**TTIT33 Algorithms and Optimization – Dalg Lecture 2**

**Lecture 2 Algorithms**

**Sorting**

Content:

Focus: complexity analysis
- Intro: aspects of sorting
- Comparison-based sorting:
  - Insertion sort, Selection sort, Quick sort, Merge sort
  - Theoretical lower bound for comparison-based sorting.
- Digital sorting:
  - bucket sort, radix sort
- Selection, median finding, quick select

**The Sorting Problem**

Input:
- A list \( L \) of data items with keys (the part of each data item we base our sorting on).

Output:
- A list \( L' \) of the same data items placed in order of non-decreasing keys.

**Aspects of Sorting:**

- Internal vs. External sorting:
  - Can data be kept in fast, random accessed internal memory – or...
- Sorting in-place vs. sorting with auxiliary data structures
  - Does the sorting algorithm need extra data structures?
- Worst-case vs. expected-case performance
  - How does the algorithm behave in different situations?
- Sorting by comparison vs. Sorting digitally
  - Compare keys, or use binary representation of data?
- Stable vs. unstable sorting
  - What happens with multiple occurrences of the same key?

**Insertion sort**

"In each iteration, insert the first item from unsorted part its proper place in the sorted part"

An in-place sorting algorithm!

Data stored in \( A[0..n-1] \)

Iterate \( i \) from 1 to \( n-1 \):

- The table consist of:
  - Sorted data in \( A[0..i-1] \)
  - Unsorted data in \( A[i..n-1] \)
- Scan sorted part for index \( i \) for insertion of the selected item
- Increase \( i \)

**Analysis of Insertion Sort**

Procedure InsertionSort (table \( A[0..n-1] \)):

1. for \( i \) from 1 to \( n-1 \) do
2. \( j \leftarrow i \)
3. \( x \leftarrow A[i] \)
4. while \( j \geq 1 \) and \( A[j-1] > x \) do
5. \( A[j] \leftarrow A[j-1] \)
6. \( j \leftarrow j-1 \)
7. \( A[j] \leftarrow x \)

- \( t_1: n-1 \) passes over this "constant speed" code
- \( t_2: n-1 \) passes...
- \( t_2: i \) passes
- \( t_3: n-1 \) passes
- \( T: 1 \times t_1 + 4 \times t_2 + 3 \times t_3 \)

Thus, we have an algorithm in \( O(n^2) \) in worst case, but .... good if the almost sorted.
Selection sort

In each iteration, search the unsorted set for the smallest remaining item to add to the end of the sorted set.

An in-place sorting algorithm!

Data stored in A[0..n-1]

Iterate i from 1 to n-1:

- The table consist of:
  - Sorted data in A[0..i-1]
  - Unsorted data in A[i..n-1]
- Scan unsorted part for index s for smallest remaining item
- Swap places for A[i] and A[s]

Analysis of Selection Sort

\[ t_1: \text{n-1 passes over this "constant speed" code} \]
\[ t_2: \text{n-1 passes...} \]
\[ t_3: I = \text{no. of iterations in inner loop:} \]
\[ I = n-2 + n-3 + n-4 + ... + 1 = (n-2)(n-1)/2 = n^2 - 3n + 2 \]
\[ t_4: I \text{ passes} \]
\[ t_5: \text{n-1 passes} \]
\[ T: t_1 + t_2 + t_3 + t_4 + t_5 = 3(n-1)+2(n^2-3n+2)/2 = n^2 - 1 \]

...thus we have an algorithm in \( O(n^2) \) ...rather bad!

Quick-Sort

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- Divide: pick a random element \( x \) (called pivot) and partition \( S \) into
  - \( L \) elements less than \( x \)
  - \( E \) elements equal \( x \)
  - \( G \) elements greater than \( x \)
- Recur: sort \( L \) and \( G \)
- Conquer: join \( L \), \( E \) and \( G \)

Quick-Sort Tree

An execution of quick-sort is depicted by a binary tree

- Each node represents a recursive call of quick-sort and stores
  - Unsorted sequence before the execution and its pivot
  - Sorted sequence at the end of the execution
- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1

Quick Sort – Analysis – worst case...

If the pivot element happens to be the min or max element of \( A \) in each call to quicksort...
(e.g., pre-sorted data)
- Unbalanced recursion tree
- Recursion depth becomes \( n \)

Sorting nearly-sorted arrays occurs frequently in practice

\[ \text{Worst-case time } T_w(n) = \sum_{k=0}^{n-1} k^2 = O(n^3) \]

Quick Sort – Analysis – best case...

Best – balanced search tree! How...
- If pivot is median of data set

\[ \text{Best-case time } T_b(n) = cn \log(n-1) + cn = O(n \log n) \]
In-Place Partitioning

Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

Repeat until j and k cross:
- Scan j to the right until finding an element > x.
- Scan k to the left until finding an element < x.
- Swap elements at indices j and k.

```
3  2  5  1  0  7  3  5  9  2  7  9  8  9  7
     j k
```

3  2  5  1  0  7  3  5  9  2  7  9  8  9  7

Merge Sort

```
25 17 19 32 41 43 31 12
25 17 19 32                           41 43 31 12
25 17                    19  32               41 43             12 31
25 17               19     32           41     43              12    31
17 25                    19  32               41 43             12 31
17 19 25 32                           12  31 41 43
12 17 19 25  31 32 41 43
```

Procedure

```
MergeSort (table T[a..b] ):
1. if a ≥ b then return else
2. Middle ← \lfloor \frac{a+b}{2} \rfloor
3. MergeSort(T[a..Middle])
4. MergeSort(T[Middle+1..b])
5. Merge(T[a..Middle] , T[Middle+1..b])
```

Comparison-based sorting

- A comparison at every step.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree.

![Decision Tree](image)

Lower bound on running time

- The height of this decision tree is a lower bound on the running time.
- Every input permutation leads to a separate leaf.
- If not, some input would have same output ordering as a separate leaf.
- \( n! = 1 \times 2 \times \ldots \times n \) leaves.
- Heights \( \log(n!) \).

Bucket-Sort – not comparison based...

- Don’t compare keys – use them as indices to a table B.
- If we know that all keys are in \([0..255]\)
  - Create a table B with room for 256 items,
  - “Move” all items into B using the key as index.
  - Copy them back in order to A.

Procedure

```
BucketSort (table <K> A[0:m]):
1. table <integer> B[0: |K|]
2. for i from 0 to m do B[A[i]]++
3. j ← 0
4. for i from 0 to |K| do
5. while B[i] > 0 do A[j++] ← i
6. |K| is the max number of different keys.
7. O(m + |K|) ... for fixed size |K|.
```
If keys are large – Radix Sort

- Divide keys in \( n \) smaller parts
  - ...suitable for Bucket Sort
- For all keys, focusing on the last part, sort using Bucket Sort.
- Redo sort using second last part.
- ...
- Sort using first part of each key.

Since Bucket sort is stable, Radix sort must be stable!

Radix Sort - Example

Example: alphabetic keys, split by character, buckets on letters

mary anna bill adam mona (input data)
anna mona bill adam mary (sorted by 4th letter)
adam bill anna mona mary (sorted by 3rd letter, keep relative order)
mary adam bill anna mona
adam anna bill mary mona

The Selection Problem

- Find the \( k \)-th smallest element in \( x_1, x_2, \ldots, x_n \).
- Sort in \( O(n \log n) \) time and index the \( k \)-th element?
- \( k = 3 \) \[ 7 \ 4 \ 9 \ 6 \ 2 \to 2 \ 4 \ 6 \ 7 \ 9 \]
- Can we solve the selection problem faster?

Quick-Select

- Quick-select is a randomized selection algorithm based on the prune-and-search paradigm:
  - Prune: pick a random element \( x \) (called pivot) and partition \( S \) into:
    - \( L \) elements less than \( x \)
    - \( E \) elements equal \( x \)
    - \( G \) elements greater than \( x \)
  - Search: depending on \( k \), either answer is in \( E \), or we need to recurse in either \( L \) or \( G \)

Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores \( k \) and the remaining sequence

Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores \( k \) and the remaining sequence
Proving time complexity of QuickSelect (cont.)

\[ t(n) \leq n \cdot g(n) + n\cdot 3n/4 \]

where \( g(n) \) is the no. of times we scan all data
(bad partitions), \( E[g(n)] = 2 \)

\[ g(n) \]

= the time to search the good partition, max
\( n\cdot 3n/4 \)

\[ E(t(n)) \leq E[n \cdot g(n) + n\cdot 3n/4] \]

\[ = n^2 E(g(n)) + E(n\cdot 3n/4) \]

\[ = n^2 + E(n\cdot 3n/4) \]

Recursively:
\[ E(t(n)) \leq n^2 \cdot 3n/4 \cdot 2 + E(n\cdot 3n/4) \]

\[ = n^2 + n^2 \cdot 3/4 + E(n\cdot 3/4) \]

Geometric series:
\[ E(t(n)) \leq n^2 + n^2 \cdot 3/4 + n^2 \cdot (3/4)^2 + \ldots \]

Hence expected running time is \( O(n) \)

We recurse into a partition of size \( n\cdot 3n/4 \), \( n\cdot 3/4 \) ... assume worst (largest) size