Radix Sorting

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Bits and Radixes

- Every binary object is defined as a sequence of bits.
- In many cases, the order in which we want to sort is identical to the alphabetical order of binary strings.
- Examples:

Bits and Radixes

- Every binary object is defined as a sequence of bits.
- In many cases, the order in which we want to sort is identical to the alphabetical order of binary strings.
- Examples:
  - Sorting positive integers (why only positive?).
  - Sorting regular strings of characters.
    - (If by alphabetical order we mean the order defined by the strcmp function, where "Dog" comes before "cat", because capital letters come before lowercase letters).

Bits and Radixes

- Every binary object is defined as a sequence of bits.
- In many cases, the order in which we want to sort is identical to the alphabetical order of binary strings.
- Examples:
  - Sorting positive integers (why only positive?).
    - Negative integers may have a 1 at the most significant bit, thus coming "after" positive integers in alphabetical order binary strings
  - Sorting regular strings of characters.
    - (If by alphabetical order we mean the order defined by the strcmp function, where "Dog" comes before "cat", because capital letters come before lowercase letters).

Bits and Radixes

- The word "radix" is used as a synonym for "base".
- A radix-R representation is the same as a base-R representation.
- For example:
  - What is a radix-2 representation?
  - What is a radix-10 representation?
  - What is a radix-16 representation?

Bits and Radixes

- The word "radix" is used as a synonym for "base".
- A radix-R representation is the same as a base-R representation.
- For example:
  – What is a radix-2 representation? Binary.
  – What is a radix-10 representation? Decimal.
  – What is a radix-16 representation? Hexadecimal.
  – We often use radixes that are powers of 2, but not always.

MSD Radix Sort

- MSD Radix sort is yet another sorting algorithm, that has its own interesting characteristics.
- If the radix is R, the first pass of radix sort works as follows:
  - Create R buckets.
  - In bucket M, store all items whose most significant digit (in R-based representation) is M.
  - Reorder the array by concatenating the contents of all buckets.
- In the second pass, we sort each of the buckets separately.
  - All items in the same bucket have the same most significant digit.
  - Thus, we sort each bucket (by creating sub buckets of the bucket) based on the second most significant digit.
- We keep doing passes until we have used all digits.

Example

- Example: suppose our items are 3-letter words:
  - cat, dog, cab, ate, cow, dip, ago, cot, act, din, any.
- Let R = 256.
- This means that we will be creating 256 buckets at each pass.
- What would be the "digits" of the items, that we use to assign them to buckets?

Example

- Example: suppose our items are 3-letter words:
  - cat, dog, cab, ate, cow, dip, ago, cot, act, din, any.
- Let R = 256.
- This means that we will be creating 256 buckets at each pass.
- What would be the "digits" of the items, that we use to assign them to buckets?
- Each character is a digit in radix-256 representation, since each character is an 8-bit ASCII code.
- What will the buckets look like after the first pass?

Example

- Example: suppose our items are 3-letter words:
  - cat, dog, cab, ate, cow, dip, ago, cot, act, din, any.
- What will the buckets look like after the first pass?
- Bucket 'a' = ate, ago, act, any.
- Bucket 'c' = cat, cab, cow, cot.
- Bucket 'd' = dog, dip, din.
- **All other buckets are empty.**
- How do we rearrange the input array?
  - ate, ago, act, any, cat, cab, cow, cot, dog, dip, din.
- What happens at the second pass?

Example

- After first pass:
  - ate, ago, act, any, cat, cab, cow, cot, dog, dip, din.
- What happens at the second pass?
- Bucket 'a' = ate, ago, act, any.
  - subbucket 'c' = act.
  - subbucket 'g' = ago.
  - subbucket 'n' = any.
  - subbucket 't' = ate.
- **All other buckets are empty.**
- Bucket 'a' is rearranged as act, ago, any, ate.

Programming MSD Radix Sort

- radixMSD_help(int * items, int left, int right, int * scratch, int digit_position)
  - If the digit position is greater than the number of digits in the items, return.
  - If right <= left, return.
  - Count number of items for each bucket.
  - Figure out where each bucket should be stored (positions of the first and last element of the bucket in the scratch array).
  - Copy each item to the corresponding bucket (in the scratch array).
  - Copy the scratch array back into items.
  - For each bucket:
    - new_left = leftmost position of bucket in items
    - new_right = rightmost position of bucket in items
    - radixMSD_help(items, new_left, new_right, scratch, digit_position+1)

Programming MSD Radix Sort

- See file radix_sort.c.
- Note: the implementation of MSD radix sort in that file is not very efficient.
- Certain quantities (like number of digits per item, number of bits per digit) get computed a lot of times.
  - You can definitely make the implementation a lot more efficient.
- The goal was to have the code be as clear and easy to read as possible.
  - I avoided optimizations that would make the code harder to read.

Programming MSD Radix Sort

- File radix_sort.c provides two implementations of MSD radix sort.
- First implementation: radix equals 2 (each digit is a single bit).
- Second implementation: radix can be specified as an argument.
  - But, bits per digit have to divide the size of the integer in bits.
  - If an integer is 32 bits:
  - Legal bits for digit are 1, 2, 4, 8, 16, 32.
  - Legal radixes are: 2, 4, 16, 256, 65536, 2^{32}.
  - 2^{32} takes too much memory…

Getting a Digit

```
// Digit 0 is the least significant digit
int get_digit(int number, int bits_per_digit, int digit_position)
{
  int mask = get_mask(bits_per_digit);
  int digits_per_int = sizeof(int)*8 / bits_per_digit;
  int left_shift = (digits_per_int - digit_position - 1) * bits_per_digit;
  int right_shift = (digits_per_int - 1) * bits_per_digit;

  unsigned int result = number << left_shift;
  result = result >> right_shift;
  return result;
}
```

If result is signed, shifting to the right preserves the sign (i.e., a -1 as most s significant digit).

LSD Radix Sort

- The previous version of radix sort is called MSD radix sort.
  - It goes through the data digit by digit, starting at the most significant digit (MSD).
- LSD stands for least significant digit.
- LSD radix sort goes through data starting at the least significant digit.
- It is somewhat counterintuitive, but:
  - It works.
  - It is actually simpler to implement than the MSD version.

LSD Radix Sort

```
void radixLSD(int * items, int length)
{
  int bits_per_item = sizeof(int) * 8;

  int bit;
  for (bit = 0; bit < bits_per_item; bit++)
  {
    radixLSD_help(items, length, bit);
    printf("done with bit %d\n", bit);
    print_arrayb(items, length);
  }
}
```

LSD Radix Sort

- void radixLSD_help(int * items, int length, int bit)
  - Count number of items for each bucket.
  - Figure out where each bucket should be stored (positions of the first and last element of the bucket in the scratch array).
  - Copy each item to the corresponding bucket (in the scratch array).
  - Copy the scratch array back into items.

MSD versus LSD: Differences

- The MSD helper function is recursive.
  - The MSD top-level function makes a single call to the MSD helper function.
  - Each recursive call works on an individual bucket, and uses the next digit.
  - The implementation is more complicated.
- The LSD helper function is not recursive.
  - The LSD top-level function calls the helper function once for each digit.
  - Each call of the helper function works on the entire data.

LSD Radix Sort Implementation

- File radix_sort.c provides an implementations of LSD radix sort, for radix = 2 (single-bit digits).
- The implementation prints outs the array after processing each bit.

LSD Radix Sort Implementation

before radix sort:

0: 4

1: 93

2: 5

3: 104

4: 53

5: 90

6: 208

LSD Radix Sort Implementation

```
done with bit 0
 0:          4 00000000000000000000000000000100
 1:        104 00000000000000000000000001101000
 2:         90 00000000000000000000000001011010
 3:        208 00000000000000000000000011010000
 4:         93 00000000000000000000000001011101
 5:          5 00000000000000000000000000000101
 6:         53 00000000000000000000000000110101
```

LSD Radix Sort Implementation

```
done with bit 1
 0:          4 00000000000000000000000000000100
 1:        104 00000000000000000000000001101000
 2:        208 00000000000000000000000011010000
 3:         93 00000000000000000000000001011101
 4:          5 00000000000000000000000000000101
 5:         53 00000000000000000000000000110101
 6:         90 00000000000000000000000001011010
```

LSD Radix Sort Implementation

```
done with bit 2
 0:         104 00000000000000000000000001101000
 1:         208 00000000000000000000000011010000
 2:          90 00000000000000000000000001011010
 3:           4 00000000000000000000000000000100
 4:          93 00000000000000000000000001011101
 5:           5 00000000000000000000000000000101
 6:          53 00000000000000000000000000110101
```

LSD Radix Sort Implementation

```
done with bit 3
 0:        208 00000000000000000000000011010000
 1:          4 00000000000000000000000000000100
 2:          5 00000000000000000000000000000101
 3:         53 00000000000000000000000000110101
 4:        104 00000000000000000000000001101000
 5:         90 00000000000000000000000001011010
 6:         93 00000000000000000000000001011101
```

LSD Radix Sort Implementation

```
done with bit 4
 0:          4 00000000000000000000000000000100
 1:          5 00000000000000000000000000000101
 2:        104 00000000000000000000000001101000
 3:        208 00000000000000000000000011010000
 4:         53 00000000000000000000000000110101
 5:         90 00000000000000000000000001011010
 6:         93 00000000000000000000000001011101
```

LSD Radix Sort Implementation

```
done with bit 5
 0:          4 00000000000000000000000000000100
 1:          5 00000000000000000000000000000101
 2:        208 00000000000000000000000011010000
 3:         90 00000000000000000000000001011010
 4:         93 00000000000000000000000001011101
 5:        104 00000000000000000000000001101000
 6:         53 00000000000000000000000000110101
```

LSD Radix Sort Implementation

```
done with bit 6
 0:          4 00000000000000000000000000000100
 1:          5 00000000000000000000000000000101
 2:         53 00000000000000000000000000110101
 3:        208 00000000000000000000000011010000
 4:         90 00000000000000000000000001011010
 5:         93 00000000000000000000000001011101
 6:        104 00000000000000000000000001101000
```

LSD Radix Sort Implementation

```
done with bit 7
 0:          4 00000000000000000000000000000100
 1:          5 00000000000000000000000000000101
 2:         53 00000000000000000000000000110101
 3:         90 00000000000000000000000001011010
 4:         93 00000000000000000000000001011101
 5:        104 00000000000000000000000001101000
 6:        208 00000000000000000000000011010000
```

LSD Radix Sort Implementation

```
done with bit 8
 0:          4 00000000000000000000000000000100
 1:          5 00000000000000000000000000000101
 2:         53 00000000000000000000000000110101
 3:         90 00000000000000000000000001011010
 4:         93 00000000000000000000000001011101
 5:        104 00000000000000000000000001101000
 6:        208 00000000000000000000000011010000
```

MSD Radix Sort Complexity

- $O(Nw + R*max(N, 2^w))$ time, where:
  - N is the number of items to sort.
  - R is the radix.
  - w is the number of digits in the radix-R representation of each item.
- O(N + R) space.
  - O(N) space for input array and scratch array.
  - O(R) space for counters and indices.

LSD Radix Sort Complexity

- O(Nw + Rw) time, where:
  - N is the number of items to sort.
  - R is the radix.
  - w is the number of digits in the radix-R representation of each item.
- As fast or faster than the MSD version!!!
  - Compare O(Nw + Rw) with O(Nw + R*max(N, 2^w)) …
  - Compare Rw with R*max(N, 2^w).
- O(N + R) space.
  - O(N) space for input array and scratch array.
  - O(R) space for counters and indices.

MSD Radix Sort Complexity

- Suppose we have 1 billion numbers between 1 and 1000.
- Then, make radix equal to 1001 (max item + 1).
- What is the number of digits per item in radix-1001 representation?

- What would be the time and space complexity of MSD and LSD radix sort in that case?

Radix Sort Complexity

- Suppose we have 1 billion numbers between 1 and 1000.
- Then, make radix equal to 1001 (max item + 1).
- What is the number of digits per item in radix-1001 representation?
  - 1 digit! So, both MSD and LSD make only one pass.
- What would be the time and space complexity of MSD and LSD radix sort in that case?
  - O(N+R) time. N dominates R, so we get linear time for sorting, **<u>best choice in this case.</u>**
  - O(N+R) extra space (in addition to space taken by the input). OK (not great).

MSD Radix Sort Complexity

- Suppose we have 1000 numbers between 1 and 1 billion.
- If radix equal to 1 billion + 1 (max item + 1):
- What would be the time and space complexity of MSD and LSD radix sort in that case?

MSD Radix Sort Complexity

- Suppose we have 1000 numbers between 1 and 1 billion.
- If radix equal to 1 billion + 1 (max item + 1):
- What would be the time and space complexity of MSD and LSD radix sort in that case?
  - O(N+R) time. R dominates, pretty bad time performance.
  - O(N+R) space. Again, R dominates, pretty bad space requirements.

Radix Sort Complexity

- Radix sort summary:
- Great if range of values is smaller than number of items to sort.
- Great if we can use a radix R such that:
  - R is much smaller than the number of items we need to sort.
  - Each item has a small number of digits in radix-R representation, so that we can sort the data with only a few passes.
  - Best cases: 1 or 2 passes.
- Becomes less attractive as the range of digits gets larger and the number of items to sort gets smaller.