CO322: DS & A
(Simple yet) efficient algorithms

Dhammika Elkaduwe

Department of Computer Engineering
Faculty of Engineering

University of Peradeniya

What is to come

We will look at;

- Efficient algorithms (for sorting/**searching**)
- Start with simple algorithms (not so efficient ones)
- Complexity analysis of recursive algorithms
- Algorithms vs. their implementation
- Features of programming languages

Searching algorithm

Look whether the given key is there.
Assume that there is no order/configuration within the given records.

```
static boolean linear_search(int [] records, int key) {
int i;
for(i=0; i < records.length; i++)
    if(records[i] == key) break;

return i != records.length;
 }
```

What is the complexity of this algorithm in best, worst and average cases. (Note: Fixed time, you call it $O(1)$)

Can we improve linear search?

```
static boolean linear_search(int [] records, int key) {
int i;
for(i=0; i < records.length; i++)
    if(records[i] == key) break;

return i != records.length;
 }
```

- One comparison takes out one element
- We need N comparisons (so $O(N)$ algorithms)
- Can we better this?

Binary search

- Records should be sorted (precondition for the algorithm)
- Compare with the middle element, if found done!
- Select the left or the right part of records based on the comparison (remember the array is sorted)
- One comparison will half the problem!!

- **Question:** I have 20million records. How many comparisons in the worst case?

Binary search

- Records should be sorted (precondition for the algorithm)
- Compare with the middle element, if found done!
- Select the left or the right part of records based on the comparison (remember the array is sorted)
- One comparison will half the problem!!

- **Question:** I have 20million records. How many comparisons in the worst case?

The binary search: a recursive implementation

```
 static boolean binary_search_impl(int start, int end,
     int [] records, int key) {
int mid = (int)((start + end) / 2);
if(start > end) return false;
if(records[mid] == key) return true;
if(records[mid] > key)
         return binary_search_impl(start, mid-1, records,
             key);
return binary_search_impl(mid+1, end, records, key);
}
```

```
/* wrapper for actual implementation */
static boolean binary_search(int [] records, int key) {
return binary_search_impl(0, records.length, records,
    key);
}
```

The binary search: Analysis

```
 static boolean binary_search_impl(int start, int end,
     int [] records, int key) {
int mid = (int)((start + end) / 2);
if(start > end) return false;
if(records[mid] == key) return true;
if(records[mid] > key)
          return binary_search_impl(start, mid-1, records,
              key);
return binary_search_impl(mid+1, end, records, key);
}
```

What is the time complexity of the above algorithm?
What is the space complexity of the above algorithm? (space that it uses)

The binary search: iterative implementation

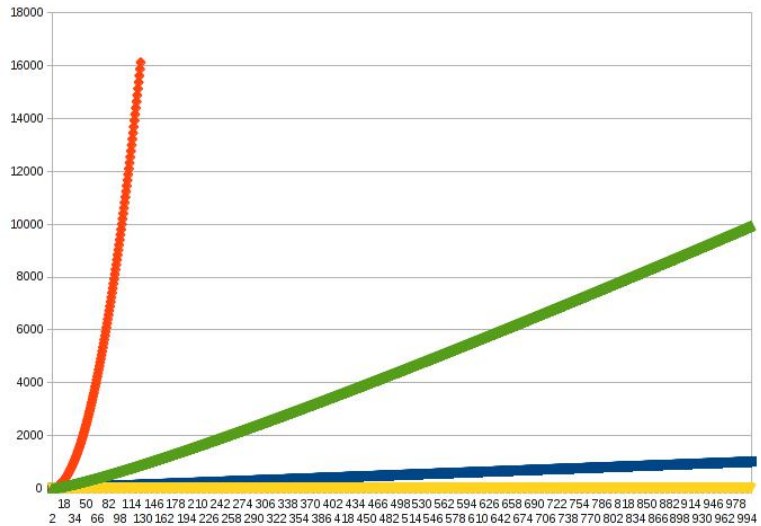
```
static boolean binary_search_ite(int [] rec, int key) {
int high = rec.length;
int start = 0;

while(start <= high) {
    int mid = (int)((high + start)/2);
    if(rec[mid] == key) return true;
    if(rec[mid] > key) high = mid-1;
    else               start = mid+1;
}
return false;
}
```

What is the time complexity of this?

Complexities

Note that graph only goes to 1000!

- $O(N^2)$ does not scale at all, not useful for any real problem
- $O(N)$ algorithms are somewhat OK
- $O(log(n))$ is the best, of cause :)



‘Power of‘ function

Another example:

```
static int power_of(int a, int k) {
  /* function should return a^k
  * Do I have preconditions?
  */
  int i = 1;
  for(; k < 1; k--)
      i = i * a;
  return i;
}
```

What is the complexity of this algorithm?
Can we better this? (recall: problem is halved)

"power of": better version

```
static int power_of(int a, int k) {
  /* function should return a^k
  * Do I have preconditions?
  */
  int i = 1;
  for(; k >= 1; k--)
        i *= a;
  return i;
}
```

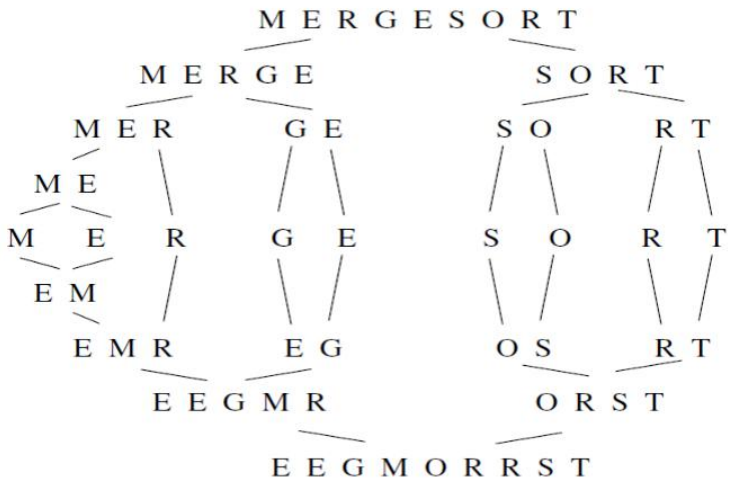
Can we improve:

- $n^{2k} = (n^k)^2$ and $n^{2k+1} = n^{2k} * n$
- Can you use this observation to make the above algorithm better? (code using recursion/iteration)
- What is the time complexity of the new version

Merge Sort: divide and conquer in a picture

Basic Idea: It is easy to merge two sorted arrays!

- Break the array into two parts (in the middle)
- (merge) sort the two halves
- Merge them together



Merge Sort: divide and conquer pseudocode

Basic Idea: It is easy to merge two sorted arrays!

- Break the array into two parts (in the middle)
- (merge) sort the two halves
- Merge them together

```
mergeSort(A, p, r)
     if p >= r, do nothing
     if p < r then
        q = |(p + r) / 2|
        mergeSort(A, p, q);
        mergeSort(A, q+1, r);
        combine(A, p, q, r)
```

Exercises:

- Implement the merge sort, using Java
- What is the time complexity of merge sort?
- What is the space complexity of merge sort? needs additional space)

Merge Sort: divide and conquer pseudocode

Basic Idea: It is easy to merge two sorted arrays!

- Break the array into two parts (in the middle)
- (merge) sort the two halves
- Merge them together

```
mergeSort(A, p, r)
     if p >= r, do nothing
     if p < r then
        q = |(p + r) / 2|
        mergeSort(A, p, q);
        mergeSort(A, q+1, r);
        combine(A, p, q, r)
```

Exercises:

- Implement the merge sort, using Java
- What is the time complexity of merge sort?
- What is the space complexity of merge sort? needs additional space)

Merge Sort: Not so good implementation

Merge function: needs new arrays each time!

```
static int [] combine(int [] a, int [] b) {
  int total = a.length + b.length;
  int [] merged = new int [total];
  int i=0, j=0, k;
  for(k=0; k < total && i < a.length && j < b.length;
      k++) {
    if(a[i] < b[j]) merged[k] = a[i++];
    else            merged[k] = b[j++];
  } /* for(k=0; .. */
  for(; k < total && i < a.length; k++, i++)
     merged[k] = a[i];
  for(; k < total && j < b.length; k++, j++)
     merged[k] = b[j];

  return merged;
}
```

Merge Sort: Not so good implementation

```
static int [] split(int [] a, int from, int to) {
  int [] newarray = new int [to - from];
  for(int i=0; i < newarray.length; i++) {
    newarray[i] = a[from + i];
  }
  return newarray;
}

static int [] merge_sort(int [] a) {
  if(a.length == 1) return a;

  int mid = (int)((a.length) / 2);
  int [] left = split(a, 0, mid); // excluding mid
  int [] right = split(a, mid, a.length);
  left  = merge_sort(left);
  right = merge_sort(right);
  return combine(left, right);
}
```

Merge Sort: Analysis

What is the time complexity of the **above implementation** of the merge sort?

- Time complexity = $O(N.log(N))$
- Space complexity = $O(N.log(N))$ (note that there are better ways of doing this. In the above implementation we allocate new arrays (while splitting) in each iteration.
- We only need $O(N)$ additional space, if we do a good implementation!!

Take home point: time and space efficiency depends on the implementation. Each algorithm has a best that it can do; but you may not get their unless implemented correctly.

Merge Sort: Better implementation

At most we need additional memory to hold N elements; use the same space for all temporary merges!

```
static void mergeSort(int [] data) {
  int [] tmp = new int [data.length];
  mergeSort(data, tmp, 0, data.length-1);
}

static void mergeSort(int [] data, int [] tmp, int start,
    int end){
  if(start < end) {
    int mid = (start + end) / 2;
    mergeSort(data, tmp, start, mid);
    mergeSort(data, tmp, mid+1, end);
    /* merge the two sorted arrays */
    merge(data, tmp, start, mid, end);
  }
}
```

Merge Sort: Better implementation

```
static void merge(int [] data, int [] tmp, int start, int
    mid, int end) {
  int total = end - start + 1;
  int i = start, j = mid+1, k;

  /* use tmp array for movements; starting from 0 */
  for(k=0; (k < total) && (i <= mid) && (j <= end); k++) {
    if(data[i] < data[j]) tmp[k] = data[i++];
    else                  tmp[k] = data[j++];
  }
  for(;k < total && i <= mid; k++, i++) tmp[k] = data[i];
  for(;k < total && j <= end; k++, j++) tmp[k] = data[j];

  /* copy the merged results back to the data array */
  for(k=0; k < total; k++)
    data[start + k] = tmp[k];
}
```

Merge Sort: Analysis

- The merge sort algorithm has, $O(N.log(N))$ time complexity and needs $O(N)$ additional memory. (need to implement properly, of cause)
- Next algorithm: can we get rid of the additional memory?

Quick sort

Basic algorithm:

- Pick some value from the array (call it **pivot**)
  - pivot can be the first element in the array, last element, or some random element.
  - You can use other ways to pick the pivot. Given above are some examples.
- Rearrange elements such that all elements less than to pivot are in front of the array followed by the pivot and elements which are greater than or equal to the pivot (called **partitioning**)
  - Partitioning is typically done by selecting an item smaller than pivot from the right side of the array and swapping it with an item larger than pivot which is in the left hand side of the array.
- After partitioning, the pivot is in the correct position.
- Quick sort the left and the right portions from the pivot

Quick sort: Partitioning example

choose pivot:	4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
search:	4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
swap:	4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
search:	4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
swap:	4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
search:	4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
swap:	4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
search: (left > right)	4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
swap with pivot:	1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

Quick sort: Implementation in Java

```
static void quickSort(int [] data) {
  quickSort(data, 0, data.length-1);
  /* including the last element, hence length-1 */
}

static void quickSort(int [] data, int start, int end) {
  if(start < end) {
    int pivotPosition = partition(data, start, end);

    /* including the last element, hence length-1
    * Pivot is in the correct position!
    */
    quickSort(data, start, pivotPosition-1);

    quickSort(data, pivotPosition+1, end);
  }
}
```

Quick sort: Analysis questions

Assume that the first element of the array is selected as the pivot. Ideally we need a pivot that would halve the problem.

- What is the best case? What is the corresponding time complexity?
- What is the worst case? What is the corresponding time complexity?
- What is the average case? What is the corresponding time complexity?

Quick sort: Analysis

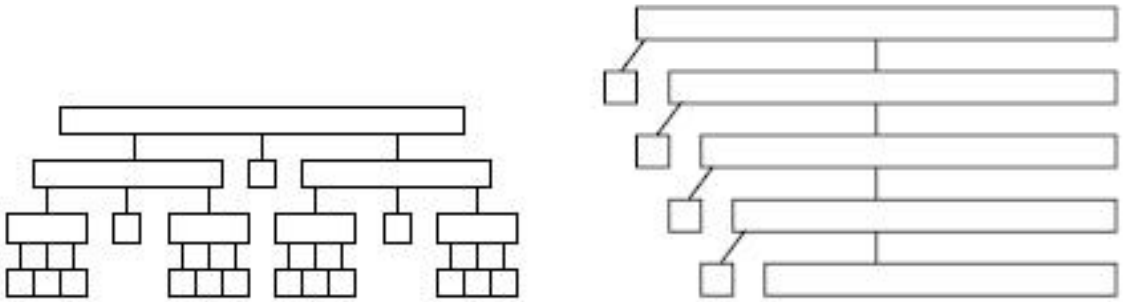


Figure 4.6: The best-case (l) and worst-case (r) recursion trees for quicksort

Case	When does it occur	Run time complexity
best case	pivot is the median	$O(N.log(N))$
worst case	pivot is the smallest	$O(N^2)$
average case	random data set	$O(N.log(N))$

Note: If you are selecting the first element as pivot, a sorted array gives the worst run time!

Quick sort: Selecting the pivot

Run time varies from $O(N.log(N))$ to $O(N^2)$ depending on the pivot.

- Ideally we need a pivot that would halve the problem. i.e. the median. But we cannot say what the median is without sorting the array!
- Methods for selecting a pivot:
  - First, last or middle element of the array
  - Random element
  - Pick 3 elements randomly and use median of those 3 as pivot.
- Make sure pivot selection is $O(1)$!
- No selection **guarantees** best case.

Quick sort: What is the average case?

So what happens in the average case? Everything depends on how the array is partitioned!

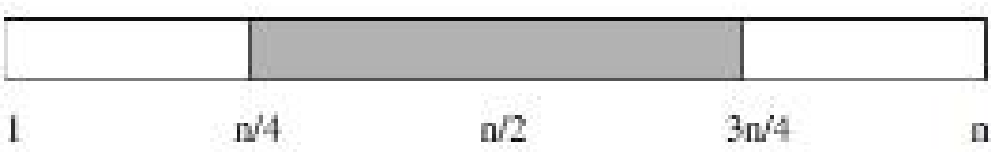
- We have N data items and 1 median value (assuming there are no duplicates)
- giving us $\frac{1}{N}$ chance of picking the median (and therefore getting the best case). Let us call it *ideal pivot*
- for a large N, this chance is very small.

continues in the next slide ..

Quick sort: What is the average case?...

We do not need the *ideal pivot*

- We need a *good pivot* which lies between $\frac{1}{4}$ and $\frac{3}{4}$ of the array (see figure)
- We have $\frac{1}{2}$ chance of getting a *good pivot*
- If we get a *good pivot* our worst case run-time is $O(log_{\frac{3}{4}} N)$ (longest path)
- If we do not get the *good pivot* (which happens $\frac{1}{2}$ the times) then partitioning will not break the array at all.
- So average number of partitions will be $O(2.log_{\frac{3}{4}} N) = O(log(N))$
- So average run-time is $O(N.log(N))$



Bucket sort: Sorting by partitioning

Basic idea (assume we need to sort student names):

- Group the names based on the first letter (giving us 26 groups or *buckets*)
- Buckets will be kept sorted
- Now we have a problem which is $\frac{1}{26}$ of the original
- Now sort each of the buckets with regards to the 2^{nd} letter
- Repeat for all α letters

Exercises:

1. Find the following for N names with α letters:
   1.1 the best case time complexity
   1.2 the best case space complexity
2. what is suitable *Java Collection* for implementing this?
3. If someone asks you to pictorially represent this sorting algorithm what will your answer looks like?

Bucket sort: Sorting by partitioning

Basic idea (assume we need to sort student names):

- Group the names based on the first letter (giving us 26 groups or *buckets*)
- Buckets will be kept sorted
- Now we have a problem which is $\frac{1}{26}$ of the original
- Now sort each of the buckets with regards to the 2^{nd} letter
- Repeat for all α letters

Exercises:

1. Find the following for N names with α letters:
   - 1.1 the best case time complexity
   - 1.2 the best case space complexity
2. what is suitable *Java Collection* for implementing this?
3. If someone asks you to pictorially represent this sorting algorithm what will your answer looks like?

Wrap-up:Practical issues

Some issues that you need to consider:

- Keys vs. records
- What to do with equal keys?
- Arrangement of data (should we pick a random permutation of given data so our *randomised* algorithms will work)
- Distribution of data (how many students have names starting with "z"? will this effect bucket sort?)
- Data in memory or in disk?

Wrap-up:What is expected from you

You should be able to:

- Implement a given algorithm using a suitable method (recursion/iteration) and data structures.
- Modify the simple algorithms to suite a new situation (tutorial).
- Analyse the time/space complexity of algorithms.
- Compare algorithms (based on their complexities as well as other considerations).
- Reason about performance of algorithms for a set of random inputs.

Wrap-up:Additional reading

- Chapter 2 & 4 of *The Algorithm Design Manual* (pdf is in Moodle)