Algorithms and Data Structures for Computational Biology

Greedy Algorithms and Sorting Algorithms

(Slides credits: these slides are a revised version of slides created by Prof. Luciano Bononi)

Prof. Ivan Lanese

Outline of the lesson

- Greedy technique:
  - definition
  - knapsack problem: continue and discrete cases
- Sorting algorithms:
  - counting sort
  - radix sort

The greedy technique

- The greedy technique is a strategy for designing algorithms, such as the “divide-et-impera” technique
- A greedy algorithm always makes the choice that looks best at the moment
- The hope: a locally optimal choice will lead to a globally optimal solution
- For some problems, it works
  - For others it will give solutions which are not optimal
- Greedy algorithms tend to be easy to code

The greedy technique

- Basic steps:
  - define the problem and corresponding greedy strategy
  - show that greedy approach leads to optimal solution
- A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions to its sub-problems

The Knapsack Problem

- The famous knapsack problem:
- A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

The Knapsack Problem: 0-1 knapsack

- More formally, the 0-1 knapsack problem:
  - the thief must choose among n items, where the i-th item worth v_i euro and weights w_i kilograms
- GOAL: carrying at most W kilograms and maximize value
- note: assume v_i, w_i, and W are all integers
- "0-1", means that each item must be taken or left in entirety

The Knapsack Problem: fractional knapsack

- A variation, the fractional knapsack problem:
  - it is a variation of the knapsack problem
  - thief can take fractions of items
  - think of items in 0-1 problem as gold lingots, in fractional problem as buckets of gold dust

Solving the knapsack problems

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm. How?
- The same greedy approach can be applied to the knapsack 0-1 problem but…
- … the optimal solution to the 0-1 problem cannot be found using the same greedy strategy (nor different greedy algorithms)

Solving the knapsack problems

- Greedy strategy: take in order of dollars/kilo
- The optimal solution to the 0-1 problem cannot be found using the same greedy strategy
- Example: 3 items weighing 10, 20, and 30 kilos,
- Knapsack can hold 50 kilos
- Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail
- E.g. 10($190)=19$/Kg, 20($200)=10$/Kg, 30($300)=10$/Kg ->
  - greedy: 10+30 total $490
  - non greedy: 20+30 total $500

Solving the knapsack problems

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - We will see that it can be solved with dynamic programming

Sorting algorithms: counting sort

- The counting sort is a sorting algorithm that is NOT based on comparisons
- It takes advantage of knowing the range of the numbers in the array to be sorted
- Basic idea: given that the numbers are in a range, the algorithm can then determine, for each input element, the amount of elements smaller than it

Counting sort

- data structures:
- A[]:initial data to be sorted
- B[]:used to store the sorted output
- C[]: used to count the occurrences of the data values

- CountingSort(A[], B[], k)

```
for i = 1 to k do
    C[i] = 0
for j = 1 to length(A) do
    C[A[j]] = C[A[j]] + 1
for i = 2 to k do
    C[i] = C[i] + C[i-1]
for j = length(A) downto 1 do
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

- *initializes C[]*
- *increments the values in C[], according to their frequencies*
- *adds all previous values, making C[] contain a cumulative total*
- *writes out the sorted data into array B[]*

Counting sort: computational complexity

- CountingSort(A[], B[], k)

```
for i = 1 to k do
    C[i] = 0
for j = 1 to length(A) do
    C[A[j]] = C[A[j]] + 1
for i = 2 to k do
    C[i] = C[i] + C[i-1]
for j = length(A) downto 1 do
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

- Given n = length(A)

  loop 1 = O(k)

  loop 2 = O(n)

  loop 3 = O(k)

  loop 4 = (n)

  = O(k+n)

- if k = O(n) then the counting sort is O(n)

Counting sort, with unknown range

- CountingSort(A[])

```
max = A[0]; min = A[0]              //Calcolo degli elementi max e min
for i = 1 to length[A] do
    if (A[i] > max) then max = A[i]
    else
        if(A[i] < min) then min = A[i]
for i = 0 to max-min-1 do           //Crea array C di dimensione max - min + 1
    C[i] = 0                        //inizializza a zero gli elementi di C
for i = 0 to length[A] do
    C[A[i] - min] = C[A[i] - min] + 1  //aumenta il numero di volte che si è incontrato A[i]
//Ordinamento in base al contenuto dell'array delle frequenze C
k = 0                               //indice per l'array A
for i = 0 to length[C] do
    while C[i] > 0 do               //scrive C[i] volte il valore (i + min) nell'array A
        A[k] = i + min
        k = k + 1
        C[i] = C[i] - 1
```

Sorting algorithms: radix sort

- The radix sort is a sorting algorithm that sorts integers by processing individual digits
- Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines
- It works by sorting the input numbers on each digit, for each of the digits in the numbers
- The numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so, up to the most significant digit

Radix sort

- RadixSort (A, d) A = array, d = number of digits

  for i = 1 to d do

  // use a stable sort to sort A on digit i

Input	Pass 1	Pass 2	Pass 3
329			
457			
657			
839			
436			
720			
355			

Radix sort

- RadixSort (A, d)   A = array, d = number of digits

    for i = 1 to d do

        // use a stable sort to sort A on digit i

Input	Pass 1	Pass 2	Pass 3
329	720		
457	355		
657	436		
839	457		
436	657		
720	329		
355	839		

Radix sort

- RadixSort (A, d)   A = array, d = number of digits

  for i = 1 to d do

    // use a stable sort to sort A on digit i

Input	Pass 1	Pass 2	Pass 3
329	720	720	
457	355	329	
657	436	436	
839	457	839	
436	657	355	
720	329	457	
355	839	657	

Radix sort

- RadixSort (A, d)   A = array, d = number of digits

  for i = 1 to d do

    // use a stable sort to sort A on digit i

Input	Pass 1	Pass 2	Pass 3
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix sort: computational complexity

- The cost of radix sort depends on the intermediate sorting algorithm that is used
- The counting sort can be a good choice
- In this case, each pass over n d-digit numbers takes $O(n + k)$ time
- There are d passes so the total time for radix sort is $\Theta(dn+kd)$
- When d is constant and $k = \Theta(n)$, the radix sort runs in linear time