Computers in Engineering COMP 208

Searching and Sorting

Michael A. Hawker

Where's Waldo?

- A common use for computers is to search for the whereabouts of a specific item in a list
- The most straightforward approach is just to start looking at the beginning and go on from there

Linear Search

```
int linear_search(int val, int arr[], int size) {
  int i;

  for(i = 0; i < size; ++i) {
    if(arr[i] == val)
       return i;
  }
  return -1;
}
```

Searching Sorted Lists

- Is that the way we would look up a name in the Montreal telephone directory?
- I hope not!

Binary Search

- To search a **sorted** array, we could check the middle element
- The value we are looking for might be there
- If not we can determine whether it is in the first or second half of the array and search that smaller array

Iterative Binary Search

```
int binary_search(int val, int arr[], int size) {
  int left = 0, right = size-1, middle;
  do {
    middle = (left + right) / 2;
    if(arr[middle] < val)
      left = middle + 1;
    else if(arr[middle] > val)
      right = middle - 1;
    else
      return middle;
  } while(left < right);
  return -1;
}
```

Sorting Data

- Sorting is one of the most common tasks given to computers
- Much work has been done on developing efficient sorting techniques
- We have seen one method and now we consider some others

Remember Bubble Sort?

```
void bubble_sort(int arr[], int size){
  int i, j;
  for (i=0; i<size-1; i++){
    for (j=size-1; j>i; --j)
      if (arr[j] < arr[j-1])
        swap (&arr[j], &arr[j-1]);
  }
}
```

An Optimization

- If no swaps are made, the array is already sorted
- We can keep track of whether any swaps were made in a pass
- If no swaps were made, the array must be sorted and we can stop

Optimized Bubble Sort

```
void bubble_sort(int arr[], int size) {
  int i, j, swapped;
  for (i=0; i<size-1; i++){
    swapped = 0;
    for (j=size-1; j>i; --j)
      if (arr[j] < arr[j-1]){
        swap (&arr[j], &arr[j-1]);
        swapped = 1;
      }
    if (!swapped) break;
  }
}
```

Selection Sort

- Another sorting technique is known as selection sort
- At each step, select the smallest value not yet in place and put it where it belongs
- Where's that?
- After the smaller elements at the front of the array

Selection Sort

- In the following program, note the use of pointer arithmetic to access the array elements
- We use arr + i instead of arr[i]
- As an argument arr + i represents an array with starting address arr[i]

Selection Sort

```
void select_sort(int arr[], int size){
  int i, index_of_min;

  for(i = 0; i < size; ++i) {
    index_of_min =
          find_min(arr + i, size - i);
    swap(arr + i, arr + i + index_of_min);
  }
  return;
}
```

Insertion Sort

- With insertion sort, we keep elements that have already been sorted at the front of the array
- At each step we look at the first of the unsorted values
- We add that value to the sorted part by "bubbling" it to the position where it belongs

Insertion Sort

```
void insertion_sort(int arr[], int size){
  int i, j;
  for(i = 1; i < size; ++i)
    for(j = i; j; --j)
      if(arr[j] < arr[j-1])
        swap(&arr[j], &arr[j-1]);
      else
        break;
}
```

The Cost of Algorithms

- We've seen multiple sorting algorithms
- Why is one better than the other?
- How can we measure this?
- In a uniform way?

Finding the Maximum

- We have already seen how to find the largest value in an array
- Here is the C code for that algorithm
- This code returns the location of the largest value (rather than the value itself)

Finding Max

```
int find_max(int arr[], int size) {
  int i, index_of_max = 0;

  for(i = 1; i < size; ++i)
    if(arr[i] > arr[index_of_max])
      index_of_max = i;

  return index_of_max;
}
```

Evaluating Algorithms

- How much “work” does the computer do to find the maximum?
- Different computers run at different speeds but we can try and count operations
- That is easier said than done

Asymptotic Analysis

- To get an approximate idea of the running time of an algorithm we count the number of operations but ignore the actual cost of each one
- The time is clearly dependent on the problem size

The Cost of Find_Max

- There is a loop that is executed n-1 times
- Each time a constant number of operations is done
- We say the algorithm for finding the maximum value runs in O(n) time if the problem is of size n

Linear Search

- The code for linear search is similar to the code for finding the maximum value
- It differs in that the algorithm does not always have to examine all values in the array
- It can stop as soon as it finds the value
- If the value isn't there, it must go all the way to the end to find out

Linear Search

```
int linear_search(int val, int arr[],
                  int size){
  int i;

  for(i = 0; i < size; ++i)
    if(arr[i] == val) return i;

  return -1;
}
```

Analysis of Linear Search

- If the value we are searching for is near the front of the array, the time taken is very small
- If the value is at the end of the array, or not in the array at all the time taken is proportional to n, i.e. O(n)

Worst Case Analysis

- When evaluating an algorithm we generally look at the worst case
- This gives us a “guaranteed” running time even if the time may be faster in many cases
- In this example we say the worst case running time is O(n)

Average Case Analysis

- In general it is difficult to determine the average time an algorithm will take
- Average case time is dependent on the distribution of the data values
- If the data is uniformly distributed and we search for a random value, the average case time for linear search is also O(n)

Binary Search

- We have also seen another algorithm for searching sorted lists, binary search
- Intuitively it seems to be much faster
- How can we show this analytically?
- How much faster is it?

Binary Search in Sorted Arrays

```
int binary_search(int val, int arr[], int size){
  int left = 0, right = size, middle;
  do {
    middle = (left + right) / 2;

    if(arr[middle] < val)
      left = middle + 1;
    else if(arr[middle] > val)
      right = middle - 1;
    else
      return middle;

  } while(left <= right);
  return -1;
}
```

The Cost of Binary Search

- The original list being searched had n values
- After checking the middle element we either find the value we are looking for or we reduce the problem size to n/2
- In the worst case, if we don't happen to find the value, the problem size becomes n/4, n/8, n/16, …

The Cost of Binary Search

- The process cannot continue forever
- Eventually $n/2^i$ becomes smaller than 1 and the value was either found or is not in the list
- This must stop after $\log_2 n$ steps
- The cost of binary search is then $O(\log n)$

The Cost of Bubble Sort

- There are n passes through the array in the worst case
- Pass j takes n-j steps
- The total number of steps is 1+2+…+n
- We say this is $O(n^2)$

Analysis

- Is the optimized version faster?
- Yes and No.
- In practice, yes
- Asymptotically, no.
- It is still $O(n^2)$ in the worst case

Other Sorting Algorithms

- How about selection or insertion sort?
- They also contain nested loops
- Note that for selection sort, the inner loop is "hidden" inside the function
- In either event, the cost is $O(n^2)$

Can we do better?

- Sorting is an important application
- Are there faster ways to sort?
- Wait and see!