Faculty of Information Technology,
Monash University

FIT2004: Algorithms and Data Structures

Week 2:
Analysis of Algorithms

Outline

- Proving correctness of algorithms
- Comparison-based sorting algorithms
  - Selection sort
  - Lower bound for comparison-based sorting
- Non-comparison-based sorting algorithms
  - Counting Sort
  - Radix Sort

Algorithmic Analysis

- In algorithmic analysis, one is interested in (at least) two things:
  - The correctness of the algorithm.
  - The amount of resources used by the algorithm.
- Let's look at how to prove correctness of an algorithm.

Proving Correctness of Algorithms

- Usually, we write programs and then test them.
- Testing detects inputs for which the program produces incorrect output.
  - Can we test all possible inputs?
- There are infinitely many possible inputs (for most programs) so we cannot test them all, therefore…
- Testing cannot guarantee that the program is always correct!
- Then, what is the solution?
- Logic **can** prove that a program is always correct.
- This is usually achieved in two parts by showing that:
  1. the program **always** terminates;
  2. It produces **correct results** when it terminates.

Example: Finding Minimum Value

```
//Find minimum value in an unsorted array of N>0 elements

min = array[1] // index starts at 1
index = 2

while index <= N

        if array[index] < min
                min = array[index]
        index = index + 1

return min
```

Does it Always Terminate?

```
//Find minimum value in an unsorted array of N>0 elements

min = array[1]
index = 2

while index <= N

       if array[index] < min
              min = array[index]
       index = index + 1

return min
```


Correct Result at Termination?

```
//Find minimum value in an unsorted array of N>0 elements

min = array[1]
index = 2

while index <= N

       if array[index] < min
              min = array[index]
       index = index + 1

return min
```

Will we get a correct result if we replace index <= N with index < N?

Correctness using Loop Invariant

- A ***loop invariant*** is used to reason about the correctness and behavior of an algorithm.
- It is a condition or property that holds true ***before and after each iteration of a loop***, throughout the execution of that loop.
- In other words, it is a ***logical assertion*** about the state of the program that remains ***consistent*** as the loop progresses, even though the loop's variables might change.

Correctness using Loop Invariant (LI)

Invariant should be true at three points:

1. Before the loop starts (Initialisation)
2. During each loop (Maintenance)
3. After the loop terminates (Termination)

Correctness using LI- Initialisation

```
min = array[1]
index = 2
//LI: min equals the minimum value in array[1 … index - 1]
while index <= N
        if array[index] < min
                min = array[index]
        index = index + 1
return min
```

- Consider the invariant just **before** the loop condition is checked.

  `min = array[1]`
  `index = 2`
- Here, min = the minimum value in array[1 … index - 1], i.e. min = the minimum value in array from array[1] to array[1] (i.e. the 1st element).
- The condition holds because min is initialized to array[1], and obviously the "minimum" value of just one element is that element itself.

Correctness using LI- Maintenance

```
min = array[1]
index = 2
//LI: min equals the minimum value in array[1 … index - 1]
while index <= N
        if array[index] < min
                min = array[index]
        index = index + 1
return min
```

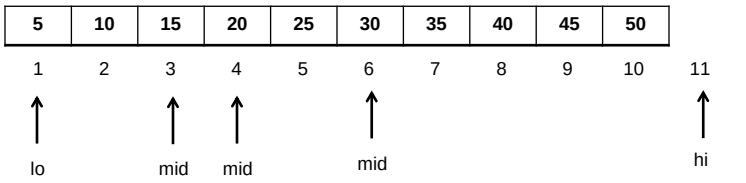
- At the start of each iteration, min equals the minimum value in `array[1 … index - 1]`.
- This is maintained because, the min variable holds the smallest value encountered up to `index - 1,` which starts with `array[1]`.
- If a smaller value is found during the current iteration, the `min` is updated to this new smaller value, `array[index]`.
- After the update, `min` still correctly represents the smallest value in the array from `array[1]` to `array[index - 1]`.

Correctness using LI- Termination

```
min = array[1]
index = 2
//LI: min equals the minimum value in array[1 … index - 1]
while index <= N
        if array[index] < min
                min = array[index]
        index = index + 1
return min
```

- We saw that LI is true for all values of `index` (in the previous step).
- We have set `index` to the value which causes termination (`N+1`).
- Check if the statement of LI with this `index` value is what we want
  `//LI: min equals the minimum value in array[1 … N]`
- After the last iteration, `min` will hold the minimum value in the entire array from `array[1]` to `array[N]`.
- Thus, the final value of `min` after the loop ends will be the minimum value in the range `array[1 … N]`, which is the desired outcome.

Another Example: Binary Search



mid = floor((lo+hi)/2)

Binary search 20 in a **sorted** array

- Since 20 < array[mid],
  - Search from lo to mid (e.g., move hi to mid)
- Since 20 > array[mid]
  - Search from mid to hi (e.g., move lo to mid)
- …

Algorithm for Binary Search

```
lo = 1//we are assuming indices range is 1 to N inclusive
hi = N + 1

while ( lo < hi )
   mid = floor( (lo+hi)/2 )
   if key >= array[mid]
      lo=mid
   else
      hi=mid

if N > 0 and array[lo] == key
   print(key found at index lo)
else
   print(key not found)
```

Is this algorithm correct?

To prove correctness, we need to show that

1. it **always** terminates, and
2. it returns correct result when it terminates

Does this algorithm always terminate?

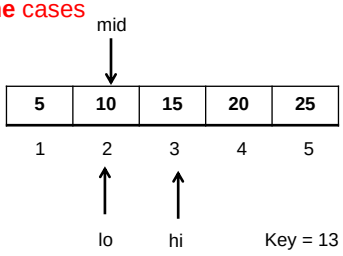
```
lo = 1
hi = N+1

while ( lo < hi )
   mid = floor( (lo+hi)/2 )
   if key >= array[mid]
      lo=mid
   else
      hi=mid

if N > 0 and array[lo] == key
   print(key found at index lo)
else
   print(key not found)
```

Let us try to fix this

This algorithm may never terminate in **some** cases



Does it always terminate?

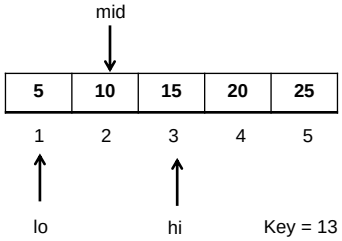
```
lo = 1//we are assuming indices range is 1 to N inclusive
hi = N + 1

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key >= array[mid]
        lo=mid
    else
        hi=mid

if N > 0 and array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

Proof that it always terminates

- **lo < hi – 1** implies that the difference between lo and hi is always at least 2
- Therefore, lo < mid < hi.
- Hence, the search space always shrinks (e.g., lo and hi get closer after every iteration of the while loop until lo ≥ hi – 1 in which case the algorithm terminates)



Correctness using Loop Invariant

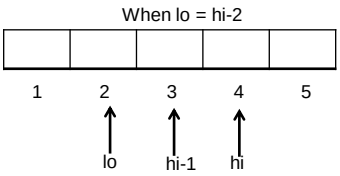
```
lo = 1
hi = N + 1
// LI: key in array[1 … N] if and only if (iff) key in array[lo … hi - 1]
while ( lo < hi - 1 )
    // LI: key in array[1 … N] iff key in array[lo … hi-1]
    mid = floor( (lo+hi)/2 )
    if key >= array[mid]
        // key in array[1 … N] iff key in array[mid … hi-1]
        lo=mid
        // LI: key in array[1 … N] iff key in array[lo … hi-1]
    else
        // key in array[1 … N] iff key in array[lo … mid-1]
        hi=mid
        // LI: key in array[1 … N] iff key in array[lo … hi-1]
// LI: key in array[1 … N] iff key in array[lo … hi-1]
```

Correctness using Loop Invariant

```
// LI: key in array[1 … N] if and only if (iff) key in array[lo … hi - 1]
while ( lo < hi - 1 )
   mid = floor( (lo+hi)/2 )
   if key >= array[mid]
     lo=mid
   else
     hi=mid
// LI: key in array[1 … N] iff key in array[lo … hi-1]
//  lo ≥ hi – 1 → lo + 1 ≥ hi                ----    (A)
//  lo < hi → lo + 1 ≤ hi                    ----    (B)
// From (A) and (B): lo + 1 = hi → lo = hi - 1
// Hence,  key in array[1 … N] iff key in array[lo … lo];   (Proof Complete)
 if N > 0 and array[lo] == key
      print(key found at index lo)
   else
      print(key not found)
```

Note: lo < hi when loop terminates, because

- lo < mid < hi in each iteration and
- we update either lo to be mid or hi to be mid



Correctness using Loop Invariant

Invariant: If key is in L, it is in L[lo…hi-1]

```
BinarySearch(L[1..n], t)
  lo = 1
  hi = N+1
  while ( lo < hi - 1 )
    mid = ⌊(lo + hi) / 2⌋
    if t ≥ L[mid]
      lo = mid
    if t < L[mid]
      hi = mid
  if L[lo] == t
    return lo
  return False
```

Initialisation

- If rightmost target is in L, then target is in `L[lo…hi-1] = L[1…N] = L`

Correctness using Loop Invariant

Invariant: If key is in L, it is in L[lo…hi-1]

```
BinarySearch(L[1..n], t)
  lo = 1
  hi = N+1
  while ( lo < hi - 1 )
    mid = ⌊(lo + hi) / 2⌋
    if t ≥ L[mid]
      lo = mid
    if t < L[mid]
      hi = mid
  if L[lo] == t
    return lo
  return False
```

Maintenance

- Assume target in `L[lo…hi-1]` before some loop iteration
- If `t >= L[mid]`, then it must be located in the range `L[mid…hi-1]`, so we should set `lo` to `mid`
- `If t < L[mid]`, then it must be in the range `L[lo…mid-1]`, so we should set `hi` to `mid` (since L[hi] is excluded)
- Since this is exactly what we do, at the start of the next iteration the invariant is still true

Correctness using Loop Invariant

Invariant: If key is in L, it is in L[lo…hi-1]

```
BinarySearch(L[1..n], t)
  lo = 1
  hi = N+1
  while ( lo < hi - 1 )
    mid = ⌊(lo + hi) / 2⌋
    if t ≥ L[mid]
      lo = mid
    if t < L[mid]
      hi = mid
  if L[lo] == t
    return lo
  return False
```

Termination

- `lo == hi-1`. Since the invariant has been correctly maintained…
- Rightmost target must be in `L[lo…hi-1]`, or it is not in the list
- `L[lo…hi-1]` is a single element, `L[lo]`.
- If the element is found at `L[lo]`, then return that index
- Otherwise return False

Outline

- Proving correctness of algorithms
- Comparison-based sorting algorithms
  - Selection sort
  - Lower bound for comparison-based sorting
- Non-comparison sorting algorithms
  - Counting Sort
  - Radix Sort

Comparison-based Sorting

- Comparison-based sorting algorithms sort the input array by comparing the items with each other:
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Merge Sort
  - Heap Sort
  - …

- The algorithms that do not require comparing items with each other are called non-comparison sorting algorithms. E.g., Counting sort, radix sort, bucket sort etc.

Comparison Cost

- Typically, we assume that comparing two elements takes O(1), e.g., array[i] <= array[j]. This is not necessarily true.
- String Comparison: The worst-case cost of comparing two strings is O(L) where L is the number of characters in the smaller string. E.g.,
  - "Welcome to Faculty of IT" <= "Welcome to FIT2004" ??
  - We compare strings character by character (from left to right) until the two characters are different – all green letters are compared in above example
- Number Comparison: Generally we assume that numbers are machine-word sized (i.e. fit in a register) and so comparison is O(1). In theory, for very large numbers, comparison would be O(digits).

Comparison Cost

- Typically assume comparison cost is O(1) (small values)
- Sometimes comparison cost is critical: genome sequences may have millions of characters – expensive comparison.
- The cost of comparison-based sorting is often taken as in terms of # of comparisons, e.g., # of comparisons in merge sort is O(N log N).
- This ignores comparison cost (which is mostly fine).

In summary:

In this unit we will generally assume that

- String comparison is O(c), where c is the number of characters which get compared
- Numerical comparison is O(1) unless otherwise specified

Stable Sorting Algorithms

- A sorting algorithm is called **stable** if it maintains the relative ordering of elements that have equal keys.

Input is sorted by names

Input

Marks	80	75	70	90	85	75
Name	Alice	Bill	Don	Geoff	Leo	Maria

Sort on Marks using a stable algorithm

Output

Marks	70	75	75	80	85	90
Name	Don	Bill	Maria	Alice	Leo	Geoff

Note: Output is sorted on marks then names.
Unstable sorting cannot guarantee this (e.g., Maria may appear before Bill)

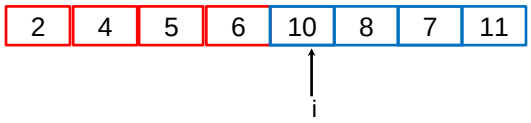
Outline

- Proving correctness of algorithms
- Comparison-based sorting algorithms
  - Selection sort
  - Lower bound for comparison-based sorting
- Non-comparison sorting algorithms
  - Counting Sort
  - Radix Sort

Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

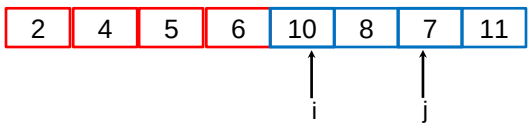
```
for(i = 1; i < N; i++) {
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
}
```



Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

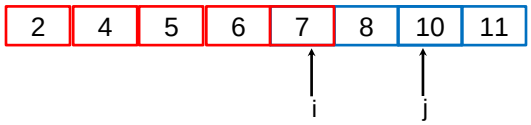
```
for(i = 1; i < N; i++) {
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
}
```



Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

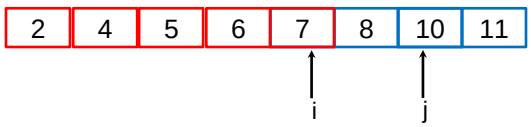
```
for(i = 1; i < N; i++) {
       j = index of minimum element in arr[i … N]
       swap (arr[i],arr[j])
}
```



Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

```
// LI: arr[1 … i-1] is sorted AND arr[1 … i-1] <= arr[i … N]
for(i = 1; i < N; i++) {
       j = index of minimum element in arr[i … N]
       swap (arr[i],arr[j])
}
// i=N when the loop terminates
// LI: arr[1 … N-1] is sorted AND arr[1 … N-1] <= arr[N]
```



Selection Sort (Correctness)

Sort an array (denoted as arr) in ascending order

```
// LI: arr[1 … i-1] is sorted AND arr[1 … i-1] <= arr[i … N]
for(i = 1; i < N; i++) {
       j = index of minimum element in arr[i … N]
       swap (arr[i],arr[j])
}
// i=N when the loop terminates
// LI: arr[1 … N-1] is sorted AND arr[1 … N-1] <= arr[N]
```

Could we use a weaker loop invariant, e.g.,

```
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

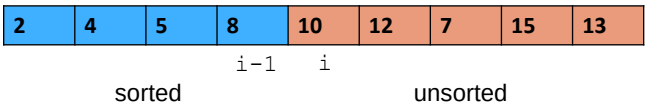

Selection Sort (Correctness)

Could we use a weaker loop invariant, e.g.,

```
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
       swap (arr[i],arr[j])
```

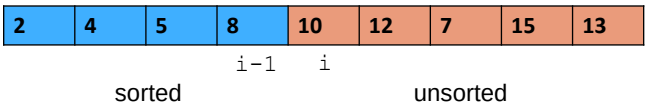


Selection Sort (Correctness)

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```



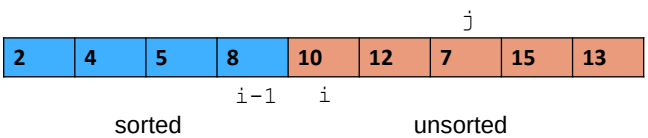
The invariant currently holds

Selection Sort (Correctness)

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```



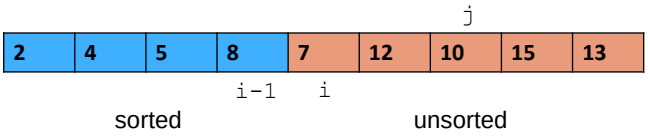
The invariant currently holds

Selection Sort (Correctness)

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```



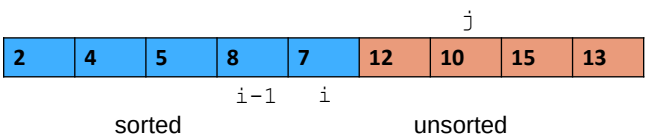
The invariant currently holds

Selection Sort (Correctness)

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i-1] is sorted (That is Insertion Sort)
```

While doing the proof, we need to show that if the LI is true at iteration k, it is true at iteration k+1

```
j = index of minimum element in arr[i … N]
      swap (arr[i],arr[j])
```



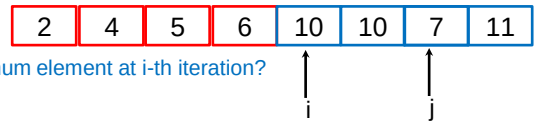
The invariant should still be true… but it is not

Selection Sort Analysis

```
for(i = 1; i < N; i++) {
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
}
```

Time Complexity?

- Worst-case
  - Complexity of finding minimum element at i-th iteration?
  - Total complexity? $O(N^2)$
- Best-case



Space Complexity? Input array: O(N)
Space for swapping operation: O(1)

Auxiliary Space Complexity?

- Selection Sort is an in-place algorithm!

Is selection sort stable?

Summary of Comparison-based Sorting Algorithms

	Best	Worst	Average	Stable?	In-place?
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	Yes
Insertion Sort	O(N)	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	O(N)	O(N log N)	O(N log N)	No	Yes
Merge Sort	O(N log N)	O(N log N)	O(N log N)	Yes	No

- Is it possible to develop a sorting algorithm with worst-case time complexity better than O(N log N)?

Outline

- Proving correctness of algorithms
- Comparison-based sorting algorithms
  - Selection sort
  - Lower bound for comparison-based sorting
- Non-comparison sorting algorithms
  - Counting Sort
  - Radix Sort

Lower Bound Complexity

- **Lower bound complexity** for a **problem** is the lowest possible complexity **any** algorithm (known or unknown) can achieve to solve the problem.
  - It is important because it gives a theoretical bound on what is best possible.
  - Unless stated otherwise, lower bound is for the worst-case complexity of the algorithm.
- What is the lower bound complexity of finding the minimum element in an array of N elements?
  - **Ans:** Ω(N), i.e. Omega(N): We (at least) need to look at every element.
- Since the finding minimum algorithm we saw has O(N) worst-case complexity, it is the best possible algorithm (i.e., optimal) in terms of time complexity.
- Generally, notation O(N) (Big O) is used for Upper bound (indicating worst case or at most this time). Ω(N) (Big Omega) is used for Lower bound (best case or at least this time).

Lower Bound Complexity

- **Lower bound complexity** for a **problem** is the lowest possible complexity **any** algorithm (known or unknown) can achieve to solve the problem.
  - It is important because it gives a theoretical bound on what is best possible.
  - Unless stated otherwise, lower bound is for the worst-case complexity of the algorithm.
- What is the lower bound complexity for sorting?
  - For comparison-based algorithm, lower bound complexity is Ω(N log N).
  - Read the lecture notes to see why the lower bound is Ω(N log N) -- the proof is not examinable.
- Next, we discuss two non-comparison sorting algorithms that do sorting in less than O(N log N).

Outline

- Proving correctness of algorithms
- Comparison-based sorting algorithms
  - Selection sort
  - Lower bound for comparison-based sorting
- Non-comparison sorting algorithms
  - Counting Sort
  - Radix Sort

Intuition

- Suppose you are given an array containing a permutation of the numbers 1…N to sort.
- Or given an array containing a subset of the numbers 1…N, sort them.
- Assume you are asked to enter a number between 1 to 200 (inclusive) on Flux. The rules of the game are as follows:
  - The person entering the smallest unique number wins
  - If two people enter the same number, both are disqualified
  - The winner is the person who enters the smallest number among the not-disqualified set
- Can you write an algorithm to determine the winner?
- What is the lower bound complexity for this problem?

Counting Sort

- For example, consider the problem of sorting positive integers.
- Determine the maximum value in the input.
- Create an array “count” of that size.

count

1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	0
2	0
3	1
4	0
5	0
6	0
7	0
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	1
2	0
3	1
4	0
5	0
6	0
7	0
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	1
2	0
3	2
4	0
5	0
6	0
7	0
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	1
2	0
3	2
4	0
5	0
6	0
7	1
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	1
2	0
3	2
4	0
5	1
6	0
7	1
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	1
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in “count”.
- If we see a 3, increment $count[3]$.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	0

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Iterate through the input and count the number of times each value occurs.
- Do this by incrementing the corresponding position in "count".
- If we see a 3, increment $count[3]$.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Counting Sort

- Create "output".
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1							

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1							

FIT2004: Lecture 2 - Analysis of Algorithms

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3				

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3				

FIT2004: Lecture 2 - Analysis of Algorithms

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5			

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5			

FIT2004: Lecture 2 - Analysis of Algorithms

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5	7	7	

Counting Sort

- Create “output”.
- For each position in count:
  - Append $count[i]$ copies of the value i to output.
  - So if $count[7] = 2$, then append 2 copies of 7.

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Output

1	3	3	3	5	7	7	8

FIT2004: Lecture 2 - Analysis of Algorithms

Analysis of Counting Sort

- Create “count” array of size max, where max is the maximum value in the input.
- For each value in **“Input”:**
  - count[value] += 1
- Output = empty
- For x=1 to len(**count**):
  - NumOfOccurrences = count[x]
  - Append x to Output NumOfOccurrences times

Let N be the size of Input array and U be the domain size (e.g., max), i.e., U is the size of count array.

Time Complexity:

- O(N+U) – worst-case, best-case, average-case all are the same.

Space Complexity:

- O(N+U)

Analysis of Counting Sort

- Create “count” array of size max, where max is the maximum value in the input.
- For each value in **“Input”:**
  - count[value] += 1
- Output = empty
- For x=1 to len(**count**):
  - NumOfOccurrences = count[x]
  - Append x to Output NumOfOccurrences times

Is counting sort stable?

No, because it counts the values but does not distinguishes between them. In fact, it loses any data associated with the values, so it is much worse than unstable. Let’s fix this!

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)
Output	1	3	3	3	5	7	7	8

Stable Counting Sort

- To fix the two problems of stability and losing data, we need a smart idea.

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)
Output	(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	(8,w)

- What information would you need to correctly place the data with key “5” in the output?

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

	count
1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	1
2	2
3	0
4	0
5	0
6	0
7	0
8	0

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

	count	position
1	1	1
2	0	2
3	3	2
4	0	0
5	1	0
6	0	0
7	2	0
8	1	0

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

	count	position
1	1	1
2	0	2
3	3	2
4	0	5
5	1	0
6	0	0
7	2	0
8	1	0

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

	count	position
1	1	1
2	0	2
3	3	2
4	0	5
5	1	5
6	0	0
7	2	0
8	1	0

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

	count	position
1	1	1
2	0	2
3	3	2
4	0	5
5	1	5
6	0	6
7	2	0
8	1	0

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	1
2	2
3	2
4	5
5	5
6	6
7	6
8	0

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	1
2	2
3	2
4	5
5	5
6	6
7	6
8	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

	count	position
1	1	1
2	0	2
3	3	2
4	0	5
5	1	5
6	0	6
7	2	6
8	1	8

Output								
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

	count	position
1	1	1
2	0	2
3	3	3
4	0	4
5	1	5
6	0	6
7	2	6
8	1	8

Output		(3,a)						
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	1
2	2
3	3
4	5
5	5
6	6
7	6
8	8

Output

	(3,a)						
1	2	3	4	5	6	7	8

Stable Counting Sort

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	1
2	2
3	3
4	5
5	5
6	6
7	6
8	8

Output

(1,p)	(3,a)						
1	2	3	4	5	6	7	8

Stable Counting Sort

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	3
4	5
5	5
6	6
7	6
8	8

Output

(1,p)	(3,a)						
1	2	3	4	5	6	7	8

Stable Counting Sort

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	3
4	5
5	5
6	6
7	6
8	8

Output

(1,p)	(3,a)	(3,c)					
1	2	3	4	5	6	7	8

Stable Counting Sort

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	4
4	5
5	5
6	6
7	6
8	8

Output

(1,p)	(3,a)	(3,c)					
1	2	3	4	5	6	7	8

Stable Counting Sort

Input

(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	4
4	5
5	5
6	6
7	6
8	8

Output

(1,p)	(3,a)	(3,c)			(7,f)		
1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	4
4	5
5	5
6	6
7	7
8	8

Output	(1,p)	(3,a)	(3,c)			(7,f)		
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	4
4	5
5	5
6	6
7	7
8	8

Output	(1,p)	(3,a)	(3,c)		(5,g)	(7,f)		
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	4
4	5
5	6
6	6
7	7
8	8

Output	(1,p)	(3,a)	(3,c)		(5,g)	(7,f)		
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	4
4	5
5	6
6	6
7	7
8	8

Output	(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)		
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	5
4	5
5	6
6	6
7	7
8	8

Output	(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)		
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	5
4	5
5	6
6	6
7	7
8	8

Output	(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	5
4	5
5	6
6	6
7	8
8	8

Output	(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

count

1	1
2	0
3	3
4	0
5	1
6	0
7	2
8	1

position

1	2
2	2
3	5
4	5
5	6
6	6
7	8
8	8

Output	(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	(8,w)
	1	2	3	4	5	6	7	8

Stable Counting Sort

Input	(3,a)	(1,p)	(3,c)	(7,f)	(5,g)	(3,b)	(7,d)	(8,w)

Construct count:
- For each key in input,
- count[key] += 1

Construct position:
- Initialise first position as a 1
- position[i] = position[i-1] + count[i-1]

Construct output
- Go through input, looking at each (key, val)
- Set output[position[key]] to the (key, val) pair from input
- Increment position[key]

	count	position
1	1	2
2	0	2
3	3	5
4	0	5
5	1	6
6	0	6
7	2	8
8	1	9

Output	(1,p)	(3,a)	(3,c)	(3,b)	(5,g)	(7,f)	(7,d)	(8,w)
	1	2	3	4	5	6	7	8

Analysis of Stable Counting Sort

- We have made count sort:
  - Stable
  - Able to keep associated data
- Time and space complexity?
  - We have not changed the complexity (Though one is faster in practice).
  - Time Complexity is O(N+U)
  - Can it be O(N)?
    - Yes, when U is O(N) (very efficient)
- Counting sort can also be very inefficient for certain kinds of input, any guesses?
  - Example: N = 2, U = 9223372036854775807
  - Input: [0, 9223372036854775807]
- Using counting sort, can we build a fast way to sort numbers?

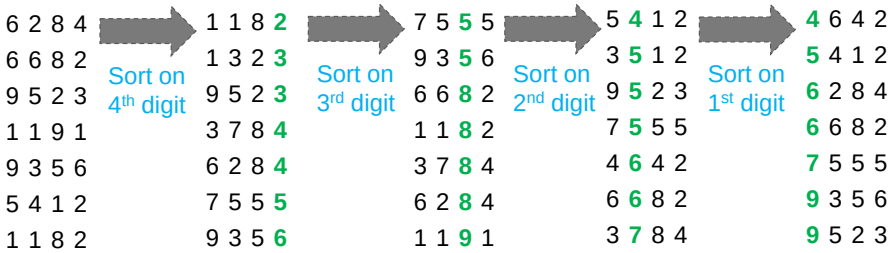
Outline

- Proving correctness of algorithms
- Comparison-based sorting algorithms
  - Selection sort
  - Lower bound for comparison-based sorting
- Non-comparison sorting algorithms
  - Counting Sort
  - Radix Sort

Radix Sort

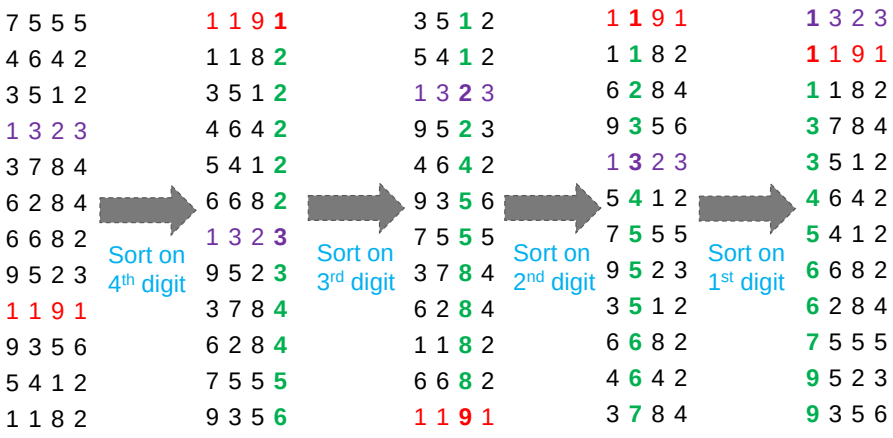
Sort an array of numbers, assuming each number has k digits (why is this often reasonable?)

- Use stable sort to sort them on the k-th digit
- Use stable sort to sort them on the (k-1)-th digit
- …
- Use stable sort to sort them on 1st digit



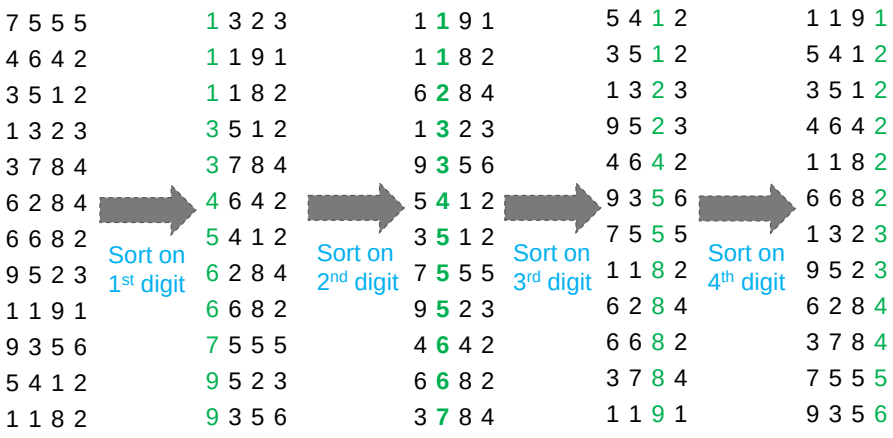
Radix Sort

What happens if we don't use stable sort?



Radix Sort

What happens if we process left to right (or most significant digit to least?)



Analysis of Radix Sort

Sort an array of numbers, assuming each number has k digits
- Use stable sort to sort them on the k-th digit
- Use stable sort to sort them on the (k-1)-th digit
- …
- Use stable sort to sort them on 1st digit

Assume that **N** numbers to be sorted and each number has **k** digits

Assume we are using stable counting sort which has time and space complexity $O(N + U)$

Time complexity of radix sort:
- Number of digits $*$ complexity of counting sort
- $O(k * (N + U)) = O(kN)$ because U is 10

Space complexity of radix sort:
- $O(k * (N + U)) = O(kN)$

Analysis of Radix Sort

But wait!

- Why base 10?
- Variable base, lets call the base “b”
- How many digits does a number have in base b?
- If the number has value M, then it has roughly $\log_b M$ digits
  - The exact number of digits is $\lfloor \log_b M \rfloor + 1$, but since we are doing complexity analysis, we can just say this is $O(\log_b M)$
- So we would need $O(\log_b M)$ count sorts to sort numbers of size M

Analysis of Radix Sort

- So we would need $O(\log_b M)$ count sorts to sort numbers of size M
- What would be the complexity of each counting sort?
- Each counting sort would take $O(N + b)$
- Total cost: $O(\log_b M * (N + b))$
- As we increase b…
  - Number of count sorts goes **down**
  - Cost of each count sort goes **up**

Analysis of Radix Sort

- Total cost: $O(\log_b M * (N + b))$

We want to find a good balance …

- Notice that each count sort will be O(N) as long as the base b is O(N)
- As an example, pick b = N (i.e. the base is the number of elements in the input)
- Total cost: $O(\log_N M * N)$
- What would be the value of M which makes the total cost O(N)?
- If M is $O(N^c)$, then total cost: $O(\log_N N^c * (N)) = O(CN) = O(N)$!!!
- Of course, practical considerations also matter:
  - Choosing a base which is a power of 2 is probably good
  - Cache/localisation considerations…

Sorting Strings

- How can we apply the count sort/radix sort idea to strings?

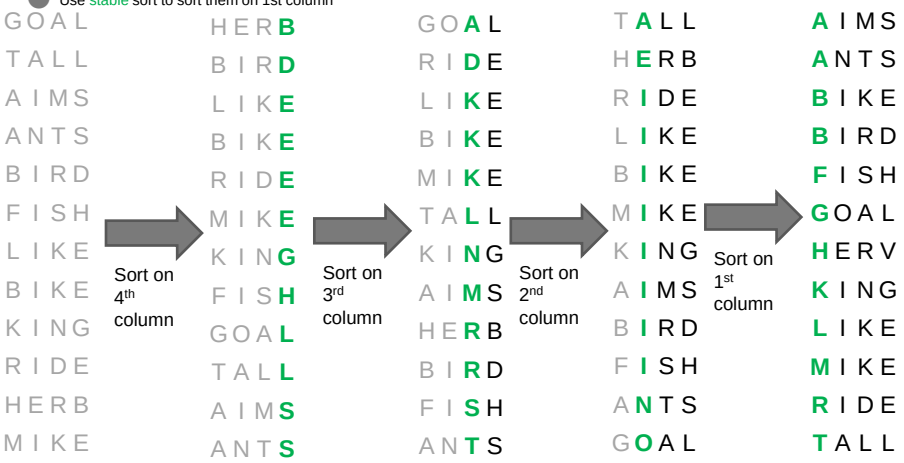
- Strings have “digits” which are letters
- The mapping can be done using ASCII:
  - e.g., in python:
  - ord(“A”) gives 65.
  - ord(“B”) gives 66.
  - and so on…

- The radix is now 26 (or however many characters we are using, e.g. 256)

Radix Sort

Sort an array of words in alphabetical order assuming each word consists of M letters each

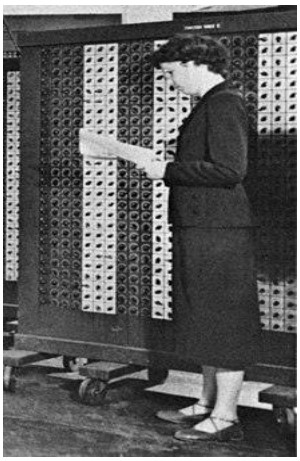
- Use stable sort to sort them on the M-th column
- Use stable sort to sort them on the (M-1)-th column
- …
- Use stable sort to sort them on 1st column



Algorithm's Heroes

Betty Holberton (March 7, 1917 – December 8, 2001)

- An American mathematician and programmer
- One of the six original programmers of the first electronic digital computer ENIAC (Electronic Numerical Integrator and Calculator)
- She was the inventor of breakpoints in computer debugging
- Once she said: By 1951 engineers had well accepted the computer, but the business world was still very sceptical about it, because they had such questions as, "How do I know the data is on the magnetic tape?" And "What is the legal implication of data on a magnetic tape?"



Reading

- Course Notes: Sections 1.1, 1.5, 3.1, 3.3 and 3.4
- You can also check algorithms' textbooks for contents related to this lecture, e.g.:
  - CLRS: Section 2.1, Chapter 8
  - Rou: Section 5.6

Concluding Remarks

Summary

- Loop invariants can be used to prove correctness
- Stable sorting, in-place algorithms
- Non-comparison sorting

Coming Up Next

- Quick Sort and its best/worst/average case analysis
- How to improve worst-case complexity of Quick Sort to O(N log N)

Preparation required before next lecture

- Make sure you understand this lecture completely
- Try to implement radix sort yourself