

31251 – Data Structures and Algorithms

Week 7

Luke Mathieson

- Heaps
- gtrioSn
- Striogn
- Soritgn
- Sortign
- Sorting

Heaps

We have seen Binary Search Trees:

- Keeps things in a complete order.
- Mostly easier to maintain the ordering than an array or linked list ($O(\log n)$ insertion and search).
- Can go wrong.

We can ameliorate the worst-case behavior with more a complicated data structure (AVL Tree, Red Black Tree, etc.).

What if we don't really need a complete order?

A Heap uses a binary tree to maintain a weak ordering:

- In a min-heap, the value stored at a vertex is smaller than the value at both its children.
- In a max-heap, the value stored at a vertex is greater than the value at both its children.

This property then holds for the entire subtree under any given vertex.

This means the thing at the root is the min/max.

Why not just keep everything fully ordered?

- Sometimes all we need is the min or max element.
- It can be done with less effort than something like an AVL-tree.
- We can do it in $O(\log n)$ time guaranteed.

What do we need to make this work?

- Keep track of the next available leaf.
- A way to insert that maintains the order.
- Restrict removal to the root only.

- ① Insert new element at next available leaf.
- ② Keep swapping with the parent to maintain the ordering.

- ① Copy root to temp. variable.
- ② Swap root value and last leaf value.
- ③ Delete the last leaf.
- ④ Swap the root value down to maintain ordering.

How do we store this information?

Remember the array embedding of a binary tree?

It's perfect for a heap!

- We know exactly where the last leaf/next available leaf is (it's just the next spot in the array).
- We can take an array and turn it into a heap without using any extra space!

Also available as a separate file with the lecture notes. (Plus a proper fancy version)

```
void insert(int i) {
    if (next_leaf >= capacity) resize();
    array[next_leaf++] = i;
    bubble_up(next_leaf - 1);
}

void bubble_up(int p){
    if (p > 0 && array[p] < array[parent(p)]) {
        swap(p, parent(p));
        bubble_up(parent(p));
    }
}
```

```
int remove() {  
    int temp = array[0];  
    array[0] = array[--next_leaf];  
    bubble_down(0);  
    return temp;  
}
```

```
void bubble_down(int p){  
    if (has_left(p)) {  
        int smallest = left(p);  
        if (has_right(p) && array[right(p)] < array[left(p)])  
            smallest = right(p);  
        if (array[p] > array[smallest]) {  
            swap(p, smallest);  
            bubble_down(smallest);  
        }  
    }  
}
```

Sorting

- Sorting is one of the basic algorithmic problems.
 - Given a list of elements, we want a permutation that's in order according to some comparator.
- Despite being an obvious problem, sorting efficiently is not necessarily as easy.
- It even still attracts attention as a problem to solve!

Comparison Based Sorting

Comparison Based Sorting

- Without knowing anything special about the input, we must compare elements to each other.
 - Hence the name “Comparison Based Sorting”.
- If we are limited to comparing elements to each other, we have a definite lower bound on performance:
 - ① For a list of n distinct items, there are $n!$ arrangements, only one of which is sorted.
 - ② If the algorithm takes $f(n)$ steps, and each step distinguishes two cases, it can distinguish at most $2^{f(n)}$ cases.
 - ③ So we need $2^{f(n)} \geq n!$, then doing some algebra,
$$f(n) \geq \log(n!) \Rightarrow f(n) \geq n \log n - n \log_2 e + O(\log n) \Rightarrow f(n) \in \Omega(n \log n)$$

Many of the algorithms you may know are comparison based sorts:

- Bubble sort.
- Insertion sort.
- Selection sort.
- Quick sort.
- Merge sort.

```
bubbleSort(int a[]) {  
    //assume array is length n  
    bool swapHappened;  
    do {  
        swapHappened = false;  
        for (int i = 0; i < n; i++){  
            if (a[i] > a[i+1]){  
                swap(a[i], a[i+1]);  
                swapHappened = true;  
            }  
        }  
    } while (swapHappened);  
}
```

- Easy to code.
- Bad at almost everything else.
- Best case time: $O(n)$, Average case: $O(n^2)$, Worst case: $O(n^2)$.
- Only $O(1)$ extra space need though!

```
insertionSort(int a[]){  
    for (int i = 1; i < n; i++){  
        int x = a[i];  
        int pos = i-1;  
        while (pos >= 0 && a[pos] > x){  
            a[pos + 1] = a[pos];  
            pos--;  
        }  
        a[pos + 1] = x;  
    }  
}
```

- Has the same complexity profile as Bubble sort ($O(n^2)$ worst case, $O(1)$ extra space).
- Turns out to be observably faster.
- This is actually a Divide-and-Conquer algorithm.
 - Just a really bad use of one.
- By minimizing the number of comparisons, it reduces the hidden constants and beats Bubble sort.

```
selectionSort(int a[]) {  
    for (int i = 0; i < n; ++i) {  
        int min_pos = i;  
        for (int j = i + 1; j < n; ++j) {  
            if (a[j] < a[min_pos]) min_pos = j;  
        }  
        swap(a[i], a[min_pos]);  
    }  
}
```

- $O(n^2)$ time and $O(1)$ space again, between Bubble Sort and Insertion Sort (usually).
- Sort of the inverse of Insertion Sort - finds the next smallest thing, rather than the place for the next thing.
- Pretty simple to implement.

- This is actually a decent sorting algorithm!
- It's a Divide-and-Conquer algorithm.
- But it has more complex code (it's that trade-off again!).

```
quicksort(int a[], int low, int high){  
    if (low < high){  
        p = partition(a, low, high);  
        quicksort(a, low, p);  
        quicksort(a, p+1, high);  
    }  
}
```

```
int partition(int a[], int low, int high){  
    int pivot = a[high];  
    int part = low - 1;  
    for (int j = low; j < high; j++){  
        if (a[j] < pivot){  
            part++;  
            swap(a[part], a[j]);  
        }  
    }  
    swap(a[part+1], a[high]);  
    return part+1;  
}
```

- $O(n \log n)$ average case time!
- But still $O(n^2)$ worst case...
 - It all depends on the pivot value in partition. In the worst case, this is just Insertion sort.
- Typically fast in practice. Lots of heuristics for picking a pivot value well.

- Also a divide and conquer algorithm.
- Very fast.
- Also more complicated code.

```
int[] mergesort(int a[]){  
    if (a.length == 1) return a;  
  
    int left[] = a[0..n/2];  
    int right[] = a[n/2+1 ..n-1];  
  
    return merge(mergesort(left), mergesort(right));  
}
```

```
int[] merge(int left[], int right[]){
    int merged[left.length + right.length];
    int i = 0;
    int lpos = 0;
    int rpos = 0;
    while (i < merged.length){
        if (left[lpos] < right[rpos]){
            merged[i] = left[lpos++];
        }
        else {
            merged[i] = right[rpos++];
        }
        i++;
    }
    return merged;
}
```

- We finally achieve $O(n \log n)$ worst case running time.
- Uses $O(n)$ extra space though.
 - But by complicating the code, we can make this $O(1)$ space!
- Also happens to be highly parallelisable.

- Uses a Heap data structure (not the same as heap memory).
- Has $O(n \log n)$ worst case time, and $O(1)$ extra space usage!
- Good for devices with small caches (embedded things etc.).
- Not so parallelisable.
- Just turn the array into a min-heap, and remove things until it's empty!

Non-Comparison Based Sorting

Sorting without directly comparing elements

- If we know something more about the data, we can sometimes do better.
- For example, if the data has hierarchical structure.
 - Like digits in a number, characters in a string for lexicographic ordering...

- ① Split your data range into sub-ranges (buckets).
- ② Go through the array and put each element in the proper bucket.
- ③ Recursively sort the buckets (using bucket sort, or something else).
- ④ Merge the buckets back together.

- If the bucket's range is 1 value, this becomes Counting Sort.
- If you use two buckets, it's a version of Quick Sort.
- Worst case time: $O(n^2)$, but average case: $O(n + k)$ where k is the number of buckets.
 - So if $k \approx n$... $O(n)$ sorting! (If everything comes out right...)
- $O(nk)$ space though...
- If you do it with the digits of the number, you get...

- 1 Start with the most significant digit, and work down to the least significant.
- 2 For the current digit split the elements into b buckets, where b is the base of the number system you are using.
- 3 Recursively sort each bucket.
- 4 Merge the buckets.

- Suuuuuper fast for small integers.
- Worst case complexity is $O(dn)$ where d is the number of digits in your numbers.
- If you have a lot of different numbers, $d > \log n$ (maybe a looooot bigger), so we don't beat comparison based sorting really, but...
- If all your numbers are small (maybe many repeats if you have a lot of numbers), this is $O(n)$.
- It's space is "good" too $O(d + n)$.

Hardware Based Sorting

- One last little example – we can trade off processors for speed.
- If we don't mind using lots of “CPUs”, we can sort really fast in parallel.
- These are called sorting networks, their speed is governed by their depth – $O((\log n)^2)$ depth is achievable, but you need polynomially many “CPUs”.