CSIT113
Problem Solving

UNIT 7
SEARCHING AND SORTING

Overview

- Searching:
  - Linear Search
  - Binary Search
- Sorting:
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Quick Sort

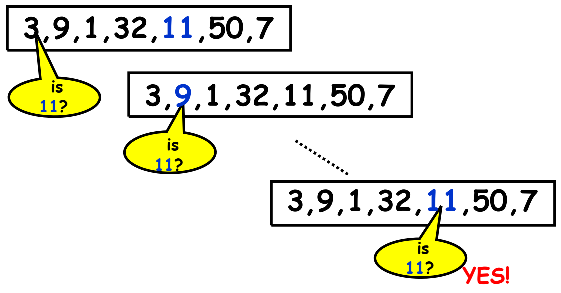
Searching

- Assume that we have to find whether a particular name is in a sequence of names.
- What is our best strategy?
- This depends on the precise nature of the sequence.
- OK – the sequence is in no special order.
- Now what is our best strategy.

Searching an unordered sequence: Linear Search

- Look at each item, in turn, until we find the one we are looking for or we run out of names.
- This is known as a linear search and, on average, we will look at around half of the names in the sequence (assuming it is there at all).
- In the worst case, the name is not on the sequence, it involves looking at all of the entries.
- In this case, the “best” strategy is actually not especially good: but this is the only way, there is no other choices.

Linear Search: Example



Improving on linear search.

- What can we do to the sequence to make it easier (more efficient) to search?
- Think about sequences of words you are familiar with.
  - Phone books;
  - Dictionaries;
  - Indexes.
- What property do all these sequences share?
- They are in alphabetical order!
- Why is this done?

Searching ordered sequences: Binary Search

- If a sequence is in order we can search it more efficiently.
- We do need to know how big the sequence is, though.
- Here is our new strategy called Binary Search:
  1. Look at the word in the middle of the sequence.
  2. If this word is the one we are looking for, we have found the target, so we stop.
  3. If this word is after the one we are looking for, replace the sequence with the first half of the sequence.
  4. If this word is before the one we are looking for, replace the sequence with the second half of the sequence.
  5. Go back to step 1. with the shorter sequence.

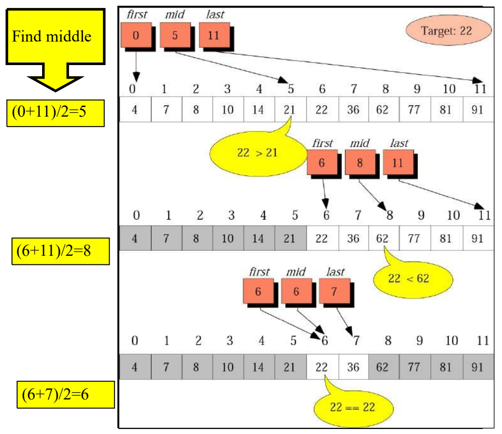
Performing Binary Search in an Sequence in non-descending order

- Initially binary search set f = 0, *l* = n-1, targetFound = “no”. Then, while ((targetFound = “no”) and (f ≤ *l*)) the following is carried out:
  - ➢ Set mid $= \lfloor \frac{f+l}{2} \rfloor$.
  - ➢ If T = A[mid], we have found the target in the sequence, A[mid] and set targetFound = “yes”.
  - ➢ If T < A[mid], then the target can only be in the first half of the sequence currently searched, so for searching the first half in the next iteration, we set f and *l* as follows:
    - ❑ f = f of the current step
    - ❑ *l* = mid of the current step - 1
  - ➢ If T > A[mid], then the target can only be in the last half of the sequence currently searched, so for searching the last half in the next iteration, we set f and *l* as follows:
    - ❑ f = mid of the current step + 1
    - ❑ *l* = *l* of the current step
- When targetFound = “yes”, the value of mid is returned, otherwise, “target not found” is returned.

Note on Binary Search in an Sequence in non-descending order

- Note that the binary search will stop under one of the following two cases:
  - ➢((targetFound = “yes”): in this case, the target is found and is equal to the middle element of the subsequence searched, that is T = A[mid].
  - ➢(f > *l*): as it is not possible for the first index more than the last index in any subsequence, this implies that there is no further subsequence to search. Hence, it implies that the target is not in the sequence. Therefore, “target not found” is returned.

Example: Binary Search in an Sequence



If target is in the first part:
First of the next step = first of current step
Last of the next step = mid of current step - 1

If target is in the last part:
First of the next step = mid of current step + 1
Last of the next step = last of current step

Comparison.

- Is this a better strategy?
- Let’s consider how much of the sequence we eliminate at each step in the strategy.
- Linear search:
  - 1 word less at each step.
  - Reduce and conquer, step size of 1.
- Binary search:
  - Halve the number at each step.
  - Reduce and conquer, step size about $n/2$.

Using Linear Search for finding a Word

- A recent estimate stated that the number of words in the English language is 1,025,109.8
  - I’m not sure how you get 4/5 of a word either!
- If I start to look for a random word in this sequence using a linear search and can check one word every second, without error, I can expect to be done in 1,025,109.8/2 seconds.
- That is around six days!
  - If I don’t take any breaks to eat, drink, sleep etc.
  - 12 days in the worst case.
- What about binary search?

Using Binary Search for finding a Word

- We can see what will happen with binary search if we build a table.

Number of checks	Size of sequence remaining	Total time
0	1,025,109.8	0 sec
1	512555	1 sec
2	256277	2 sec
3	128138	3 sec
4	64069	4 sec
5	32034	5 sec
6	16017	6 sec

- How long will it take to get to a sequence of size 1?
- 20 seconds!

Comparing Linear Search and Binary Search for finding a Word

- So, that is 6 days or 20 seconds.
- I know which looks better to me!
- Hang on, though!
- I may have cheated.
- Can you see how?
- How did the sequence get sorted?
- How long did that take?

Sorting a sequence, a good idea?

- As we will soon see, it takes a lot of work to sort a sequence.
- Much more than it takes to search it!
- Even with linear search!
- This means that we would be crazy to sort the sequence before searching it.
- Except…
- what if we want to search it many times?
- We only have to sort it once.

Sorting a sequence, a good idea?

- So, provided we are going to search the sequence a lot, it makes sense to sort it.
  - Dictionaries.
  - Telephone books.
- Otherwise, don't bother.
- So… we have decided the sequence is worth sorting.
- How do we do it?
- We have lots of choices…

All sorts of sorts.

- We will start by looking at three simple sorting strategies:
  - Selection sort.
  - Insertion sort.
  - Bubble sort.
- Each strategy works in a slightly different way but includes two basic operations:
  - Compare two items in a sequence.
  - Swap two items in a sequence.
- What changes is the way these two operations are used.

What to sort.

- For the examples of sorting in this lecture I will use a sequence of positive integers instead of words.
- Why?
  - They are easier to type.
  - They take up less room.
  - It is easier to check if they are in order.
- The same sorting strategies will work on words too.
- In fact they will work on any sequence as long as we have well-defined compare and swap operations.

Selection Sort.

- Selection sort uses the following strategy:
  1. Start with the whole sequence.
  2. Find the smallest element in the sequence.
  3. Swap the first element with the smallest.
  4. Shorten the sequence by ignoring its first element (because it is in the right place).
  5. If the sequence has only one element, stop.
  6. Otherwise, go to step 1. with the shorter sequence.
- We can see how this works with an example.

Selection sort example.

- Starting sequence:

14	20	7	8	5	15	18	17	6	13

- Where does the sequence start?

14	20	7	8	5	15	18	17	6	13

- Where is the smallest number?

14	20	7	8	5	15	18	17	6	13

- Swap these numbers.

5	20	7	8	14	15	18	17	6	13

- Shorten the sequence by one.

5	20	7	8	14	15	18	17	6	13

Selection sort example.

- Starting sequence:

5	20	7	8	14	15	18	17	6	13

- Where does the sequence start?

5	20	7	8	14	15	18	17	6	13

- Where is the smallest number?

5	20	7	8	14	15	18	17	6	13

- Swap these numbers.

5	6	7	8	14	15	18	17	20	13

- Shorten the sequence by one.

5	6	7	8	14	15	18	17	20	13

Selection sort example.

- Starting sequence:

5	6	7	8	14	15	18	17	20	13

- Where does the sequence start?

5	6	7	8	14	15	18	17	20	13

- Where is the smallest number?

5	6	7	8	14	15	18	17	20	13

- It's the same number. No need to swap.

5	6	7	8	14	15	18	17	20	13

- Shorten the sequence by one.

5	6	7	8	14	15	18	17	20	13

Selection sort example.

- Starting sequence:

5	6	7	8	14	15	18	17	20	13

- Where does the sequence start?

5	6	7	8	14	15	18	17	20	13

- Where is the smallest number?

5	6	7	8	14	15	18	17	20	13

- It's the same number. No need to swap.

5	6	7	8	14	15	18	17	20	13

- Shorten the sequence by one.

5	6	7	8	14	15	18	17	20	13

Selection sort example.

- Let's speed this up

5	6	7	8	14	15	18	17	20	13

- Each step (cycle)

5	6	7	8	13	15	18	17	20	14
5	6	7	8	13	14	18	17	20	15
5	6	7	8	13	14	15	17	20	18
5	6	7	8	13	14	15	17	20	18
5	6	7	8	13	14	15	17	18	20
5	6	7	8	13	14	15	17	18	20

- And we are done!

How hard was that?

- To sort a sequence of 10 numbers we completed 9 cycles.
- But I cheated again!
- How did I find the smallest number each time?
- What I really need to do involves a bit (a lot) more work:

1. Assume the first element in the sequence is the smallest.
2. Note its value and location.
3. Compare the smallest value with each element in the sequence.
4. If it is smaller remember its value and location instead.

Selection Sort: Work done in Each Cycle

- Let's see that in action.
- Starting position:

5	6	7	8	13	14	18	17	20	15

- Smallest = 18
- Look at next element, is it smaller?

5	6	7	8	13	14	18	17	20	15

- Yes, new Smallest = 17

5	6	7	8	13	14	18	17	20	15

- Still 17

5	6	7	8	13	14	18	17	20	15

- Smallest =15, and we are done.
- Now we can swap 18 and 15.

Bridging Theory and Practice: A Simple Example

- All the problem solving strategies and techniques we are studying in this course can be applied to real world practices.
- Once we have worked out a solution, an algorithm can be designed and implemented as a computer program accordingly.
- Next slide will show the algorithms designed for selection sort discussed.
- This algorithm can be directly implemented in any programming language,

Bridging Theory and Practice: A Simple Example

```
selectionSort(A, n) {
// A is the sequence to be sorted, and n is the no of ele in the sequence
// Extend the sorted subsequence by including the smallest ele in the unsorted
subsequence
  for (i = 0 to (n-2)) {
    // Find the smallest ele in the unsorted sequence
    min_idx = i
    for (j = (i+1) to (n-1)) {
      if (A[j] < A[min_idx])
        min_idx = j
    }
    // Swap the smallest ele found with the 1st ele in the unsorted subsequence
    swap(A[min_idx], A[i])
}
```

How much work?

- So, to find the smallest element in each cycle we perform a comparison for each item left in the unsorted sequence.
- To sort a sequence of 10 items that is 9+8+…+2+1 = 45 comparisons.
- To sort a sequence of 10 items, 9 steps will be performed.
- And up to 9 swaps.
- To sort a sequence of N items, (N-1) steps will always be performed.
- For a sequence of *N* numbers we will carry out:
  - *1 + 2 +…..+* $(N - 1) = N \times (N - 1) / 2$ comparisons and $N - 1$ swaps
  - *The sum is an arithmetic series*
- That is, total no of operations = $N\times(N - 1)/2 + (N-1) \approx N^2/2$ operations, roughly (only consider the terms with highest degree).
- Perhaps we can do better.

Insertion Sort.

- Insertion sort uses the following strategy:
  1. Start with the second element in the sequence.
  2. Insert it in the right place in the preceding sequence.
  3. Repeat with the next unsorted element.
  4. Keep going until we have placed the last element in the sequence.
- We can see how this works with an example.

Insertion sort example.

- Starting sequence:

14	20	7	8	5	15	18	17	6	13

- Start at the second element.

14	20	7	8	5	15	18	17	6	13

- It's in the right place

14	20	7	8	5	15	18	17	6	13

- We have finished the step.

14	20	7	8	5	15	18	17	6	13

- Repeat.

Insertion sort example.

- Starting sequence:

14	20	7	8	5	15	18	17	6	13

- Start at the next element.

14	20	7	8	5	15	18	17	6	13

- Put it in the right place.

7	14	20	8	5	15	18	17	6	13

- We have finished the step.

7	14	20	8	5	15	18	17	6	13

- Repeat.

Insertion sort example.

- Starting sequence:

7	14	20	8	5	15	18	17	6	13

- Start at the next element.

7	14	20	8	5	15	18	17	6	13

- Put it in the right place.

7	8	14	20	5	15	18	17	6	13

- We have finished the step.

7	8	14	20	5	15	18	17	6	13

- Repeat.

Insertion sort example.

- Starting sequence:

7	8	14	20	5	15	18	17	6	13

- Start at the next element.

7	8	14	20	5	15	18	17	6	13

- Put it in the right place.

5	7	8	14	20	15	18	17	6	13

- We have finished the step.

5	7	8	14	20	15	18	17	6	13

- Repeat.

Insertion sort example.

- Again, lets speed it up.

5	7	8	14	20	15	18	17	6	13

- Each Cycle.

5	7	8	14	15	20	18	17	6	13
5	7	8	14	15	18	20	17	6	13
5	7	8	14	15	17	18	20	6	13
5	6	7	8	14	15	17	18	20	13
5	6	7	8	13	14	15	17	18	20

- And we are done

How hard was that?

- Again, to sort a sequence of 10 numbers we completed 9 cycles.
- But I cheated again!
- This time, how did I get the number into the right place?
- Again, I need to do more work:
  1. Find where the new element should be in the sequence so far.
  2. Move every element above this up by one place.
  3. Move the new element into the hole.

Insertion Sort: Work done in Each Cycle

- Let’s see that in action.
- Starting position:

7	14	20	8	5	15	18	17	6	13

- We need to insert the value 8, where does it go?

7	14	20	8	5	15	18	17	6	13

- Here!
- Move the elements up by 1.

7	14		20	5	15	18	17	6	13
7		14	20	5	15	18	17	6	13

- Insert the 8

7	8	14	20	5	15	18	17	6	13

Another way to look at it.

- We can combine the two steps:
  - Find the right place;
  - Put the element there.
- To do this, we simply move each element above the new element one place up until the right place for the new element is found.
- This, however, replaces moves with swaps which are slower.

How much work this time?

- So, to find the smallest element in each cycle we perform a comparison for each item in the sorted part of the sequence that is greater then the new item.
- To sort a sequence of 10 items, 9 steps will always be performed.
- This will involve, in the worst case, looking at each element.
- To sort a sequence of 10 items that is $1 + 2 + 3 + \ldots + 8 + 9 = 45$ comparisons.
- The same number of moves (excluding moving the new element).
- To sort a sequence of N items, (N-1) steps will always be performed.
- For a sequence of N numbers we will carry out:
  - $1 + 2 + \ldots + (N - 1) = N \times (N - 1)/2$ comparisons and up to $N \times (N - 1)/2$ moves.
  - The sum is an arithmetic series.
  - That is, total no of operations = $N\times(N - 1)/2 + N \times (N - 1)/2 \approx N^2$ operations, roughly (only consider the terms with highest degree).
- This does not look a lot better.

Bubble Sort.

- Bubble sort uses the following strategy:

1. Compare the first element with the next one.
2. If they are in the wrong order swap them.
3. Continue comparing until the end of the sequence.
4. Shorten the sequence by one – the last element is now in the right place.
5. Repeat from step 1.

- If you don’t do any swaps on any pass you stop.

Bubble sort example.

- Starting sequence:

14	20	7	8	5	15	18	17	6	13

- Compare elements 1 and 2.

14	20	7	8	5	15	18	17	6	13

- No swap needed. Now compare elements 2 and 3.

14	20	7	8	5	15	18	17	6	13

- Swap them. Compare elements 3 and 4.

14	7	20	8	5	15	18	17	6	13

- Repeat.

Bubble sort example.

- Keep going, swapping as needed.

14	7	20	8	5	15	18	17	6	13
14	7	8	20	5	15	18	17	6	13
14	7	8	5	20	15	18	17	6	13
14	7	8	5	15	20	18	17	6	13
14	7	8	5	15	18	20	17	6	13
14	7	8	5	15	18	17	20	6	13
14	7	8	5	15	18	17	6	20	13
14	7	8	5	15	18	17	6	13	20

- That is the first pass complete.

Bubble sort example.

- Let's do it again.

14	7	8	5	15	18	17	6	13	20
7	14	8	5	15	18	17	6	13	20
7	8	14	5	15	18	17	6	13	20
7	8	5	14	15	18	17	6	13	20
7	8	5	14	15	18	17	6	13	20
7	8	5	14	15	18	17	6	13	20
7	8	5	14	15	17	18	6	13	20
7	8	5	14	15	17	6	18	13	20
7	8	5	14	15	17	6	13	18	20

- That's two passes.

Bubble sort example.

- Let's just look at the end of each pass

7	8	5	14	15	17	6	13	18	20
7	5	8	14	15	6	13	17	18	20
5	7	8	14	6	13	15	17	18	20
5	7	8	6	13	14	15	17	18	20
5	7	6	8	13	14	15	17	18	20
5	6	7	8	13	14	15	17	18	20
5	6	7	8	13	14	15	17	18	20

- We didn't swap anything that time. We are finished.

5	6	7	8	13	14	15	17	18	20

Bubble Sort

- To sort a sequence of N items, upon completion of a pass, the sorting is completed, when any one of the following conditions is met:

  - No of passes = N-1, or
  - No swap during the pass

- Hence, to sort a sequence of N items, at most (N-1) passes is required.

How hard this time?

- Again, to sort a sequence of 10 numbers we complete at most 9 passes.
- For each pass we performed one less comparison.
- We also perform up to the same number of swaps.
- That is 9 + 8 + 7 + ... + 2 + 1 = 45 comparisons.
  - Actually, it was less than that because we stopped early.
- And some number of swaps, again at most 45.
- So, that is up to $N \times (N - 1) / 2$ comparisons and $N \times (N - 1) / 2$ swaps.
- Again, about N^2 operations.

What is so special about N^2 operations?

- Notice that the number of operations in all of these sorts involves around N^2 operations.
- Ok, selection sort uses half as many in the worst case.
- Does this mean all sorts take around N^2 operations?
- No.
- Notice that all three sorts we have looked at use reduce and conquer.
- We can do better.
- We can also do a lot worse!
- Let us look at one last sort.

QuickSort

- The quicksort strategy is a bit more difficult to understand but let us first try and explain it in English.
- Partition the sequence into two parts: the next few slides will explain the partitioning.
- The first part contains all the elements smaller than the original first element.
- The second part contains all the elements greater than or equal to the original first element.
- Repeat on each part.
- Keep splitting each sub sequence into two parts until we have a sorted sequence.

Partitioning.

- Let us examine in more detail how the partitioning process of quicksort works.
- Take the sequence [4, 8, 3, 5, 7, 2, 1] as an example.
- Our partition (or *pivot*) value is 4.
- Let us also mark the two ends of the remainder of the sequence.
  - Let us call these values *head* and *tail*.
- We now proceed as follows:

Partitioning.

1. Compare the pivot to the head.
   - If it is larger than head and the head has not reached the end, move head to the right (by one place) and repeat step 1.
   - Otherwise go to step 2.
2. Compare the pivot to the tail.
   - If it is smaller than or equal with tail and the tail has not reached the beginning, move tail to the left (by one place) and repeat step 2.
   - Otherwise go to step 3.
3. If head and tail have met or crossed over, swap the pivot with tail and stop.
   - Otherwise go to step 4.
4. Swap the values at head and tail
   - Move head to the right
   - Move tail to the left
   - Go to step 1

Partitioning in action

- Start:[4, 8, 3, 5, 7, 2, 1] (head = 8, tail =1)

- Step 1. compare 4 and 8
  - 8 >4 so go to step 2.
- Step 2. compare 4 and 1
  - 1 <4 so go to step 3.
- Step 4. swap head and tail and move them.
  - [4, 1, 3, 5, 7, 2, 8] (head = 3, tail =2)
- Step 1. compare 4 and 3
  - 3 < 4 so move head to the right and repeat step 1
  - [4, 1, 3, 5, 7, 2, 8] (head = 5)
- Step 1. compare 4 and 5
  - 5>4 so go to step 2
- Step 2. compare 4 and 2
  - 2 < 4 so go to step 3
- Step 4. swap head and tail and move them.
  - [4, 1, 3, 2, 7, 5, 8] (head = 7, tail =7)
- Step 1. compare 4 and 7
  - 7>4 so go to step 2
- Step 2. compare 4 and 7
  - 7>4 so move tail to the left and repeat step 2
  - [4, 1, 3, 2, 7, 5, 8] (tail = 2)
- Step 2. compare 4 and 2
  - 2<4 so go to step 3
- Step 3. swap 4 and 2 and stop.
  - [2, 1, 3, 4, 7, 5, 8]

Quicksort example.

- Starting sequence:

14	20	7	8	5	15	18	17	6	13

- Partition the sequence. Note that the 14 is in the right place.

6	13	7	8	5	14	18	17	15	20

- Partition the first half.

5	6	7	8	13	14	18	17	15	20

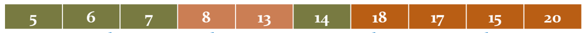
- Partition again. Only one element so it is in the right place.

5	6	7	8	13	14	18	17	15	20

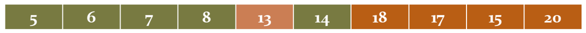
- Partition the next sub sequence. Nothing moved so 7 is in the right place too.

5	6	7	8	13	14	18	17	15	20

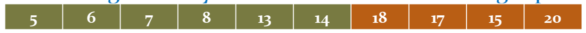
Quicksort example.



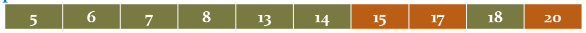
- Partition the next sub sequence. Nothing moved so 8 is in the right place too.



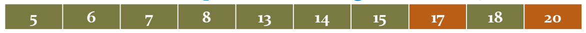
- Partition again. Only one element so it is in the right place.



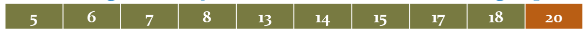
- Partition the second half of the sequence. Note that 18 is at the right place.



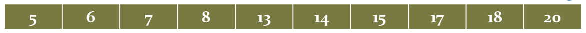
- Partition the sub sequence. Nothing moved so 15 is in the right place.



- Partition again. Only one element so it is in the right place.



- Partition the last part. Only one element so it is in the right place.



Q

How much work this time?

- To perform each partition we need to look at each element in the sub sequence.
- Each time the sub sequences split into two parts. Ideally, into halves.
- Ideally, we can show that quicksort uses around $N \times \log(N)$ operations: Here, we assume the ideal case, hence, the sequence will be partitioned k times such that $N/2^k = 1$, and each time requires N operations. Hence, $N = 2^k$. Therefore, $k = \log(N)$.
- This is considerably better than N^2 operations!
- So, should we always use quicksort?

Which sort is best?

- The best sort to use depends on a number of factors.
- How many items are we sorting?
  - If the number of items is small, quicksort is overkill.
- How disordered is the sequence?
  - If the sequence is nearly in order insertion sort or bubble sort work really well.
- If there are only a small number of items out of order:
  - Use insertion sort.
- If the items are nearly in order:
  - Use bubble sort.