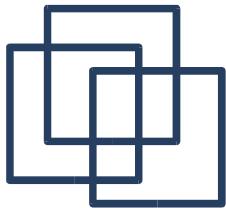


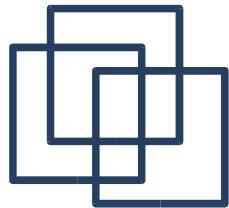
Algorithms and Architecture

Sorting



The Problem

- **Input:** a sequence of n numbers $\langle a_1, a_2 \dots a_n \rangle$
- **Output:** a permutation (reordering)
 $\langle a'_1, a'_2 \dots a'_n \rangle$ of the input sequence such that
 $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- Numbers to sort are also known as the **keys**



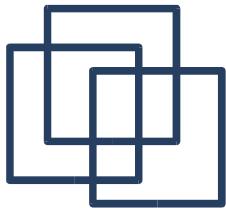
Sorting Algorithms

- Bubble sort
 - Selection sort
 - Insertion sort
 - Merge sort
 - Quick sort
 - Heap sort
 - application: priority queues
- n^2
- $n \log_2(n)$



Bubble Sort

- › Pseudo code, array from 1 to n , pb 2-2 p38
- › **for** $i:=1$ **to** $n-1$ **do**
 for $j:=n$ **downto** $i+1$ **do**
 if $a[j] < a[j-1]$ **then** swap($a[j], a[j-1]$)
- › Loop invariant: sub-array $a[1..i]$ is sorted before loop (on i).
- › Running time: $n-1+n-2+\dots+1 = n*(n-1)/2$
 $\Theta(n^2)$



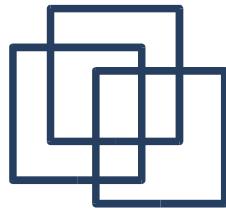
Selection Sort

- › Exercise 2.2-2 p27
 - › **for** $i := 1$ **to** $n - 1$ **do**
 for $j := i + 1$ **to** n **do**
 if $a[i] > a[j]$ **then** *swap(a[i], a[j])*
 - › Loop invariant: sub-array $a[1..i]$ is sorted before loop (on i).
 - › Running time: $n-1+n-2+\dots+1=n*(n-1)/2$
 $\Theta(n^2)$
-



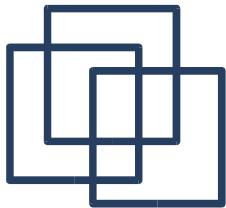
Insertion Sort

- › **for** $j := 2$ **to** n **do**
 - key := $a[j]$
 - $i := j - 1$
 - while** $i > 0$ and $a[i] > \text{key}$ **do**
 - $a[i+1] := a[i]$
 - $i := i - 1$
 - $a[i+1] := \text{key}$
 - › Loop invariant: sub-array $a[1..j-1]$ is sorted before loop (on j).
 - › Running time: $\Theta(n^2)$
-



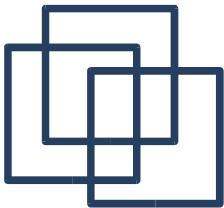
Analysis of Insertion Sort

- Count executions of loops
 - test n times, execute $n-1$ times (outer loop)
 - test t_j times, execute t_j-1 times (inner loop)
 - $c_1 + c_2 n + c_3 \sum_{j=2}^n t_j$
- Analysis
 - best case: $t_j=1$, $T(n)=\Omega(n)$
 - worst case: $t_j=j$, $T(n)=O(n^2)$
 - average case: $t_j=j/2$, $E[T(n)]=O(n^2)$



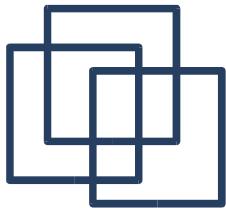
Merge Sort

- Divide-and-conquer approach
 - apply to algorithm that are *recursive*
 - **divide** main problem into sub-problems
 - **conquer** the sub-problems (solve recursively)
 - **combine** the solutions of the sub-problems
- Merge sort:
 - divide array n into 2 arrays of size $n/2$
 - sort 2 smaller arrays with merge sort
 - combine smaller arrays



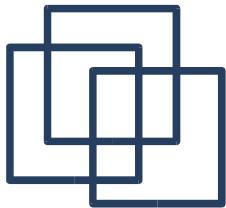
Merge Sort

- Key: merge sorted sub-arrays $A[p..q]$ and $A[q+1..r]$. For n elements merged $T(n)=\Theta(n)$.
 - **Merge-sort(a,p,r):**
if $p < r$ **then**
 $q := (p+r)/2$
 Merge-sort(a,p,q)
 Merge-sort(a,q+1,r)
 Merge(a,p,q,r)
 - **Merge** described using copies and sentinel values
-



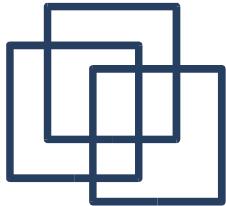
Analysis of Merge Sort

- › Running time $T(n)$ given as a recurrence. In general for divide-and-conquer algorithms:
 $T(n)=\Theta(1)$ if $n \leq c$ (c : small enough) otherwise
 $T(n)=aT(n/b)+D(n)+C(n).$
- › You are masters in recurrences, easy!
- › Merge-sort: $c=1$, $D(n)=1$, $C(n)=\Theta(n)$ (combine), divide: $a=2$ and $b=2$.
 $T(n)=\Theta(n \lg n)$



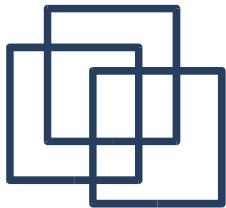
Quick Sort

- › Divide-and-conquer algorithm. Worst case $\Theta(n^2)$ but average case $\Theta(n \lg n)$ and in practice ***VERY*** efficient.
- › In place algorithm with small constants for the complexity. Fastest sort in practice except for “almost-sorted” inputs.
- › We will check it in the exercises.



Quick Sort

- › **Divide:** partition $A[p..r]$ into $A'[p..q-1]$ and $A'[q+1..r]$ s.t. $a'_{p..q-1} \leq a'_{q} \leq a'_{q+1..r}$. Compute q .
- › **Conquer:** sort $A'[p..q-1]$ and $A'[q+1..r]$ by recursive calls to quicksort.
- › **Combine:** $A[p..r]$ is sorted, nothing to do!
- › **Key:** partitioning. We can choose a'_{q} arbitrarily.



Quick Sort Partitioning

- › **Partition(A,p,r)**

```
x:=A[r] // pivot element
```

```
i:=p-1
```

```
for j:=p to r-1 do
```

```
  if A[j]≤x then
```

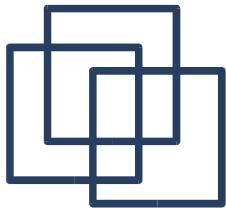
```
    i:=i+1
```

```
    swap(A[i],A[j])
```

```
swap(A[i+1],A[r])
```

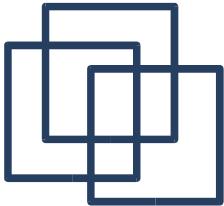
```
return i+1
```

- › Equivalent to selection-sort for worst case.
-



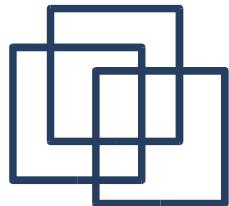
Performance of Quick Sort

- Depends on the input (balanced array or not) and the choice of the pivot.
 - Worst-case (sorted, reversed, or equal elements): $\Theta(n^2)$.
 - Best-case (balanced): $T(n) \leq 2T(n/2) + \theta(n)$, i.e.,
 $T(n) = O(n \lg n)$.
 - Average case?
 - remark: case 1/10 – 9/10 is still in $n \lg n$.
 - it is $O(n \lg n)$.
-



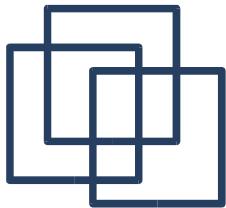
More Quick Sort

- › Randomized quicksort
 - There is no worst-case input, but an unlucky run may result in $T(n)=n^2$.
 - Choose the pivot randomly, i.e., swap a random element with $A[r]$.
- › Hoare partitioning (pb 7-1)
 - possibly fewer swaps
 - common $pivot:=A[(p+r)/2]$



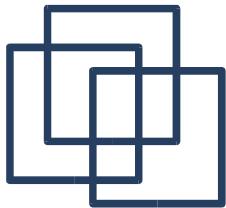
Heap Data Structure

- See it as a binary tree
- **root**= $A[1]$ (root of the tree)
- **parent(i)**= $i/2$ (for a node i)
- **left(i)**= $2i$ (left child)
- **right(i)**= $2i+1$ (right child)
- **max-heap** property: $A[\text{parent}(i)] \geq A[i]$
- or **min-heap** property: $A[\text{parent}(i)] \leq A[i]$
- **height** is $\Theta(\lg n)$.



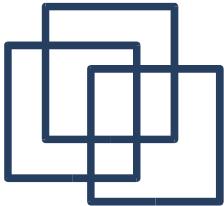
Heap Sort

- › Max-heap for sorting (ascending).
- › Basic procedures:
 - **max-heapify**: maintains max-heap property $O(\lg n)$.
 - **build-max-heap**: computes a max-heap $O(n)$.
 - **heapsort**: sorts an array in place $O(n \lg n)$.
 - others for priority queues (later).



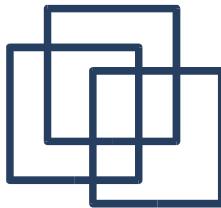
Maintaining a Heap

- **max-heapify(A,i)**
 - assume that **left(i)** and **right(i)** are max-heaps.
 - $A[i]$ may be smaller than its children.
 - (simplified) algorithm:
 $\text{child} := A[\text{left}(i)] > A[\text{right}(i)] ? \text{left}(i) : \text{right}(i)$
if $A[i] < A[\text{child}]$ **then**
 - $\text{swap}(A[i], A[\text{child}])$
 - $\text{max-heapify}(A, \text{child})$
 - when the algorithm terminates, the max-heap property is restored. Clearly $O(\lg n)$.



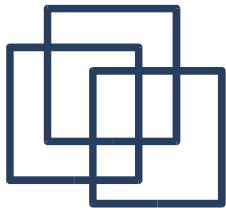
Building a Heap

- We can use **max-heapify** in a bottom-up manner: build the tree incrementally.
- **build-max-heap(A):**
 $\text{heap_size}[A]:=\text{length}(A)$
for $i:=\text{length}[A]/2$ **downto** 1 **do**
 max-heapify(A, i)
- Clearly $O(n \lg n)$.
- Why do we start from $\text{length}[A]/2$?



Heap Sort

- › Build a max-heap, put the root element at the end of the heap, max-heapify with a smaller heap, repeat.
- › **heap-sort(A):**
build-max-heap(A)
for $i := \text{length}(A)$ **downto** 2 **do**
 swap($A[1], A[i]$)
 heap_size(A)--
 max-heapify($A, 1$)
- › Clearly $O(n \lg n)$, but also $\Omega(n \lg n)$ (bad).



Priority Queues

- Heaps are useful for priority queues because of the max(or min)-heap property.
- Operations:
 - insert an element
 - get the max (or the min) element
 - remove the max (or the min) element
 - increase key of an element
- Used in the STL