# Course Notes - what did we study, what are we going to study

Dear student: The purpose of these notes is to list what we studied so far, what we are going to study, give pointers to textbooks and web resources etc. These notes do not intend to replace being in the classroom, but are trying to help in the unfortunate event that it occurred.

**Till 9/9/02** Introduction, several methods to find the largest empty rectangles in a binary array.

**Till 9/18/02** Examples of divide-and-concur algorithms, and related subjects:

1. Partition of an array, according to a pivot (a procedure similar to the one shown in CLR).
2. Randomized methods for selecting the $i$th smallest element in an array, and mentioned the idea of picking 5 elements at random, and use their median as the pivot.
3. A linear time deterministic algorithm for the $i$th smallest element (CLR).
4. We show that the expected time of QuickSort algorithm for sorting an array of $S$ of $n$ elements is $O(n \log n)$, by showing that this is the expected number of comparisons between pairs of elements. We next showed that the probability that two elements $x, y \in S$ (for $x < y$) are compared in the course of the algorithm is $2/|\{z \in S | x \leq z \leq y\}|$. The expected running time is therefor the sum of these expression over all pairs $x, y \in S, (x < y)$.
5. We study how to dynamically maintain a set of elements (i.e. when elements can be inserted and deleted) and maintain their median. This was obtained by storing the elements in a balanced search tree, and keeping a pointer to the median element. This pointer is updated using *successor* and *predecessor* operations in the tree, after each insertion and deletion.
6. We study how to maintain a dynamically a set of elements, and be able to answer $i$th largest element queries, where each operation takes $O(\log n)$ time where $n$ is the number of elements. This was obtained by augment a balanced search tree, where each node $v$ of the tree maintains the number of elements in the subtree rooted at $v$. You can find more about this idea in CLR.
7. Started amortized analysis.

**9/26/02** We discussed several issues in this meeting. We showed the the time analysis for the dynamic array (array which expands and shrinks) can be analysis also by the accounting method, where every element inserted to the array has \$3, one for itself, one for the time it would be copies and one for another element which would be copied at some time in the future. Once an element is deleted, it contributes its dollar (if exists), plus another dollar receied we invest, to another element. We show that once an element need to be copied,

Given a set $L$ of $n$ lines in the plane, the lower envelope of $L$ is defined as the set of points which are on a line of $L$, and no other line of $L$ is vertically strickly below these points. We shown an $O(n \log n)$ time algorithm, for computing the lower envelope of $L$, as follows. We sort the lines of $L$, based on their slope. Let $\ell_1 \dots \ell_n$ be the resulting sequence. We maintain the lower envelope $F_i$ of $\{\ell_1 \dots \ell_i\}$, as a linked list of segments. To insert $\ell_{i+1}$, we observe that all segments of $F_i$ to the right of $\ell_{i+1}$ do not participate in any $F_j$, for $j > i$. So we scan $F_i$ from right to left, seeking the intersection point with $\ell_{i+1}$. We spend $O(1)$ time for every segment we scan, and does not intersect with $\ell_{i+1}$. The line containing this segment could be avoided for future considerations. Hence only $O(n)$ such case can occur.

Finally we study the basic operations in splay trees.

**Near future** Union/Find data structure (can be found in CLR).

**Initialization of an array in $O(1)$ time** We assume that an arrays $A$ is going to be used — maybe more than once. The interfene at which the user is going to access the arrays are the opertaions `init`$(A, d)$ which virtually put the deault value $default[A] = d$ in each cell of $A$). The user also uses the command `put`$(A, i, x)$ which put the value $x$ in the cell $A[i]$. We say that $A[i]$ is garbadge if we the user did not insert a value to $A[i]$. The thier command is `Get`$(A, i)$ which returns either $default[A]$ if $A[i]$ is garbadge, and $A[i]$ otherwise.

To be able to accomplished each operation in time $O(1)$, we use two extra arrays $S$ and $P$, each contains $n$ intergerst between 1 and $n$ ($n$ here is the size of $A$). We also use a counter $cnt$.

We implement the procdures as follows:

```
Function Get$(A, i)$
    If garbage$(A, i)$   Then Return $d[A]$
        Else Return $A[i]$

Function Init$(A, d)$
    cnt[A] =0

Function Put$(A, i, x)$
    If Not garbage$(A, i)$   Then $A[i] = x$
    Else
        A[i]=x
        $cnt[A]++$
        $S[cnt[A]] = i$
        $P[i] = cnt[A]$

Function garbage$(A, i)$ /* Boolean function — return "true" if $A[i]$ is garbage */
    Return ( $P[i] > cnt[A]$ ) Or ($i \neq S[P[i]]$ )
```

**10/26/02** Dynamic Programming (including edit distance and triangulation of convex polygon. The discussion about application to speach recognition is not included in the material for the exam)

**Greedy Algorithms**

## 1 Dijkstra's algorithm

Given a graph $G(V, E)$ in which every edge $(u, v) \in E$ is associated with a weight $w(u, v) \geq 0$, and a vertex $s \in V$, we studied Dijskstra algorithm for finding the shortest path from a $s$ to every vertex in a $V$. Here the length of a path is the sum of weights of edges on the path. For simplicity, we assume that no two different paths of the graph has the same weight. For every $v \in V$ let $\pi^*(v)$ denote the shortest path from $s$ to $v$. Let $\delta(v)$, denote the length of this path.

The output of the algorithm is the arrays $d[\cdot], P[\cdot]$, where $d[v]$ contains upon terminations of the algorihm (as we show below), $\delta(v)$ $(\forall v \in V)$, and $P[v]$ contains the vertex proceeding $v$ on $\pi^*[v]$.

```
Dijkstra Algorithm
$d[s] = 0, \quad P[v] = NULL, \quad d[v] = \infty, \forall v \neq s, \quad S = \emptyset$
While $(S \neq V)$
    Pick $u \in V \setminus S$ that minimizes $d[v]$
    Add $u$ to $S$
    For every edge $(u, v) \in E$
        If $d[v] > d[u] + w(u, v)$   Then
            $d[v] = d[u] + w(u, v)$
            $\pi[v] = u$
Output $P[\cdot], d[\cdot]$
```

The proof of the following lemma is left to the reader (that is, you!).

**Lemma 1.1**

1. *At every stage of the algorithm, and for every $v \in V$ such that $d[v] < \infty$, $d[v]$ contains the length of a path $s \to v$ (though not necessarily optimal. Hence at any stage, $d[v] \geq \delta(v)$.*
2. *Let $x$ be a vertex on $\pi^*[v]$, and let $\pi'$ be the portion of $\pi^*[v]$ from $s$ to $x$. The $\pi' = \pi^*[x]$.*
3. *Let $s = u_1 \ldots u_n$ denote the vertices of $V$ in the order they join $S$, and let $d^J[u_i]$ denote the value of $d[u_i]$ when $u_i$ joined $S$. Then $d^J[u_1] \leq d^J[u_2] \leq d^J[u_n]$.*

We define a path $\pi$ to be *$S$-internal*, if all its nodes, except maybe the last one, belong to $S$. We define $\pi$ to be $S$-optimal if it the shortest among all paths which are $S$-internal, and connect $s$ to the endpoint of $\pi$.

**Lemma 1.2**

1. *At every stage of the algorithm, after finishing the inner loop, and for every $v \in V$, $d[v]$ contains the length of an $S$-optimal path.*
2. *For every $u \in S$, $\delta(u) = d[u]$.*

*Proof:* The proof is by induction on $|S|$. Both claims are are clearly true when $|S| = 1$, since then $S$ contains only $s$, and the claim is easily established. So assume that the cliams are true for $|S| = k - 1$, and we prove them for $|S| = k$, when $u$ joined $S$. Let $S'$ denote $S$ before

$u$ joined (i.e. when $|S| = k-1$). Let $\pi^*_S(u)$ and $\pi^*_{S'}(u)$ denote the $S$-optimal and $S'$-optimal paths to $u$, respectively.

The first claim is obviously true for $u$ itself, since it was true before $u$ joined $S$, and the (new) $S$-optimal path must be identical to the $S'$-optimal path.

Next consider $y \in V \setminus S$. If $\pi^*_S(y)$ does not pass through $u$, then $\pi^*_S(y)$ is also an $S'$-optimal path, and there is nothing to prove.

Hence assume that $\pi^*_S(u)$ passes through $u$. Observe that $u$ must be the vertex of $\pi^*_S(u)$ proceeding $y$. If this is not the case, then $\pi^*_S(y)$ passes through a vertex $w \in S$ immediately after reaching $u$. Based on the Lemma 1.1, the portion $\pi'$ of $\pi^*_S(y)$ from $s$ to $w$ must be also be $S$-optimal, contradicting the induction hypothesis that the $S'$-optimal path to $w$ is also global optimal (i.e. $\pi^*_{S'}(w) = \pi^*(w)$).

Finally, we need to show that the $\pi^*_S[u] = \pi^*[u]$ (the second claim of the lemma). Assume that this is not true. $\pi^*[u]$ starts in $s \in S$, so it must leave $S$ at some vertex (otherwise it is $S$-internal). Let $z$ be the first vertex of $\pi^*[u]$ outside $S$, and let $\pi'$ be the portion of $\pi^*[u]$ from $s$ to $z$. Note that $\pi'$ is $S$-internal, but is also on a (globally) optimal path, hence $\pi' = \pi^*[z]$, and by the induction hypothesis, $d[z] = \delta(z)$. The length of $\pi^*[u]$ is at least the length of $\pi^*[z]$. However, since $u$ was chosen (and not $w$) we deduce that

$$d[u] > d[z] = \delta(z) > \delta(u)$$

Which contradicts Lemma 1.1. ■

**Ford & Fulkerson Algorithm** Finding shortest path from $s$ to every vertex — arbitrary weights.

**Stable marriage**

**Skip List**

**11/4/02** Finding the closest pair in $O(n)$ expected time.

**All pairs shortest path**

**Network flow** Edmons & Karp algorithm.