CS2521: Sorting

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Why?

- Sorting is one of the most fundamental problems in CS
- Used as a building block for many other algorithms
- Many interesting generic ideas emerge
- Computers spend most of their time sorting (1/4 of all mainframe time was spent sorting)
- It's the most studied problem in CS
- Many problems, once sorted, become a lot easier. E.g. binary search (see Skiena for many others)

Spoilers

- Good sorting algorithms run in $O(n \log n)$.
- Naïve ones in $O(n^2)$
- For 1000000 elements, it's 1000 seconds vs 0.6 seconds
- Applications: bioinformatics, astronomy, particle physics ($n > 10^9$ typically)

Sample Application

- How can we determine whether two sets are disjoint (assume sets of size m and n, $m \ll n$)?

for all $i \in S_m$ **do**
    j=0, found=false
    while $j < |S_n|$ and !found **do**
        if $i == S_n[j]$ **then** found=true
    end while
    if found==false **then return** false
end for
return true

- Complexity?

Sample Application

- How can we determine whether two sets are disjoint (assume sets of size m and n, $m \ll n$)?

- Sort the big set, do a binary search for each element in the smaller set. Complexity: $O(n \log n)$ for sorting, $O(m \log n)$ for search. Total: $O((n+m) \log n)$
- Sort the small set and search from big set. Complexity: $O((n+m) \log m)$

Sample Application

- How can we determine whether two sets are disjoint (assume sets of size m and n, $m \ll n$)?
- Sort both sets.
- Now no need to do a binary search: compare smallest elements, discard smaller one if not identical. Keep repeating on the now smaller sets. Complexity?

Sample Application

- How can we determine whether two sets are disjoint (assume sets of size m and n, $m \ll n$)?

- Sort both sets.
- Now no need to do a binary search: compare smallest elements, discard smaller one if not identical. Keep repeating on the now smaller sets. Complexity?
- We visit each element of each set once at most, i.e. $O(n + m)$
- Total complexity: $O(n \log n + m \log m + n + m)$

Sample Application

- How can we determine whether two sets are disjoint (assume sets of size m and n, $m \ll n$)?

- $O(nm)$, $O((n+m)\log n)$, $O((n+m)\log m)$, $O(n\log n + m\log m + n + m)$ Which is fastest?
- Clearly not $O(nm)$.
- Now $\log m < \log n$ so we eliminate $O((n+m)\log n)$
- Expanding, we see this is better than $O(n\log n + m\log m + n + m)$
- So best to sort the small set only. If m is constant, complexity is linear!
- Note that we could build a hash table with elements of set n, and verify that collisions occur for all m elements in linear time.

Choices...

- Increasing or decreasing order?
- Sort just key or entire record?
- What should be done with equal keys?
- What do we do with non-numerical data?

Choices...

- Increasing or decreasing order?
- Sort just key or entire record?
- What should be done with equal keys?
- What do we do with non-numerical data?
- We assume the existence of a comparison function

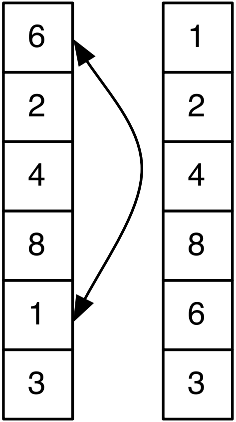
Selection Sort

```
function SelectionSort(s)
    for all i=0 to n-1 do
        min=i
        for all j=i+1 to n-1 do
            if s[j]<s[min] then
                min=j
            end if
        end for
        swap s[i],s[min]
    end for
end function
```

6
2
4
8
1
3

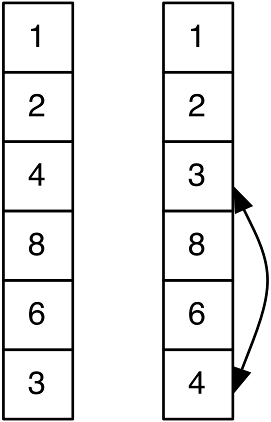
Selection Sort

```
function SelectionSort(s)
    for all i=0 to n-1 do
        min=i
        for all j=i+1 to n-1 do
            if s[j]<s[min] then
                min=j
            end if
        end for
        swap s[i],s[min]
    end for
end function
```



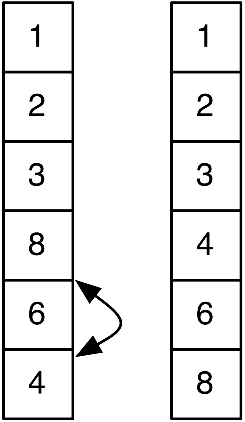
Selection Sort

```
function SelectionSort(s)
    for all i=0 to n-1 do
        min=i
        for all j=i+1 to n-1 do
            if s[j]<s[min] then
                min=j
            end if
        end for
        swap s[i],s[min]
    end for
end function
```



Selection Sort

```
function SelectionSort(s)
    for all i=0 to n-1 do
        min=i
        for all j=i+1 to n-1 do
            if s[j]<s[min] then
                min=j
            end if
        end for
        swap s[i],s[min]
    end for
end function
```



Selection Sort

```
function SelectionSort(s)
    for all i=0 to n-1 do
        min=i
        for all j=i+1 to n-1 do
            if s[j]<s[min] then
                min=j
            end if
        end for
        swap s[i],s[min]
    end for
end function
```

```
function SelectionSort(S)
    N=new array of size |S|
    for all i=0 to n-1 do
        sml=find smallest(S)
        N[i] = sml
        delete sml from S
    end for
    return N
end function
```

Selection Sort

- find smallest=$O(n)$ (sweep through average array size $n/2$)
- For loop $O(n)$
- Total time $O(n^2)$
- But we can speed up "find smallest"?

```
function SelectionSort(S)
    N=new array of size |S|
    for all i=0 to n-1 do
        sml=find smallest(S)
        N[i] = sml
        delete sml from S
    end for
    return N
end function
```

Selection Sort

- find smallest=$O(n)$ (sweep through average array size $n/2$)
- For loop $O(n)$
- Total time $O(n^2)$
- But we can speed up "find smallest"?
- Yes, through a better data structure: balanced binary tree or heap. $O(\log n)$
- New time: $O(n \log n)$

```
function SelectionSort(S)
    N=new array of size |S|
    for all i=0 to n-1 do
        sml=find smallest(S)
        N[i] = sml
        delete sml from S
    end for
    return N
end function
```

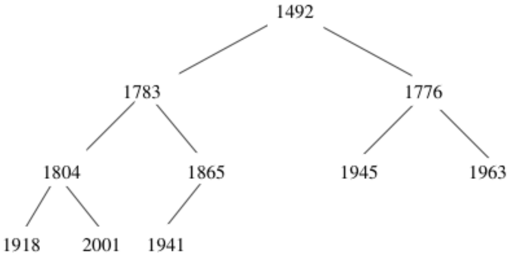
Heaps

- Given that we don't (yet) know how to construct a balanced binary tree, we'll look at heaps.
- A heap is a data structure that efficiently supports priority queue operations insert and find/delete-min.
- A heap keeps some sort of loose ordering on the set of elements.
- A heap encodes a tree structure, wherein each element in the tree <u>dominates</u> its children.
- A organisational hierarchy is a good example of a heap; the boss sits on the top, then managers below, then "normal" employees (whose boss is a specific manager) etc. The employer dominates the employees.
- In a <u>min-heap</u> a node dominates its children by containing a smaller key than they do.
- In a <u>max-heap</u> a node dominates its children by containing a larger key than they do.

Heaps

- Keys are typically small; if we use a binary tree representation, the pointers can use up more space than the keys themselves. We seek a different representation.
- Heaps allow us to represent binary trees without pointers.

Heaps



1	1492
2	1783
3	1776
4	1804
5	1865
6	1945
7	1963
8	1918
9	2001
10	1941

- We store the keys in an array, with the position of the keys implicitly acting as pointers.
- The first index stores the root, index 2 stores the left hand child, 3 stores the right, etc.
- Indexes 2^{l-1} to $2^l - 1$ store the 2^{l-1} keys at depth l of the tree.

Heaps

- The left child of an element at index k sits at position $2k$ in the tree, the right child at $2k + 1$.
- The parent of k at position $\lfloor k/2 \rfloor$
- What's the catch?

Heaps

- The left child of an element at index k sits at position $2k$ in the tree, the right child at $2k + 1$.
- The parent of k at position $\lfloor k/2 \rfloor$
- What's the catch?
- All missing internal nodes take up space as we need to represent the full tree (except the right part of the last level) to maintain positional mapping.
- Without this requirement, worst case is array of size 2^n to store n elements.
- Tradeoff between flexibility and space.

Using heaps

- How can we search for a key in a heap?

Using heaps

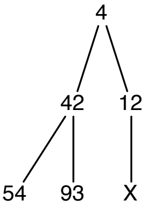
- How can we search for a key in a heap?
- We can't. A heap is not a binary search tree.
- We have to do a linear search.

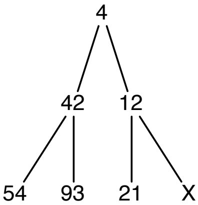
Where are we?

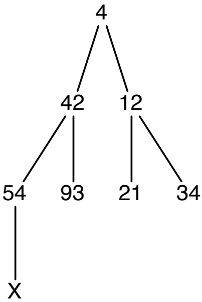
- We've described the structure of a heap, a pointer free representation of trees.
- In a heap, each node dominates its children.
- We want to show that we can find the minimum element of a heap quickly, and feed it into our sort operation to speed it up.
- We need to be able to
  - Build the heap
  - Find its minimum
  - Delete its minimum

Building a Heap

- We can build a heap incrementally inserting each new element into the left-most open spot in the array (i.e. the n+1'st position of a *n* element heap).
- This maintains balance of the heap.
- BUT, does not maintain dominance.



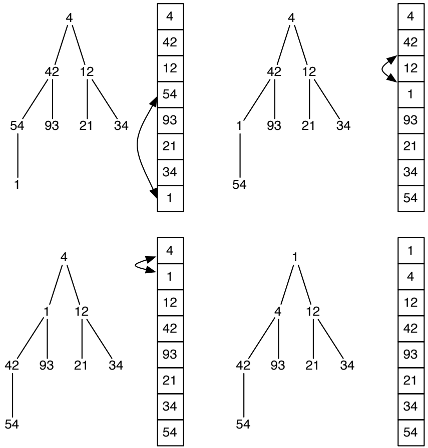




Building a Heap

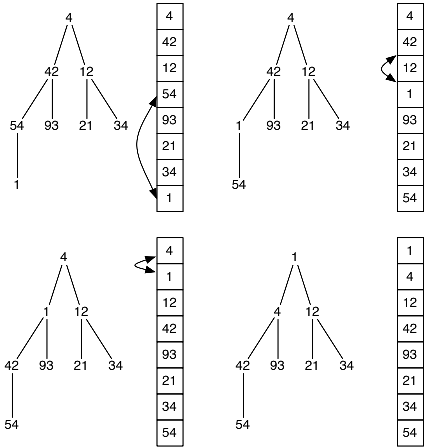
- To solve: we swap the new element with its parent as necessary.
- Old parent is clearly dominated.
- Other child is clearly dominated (even more).
- But new parent may still dominate its parent.
- So repeat, bubbling up as far as needed.
- Heap order is preserved.

Bubbling




```
function Insert(x,heap)
    heap[|heap|]=x
    bubble(heap,|heap|)
end function
function bubble(heap,index)
    if heap[index]>heap[index/2] then
        swap(heap[index],heap[index/2])
        bubble(heap,index/2)
    end if
end function
```

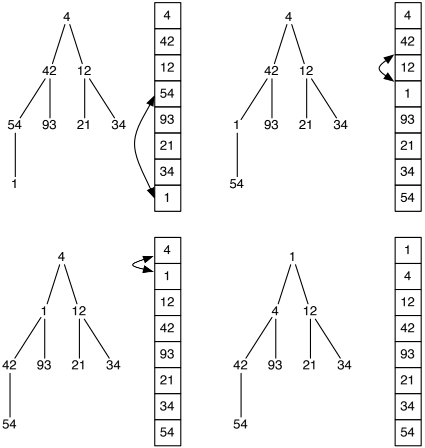
- Complexity?




```
function Insert(x,heap)
    heap[|heap|]=x
    bubble(heap,|heap|)
end function
function bubble(heap,index)
    if heap[index]>heap[index/2] then
        swap(heap[index],heap[index/2])
        bubble(heap,index/2)
    end if
end function
```

- n inserts
- $\log n$ bubbles per insert

Bubbling



```
function Insert(x,heap)
    heap[|heap|]=x
    bubble(heap,|heap|)
end function
function bubble(heap,index)
    if heap[index]>heap[index/2] then
        swap(heap[index],heap[index/2])
        bubble(heap,index/2)
    end if
end function
```

- n inserts
- $\log n$ bubbles per insert
- $O(n \log n)$

Making the heap

```
function makeHeap(array)
    for all i=0 to |array| do
        heap=array[0..i]
        insert(array[i],heap)
    end for
end function
```

- A heap is made in-place, using no additional memory.
- Since the original array was unsorted, we end up with a new unsorted array, which happens to be a heap.

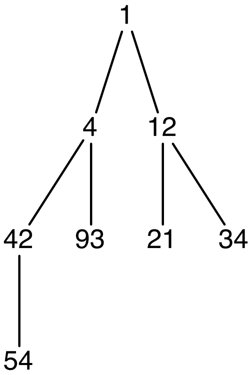
Finding the Minimum

- Easy: look at the first element of the heap.
- $O(1)$

Deleting the Minimum

- Once we delete the minimum, we have a hole in our heap.
- Replace by moving right most leaf into first position.
- Tree shape is restored. BUT heap property might be violated.
- The root could be dominated by both of its children.
- Swap root with its smallest child.
- Repeat for next level until dominance is restored for heap.
- Non-dominated element bubbles down to its place. Also called heapify as the two subtrees (heaps) below original root are merged into a new heap.

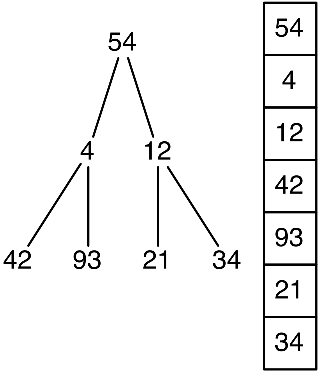
Heapifying



1
4
12
42
93
21
34
54

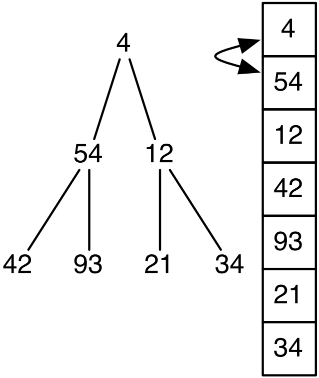
```
function ExtractMin(heap)
    ans=heap[0]
    move heap[|heap|-1] to heap[0] and resize
    heapify(heap,0)
    return ans
end function
function heapify(heap,index)
    sIndex=index
    if 2*index>|heap| then return
    if heap[2*index]<heap[sIndex] then
sIndex=2*index
    if heap[1+2*index]<heap[sIndex] then
sIndex=1+2*index
    swap heap[index] and heap[sIndex]
    heapify(heap,sIndex)
end function
```

Heapifying



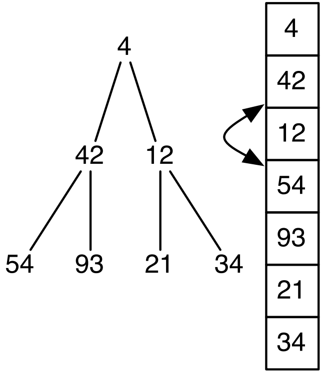

```
function ExtractMin(heap)
    ans=heap[0]
    move heap[|heap|-1] to heap[0] and resize
    heapify(heap,0)
    return ans
end function
function heapify(heap,index)
    sIndex=index
    if 2*index>|heap| then return
    if heap[2*index]<heap[sIndex] then
sIndex=2*index
    if heap[1+2*index]<heap[sIndex] then
sIndex=1+2*index
    swap heap[index] and heap[sIndex]
    heapify(heap,sIndex)
end function
```

Heapifying



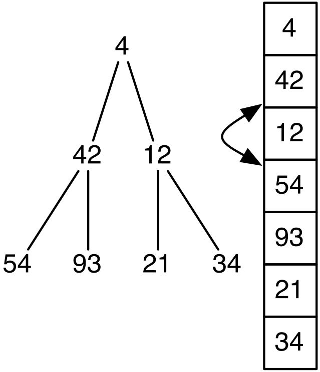

```
function ExtractMin(heap)
    ans=heap[0]
    move heap[|heap|-1] to heap[0] and resize
    heapify(heap,0)
    return ans
end function
function heapify(heap,index)
    sIndex=index
    if 2*index>|heap| then return
    if heap[2*index]<heap[sIndex] then
sIndex=2*index
    if heap[1+2*index]<heap[sIndex] then
sIndex=1+2*index
    swap heap[index] and heap[sIndex]
    heapify(heap,sIndex)
end function
```

Heapifying




```
function ExtractMin(heap)
    ans=heap[0]
    move heap[|heap|-1] to heap[0] and resize
    heapify(heap,0)
    return ans
end function
function heapify(heap,index)
    sIndex=index
    if 2*index>|heap| then return
    if heap[2*index]<heap[sIndex] then sIndex=2*index
    if heap[1+2*index]<heap[sIndex] then sIndex=1+2*index
    swap heap[index] and heap[sIndex]
    heapify(heap,sIndex)
end function
```

Heapifying



- Get root and swap last/first: $O(1)$
- Maximum of $\lfloor \log n \rfloor$ heapify operations.
- Total complexity $O(\log n)$

Where are we?

- Making the heap: $O(n \log n)$
- Extracting minimal element n times $O(n \log n)$
- We can shrink the heap each time (incrementing its start position), leaving extracted element in place.
- Result: sorted list, in $O(n \log n)$ time which uses no additional memory.
- This is heap sort
- Heap construction can actually be done in linear time, is this useful?

Faster heap construction

- An array is almost a heap - it doesn't have the heap property, but can be represented as a tree.
- But the leaves in this pseudo-heap do have the heap property.
- Consider parents of leaves - they may not dominate children, but we can swap things around so they do (how?)

Faster heap construction

- An array is almost a heap - it doesn't have the heap property, but can be represented as a tree.
- But the leaves in this pseudo-heap do have the heap property.
- Consider parents of leaves - they may not dominate children, but we can swap things around so they do (how?)
- By calling `heapify` at that index.

Faster heap construction

- Total number of heapify calls is n, and running time is $O(\log n)$ so no saving.
- But only the call at the root would take all $\log n$ steps.
- Heapify takes time proportional to the height of tree it is merging, and most heaps are small. Given a tree of n elements
  - $n/2$ are leaves
  - $n/4$ are parents of leaves
  - ...
- So there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

Faster heap construction

- Total cost of building heap is therefore

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil h \leq n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h$$

- $\sum h/2^h$ converges (below 2)
- Therefore, complexity is $O(n)$.

A different approach to sorting

- Basic idea: select a random element from the unsorted set, and put it in the proper position in the sorted set.

```
function InsertionSort(A)
    for i = 2 to n do
        j=i
        while (A[j] < A[j − 1]) & j > 0 do
            swap(A[j],A[j-1])
            j=j-1
        end while
    end for
end function
```

- $O(n^2)$ in the worst case.
- Works well if array is almost sorted already
- An example of incremental insertion; building up a structure on n by first building on $n-1$.

Divide and Conquer

- We've seen before (e.g. in binary search) that dividing and conquer is a useful approach in many situations.
  - divide the problem into smaller problems.
  - solve each of the smaller problems (typically by further division and solving).
  - combine the solutions of the subproblems into a solution of the bigger problem.
- Works easily for binary search as there are no interdependencies between subproblems.
- Can we use the approach for sorting?

Mergesort

function mergeSort($A[1..n]$)
    merge(mergeSort($A[1..\lfloor n/2 \rfloor]$), $mergeSort(A[\lfloor n/2 \rfloor + 1..n])$)
end function

- Base case?

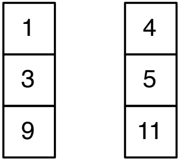
Mergesort

function mergeSort($A[1..n]$)
    merge(mergeSort($A[1..\lfloor n/2 \rfloor]$), $mergeSort(A[\lfloor n/2 \rfloor + 1..n])$)
end function

- Base case? Occurs when there's only a single element as there's nothing to sort.
- Efficiency of mergesort critically depends on how efficiently we can merge the sorted sublists.
- It would be silly to call heapsort or the like to do the combining.

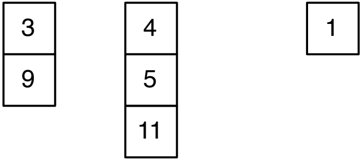
Merging

- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.



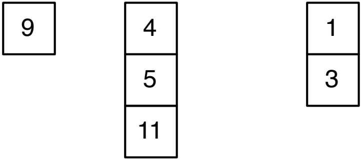
Merging

- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.



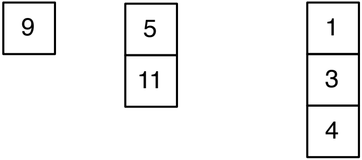
Merging

- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.



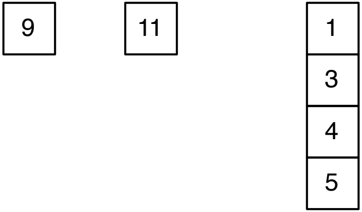
Merging

- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.

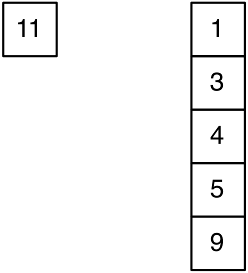


Merging

- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.



- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.



Merging

- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.

1
3
4
5
9
11

Merging

- Assume we have 2 sorted lists.
- The smallest item must be at the top of one of them.
- We pick that item, put it in the new list, and repeat until no items exist in either list.

- Merging takes $n - 1$ comparisons, i.e. $O(n)$

1
3
4
5
9
11

Merging

```
function merge(A_1, A_2)
    A_3 = []
    i = 0
    while |A_1 ∪ A_2| > 1 do
        if A_1[0] ≤ A_2[0] then
            A_3[i] = A_1[0]
            Remove A_1[0] from A_1
        else
            A_3[i] = A_2[0]
            Remove A_2[0] from A_2
        end if
        i=i+1
    end while
    add last element of A_1 ∪ A_2 to A_3[i]
end function
```

1
3
4
5
9
11

Mergesort

- To compute the running time of mergesort, we must consider how much work is done at each step of the divide & conquer approach.
- Let us assume we have n elements, such that $\log n \in \mathbb{Z}$
- The top level consists of 1 call to merge, processing all n elements.
- Level 2 consists of 2 calls to mergesort, and each processing $n/2$ elements.
- Level 3 has 4 calls to mergesort, each processing $n/4$ elements.
- Level k calls mergesort 2^k, each processing $n/2^k$ elements.

Mergesort

- Level k calls mergesort 2^k, each processing $n/2^k$ elements.
- At level 0, we merge 2 sorted lists, each of size $n/2$, so maximum of $n-1$ comparisons, $O(n)$.
- At level 1, two pairs of sorted lists are merged, each of size $n/4$, for a total of $n-2$ comparisons maximum.
- At level k we merge 2^k sorted lists, each of size $n/2^{k+1}$, for a total of at most $n-2^k$ operations.
- linear work is done merging all the elements on each level
- The number of elements in the subproblem is halved at each level.
- So recursion goes $\log n$ levels deep, and linear work done at each level.
- Worst case: $O(n \log n)$.

Mergesort

- Mergesort does not rely on random access to the elements in order to sort; it moves from start to end.
- Great for linked lists.
- By rearranging pointers, no extra space is used.
- But for arrays, a third array is required to store merge results, otherwise overwriting will occur.
- `http://youtu.be/XaqR3G_NVoo`

Sorting and Randomness

- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

5	1	7	3	4	6	8	2

Sorting and Randomness

- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

5	1	7	3	4	6	8	2

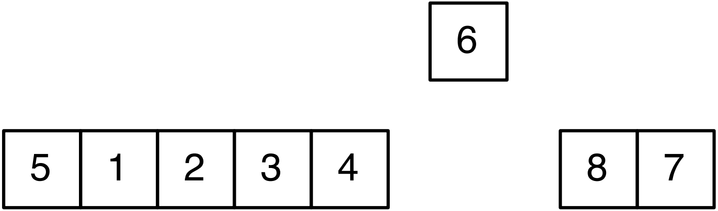
Sorting and Randomness

- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

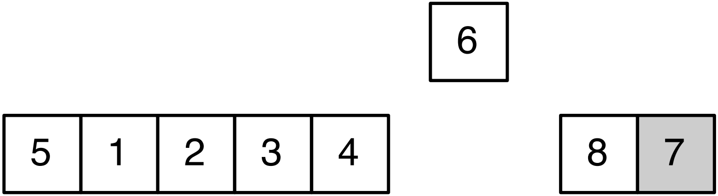
5	1	2	3	4	6	8	7

Sorting and Randomness

- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

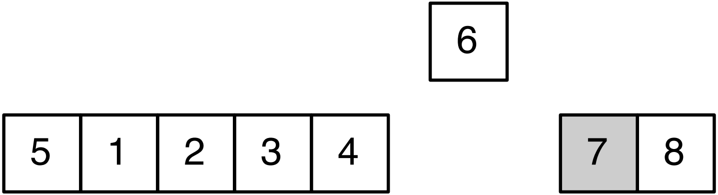


- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

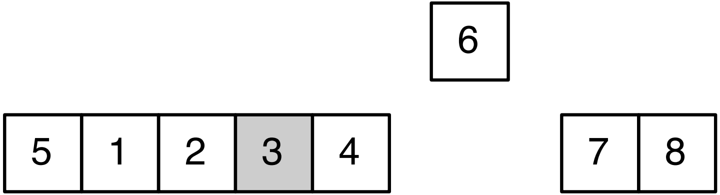


Sorting and Randomness

- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

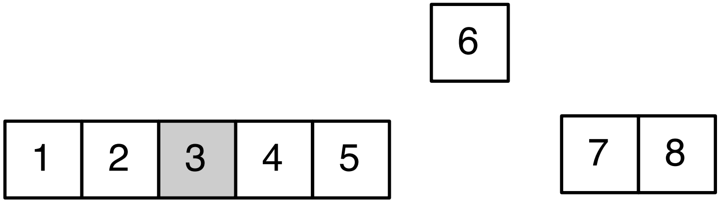


- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

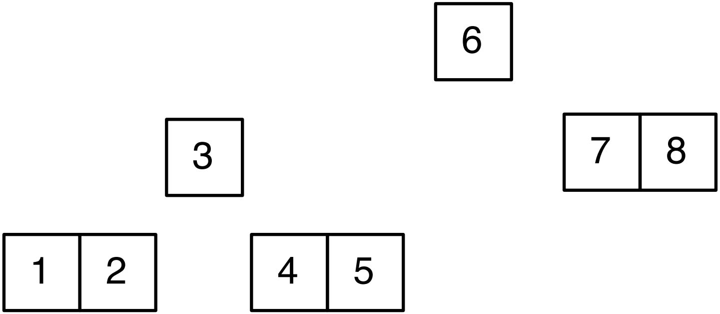


Sorting and Randomness

- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.



- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.



Sorting and Randomness

- Pick a spot in the array. This is the pivot
- Move all elements around pivot. If smaller than pivot, to the left of it, if bigger, to the right.
- Repeat for sub-arrays to the left and right.

1	2	3	4	5	6	7	8

Formalising the idea

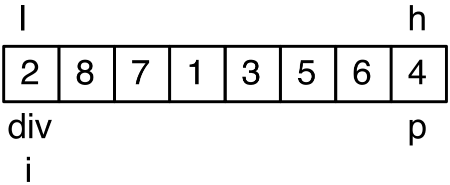
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.

```
function quicksort(A, l, h)
    if l < h then
        p=partition(A, l, h)
        quicksort(a, l, p − 1)
        quicksort(a, p + 1, h)
    end if
end function
```

```
function partition(A, l, h)
    p = h                      ▷ pivot element index
    div = l                    ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div + +
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

Formalising the idea

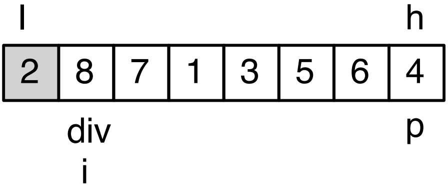
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                      ▷ pivot element index
    div = l                    ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div + +
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

Formalising the idea

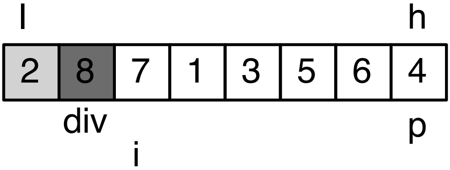
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                        ▷ pivot element index
    div = l                      ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div ++
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

Formalising the idea

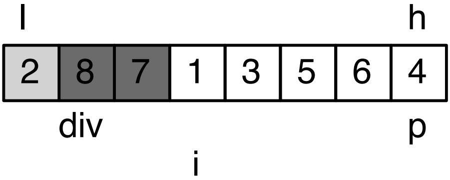
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                    ▷ pivot element index
    div = l                  ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div++
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

Formalising the idea

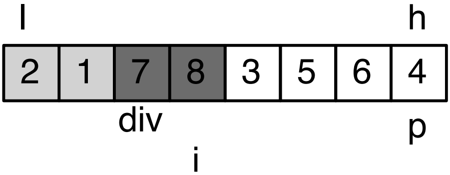
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                          ▷ pivot element index
    div = l                        ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div++
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

Formalising the idea

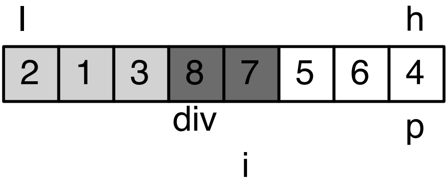
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                          ▷ pivot element index
    div = l                        ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div++
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

Formalising the idea

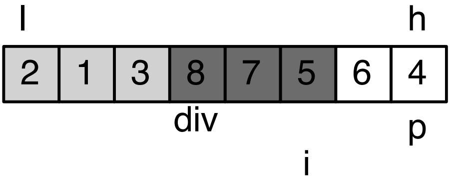
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                          ▷ pivot element index
    div = l                        ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div++
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

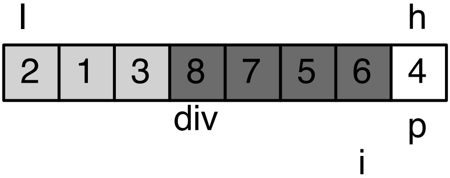
Formalising the idea

- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                        ▷ pivot element index
    div = l                      ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div++
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

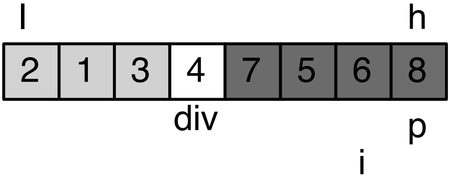
- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.



function partition(A, l, h)
    $p = h$ ▷ pivot element index
    $div = l$ ▷ divider element
    for all $i = l$ to $h - 1$ **do**
        if $A[i] \leq A[p]$ **then**
            swap $A[i], A[div]$
            $div{+}{+}$
        end if
    end for
    swap $A[p], A[div]$
    return div
end function

Formalising the idea

- This algorithm (known as quick sort) is another divide and conquer algorithm.
- The partition function does an in-place partitioning in a single scan.




```
function partition(A, l, h)
    p = h                        ▷ pivot element index
    div = l                      ▷ divider element
    for all i = l to h − 1 do
        if A[i] ≤ A[p] then
            swap A[i], A[div]
            div++
        end if
    end for
    swap A[p], A[div]
    return div
end function
```

Running Time

- Partition: $O(n)$
- But how many partitions are made?

Running Time

- Partition: $O(n)$
- But how many partitions are made?
- Depends on the pivot element picked.
- If in the middle, then each subarray created will be half the size of its parent, and time will be $O(n \log n)$ in total.
- But what happens if pivot element is the biggest or smallest element in the subarray?
- Then pivot is placed into "correct" position, and the new subarray is 1 element smaller than the original array.
- Total tree depth will be $n - 1$, with n operations in each, worst case time is $\Theta(n^2)$
- So why is quicksort called quicksort?

Average Case Running Time

- Assume that the typical partition produces a terrible split of x/y (e.g. x=9,y=10 means 9 elements are always to the left or right of the partition, and only 1 on the other side). How deep is our tree?
- If our tree is n elements in size, at level 1, we have 2 subarrays of size $n(1-x/y)$ and nx/y. The latter is bigger, and will lead to the deeper tree, so let's consider it.
- At level 2, the subarray will have $n(x/y)^2$ elements, at level 3 $n(x/y)^3$ etc, until at level h we will have 1 element.

$$n(x/y)^h = 1 \Rightarrow n = (y/x)^h$$

- $h = \log_{y/x} n$
- Remember we can change bases of logs by constant multiplication.
- As long as x and y are constants, the height of the tree is $\Theta(\log n)$
- Given n partitioning operations per level, total work is $\Theta(n \log n)$ in the average case!

More about running time

- Our claim is, that given randomly ordered data, quicksort will run in $\Theta(n \log n)$ time with a very high probability.
- For any pivot selection method, there is some worst case input which guarantees $\Theta(n^2)$ running time.
- If we use the same pivot selection method all the time, then will always fall foul of the same worst case input.
- One way to overcoming this involves randomising the pivot element.

Pivot Element Randomization

- Randomly permute the array before starting.
- At every step of the tree, select a random element to be the pivot (e.g. swap a random element from the subarray with the pivot element).
- Randomised quick sort runs in $\Theta(n \log n)$ time on any input with high probability!

An aside, Randomization

- Given a collection of n nuts and n bolts, you need to match each bolt to each nut.
- They are all a similar size, so you can't compare two nuts or two bolts.
- $O(n^2)$ brute force approach: for each nut, compare to a bolt until a match is found.
- Randomly selecting a nut means we'll work our way through half the bolts before a match is found, still $O(n^2)$.

An aside, Randomization

- Given a collection of n nuts and n bolts, you need to match each bolt to each nut.
- They are all a similar size, so you can't compare two nuts or two bolts.
- $O(n^2)$ brute force approach: for each nut, compare to a bolt until a match is found.
- Randomly selecting a nut means we'll work our way through half the bolts before a match is found, still $O(n^2)$.
- A quicksort like approach:
    - Select a nut at random, and partition bolts to those less than the nut size, and those bigger.
    - Once we find the matching bolt, partition the nuts in the same way. Total cost: $2n - 2$.
    - We can repeat this for the partitions, and the cost (as per quicksort) is $\Theta(n \log n)$ in the average case.
- No simple efficient algorithm exists for this problem!

Performance

- Mergesort, heapsort and quicksort all outperform selection sort (and insertion sort) on sufficiently large instances, as they're all $\Theta(n \log n)$
- But which is the fastest?

Performance

- Mergesort, heapsort and quicksort all outperform selection sort (and insertion sort) on sufficiently large instances, as they're all $\Theta(n \log n)$
- But which is the fastest?
- O notation doesn't tell us, implementation details are typically the most important in this case.
- Experiments have shown that a good quick sort is 2-3 times faster than the others due to the simplicity of the inner loop operations.
- `http://youtu.be/ywWBy6J5gz8`
- **Homework (to be discussed in practical):** Which type of sort should Google use?

Can we do better?

- So can we do better than $\Theta(n \log n)$?

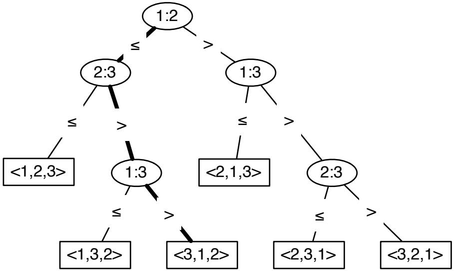
Can we do better?

- So can we do better than $\Theta(n \log n)$?
- yes and no...

Comparison Sorts

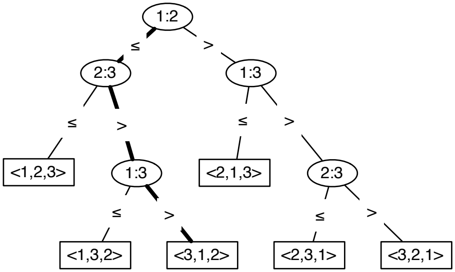
- All our sorts so far have, in one way or another, compared 2 elements and made a decision based on the difference between those two elements.
- Such comparison sorts did not use any other information about the elements (e.g. the actual value of the element).
- We can view any comparison sort abstractly in terms of decision trees.
- A decision tree is a full binary tree that represents the comparisons between elements that are performed by a sorting algorithm operating on an input of a given size.

Decision Trees



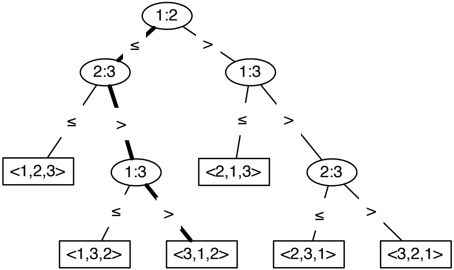
- We ignore control, datamovement etc.
- Each internal node is labelled $i : j$ for $1 \leq i, j \leq n$ for n input elements.
- Leaves are labelled with permutations of $1 \ldots n$.
- Executing a sorting algorithm traces a path from the root of the tree down to a leaf.

Decision Trees



- Each internal node represents a comparison, with edges representing decisions based on this comparison.
- When a leaf is reached, the permutation shows the ordering required for the list to be sorted.

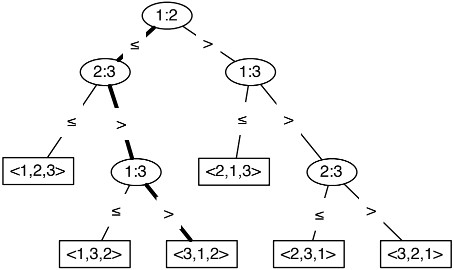
Decision Trees



- Input: 6 9 1

function InsertionSort(A)
    for $i = 2$ to n **do**
        j=i
        while $(A[j] \leq A[j-1])$ & $j > 0$ **do**
            swap(A[j],A[j-1])
            j=j-1
        end while
    end for
end function

Decision Trees



- How many leaf nodes must exist?
- The leaves must represent any possible ordering of the list, so n! leaves.
- Each leaf must be reachable from the root, as every possible ordering is possible.

A Theorem!

- The length of the longest simple path from the root of a tree to any of its reachable leaves represents the worst case number of comparisons that the corresponding sorting algorithm would perform.
- This is equivalent to the height of the tree.
- A lower bound on the height of any decision tree in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sorting algorithm.
- Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.
- Why?

Proof

- Assume a tree of height h with l reachable leaves for a sort on n elements.
- Since $n!$ permutations must appear in the leaves, $n! \leq l$. Why $\leq$ and not $=$?
- But a binary tree of height h has no more than 2^h leaves. So,

$$n! \leq l \leq 2^h$$

- Taking logarithms we obtain $h \geq \log(n!)$
- Now it can be shown that $\log(n!) = \Theta(n \log n) = \Omega(n \log n)$ $\square$
- Since heapsort and mergesort are $O(n \log n)$ they are asymptotically optimal comparison sorts (as they match the worst case lower bound from the above).

So how can we do better?

- In order to do better, we need to find a sort that doesn't utilise comparisons.
- The counting sort assumes that all input lies in the range $0 \ldots k$. If $k = O(n)$ it runs in $\Theta(n)$ time.
- Core idea is to determine for any input x, the number of elements less than x.
- This element is then placed in the correct position in the output array. E.g. if 17 elements are less than x, x will be in position 18.
- We must be able to handle elements of the same value.

Counting Sort

```
1:  function CountingSort(A,k)
2:      for all i=0 to k do
3:          C[i]=0
4:      end for
5:      for all j=0 to |A|-1 do
6:          C[A[j]]=C[A[j]]+1
7:      end for
8:      for all i=1 to k do
9:          C[i]=C[i]+C[i-1]
10:     end for
11:     for all j=|A|-1 down to 0 do
12:         B[C[A[j]]-1]=A[j]
13:         C[A[j]]=C[A[j]]-1
14:     end for
15: end function
```

- Array C is used to initially stores the number of times an element (used as the index) is encountered (line 6).
- Line 9 modifies C to count how many elements are less than or equal to the index.
- Line 11 populates the output array by obtaining the appropriate number from C and inserting it into the output, before decrementing the number (line 13).

Counting Sort

```
function CountingSort(A,k)
    for all i=0 to k do
        C[i]=0
    end for
    for all j=0 to |A|-1 do
        C[A[j]]=C[A[j]]+1
    end for
    for all i=1 to k do
        C[i]=C[i]+C[i-1]
    end for
    for all j=|A|-1 down to 0 do
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
    end for
end function
```

A	2	5	3	0	2	3	0	3

C	2	0	2	3	0	1

Counting Sort

```
function CountingSort(A,k)
    for all i=0 to k do
        C[i]=0
    end for
    for all j=0 to |A|-1 do
        C[A[j]]=C[A[j]]+1
    end for
    for all i=1 to k do
        C[i]=C[i]+C[i-1]
    end for
    for all j=|A|-1 down to 0 do
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
    end for
end function
```

C	2	2	4	7	7	8

Counting Sort

```
function CountingSort(A,k)
    for all i=0 to k do
        C[i]=0
    end for
    for all j=0 to |A|-1 do
        C[A[j]]=C[A[j]]+1
    end for
    for all i=1 to k do
        C[i]=C[i]+C[i-1]
    end for
    for all j=|A|-1 down to 0 do
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
    end for
end function
```

A	2	5	3	0	2	3	0	3
B							3	
C	2	2	4	6	7	8		

Counting Sort

```
function CountingSort(A,k)
    for all i=0 to k do
        C[i]=0
    end for
    for all j=0 to |A|-1 do
        C[A[j]]=C[A[j]]+1
    end for
    for all i=1 to k do
        C[i]=C[i]+C[i-1]
    end for
    for all j=|A|-1 down to 0 do
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
    end for
end function
```

A	2	5	3	0	2	3	0	3
B		0					3	
C	1	2	4	6	7	8		

Counting Sort

```
function CountingSort(A,k)
    for all i=0 to k do
        C[i]=0
    end for
    for all j=0 to |A|-1 do
        C[A[j]]=C[A[j]]+1
    end for
    for all i=1 to k do
        C[i]=C[i]+C[i-1]
    end for
    for all j=|A|-1 down to 0 do
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
    end for
end function
```

A	2	5	3	0	2	3	0	3

B		0				3	3	

C	1	2	4	5	7	8

Counting Sort

```
function CountingSort(A,k)
    for all i=0 to k do
        C[i]=0
    end for
    for all j=0 to |A|-1 do
        C[A[j]]=C[A[j]]+1
    end for
    for all i=1 to k do
        C[i]=C[i]+C[i-1]
    end for
    for all j=|A|-1 down to 0 do
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
    end for
end function
```

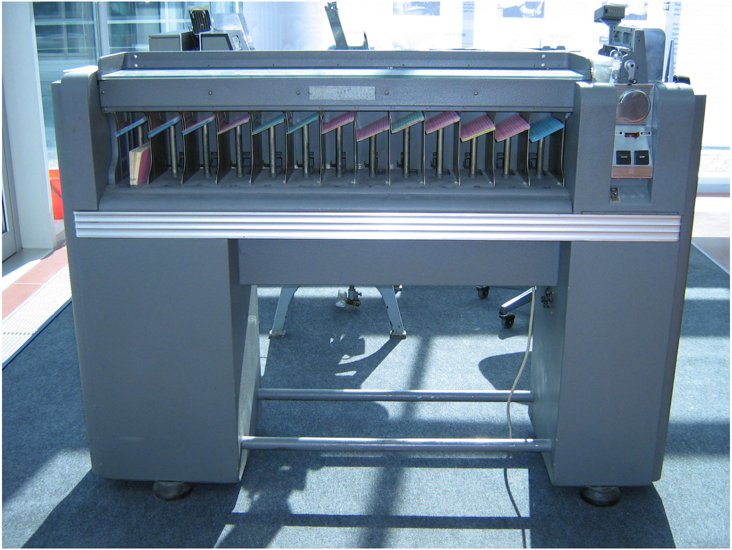
B	0	0	2	2	3	3	3	5

Counting Sort

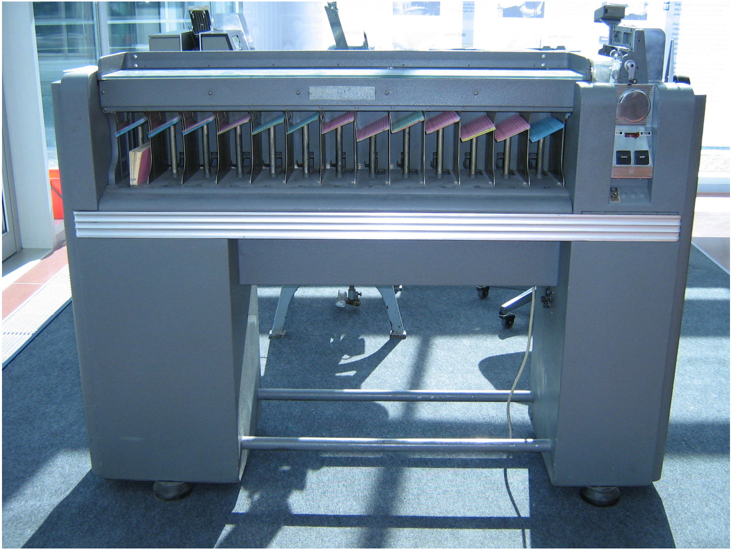
```
1:  function CountingSort(A,k)
2:      for all i=0 to k do
3:          C[i]=0
4:      end for
5:      for all j=0 to |A|-1 do
6:          C[A[j]]=C[A[j]]+1
7:      end for
8:      for all i=1 to k do
9:          C[i]=C[i]+C[i-1]
10:     end for
11:     for all j=|A|-1 down to 0 do
12:         B[C[A[j]]-1]=A[j]
13:         C[A[j]]=C[A[j]]-1
14:     end for
15: end function
```

- Loop in line 5 is $O(n)$
- Loop in line 8 is $O(k)$
- Loop in line 11 is $O(n)$
- $O(2n+k) = O(n+k) = O(n)$ when $k = O(n)$
- This is a stable sort: numbers with the same value appear in the output array in the same order as they do in the input array.

A challenge!

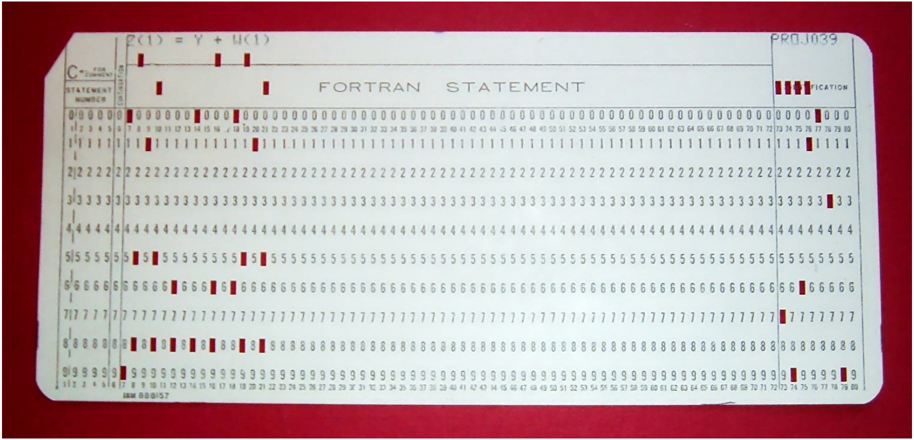


A challenge!



IBM Model 82 card sorter, capable of sorting 650 cards/minute!

A challenge!



- Scanning only one column at a time, how can we sort these punch cards <u>quickly</u>?
- We only have 13 bins, and someone must handle intermediate piles.

Radix Sort

- Most intuitive approach is to sort most significant digit first, then 2nd most significant digit, etc.
- But this requires many intermediate piles.
- The radix sort sorts on the least significant digit.
- Cards are then gathered from various bins in order and sort proceeds on 2nd least significant digit, etc.
- Given d digits n cards and k possible values per digit, and using Counting sort ($O(n+k)$ to do the sorting), radix sort is $\Theta(d(n+k))$. If d is a constant, and $k = O(n)$, then sort proceeds in $\Theta(n)$

Radix Sort

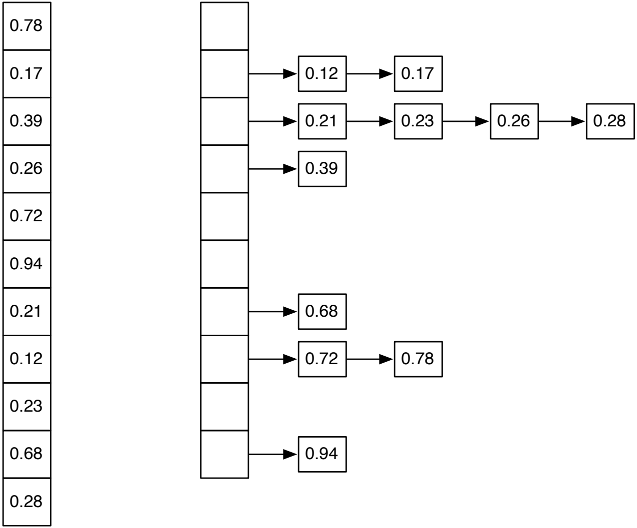
```
function RadixSort(A, d)
    for all i=1 to d do
        use a stable sort to sort A on digit d
    end for
end function
```

- **Homework (for practical)**: Prove that it works!
- There are several tweaks that can be performed to lower the constants hidden in the running time.
- If Radix sort is $\Theta(n)$ and comparison sorts are (at best) $\Theta(n \log n)$, why not always use Radix sort?

Bucket Sort

- The counting sort works when we have a small range of numbers.
- What if we have an arbitrary range instead $0 \dots max$?
- We can create m buckets:
  $[0..max/m], [max/m..2max/m], \dots [(m-1)max/m..max]$
- We then distribute the n inputs into the buckets.
- We can then take the numbers out of the buckets in order, sorting the numbers in the buckets with another sort (e.g. heapsort) if needed.
- If the items are uniformly distributed and $m = O(n)$ then there will be very few buckets containing multiple elements, sort will be $\Theta(n)$.
- But non-uniform distributions can be relatively common (see Skiena for an example).

Bucket Sort


0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
0.28
0.12
0.17
0.21
0.23
0.26
0.28
0.39
0.68
0.72
0.78
0.94

Where are we?

- We've now covered different sorting algorithms in detail.
- Comparison sort ($\Theta(n \log n)$)
  - Selection sort, Insertion sort, mergesort, heapsort, quick sort, ...
- Non-comparison sorts ($\Theta(n)$)
  - Counting sort, radix sort, bucket sort
- Different applications require different sorting algorithms.
- We've been able to analyse the complexity of many of these algorithms. Next, we will look at how to perform the analysis in a more principled manner.