Part 3

1 Sorting

For any comparison based sorting algorithm $\mathcal{A}$ and any $n \in \mathbb{N}$ large enough there exists an input of length n that requires $\mathcal{A}$ to perform $\Omega(n \log n)$ comparisons

1.1 Bucket Sort

Listing 1: BucketSort($a_1, ..., a_n \in \{0, \dots K-1\}, n \geqslant 2$)

```
1 create array C[0,...,K-1] and initialise each entry C[i] to zero
2 for i=1 to n do
3     increment value C[a_i] by one
4 end for
5 for i=0 to K-1 do
6     for j-1 to C[i] do
7         print i
8     end for
9 end for
```

- This puts elements with key i into the ith bucket, then empties one bucket after another.
- Its running time is $O(n + k)$
- This means that if K is small then the running time is $o(n \log n)$, apparently beating our lower bound.
- The reason for this is that bucket sort doesn't do any comparisons
- Also as K gets really large it becomes the worst sorting algorithm we've seen

1.2 Radix Sort

- With bucket sort if the range of our items is large, then we need a large number of buckets.
- An improvement is to not look at item values but "one level below"

The idea of radix sort:

- Have as many buckets as you've got different digits, so for base 10, 10 buckets
- Repeatedly bucket sort by given digit
- The number of rounds will depend on values, but number of buckets depends only on number of different digits

Radix sort has running time $\Theta(d \cdot n)$ where d is the number of rounds
In comparison to bucket sort in range $\{0, ..., K-1\}$

- Radix sort $d = \log_{10} K$, giving $\Theta(n \cdot \log K)$
- Bucket sort $\Theta(n + K)$

2 Searching and Selecting

Suppose:

- You are given an array of n numbers
- The numbers are in sorted order
- The n numbers are pairwise distinct
- The number to be searched for is one of the numbers in the array

Listing 2: int trivial_search (int A[1...n], int x)

```
p=1
while (A[p] != x) do
    p=p+1
end while
return p
```

2.1 Binary Search

Same assumptions as before

1. Peek right into the middle of the given array, at position $p = \lceil n/2 \rceil$
2. If $A[p] = x$ then we're lucky and done and can return p
3. Otherwise, if x is greater than A[p], we may focus our search on the elements to the right of A[p] and completely ignore anything to its left. Vice versa with if x is less than A[p]
4. Then recursively call this function on the smaller list until we find the element

Listing 3: int search (int A[1..n], int left, int right, int x)

```
if (right == left and A[left] != x)
    handle error; leave function
p = middle-index between left and right
if (A[p] == x) then
return p
// here come the recursive calls (if x not yet found)
if (x > A[p]) then
    return search(A,p+1,right,x) // in right half
else // x < A[p]
    return search(A,left,p-1,x) // in left half
```

The initial call would be `search(A,1,n,x)`

For binary search we have the recurrence

$$T(n) = T(n/2) + O(1) = O(\log n)$$

for large n, which is an awful lot quicker

2.2 Selection

So far, we had assumed that

- The input is sorted
- We are looking for the position p of an element of given value x

Now we're changing the setup

- Input is unsorted
- We're looking for value of the ith smallest element in the input

A solution to this is to sort the input in time $O(n \log n)$ and return the ith element from the left

2.3 QuickSelect

This has much the same idea as QuickSort

- Recursive
- `Partition()` function for selecting pivot and partitioning into LOW and HIGH
- not two recursive calls to sort but now only one: dividing into the part (LOW or HIGH) where we know the ith smallest element is to be found

Listing 4: QuickSelect (int A[1..n], int left, int right, int i)

```
1  if (left == right) then
2      return A[left]
3  else
4      // rearrange/partition in place
5      // return value "pivot" is index of pivot element
6      // in A[] after partitioning
7      pivot = Partition (A, left, right)
8      // Now:
9      // everything in A[left...pivot-1] is smaller than pivot
10     // everything in A[pivot+1...right] is bigger than pivot
11     // the pivot is in correct position w.r.t. sortedness
12     if (i == pivot) then
13         return A[i]
14     else if (i < pivot) then
15         return QuickSelect (A, left, pivot-1, i)
16     else // i > pivot
17         return QuickSelect (A, pivot+1, right, i)
18     end if
19 end if
```

In the worst case this will be just as slow as worst case QuickSort

$$T(n) = T(n-1) + O(n) \text{ which means } T(n) = \Theta(n^2)$$

However choosing the pivot at random makes it $O(n)$, this is just expectation though, and won't be true all the time

2.4 Median-of-Medians

This has a guaranteed linear time complexity

1. If length(A)$\leqslant$5, then sort and return the ith smallest
2. Divide n elements into $\lfloor n/5 \rfloor$ groups of 5 elements each, plus at most one group containing the remaining n mod 5 elements
3. Find the median of each of the $\lceil n/5 \rceil$ groups by sorting each one, and then picking median from sorted group elements
4. Call **select** recursively on set of $\lceil n/5 \rceil$ medians found above, giving median-of-medians, x
5. Partition the entire input around x. Let k be the number of elements on low side plus one
   - x is the kth smallest element, and
   - there are n-k elements on high side of the partition
6. if i=k, return x. Otherwise use Select recursively to find ith smallest element of low side if $i < k$, or (i-k)th smallest on high side if $i > k$

3 Trees

Tree - A connected graph without cycles

Trees can be used to store data in a similar manner to a linked list, so each node has:

- Pointer to parent (or NIL, or to itself, if root)
- Pointer to left child (or NIL, or to itself, if there isn't one)
- Pointer to right child (or NIL, or to itself, if there isn't one)
- Payload

Trees are good as they have fast insert, lookup and delete operations

3.1 Binary Search Tree

Binary Search Tree - A tree in which no node has more than two children

The one crucial property of a BST is that you must build and maintain the tree such that it's true for every node v of the tree that:

- All elements in its left sub-tree are "smaller" than v
- All elements in its right sub-tree are "bigger" than v

Different ways to do tree traversals:

- In order
  - Recurse into left sub-tree, print payload, recurse into right sub-tree
- Pre-order
  - Print payload, recurse into sub-trees
- Post-order
  - Recurse into sub-trees, print payload

3.2 RedBlack trees

All simple BST operations take $O(h)$ time, where h is the height of the BST

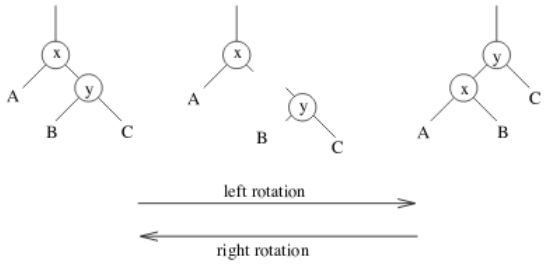
- This is OK if the BST is balanced, when $h = O(\log n)$
- This is Bad if the BST is degenerated, then $h = \Omega(n)$

A BST is a red-black tree if:

- Every node is either red or black
- The root is black
- Every leaf (`NULL`) is black
- Red nodes have black children
- For all nodes, all paths from node to descendant leaves contain the same number of black nodes

A red-black tree with n internal nodes has height at most 2log(n+1)

To maintain the red-black property the main building block is rotations



This can be done in $O(\log n)$

A red-black tree has worst case complexities:

- space $O(n)$
- lookup $O(\log n)$
- insert $O(\log n)$
- delete $O(\log n)$

3.3 Heaps

Heaps are trees, but typically assumed to be stored in a flat array

- Each tree node corresponds to an element of the array
- The tree is complete except perhaps the lowest level, filled left-to-right
- Heap property: for all nodes v in the tree `v.parent.data>= v.data` (assuming we have a node class like for BSTs)
- This is a max-heap (for min-heaps `v.parent.data =< v.data`)

Heap represented as an array A has two attributes

- `Length(A)` - the size of the array
- `HeapSize(A)` - the size of the heap stored within the array

Assume we start counting at position 1, then:

- The root is in A[1]
- parent(i)=A[i/2] - integer division, rounds down
- left(i)=A[2i]
- right(i) = A[2i+1]

Heaps are a very good data structure for priority queues and for sorting. Suppose we want to quickly extract the maximum element from a collection

Listing 5: HeapExtractMax(A)

```
1 ret = A[1] // biggest element (highest priority)
2 A[1] = A[HeapSize(A)]
3 HeapSize(A) = HeapSize(A)-1
4 Heapify(A,1,HeapSize(A))
5 return ret
```

Idea:

- Starting at the root, identify largest of current node v and its children

- Suppose largest element is in w
- If $w \neq v$
  - Swap A[w] and A[v]
  - Recurse into w (contains now what root contained previously)

Listing 6: Heapify(A,v,n)

```
// n is heap size
// find largest among v and 2v (left child)
largest = v
if 2v <= n and A[2v]>A[v] then largest = 2v
// find largest among winner above and
// 2v+1 (right child)
if 2v+1 <= n and A[2v+1]>A[largest] then
    largest = 2v+1
if largest != v then
    swap A[v], A[largest]
    Heapify (A, largest, n)
```

To take an array and turn it into a heap

Listing 7: BuildHeap(A,n)

```
for i=n downto 1 do
    Heapify(A,i,n)
endfor
```

Each call to heapify takes $O(\log n)$ time, so BuildHeap has an overall bound of $O(n \log n)$
But this can be better calculated as

$$T(n) = \sum_{h=1}^{\log n} (\#\text{ nodes at height } h) \cdot O(h) \leq \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot O(h) = O\left(\sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h\right) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right)$$

It is well known that for $x \in (0, 1)$,

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$$

Use this (x=1/2)

$$\sum_{h=1}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = \frac{1/2}{1/4} = 2$$

And hence:

$$T(n) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right) = O(n)$$

That is, can turn any array into heap in time $O(n)$ **Heapsort**:

- Call BuildHeap on unsorted data
- Repeatedly call HeapExtractMin until empty

Running time:

$$O(n) + n \cdot O(\log n) = O(n \log n)$$

Listing 8: HeapSort(A)

```
1 BuildHeap(A, Length(A))
2 for i = Length(A) downto 2 do
3     swap A[i] and A[1]
4     HeapSize(A) = HeapSize(A)-1
5     Heapify(A, 1, HeapSize(A))
6 endfor
```

4 Lower Bounds

A decision tree is a **full binary tree**
Only **comparisons** are relevant, everything else is ignored
Internal nodes are labelled $i : j$ for $1 \leqslant i, j \leqslant n$, meaning elements i and j are compared
Downward-edges are labelled $\leqslant$ or $>$, depending on the outcome of the comparisons
Leaves are labelled with come permutation of $\{1, ..., n\}$
A **branch** from root to leaf describes sequence of comparisons (nodes and edges) and ends in some resulting sequence (leaf)

4.1 Proof

Any comparison based sorting algorithm requires $\Omega(n \log n)$ *comparisons in the worst case*

- Sufficient to determine minimum height of a decision tree in which each permutation appears as leaf
- Consider decision tree of height h with ℓ leaves corresponding to a comparison sort on n elements
- Each of the n! permutations of input appears as some leaf: $\ell \geqslant n!$
- Binary tree of height h has at most 2^h leaves: $\ell \leqslant 2^h$
- Together: $n! \leqslant \ell \leqslant 2^h$ and therefore $2^h \geqslant n!$
- Take logs: $h \geqslant \log(n!) = \Omega(n \log n)$

4.2 Selection and adversaries

Adversary:

- Is a second algorithm intercepting access to input
- Gives answers so that there's always a consistent input
- Tries to make original algorithm delay a decision by dynamically constructing a bad input for it
- Doesn't know what original algorithm will do in the future, must work for any original algorithm

4.3 Representing Adversary's Strategy

As a digraph:

- The elements of array are the nodes
- If i loses to j, draw edge $i \to j$

As a status update table with bits of info revealed

Status for i, j	Adv Reply	newStatus	Info
N;N	$a_i > a_j$	N;L	1
N;L	$a_i > a_j$	No Change	0
L,N	$a_i < a_j$	No Change	0
L;L	Consist	No Change	0