{n} i^k \in \theta(n^{k+1})$

4) $n! \in \theta(\dfrac{n^{\frac{n+1}{2}}}{e^n})$

5) $\log n! \in \theta(n \log n)$

3 Heaps

A *max-heap* is a binary tree with the following two properties (min-heap has opposite order property):

- Structural Property: All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are *left-justified.*

- Heap-Order Property: For any node i, key (priority) of parent of i is larger than or equal to key of i.

Theorem. Height of a heap with n nodes is $\theta(\log n)$.

Proof
There are $k-1$ filled levels above n^{th} node, and these levels contain 2^{k-1} nodes.
Then the n^{th} node is on the k^{th} level, $n-(2^k-1)$ from left.
Note we can always choose unique k such that $2^k \leq n \leq 2^{k+1}-1$ via induction

$$\underbrace{1+(2^0+2^1+\cdots+2^{k-1})}_{\substack{\text{Total number of nodes in a full} \\ \text{tree of height } k-1\text{, plus one,} \\ \text{left justified node } n \text{ in level } k}} \leq n \leq \underbrace{2^0+2^1+\cdots+2^k}_{\substack{\text{Total number of nodes in} \\ \text{a full tree of height } k}}$$

Since nodes are added from left to right on the bottom level, the tree with n nodes must be height k; since it is bounded on both sides by trees of height k. Thus

$$2^k \leq n \leq 2^{k+1} - 1$$

$$k \leq \log n \leq \log(2^{k-1} - 1) < \log 2^{k+1} = k + 1$$

$\therefore$ height of an heap is $\theta(\log n)$. $\square$

3.1 Insertion in Heaps

Place the new key at the first free leaf. The heap-order property might be violated, so perform a *bubble-up*. The new item bubbles up until it reaches its correct place.

function BUBBLE-UP(v)
v: a node of the heap
  while parent(v) exists **and** key(parent(v)) < key(v) **do**
    swap v and parent(v)
    $v \leftarrow$ parent(v)
  end while
end function

Priority queue realization using heap:

function HEAP-INSERT(A, x)
A: an array-based heap, x: a new item
  size(A) $\leftarrow$ size(A) + 1
  A[size(A) - 1] $\leftarrow$ x
  BUBBLE-UP(A[size(A) - 1])
end function

Running time: $O(\log n)$

3.2 Delete Max In Heaps

Maximum item of a heap is just the root node; replace the root by the last leaf (last leaf is then taken out). Perform a *bubble-down* since heap-order property may be violated.

function BUBBLE-DOWN(v)
v: a node in the heap
  while v is not a leaf **do**
    u $\leftarrow$ child of v with largest key

```
            if key(u) > key(v) then
                swap v and u
                v ← i
            else
                break
            end if
        end while
    end function
```

Priority queue realization using heap:

```
    function HEAP-DELETE-MAX(A)
    A: an array-based heap
        max ← A[0]
        swap(A[0], A[size(A) - 1])
        size(A) ← size(A) - 1
        BUBBLE-DOWN(A[0])
        return max
    end function
```

Running time: $(\log n)$

3.3 Storing Heaps in Arrays

Let H be a heap (binary tree) of n items and let A be an array of size n. Store root in $A[0]$ and continue with elements *level by level* from top to bottom, in each level *left to right*.

The *left* child (if it exists) of $A[i] = A[2i+1]$

The *right* child (if it exists) of $A[i] = A[2i+2]$

The *parent* $(i \neq 0)$ of $A[i] = A[\lfloor \frac{i-1}{2} \rfloor]$

3.4 Building Heaps

```
    function HEAPIFY(A)
    A: an array
        n ← size(A) - 1
        for i ← ⌊n/2⌋ down to 0 do
            BUBBLE-DOWN(A[i])
        end for
    end function
```

Running time: $\theta(\log n)$

4 Sorting/Random Algorithms

quick-select and the related algorithm *quick-sort* rely on two subroutines:

- *choose-pivot*(A): Choose an index i such that $A[i]$ will make a good pivot (hopefully near the middle of the order)
- *partition*(A,p): Using pivot $A[p]$, rearrange A so that
  all items less than or equal to the pivot come first,
  followed by the pivot,
  followed by all items greater than the pivot.

A *randomized algorithm* is one which relies on some random number in addition to the input.
The cost will depend on the input and the random numbers used.

4.1 Expected Running Time - Randomized Algorithms

Define $T(I, R)$ as the running time of the randomized algorithm for a particular input I and the sequence of random numbers R.
The *expected running time* $T^{(exp)}(I)$ of a randomized algorithm for a particular input I is the "expected" value for $T(I, R)$:

$$T^{(exp)}(I) = \mathbf{E}[T(I,R)] = \sum_R T(I,R) \cdot Pr[R]$$

The *worse-case expected running time* is then

$$T^{(exp)}(n) = \max_{size(I)=n} T^{(exp)}(I)$$

The worst-, best-, and average-case expected times are the same for many randomized algorithm.

4.2 Partition Algorithm

function PARTITION(A,p)
A: array of size n, p: integer such that $0 \le p < n$
    swap(A[0], A[p])
    $i \leftarrow 1$, $j \leftarrow n - 1$
    loop
        while $i < n$ **and** $A[i] \le A[0]$ **do**
            $i \leftarrow i + 1$

end while
while $j \geq 1$ **and** $A[j] > A[0]$ **do**
$j \leftarrow j - 1$
end while
if $j < i$ **then**
break
else
swap(A[i], A[j])
end if
end loop
swap(A[0], A[j])
return j
end function

4.3 Selecting a Pivot and Quick Select

4.3.1 First Idea

Always select first element in array

function CHOOSE-PIVOT-1(A)
return A[0]
end function
function QUICK-SELECT-1(A, k)
A: array of size n, k: integer such that $0 \leq k < n$
$p \leftarrow$ CHOOSE-PIVOT-1(A)
$i \leftarrow$ PARTITION(A,p)
if $i = k$ **then**
return A[i]
else if $i > k$ **then**
return QUICK-SELECT-1$(A[0, 1, \ldots, i-1], k)$
else if $i < k$ **then**
return QUICK-SELECT-1$(A[i+1, i+2, \ldots, n-1], k-i-1)$
end if
end function

Worst-Case Analysis
Recursive call could always have size $n - 1$.
Recurrence given by $T(n) = \begin{cases} T(n-1) + cn & n \geq 2 \\ d & n = 1 \end{cases}$
Solution: $T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + d \in \theta(n^2)$

Best-Case Analysis
First chosen pivot could be the k^{th} element.
No recursive calls; total cost if $\theta(n)$.

Average-Case Analysis
Assume all $n!$ permutations are equally likely.
Average cost is sum of costs for all permutations, divided by $n!$
Define $T(n,k)$ as average cost for selecting k^{th} item from size-n array:

$$T(n,k) = cn + \frac{1}{n}\left(\sum_{i=0}^{k-1} T(n-i-1, k-i-1) + \sum_{i=k+1}^{n-1} T(i,k)\right)$$

For simplicity, define $T(n) = \max_{0 \le k < n} T(n,k)$
The cost is determine by i, the position of the pivot A[0].
For more than half of the $n!$ permutations, $\frac{n}{4} \le i \le \frac{3n}{4}$

In this case, the recursive call will have length at mosts $\lfloor \frac{3n}{4} \rfloor$, for any k. The average cost is then given by

$$T(n) \le \begin{cases} cn + \frac{1}{2}\left(T(n) + T(\lfloor \frac{3n}{4} \rfloor)\right) & n \ge 2 \\ d & n = 1 \end{cases}$$

Rearranging gives:

$$\begin{aligned} T(n) &\le 2cn + T(\lfloor \frac{3n}{4} \rfloor) \\ &\le 2cn + 2c(\frac{3n}{4}) + 2c(\frac{9n}{16}) + \cdots + d \\ &\le d + 2cn \sum_{i=0}^{\infty} (\frac{3}{4})^i \in O(n) \end{aligned}$$

T(n) *must* be $\Omega(n)$, so $T(n) \in \theta(n)$.

4.3.2 Second Idea

With the probability at least $\frac{1}{2}$, the random pivot has position $\frac{n}{4} \le i \le \frac{3n}{4}$.

```
function CHOOSE-PIVOT-2(A)
    return RANDOM(n)
end function
```

```
function QUICK-SELECT-2(A, k)
A: array of size n, k: integer such that 0 ≤ k < n
    p ← CHOOSE-PIVOT-2(A)
    i ← PARTITION(A,p)
    if i = k then
        return A[i]
    else if i > k then
        return QUICK-SELECT-2(A[0, 1, ..., i − 1], k)
    else if i < k then
        return QUICK-SELECT-2(A[i + 1, i + 2, ..., n − 1], k − i − 1)
    end if
end function
```

4.3.3 Third Idea

"Medians-of-five" algorithm for pivot selection.
This *mutually recursive* approach is to be $\theta(n)$ in the worst case.

```
function CHOOSE-PIVOT-3(A)
    m ← ⌊n/5⌋ − 1
    for i ← 0 to m do
        j ← index of median of A[5i, ..., 5i + 4]
        swap(A[i], A[j])
    end for
    return QUICK-SELECT-3(A[0, ..., m], ⌊m/2⌋)
end function
function QUICK-SELECT-3(A, k)
A: array of size n, k: integer such that 0 ≤ k < n
    p ← CHOOSE-PIVOT-3(A)
    i ← PARTITION(A,p)
    if i = k then
        return A[i]
    else if i > k then
        return QUICK-SELECT-3(A[0, 1, ..., i − 1], k)
    else if i < k then
        return QUICK-SELECT-3(A[i + 1, i + 2, ..., n − 1], k − i − 1)
    end if
end function
```

4.4 QuickSort

function QUICK-SORT(A)
A: array of size n
  if $n \leq 1$ **then**
    return
  end if
  $p \leftarrow$ CHOOSE-PIVOT(A)
  $i \leftarrow$ PARTITION(A,p)
  QUICK-SORT($A[0, 1, \ldots, i-1]$)
  QUICK-SORT($A[i+1, i+2, \ldots, size(A)-1]$)
end function

Worst case:
If using first idea (Section 4.3.1) or second idea (Section 4.3.2), the worst-case running time is

$$T^{(worst)}(n) = T^{(worst)}(n-1) + \theta(n) \in \theta(n^2)$$

If using third idea (Section 4.3.3), the worst-case running them is then $\theta(n \log n)$

Best case:
Regardless of pivot idea, the best running time is

$$T^{(best)}(n) = T^{(best)}(\lfloor \frac{n-1}{2} \rfloor) + T^{(best)}(\lceil \frac{n-1}{2} \rceil) + \theta(n) \in \theta(n \log n)$$

5 Review

[A] Prove $\frac{1}{n} \in o(1)$

$\dfrac{1}{n} < c$

$1 < cn$

$n > \dfrac{1}{c}$

Choose $n_0 = \dfrac{2}{c}$

Given $c > 0$, set $n_0 = \dfrac{2}{c}$

$\dfrac{1}{n} \leq \dfrac{1}{n_0} \leq \dfrac{1}{2/c}$

$= \dfrac{c}{2}$

$< c$ ■

[B] Prove $\frac{1}{n\sqrt{n}} \notin O(\frac{1}{n^2}$

$\exists c, n_0 > 0$ such that

$\dfrac{1}{n\sqrt{n}} \leq \dfrac{c}{n^2}$

$\dfrac{n^2}{n\sqrt{n}} \leq \dfrac{n^2 c}{n^2}$

$\dfrac{n}{\sqrt{n}} \leq c$

$\sqrt{n} \leq c$

$n \leq c^2$

Contradiction. Thus $n > c^2$ ■

[C] Suppose you own n electronic devices. You have n charger cables associated with each phone. Each plug is slightly different, but you can't compare plugs with each other. You can only find which charger fits with each phone by plugging it in.
Give a randomized $O(n \log n)$ algorithm: