CS 240 — Data Structures and Data Management

Kevin James

Spring 2014

---

1 Algorithms

An **algorithm** is a step-by-step process for carrying out a set of operations given an arbitrary problem instance. An algorithm **solves** a problem if, for every instance of the problem, the algorithm finds a valid solution in finite time.

A **program** is an implementation of an algorithm using a specified programming language.

For each problem we can have several algorithms and for each algorithm we can have several programs (implementations).

In practice, given a problem:

1. Design an algorithm.
2. Assess the correctness of that algorithm.
3. If the algorithm is acceptable, implement it. Otherwise, return to step 1.

When determining the efficiency of algorithms, we tend to be primarily concerned with either the runtime or the memory requirements. In this course, we will focus mostly on the runtime.

To perform runtime analysis, we may simply implement the algorithm and use some method to determine the end-to-end time of the program. Unfortunately, this approach has many variables: test system, programming language, programmer skill, compiler choice, input selection, ... This, of course, makes manual implementation a bad approach.

An idealized implementation uses a **Random Access Machine (RAM)** model. RAM systems have constant time access to memory locations and constant time primitive operations, thus the running time is determinable (as the number of memory operations plus the number of primitive operations).

We can also generally use **order notation** to compare multiple algorithms. For the most part, we compare assuming n is very large, since for small values of n the runtime will be miniscule regardless of algorithm.

We denote the runtime of a function as $T(f(x))$, for example: $T(3 \times 4)$ may be equal to $0.8ns = 8\text{ops}$. The return value is the number of operations required in the worst-case scenario.

Example: given $T_A(n) = 1000000n + 2000000000$ and $T_B(n) = 0.01n^2$, which is 'better'? For $n < 100000000$, algorithm B is better. Since we only care about large inputs, though, we say A is better overall.

Example 1.1. *Prove that* $2010n^2 + 1388 = \mathbb{O}(n^3)$.

Proof. $\forall c > 0$, $2010n^2 + 1388 \leq cn^3$
$n > 1388 \implies 2010n^2 + 1388 \leq 2011n^2 \leq cn^3$
$2011n^2 \leq cn^3 \iff 2011 \leq cn$
$n > \frac{2011}{c} = n_0$ □

Definition 1.1. *$f(n) = \mathbb{O}(g(n))$ if there exists a positive real number c and an integer $n_0 > 0$ such that $\forall n \geq n_0$, $f(n) \leq cg(n)$.*

More concretely, we can say that $f(n) = \mathbb{O}(af(n))$ and $a'f(n) = \mathbb{O}(f(n))$. It's also worth noting that order notation is transitive (e.g. $f(n) = \mathbb{O}(g(n))$ and $g(n) = \mathbb{O}(h(n))$ implies $f(n) = \mathbb{O}(h(n))$).

We use five different symbols to denote order notation:

- o denotes a function *always less* than a given order
- $\mathbb{O}$ denotes a function *less than or equal* to a given order
- Θ denotes a function *exactly equal* to a given order
- Ω denotes a function *greater than or equal* to a given order
- ω denotes a function *always greater* than a given order

Example 1.2. *For the psuedo-function*

```
function(n):
    sum = 0
    for i=1 to n:
        for j=i to n:
            sum = sum + (i-j)^2
            sum = sum^2
    return sum
```

we find the order equation

$$
\begin{aligned}
&= \Theta(1) + \sum_{i=1}^{n}\sum_{j=i}^{n}\Theta(1) + \Theta(1) \\
&= \Theta(1)\sum_{i=1}^{n}\sum_{j=1}^{n}1 \\
&= \Theta(1)\sum_{i=1}^{n}(n-i+1) \\
&= \Theta(1)\Big(\sum_{i=1}^{n}n - \sum_{i=1}^{n}i + \sum_{i=1}^{n}1\Big) \\
&= \Theta(1)\big(n^2 + i^n + n\big) \\
&= \Theta(n^2) + \Theta(i^n) + \Theta(n)
\end{aligned}
$$

Example 1.3. *For the psuedo-function*

```
function(A,n):
    max = 0
    for i=1 to n:
        for j=i to n:
        sum = 0
        for k=1 to j:
         sum = A[k]
         if sum > max:
         max = sum
    return max
```

we find the order equation

$$
\begin{aligned}
&= \sum_{i=1}^{n}\sum_{j=i}^{n}\big(1 + \sum_{k=i}^{j}c\big) \\
&= \sum_{i=1}^{n}\sum_{j=i}^{n}c(j-i+1) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n-i+1}j \\
&= \sum_{i=1}^{n}\Theta(n-i+1) \\
&= \sum_{i=1}^{n}\Theta(i)
\end{aligned}
$$

Example 1.4. *For the psuedo-function*

```
function(n):
    sum = 0
    for i=1 to n:
     j = i
        while j >= 1:
         sum = sum + i/j
         j = j/2
    return sum
```

we find the order equation

$$\sum_{i=1}^{n}\sum_{j=1}^{\log_2 i} c = \sum_{i=1}^{n}(c\log_2 i)$$
$$= c\big(\log 1 + \log 2 + \log 3 + \cdots + \log n\big)$$

all n of our terms are below $\log n$ *half of our n terms are above* $\frac{n}{2}$

$$= \mathbb{O}(n\log n) \qquad = \Omega(\frac{n}{2}\log\frac{n}{2})$$
$$= \Omega(n\log n)$$
$$= \Theta(n\log n)$$

2 Data Types

2.1 Dynamic Arrays

Linked lists support $\mathbb{O}(1)$ insertion and deletion, $\mathbb{O}(n)$ accessing. Arrays are vice-versa.

Dynamic arrays offer a compromise: $\mathbb{O}(1)$ for both, but can only insert or delete from the end of the list.

2.2 Heaps

Given a binary tree such that all levels are filled except for the lowest (**heap structure property**) where all nodes have a lower value than that of their parent (**heap ordering property**), we have the size of the heap h given by $\log(n+1)-1 \leq h \leq \log h$ or $h = \lfloor \log n \rfloor$.

Note that we do not implement tree-like data structures as such, but rather as arrays.

Heap functions have worst case running times bounded as follows

- Heap insertion: $\Theta(\log n)$
- Heap deletion: $\Theta(\log n)$
- Top-down heap creation: $\Theta(n\log n)$

- Bottom-up heap creation: $O(n)$

2.2.1 Heap Insert

The item to-be-inserted must be placed in the only possible location to preserve integrity: i.e. the bottom level in the first available position. However, this may violate heap ordering procedure; we must perform bottom-up heap swaps until the ordering is satisfied.

To perform a bottom-up heap swap, we must

- compare node with parent
- if the node is larger than the parent
  - swap
  - recurse
- else
  - quit

These swaps can be performed in $\Theta(1)$ time and since the maximum number of swaps is equal to the height of the heap we have an overall time complexity of $\Theta(\log n)$. This worst case would occur when the item to-be-inserted is larger than anything else on the heap, since it would need to swap all the way to the top.

2.2.2 Heap Delete

The item to-be-deleted can simply be removed form our heap; however, we must replace this vacated position with the lowest item in our heap. This satisfies the structural integrity of the heap. Afterward, we must follow top-down heap swapping in order to satisfy ordering integrity.

To perform a top-down heap swap, we must

- compare node with children
- if the node is smaller than at least one child
  - swap with the largest child
  - recurse
- else
  - quit

Again, these swaps can be performed in $\Theta(1)$ time and we can perform no more than $\lfloor \log n \rfloor$ swaps. As such, we have worst case time complexity $\Theta(\log n)$ when the deleted item is at the top of the heap and the replacement item is the smallest in the heap.

2.2.3 Heap Creation

Given an array in no particular order, our goal is to adjust the ordering such that we have a valid heap.

The obvious approach is to call the heap insertion method once for each item in our input array. This is the **top-down** approach to heap creation.

This can generally done in place, as we can treat the input array as a structurally valid but orderingly incorrect binary tree. As such, each insertion operation simply 'ignores' values to its right. Since we are calling out $\Theta(\log n)$ insertion method n times, we see that top-down heap creatoin has runtime complexity $\Theta(n \log n)$.

More formally, we see we have an upper bound such that the cost of the swaps is proportional to the depth of the node (i.e. which level it is on) and the number of nodes to process. This depth is given by $O(\log n)$ and the number of nodes by n, so we have $O(n \log n)$ as our upper bound. Our lower bound is given as such: the worst case occurs when the input array is sorted in ascending order (i.e. requires we perform the maximal number of swaps for each insertion). In this case, we would be required to swap $\lfloor \log n \rfloor$ times for each of our n insertions.

Lemma 2.1. *At least half of the nodes in a heap are in the last two levels (have depth $d \geq h - 1$)*

Based on lemma 2.1, we can ignore all levels above the bottom two when computing the lower bound. The worst case number of swaps at level $h - 1$ or h is at least $h - 1$, thus the overall number of swaps $s \geq \frac{n}{2}(h - 1)$ which is $\Omega(n \log n)$.

We can also creation a heap in the **bottom-up** fashion as such: the entire array is transformed into a heap in one iteration, by finding the elements from largest to smallest and inserting them in that order.

We begin in the same way as for top-down creation, by placing our array into a incomplete binary tree (or associating our already-made array with a binary tree). Now, we start with the element in the last position and perform a top-down swap operation for each element in the array (recursing backwards).

This is very similar to the top-down heap creation method, but has different boundaries.

Since our swaps can now be made in the opposite direction, our lowest level of the tree no longer needs to swap and our second lowest level can only possible swap once. As such, over half of our tree has a maximal swapping runtime of $O(1)$ and the remainder of the tree has $O(\log n)$, whereas the top-down approach uses the constant boundary $O(\log n)$.

More concretely, the total number of swaps we can perform is equal to

$$\begin{aligned} \sum_{i=0}^{h} i 2^{h-i} &< n \sum_{i=0}^{\infty} \frac{i}{2^i} \\ &< 2n \end{aligned}$$

and thus we have the upper bound of $O(n)$.

The lower bound is at least $\Omega(n)$ (since we must run our heap creation on n elements), thus we have an overall runtime complexity of $\Theta(n)$.

2.2.4 Heap Sort

The heap sort algorithm is

```
heapSort(A, n):
    H = new Heap(n)
    for i in range(0, n - 1):
        heapInsert(H, A[i])
    for i in range(0, n - 1):
        A[n-1 - i] = heapDeleteMax(H, n-1)
```

or given our "heapify" algorithm

```
heapSort(A, n):
    H = heapify(A, n)
    for i in range(0, n - 1):
        A[n-1 - i] = heapDeleteMax(H, n-1)
```

which gives us a complexity of

$$\begin{aligned} O(n) + \sum_{i=0}^{n-1} \Theta(\log n) &= O(n) + \Theta(n \log n) \\ &= \Theta(n \log n) \end{aligned}$$