3137 Data Structures and Algorithms in C++

Lecture 6

July 24 2006

Shlomo Hershkop

Announcements

- online syllabus updated with detailed readings
- Will do review on Wednesday
- what ever covered until today will be on midterm
- will release midterm this Wednesday (will talk about this)

Outline

- wrap up hashing
- Search engines
  - DS point of view
- Compression
- Search Algorithms

- Chapter 7-7.7

Hashing

- review

- what is a hash data structure
  - operations ?
  - run times?

strategies

- for collisions
- for clustering
- for table size
- for growing the hash table

Extendible hashing

- great when cant fit hash in memory
- use keys themselves to point to the location necessary to retrieve data
  - related to how B trees work
- allow quickly grow the hash table at constant cost
- Example

Application

- anyone know how Google works from a data structure point of view
- runtime ??

Search engine technology

- generally search engines work in the following way:
- collect documents e.g. webpages
- index information
- wait for search
  - understand query
  - search and match
  - scoring system

- Any ideas how to design a search engine so that you can quickly find results ?

- hash table of search words
- inverted index table

Vector Model

- Each document is a vector in an n dimensional vector space of search terms
- take query and find closets points

- sparse (very)
- if one word tokens, order will be ignored

algorithm

- First we generate a master word list

- can strip out stop words

- Stemming: can also calculate related words i.e. runs and run worry and worrying

master word list

- cat
- dog
- fine
- good
- got
- hat
- make
- pet

```
# A cat is a fine pet
$vec = [ 1, 0, 1, 0, 0, 0, 1 ];
```

- many ways of calculating similarity between search term and documents
- cosine
- can generate relevance scoring

General issues

- Better parsing
- Non-English Collections
  - stemming
  - stop words
- Similarity Search
  - can combine a few docs to find similarity
- Term Weighting
- Incorporating Metadata
- Exact Phrase Matching

Switch Back

- lets get back to our data structures

- Lempel-Ziv compression
  - how it works
  - LZW
    - where used

More DS

- Searching

Simple

- So its straightforward to sort in $O(N^2)$ time

- Insertion sort
- Selection sort
- Bubble sort

More complicated

- Shell Sort
  - This is an $O(N^{1.5})$ algorithm that is simple and efficient in practice
  - originally presented as an $O(N^2)$ algorithm
  - complicated to analyze
  - took many years to get better bounds

More Complex

- O(N log N) algorithms
  - merge sort
  - heapsort

Quicksort

- worst case O(n2)
- average case O(N log N)
  - will learn how to make the worst case occur with such low probability that we will end up dealing with average case

Selection sort

- anyone remember how this one works ??

- 2 arrays, sorted and unsorted
- keep choosing min from the unsorted list and append to sorted

Bubble Sort

- Anyone ??

- iterate and swap out of ordered elements

Insertion sort

- this is the quickest of the $O(N^2)$ algorithms for small sets

Insertion

- sort 1st element
- sort first 2
- sort first 3
- etc

code ??

```
insertionSort(array a, int length) {
    int i := 1;
    while (i < length) {
        insert(a, i, a[i]);
        i := i + 1;
    }
}

insert(array a, int length, value) {
    int i := length - 1;
    while (i ≥ 0 and a[i] > value) {
        a[i + 1] := a[i];
        i := i - 1;
    }
    a[i + 1] := value;
}
```

```
1   /**
2    * Simple insertion sort.
3    */
4   template <typename Comparable>
5   void insertionSort( vector<Comparable> & a )
6   {
7       int j;
8
9       for( int p = 1; p < a.size( ); p++ )
10      {
11          Comparable tmp = a[ p ];
12          for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
13              a[ j ] = a[ j - 1 ];
14          a[ j ] = tmp;
15      }
16  }
```

```
1   template <typename Iterator>
2   void insertionSort( const Iterator & begin, const Iterator & end )
3   {
4       if( begin != end )
5           insertionSortHelp( begin, end, *begin );
6   }
7
8
9   template <typename Iterator, typename Object>
10  void insertionSortHelp( const Iterator & begin, const Iterator & end,
11                          const Object & obj )
12  {
13      insertionSort( begin, end, less<Object>( ) );
14  }
```

implementation

- so would implementation of the underlying list affect the runtime ?
  - how ?
- any ideas why these are slow ??
  - can you prove it?

Lower Bound

- This is an analysis for simple sorts
- Inversion:
  - an ordered pair (i,j) such that i ‹ j and a[i] › a[j]
- Can you find the inversions ?
- [45, 34, 23, 35, 59]

swap

- So if we swap adjacent items, we only solve at most one inversion
- this leads to our slowdown
- any ideas ?

Theory

- before continuing….
- What would be the average number of inversion on an array of N elements ??

Average inversions

$$\frac{N(N-1)}{4}$$

- Let L be an unsorted list of elements
- Let L_r be the reverse of that list
- Any two elements are inverted either in L or L_r

- need to look at the pairs

$$\frac{N(N-1)}{2}$$

- pairs in L

- on average ½ will be inverted

- so how does swapping affect the number ?

- so how to do better than N^2?

Shell sort

- idea was to look at elements which are not adjacent
- Example:
  - look at every 8th element and do insert sort on those
    - slide window
  - Now look at every 4th
  - Every 2nd
- Increment series

Increment series

- we have an increment series $h_1, h_2, .., h_k$
- h_k must be less than N
- h_1 must be 1
  - why?
- each step keeps it sorted for last step

h_k sorted

- An array is h_k sorted
- for every i $a[i] \leq a[i + h_k]$
- we use diminishing increments
- Example

- as long as last increment is 1 , we are guaranteed to sort
- if we only do 1
  - what is it ?
- lets look at the code

```
void shellsort(int a[], int len) {
for( int gap = len/2; gap > 0; gap /=2)
  for(int i=gap; i<len; i++) {
      int tmp = a[i];
      int j=i;
      for(;j>=gap && tmp < aj-gap]; j-=gap)
      {     a[j] = a[j-gap];
      }
      a[j] = tmp;
  }
}
```

- So what is the increment series here ??
- 1 2 4 8 16 .. 2^k $\Theta(N^2)$
- Hubert
  - 1 3 7 .. 2^k-1 $\Theta(N^{1.5})$
- bizare sequences
  - $\Theta(N^{1.3})$

worst case runtime

Start	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Heapsort

- Heap sort worst case O(N log N)
  - average is slightly better
    - 2N(log N – log log N -4)
  - can save space using the same array
    - example

Better times

- lets start with better than n^2 sorting

merge sort

- if list has one element
  - return
- else
  - mergesort left half
  - mergesort right half
  - merge 2 halves

- Example

```
1   /**
2    * Mergesort algorithm (driver).
3    */
4   template <typename Comparable>
5   void mergeSort( vector<Comparable> & a )
6   {
7       vector<Comparable> tmpArray( a.size( ) );
8
9       mergeSort( a, tmpArray, 0, a.size( ) - 1 );
10  }
11
12  /**
13   * Internal method that makes recursive calls.
14   * a is an array of Comparable items.
15   * tmpArray is an array to place the merged result.
16   * left is the left-most index of the subarray.
17   * right is the right-most index of the subarray.
18   */
19  template <typename Comparable>
20  void mergeSort( vector<Comparable> & a,
21                  vector<Comparable> & tmpArray, int left, int right )
22  {
23      if( left < right )
24      {
25          int center = ( left + right ) / 2;
26          mergeSort( a, tmpArray, left, center );
27          mergeSort( a, tmpArray, center + 1, right );
28          merge( a, tmpArray, left, center + 1, right );
29      }
30  }
```

Analysis

- Lets do some simple analysis on mergesort running times
- Assume we have N items
  - N being a power of 2 so we can split nicely
  - if N is one, constant time to mergesort
  - else its 2 * N/2 mergesorts

- Define function
- T(N) = time to mergesort N items
- T(1) = 1
- T(N) = 2T(n/2)+N
- how to solve this ??

- this is a recurrence relationship
- in discrete do this all the time

First method: Telescoping

- trick is what to divide by
- what happens when you add 2 consecutive ones ??
- add all together ?

$$\frac{T(N)}{N} = \frac{2\,T\left(\frac{N}{2}\right)}{N} + 1$$

$$\frac{T(N)}{N} = \frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} + 1$$

now – for – next

$$\frac{T\left(\frac{N}{2}\right)}{\left(\frac{N}{2}\right)} = \frac{T\left(\frac{N}{4}\right)}{\left(\frac{N}{4}\right)} + 1$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Solution

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

$$T(N) = N * T(1) + N \log N$$

limitations

- telescoping is great, but sometimes hard to find what to divide by
- substitution is another method

substitution

- T(N) = 2T(N/2)+N
- sub N/2
- T(N/2) = 2T(N/4)+N/2
- go back to original
- T(N) = 4T(N/4) + 2N

- what do you get in the end ??

- $T(N) = 2^K T(N/2^K) + KN$

bottom line

- telescoping
  - more scratch work
- substitution
  - more brute force
  - easier when don't have a clue

end of the day

- Mergesort
  - O(nlogn)
  - if so good why not the default one?

reality

- requires extra temporary array
- copying is slow….sometimes
  - constant time to the big O runtime will catch up to you
- Great for external sorting

Next

- cue dramatic music
- QUICKSORT

Quick sort

- fastest currently known sort
  - Average N log N
  - Worst: N^2

Quicksort

- if one element return
- else
  - pick a pivot from the list
  - split the list around the pivot
  - return quicksort(left) + pivot + quicksort(right)
- Lets do an example

issues

- How does worst case happen ?
- how to pick the pivot ??

Pivot #1

- use the first element of the list
- pro/cons ?

- sorted list will always be N^2

Pivot #2

- choose random element for pivot
- pro/cons ?

- great performance
- expensive to generate random number

Pivot #3

- Choose median value from the list
- pro/cons ?

- hmmm don't you need a sorted list to get median?
- actually there is a linear algorithm for this ☺ will be doing it on homework

Pivot #4

- Median of 3
- since #3 isnt cheap, can grab 3 elements and take median
  - can even use random if you don't mind

- OK lets have a quiz!!
- actually, just submit feedback after next slide
- and let me know which topics you would like to see covered in Wednesday

For next time

- please complete the homework, and make sure you understand the solutions (correct ones)
  - if late, let weijen know you are submitting late
  - in general (for last 2) submit theory as early as possible and let him know you are doing it
- do all reading (see online)
- review before the exam (will be limited timed)
  - If I can get it to work ☺