

CS445

Brief Class Notes

In this note, I will provide some brief descriptions and pointers to the material discussed in class. I will try to give some useful information, but these notes cannot replace being in class and/or read the text.

Asymptotic notations and recursive formulas After a brief introduction, we discussed the asymptotic running time notations: big- O , Ω and Θ . We showed several examples. This material is well covered in the text-book, and hence we do not provide more details here. We analysed the running time of a few examples. One, of particular interest, is the following

```
read(n)
for( i = 1 ; i ≤ n ; i ++ )
    for( j = 1 ; j ≤ n ; j + = i )
        print( "*" );
```

We showed that its running time is $\sum_1^n 1/i = O(n \log n)$.

Next we moved to recursive formulas. We analysed using the iterative method (see details below) simple recursions, such as the function *NoNeed(n)*.

```
NoNeed(n)
read(n)
for( i = 1 ; i ≤ n ; i ++
```

Whose recursion formula is $T(n) = cn + T(n - 1)$, (for some constant c) and $T(1) = c$. Here as easily seen,

$$T(n) = cn + T(n) =$$

$$Tcn + (c(n - 1) + T(n - 1)) =$$

$$cn + c(n - 1) + c(n - 2) + T(n - 2) = \dots \text{ after } k \text{ stages}$$

$$c(n + (n - 1) + (n - 2) + (n - 3) + \dots + (n - k)) + T(n - k) = \text{ setting } k = n - 1$$

$$c(n + (n - 1) + (n - 2) + \dots + 2) + T(1) = c \sum_{i=1}^n i = cn(n + 1)/2$$

Hence $T(n) = \Theta(n^2)$.

Master theorem Prof. Kececioglu, who replaced me, introduced the Master Theorem, which is used to solve a large class of recursion formulas. The textbook covers this area.

Counting Sort and Radix Sort I showed Counting and Radix sorts methods. I taught it from the slides, who cover this area very well. This is also a good place to thank Charles E. Leiserson, Piotr Indyk and Carola Wenk, who all prepared previous versions of the slides I am using.

Lower bound on sorting in a comparison model Please look at the slides.

SkipList I taught a version of the skip list at which each element is a single cell (other versions, such as the one from described in the textbook “Structures & Their Algorithms, (Larry & Denenberg)” at which each key is stored in an array. We give intuitive arguments (handwaving) for the following claims, which would be proven formally later.

1. The expected number of levels in the SL is $O(\log n)$.
2. The expected size (memory used) is $O(n)$.
3. The expected search time is $O(\log n)$. This dominates the expected time for insertion and deletion operations, and also of **successor** operation. The operations $\text{succ}(x)$ finds the smallest element in the data structure which is strictly larger than x .

Analyzing the running time of QuickSort Consider sorting the keys $\{k_1 \dots k_n\}$.

We assume that we use a version of QuickSort at which all elements are different, and the probability of each element to be picked as a pivot is uniform. We showed that the running time is proportional to the number of pairs of elements which are compared to each other (which happens when one of these elements is the pivot). We define a random variable X_{ij} which is 1 if i and j are compared at some point at the course of the

algorithm k_i and k_j are compared, and 0 otherwise. Note that $E(X_{ij})$, the expected value of X_{ij} is

$$E(X_{ij}) = 1 \cdot Pr(X_{ij} = 1) + 0 \cdot Pr(X_{ij} = 0) = Pr(X_{ij} = 1) .$$

Note that the running time is $O(\sum_{1 \leq i < j \leq n} X_{ij})$, and the expected running time is

$$E\left(\sum_{1 \leq i < j \leq n} X_{ij}\right) = \sum_{1 \leq i < j \leq n} E(X_{ij}) = \sum_{1 \leq i < j \leq n} Pr(X_{ij} = 1).$$

To evaluate $Pr(X_{ij} = 1)$, we only need to note that the algorithm compares k_i and k_j if and only if either k_i or k_j were the first key to be picked as pivot, from the set $\{k_i, k_{i+1} \dots k_j\}$. Since each of them has the same probability to be picked, and there are $j - i + 1$ keys between (and including) k_i and k_j , we figure that $Pr(X_{ij} = 1) = \frac{2}{j-i+1}$. Thus

$$E\left(\sum_{1 \leq i < j \leq n} X_{ij}\right) = \sum_{1 \leq i < j \leq n} \frac{2}{j-i+1}.$$

Since $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = O(\log n)$, we conclude that

$$E\left(\sum_{1 \leq i < j \leq n} X_{ij}\right) = \sum_{1 \leq i < j \leq n} \frac{2}{j-i+1} = n \cdot O(\log n) = O(n \log n)$$

Augmenting data structures We demonstrated this structure both on SkipLists and on binary search trees. The idea was to assign extra information to each node of the SL. For example, if our goal is to be able to answer how many elements are smaller than a query key x , we maintain with each node c of the SL the field $c.n$ containing the number of keys between $key[c]$ and $key[next[c]]$. To find the number of elements smaller than x , we perform $\mathbf{find}(x)$, and sum the values of the fields $c.n$ of the nodes at which the search path turns right at node. These fields can also help is find the k 'th smallest element in the list, in time $O(\log n)$.

Details about this implementation in search trees can be found in the chapter on augmented data structures in the textbook.

—— Material for the final starts here ——

Graphs — in general General graphs. The handshaking Lemma. We proved that in any party there is an even number of people who shook an odd number of hands. We discussed two main issues to store graph in computers — adjacency matrix and adjacency lists. We next discussed connectivity in graphs, and Prim algorithms for finding MST (Minimal Spanning Trees)

Graphs and Spanning Trees — in general General graphs. The handshaking Lemma. We proved that in any party there is an even number of people who shook an odd number of hands. We discussed two main issues to store graph in computers — adjacency matrix and adjacency lists. We next discussed connectivity in graphs, and Prim algorithms for finding MST (Minimal Spanning Trees)

Shortest Paths in graphs with positive weights. We discussed the “standard” Dijkstra algorithm, for finding the shortest paths from a vertex s to all vertices of a graph at which every edge is associated with a positive weight. See the relevant slides. We discussed separately how to find the lengths of these paths, and then how to find the actual path, using the notion of *shortest paths trees*. We studied BFS as a simple special case of Dijkstra’s algorithm for graphs, and showed that in this case the running time can be improved by a factor of $O(\log |V|)$, since a queue can replace the heap as the main data structure.

Shortest Paths in graphs with positive weights. (5/18/05) Prof. Kececioglu showed the Bellman-Ford (BF) algorithm for finding the shortest paths from a vertex s to all vertices of a graph at which every edge is associated with a weight (which can be positive or negative).

Again — see the slides. This algorithm also checks for the existence of *negative cycle* in the graph.

Johnson algorithm We discussed Johnson algorithm for finding the shortest path between **every** pair of vertices in the graph with positive and negative weights. This algorithm converts the graph $G(V, E)$ into a graph $\hat{G}(V, E)$ with different weights to its edges, all positive, with the property that for each $u, v \in V$, the shortest path between u and v in G and in \hat{G} is the same. See the textbook, and the slides.

Max Flow in networks We studied the min-cut, max-flow theorem, Ford&Fulkerson algorithm. We also describe (without the analysis) of Edmonds&Karp algorithm, which is similar to Ford&Fulkerson, but always augments along the path with the minimum number of edges in G_f . See the slides.

Tues April 5 and Thursday April 7 — Computational Geometry. **Line Sweep algorithm.** Using cross products to find intersections between segments. We presented a Line-sweep algorithm that checks in time $O(n \log n)$, whether a set of S segments in the plane contains any pair of segments that cross each other. We also discussed in class a slight modification of the algorithm that computes all the intersection points. Knowing this modification is not required. We started to speak about the next subject, which is **closest pair**.

(April 12): Closest pair and convex Hull. In this problem we are given a set P of n points in the plane, and need to find the closest pair. We present an algorithm that solves this problem in time $O(n \log n)$. It can be shown (not in this course) that $\Omega(n \log n)$ is a lower bound to this problem, but an algorithm with expected time $O(n)$ exists.

Next we present the convex hull of a set of points in the plane, and present an algorithm that computes the convex hull in time $O(n \log n)$.

4/20/04 Dynamic programming. Longest Common subsequence. Matrix Multiplications. Floyd-Floyd-Warshall Algorithm for finding all pairs shortest paths (found in Section 25.2 of CLRS).

4/22/04 Finishing all pairs shortest paths, and starting stable marriage. Material is in the slide, and in different textbooks. It can be found on the rack next to Alon's office's door.

4/22/04 Finishing stable marriage, and introducing approximation algorithms for NP-hard problems. In particular, we would discuss 2-approximation algorithm for vertex-cover problems, and for Travelling Salesperson Problem (TSP).