Chapter 6

Linear time algorithms

By Sariel Har-Peled, December 2, 2021[1] Version: 0.3

Here, we investigate some cool algorithms for solving problems in linear time. All these algorithms use the ***search & prune*** technique – the idea is that you repeatedly reduce the input size, till it is small enough that you can

6.1. Deterministic median selection in linear time

6.1.1. Preliminaries

Definition 6.1.1 (Rank of element.). Let X be an unsorted array (i.e., set) of n integers. For j, $1 \le j \le n$, element of ***rank*** j is the jth smallest element in X. See Figure 6.1 for an example.

The ***median*** of X is the element of rank $j = \lfloor (n+1)/2 \rfloor$ in X.

(For simplicity, we assume all the elements of X are distinct.)

Unsorted array	16	14	34	20	12	5	3	19	11
Ranks	6	5	9	8	4	2	1	7	3
Sorted array	3	5	11	12	14	16	19	20	34

Figure 6.1: An example of an array and the ranks of its elements.

Problem 6.1.2 (Selection). The input is an unsorted array X of n integers, and an integer j.

The task is to compute the jth smallest number in X (i.e., the element of rank j) in X.

Naive algorithms for selection. The naive algorithm sorts the inputs of X, and returns the jth element in the sorted array. This takes $O(n \log n)$ time. It is natural to ask if one can avoid sorting and get a linear time algorithm.

If j is small or $n - j$ is small then one can compute the j smallest/largest elements in X in $O(jn)$ time (by finding the minimum, deleting it, and repeating). This still fails for the median, as $j = n/2$, and the resulting running time is $O(n^2)$.

6.1.2. Divide and conquer

6.1.3. quickSelect: The divide and conquer approach

The *randomized* quickSelect(X, j) works as follows:

- It "picks" a pivot element x from X.
- Partition X based on x into two sets: $X_{\text{less}} = \{y \in X \mid y \leq x\}$ and $X_{\text{greater}} = \{x \in A \mid y > x\}$.
- If $|X_{\text{less}}| = j$: return a
- **if** $|X_{\text{less}}| > j$: recursively find jth smallest element in X_{less}
- **if** $|X_{\text{less}}| < j$: recursively find kth smallest element in X_{greater}, where $k = j - |X_{\text{less}}|$.

Running time analysis. The partitioning step takes $O(n)$ time. But how do we choose the pivot?

Say the algorithm always choose the pivot to be $X[1]$, but then if X is sorted in increasing order, and $j = n$. then the running time of quickSelectis $\Omega(n^2)$.

We need a better pivot. Suppose we somehow magically (using wishful thinking) chose the pivot so it is the ℓth smallest element in X, where $n/4 \leq \ell \leq 3n/4$. Namely, the pivot is *approximately* in the middle of X.

Then, it is easy to verify that

$$n/4 \leq |X_{\text{less}}| \leq 3n/4 \qquad \text{and} \qquad n/4 \leq |X_{\text{greater}}| \leq 3n/4.$$

We get the following recurrence for the running time:

$$T(n) \leq T(3n/4) + O(n),$$

and it is not hard to verify that $T(n) = O(n)$.

In the standard implementation of the algorithm, one chooses the pivot as a random element in X. The analysis is more complicated, and we will address it later. While this *randomized* algorithm can, in the worse case, run in linear time, in practice it is amazingly fast.

6.1.4. A deterministic algorithm: Median of medians

The natural approach is to try to try to divide the problem into many subarrays, compute the medians of the subarrays, and merge them together. The resulting algorithm is the following:

(I) Break input X into many subarrays: $L_1, \ldots L_k$.
(II) Find median m_i in each subarray L_i.
(III) Find the median x of the medians $m_1, \ldots, m_k$.
(IV) Intuition: The median x should be close to being a good median of all the numbers in X.
(V) Use x as pivot in previous algorithm.

Example 6.1.3. The input is the array (which is already written as matrix with five rows).

75	31	13	26	83	110	60	120	63	30	3	41	44	107	30	23	91	17	6	110
68	24	41	26	58	57	61	20	52	45	13	79	86	91	55	66	13	103	36	60
19	40	45	111	56	74	17	95	96	77	29	65	36	96	93	119	9	61	3	9
100	3	88	47	115	107	79	39	109	20	59	25	92	81	36	10	30	113	73	116
72	58	24	16	12	69	40	24	19	92	7	65	75	41	43	117	103	38	8	20

We compute the median for each column. This results in the following numbers:

75	**31**	13	26	83	110	**60**	120	**63**	30	3	41	44	107	30	23	91	17	6	110
68	24	**41**	**26**	**58**	57	61	20	52	**45**	**13**	79	86	**91**	55	**66**	13	103	36	**60**
19	40	45	111	56	**74**	17	95	96	77	29	**65**	36	96	93	119	9	**61**	3	9
100	3	88	47	115	107	79	**39**	109	20	59	25	92	81	36	10	**30**	113	73	116
72	58	24	16	12	69	40	24	19	92	7	65	**75**	41	**43**	117	103	38	**8**	20

We compute the median of these medians. Namely, the median of the numbers in the array:

72	74	13	66
31	60	65	30
41	39	75	61
26	63	91	8
58	45	43	60

The returned value is 60, and we now use it to partition After partition (pivot 60), the resulting array is:

19	3	13	16	12	57	17	20	19	20	3	25	92	109	96	79	110	69	83	75
41	24	24	26	56	17	40	24	52	30	7	60	77	81	63	61	107	115	111	72
20	31	41	26	58	30	60	39	36	45	13	65	75	91	120	66	74	61	88	68
9	40	45	47	3	13	23	55	30	44	29	65	86	96	95	117	91	103	100	110
36	58	8	6	38	9	10	43	41	36	59	79	92	107	93	119	103	113	73	116

The pivot has rank 57. As such, we recurse on the subarray containing the smaller elements. In this array:

19	3	13	16	12	57	17	20	19	20	3	25
41	24	24	26	56	17	40	24	52	30	7	
20	31	41	26	58	30	60	39	36	45	13	
9	40	45	47	3	13	23	55	30	44	29	
36	58	8	6	38	9	10	43	41	36	59	

we are computing the element of rank 50.

Algorithm description in detail. The resulting algorithm works as follows:

(I) Partition array X into $\lceil n/5 \rceil$ lists of 5 items each.

$L_1 = \{X[1], X[2], \ldots, X[5]\}$, $L_2 = \{X[6], \ldots, X[10]\}$, ..., $L_i = \{X[5i+1], \ldots, X[5i-4]\}$, ..., $L_{\lceil n/5 \rceil} = \{X[5\lceil n/5 \rceil - 4, \ldots, X[n]\}$.

(II) For each i find median b_i of L_i using brute-force in $O(1)$ time. Total $O(n)$ time

(III) Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$

(IV) Find median b of B

See also Figure 6.2.

6.1.5. Analysis

We prove the following below, but for the time being assume the following lemma is correct.

Lemma 6.1.4. *Median of B is an* approximate *median of X. That is, if b is used a pivot to partition X, then* $|X_{less}| \leq 7n/10 + 6$ and $|X_{greater}| \leq 7n/10 + 6$.

The running time recurrence is

$$T(n) \leq T(\lceil n/5 \rceil) + \max\{T(|X_{\mathrm{less}}|), T(|X_{\mathrm{greater}})|\} + O(n).$$

We have that

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n) \qquad \text{and} \qquad T(n) = O(1) \qquad \text{for } n < 10.$$

```
select(X[1...n], j):
   Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {X[5i − 4], ..., X[5i]}
   Find median b_i of each L_i using brute-force
   Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
   Partition X into X_less and X_greater using b as pivot
   if |X_less| = j then
     return b
   if |X_less| > j then
     return select(X_less, j)
   else
     return select(X_greater, j − |X_less|)
```

Figure 6.2: Pseudo-code for deterministic median selection.

Lemma 6.1.5. *For $T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$, it holds that $T(n) = O(n)$.*

Proof: We claim that $T(n) \leq cn$, for some constant c. We have that $T(i) \leq c$ for all $i = 1, \ldots, 1000$, by picking c to be sufficiently large. This implies the base of the induction. Similarly, we can assume that the $O(n)$ in the above recurrence is smaller than $cn/100$, by picking c to be sufficiently large.

So, assume the claim holds for any $i < n$, and we will prove it for n. By induction, we have

$$\begin{aligned} T(n) &\leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n) \\ &\leq c(n/5 + 1) + c(7n/10 + 6) + cn/100 \\ &= cn(1/5 + 7/10 + 1/100 + 1/n + 6/n) \leq cn, \end{aligned}$$

for $n > 1000$. ■

Claim 6.1.6. *There are at least $3n/10 - 6$ elements smaller than the median of medians b.*

Proof: There are $n/5$ medians, and as such there are $(n/5)/2 - 1$ columns where their median is strictly smaller than the median of medians. Each of these columns contribute three elements smaller than the median. There are also the two elements from the columns of the median itself. This illustrated in Figure 6.3.

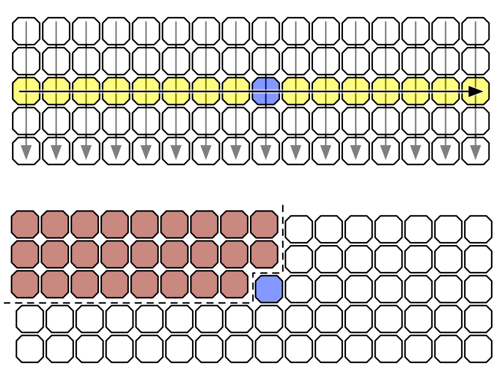

Figure 6.3

Doing this calculation more carefully, we have that the number of elements smaller than b is

$$3\left\lfloor \frac{\lfloor n/5 \rfloor + 1}{2} \right\rfloor - 1 \geq 3n/10 - 6. \qquad \blacksquare$$

This implies the following.

Corollary 6.1.7. *We have that* $|X_{greater}| \leq 7n/10 + 6$ *and* $|X_{less}| \leq 7n/10 + 6$.

6.1.6. Summary

Theorem 6.1.8. *The algorithm select($X[1 \,..\, n]$, k) computes in $O(n)$ deterministic time the kth smallest element in X.*

On the other hand, we have (for now).

Lemma 6.1.9. *The algorithm quickSelect($X[1 \,..\, n]$, k) computes the kth smallest element in X. The running time of quickSelect is $\Theta(n^2)$ in the worst case.*

Question 6.1.10. *Consider the following:*
- *Why did we choose lists of size 5? Will lists of size 3 work?*
- *Write a recurrence to analyze the algorithm's running time if we choose a list of size k.*

6.2. The lowest point above a set of lines

Let L be a set of n lines in the plane. To simplify the exposition, assume the lines are in general position:
(A) No two lines of L are parallel.
(B) No line of L is vertical or horizontal.
(C) No three lines of L meet in a point.

We are interested in the problem of computing the point with the minimum y coordinate that is above all the lines of L. We consider a point on a line to be above it.

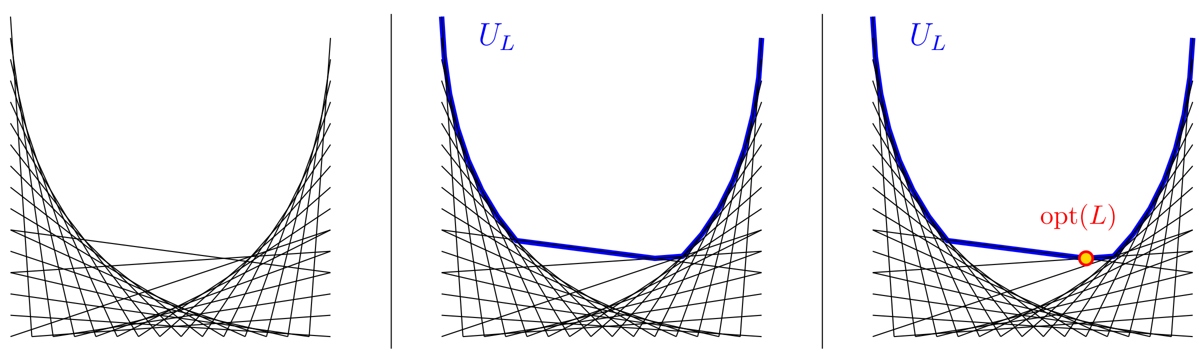

Figure 6.4: An input to the problem, the critical curve U_L, and the optimal solution – the point opt(L).

For a line $\ell \in L$, and a value $\alpha \in \mathbb{R}$, let $\ell(x)$ be the value of ℓ at α. Formally, consider the intersection point of $p = \ell \cap (x = \alpha)$ (here, $x = \alpha$ is the vertical line passing through $(\alpha, 0)$). Then $\ell(x) = y(p)$.

Let $U_L(\alpha) = \max_{\ell \in L} \ell(\alpha)$ be the ***upper envelope*** of L. The function $U_L(\cdot)$ is convex, as one can easily verify. The problem asks to compute $y^* = \min_{x \in \mathbb{R}} U_L(x)$. Let x^* be the coordinate such that $y^* = U_L(x^*)$.

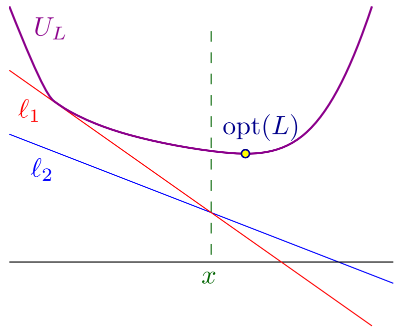

Figure 6.5: Illustration of the proof of Lemma 6.2.4.

Definition 6.2.1. Let $\text{opt}(L) = (x^*, y^*)$ denote the optimal solution – that is, lowest point on $U_L(x)$.

Remark 6.2.2. There are some uninteresting cases of this problem. For example, if all the lines of L have negative slope, then the solution is at $x^* = +\infty$. Similarly, if all the slopes are positive, then the solution is $x^* = -\infty$. We can easily check these cases in linear time. In the following, we assume that at least one line of L has positive slope, and at least one line has a negative slope.

Lemma 6.2.3. *Given a value x, and a set L of n lines, one can in linear time do the following:*
(A) Compute the value of $U_L(x)$.
(B) Decide which one of the following happens: (I) $x = x^*$, (II) $x < x^*$, or (III) $x > x^*$.

Proof: (A) Computing $\ell(x)$, for $x \in \mathbb{R}$, takes $O(1)$ time. Thus computing this value for all the lines of L takes $O(n)$ time, and the maximum can be computed in $O(n)$ time.

(B) For case (I) to happen, there must be two lines that realizes $U_L(x)$ – one of them has a positive slope, the other has negative slope. This clearly can be checked in linear time.

Otherwise, consider $U_L(x)$. If there is a single line that realizes the maximum for x, then its slope is the slope of $U_L(x)$ at x. If this slope is positive than $x^* < x$. If the slope is negative then $x < x^*$.

The slightly more challenging case is when two lines realizes the value of $U_L(x)$. That is $\big(x, U_L(x)\big)$ is an intersection point of two lines of L (i.e., a ***vertex***) on the upper envelope of the lines of L). Let ℓ_1, ℓ_2 be these two lines, and assume that $\text{slope}(\ell_1) < \text{slope}(\ell_2)$.

If $\text{slope}(\ell_2) < 0$, then both lines have negative slope, and $x^* > x$. If $\text{slope}(\ell_1) > 0$, then both lines have positive slope, and $x^* < x$. If $\text{slope}(\ell_1) < 0$, and $\text{slope}(\ell_1) > 0$, then this is case (I), and we are done. ■

Lemma 6.2.4. *Let (x, y) be the intersection point of two lines $\ell_1, \ell_2 \in L$, such that* $\text{slope}(\ell_1) < \text{slope}(\ell_2)$, *and $x < x^*$. Then $\text{opt}(L) = \text{opt}(L - \ell_1)$, where $L - \ell_1 = L \setminus \{\ell_1\}$*

Proof: See Figure 6.5. Since $x < x^*$, it must be that $U_L(\cdot)$ has a negative slope at x (and also immediately to its right). In particular, for any $\alpha > x$, we have that $U_L(\alpha) \geq \ell_2(x) > \ell_1(x)$. That is, the line $\ell_1(x)$ is "buried" below ℓ_2, and can not touch $U_L(\cdot)$ to the right of x. In particular, removing ℓ_1 from L can not change $U_L(\cdot)$ to the right of x. Furthermore, since $U_L(\cdot)$ has negative slope immediately after x, it implies that minimum point can not move by the deletion of ℓ_1. Thus implying the claim. ■

Lemma 6.2.5. *Let (x, y) be the intersection point of two lines $\ell_1, \ell_2 \in L$, such that* $\text{slope}(\ell_1) < \text{slope}(\ell_2)$, *and $x^* < x$. Then $\text{opt}(L) = \text{opt}(L - \ell_2)$.*

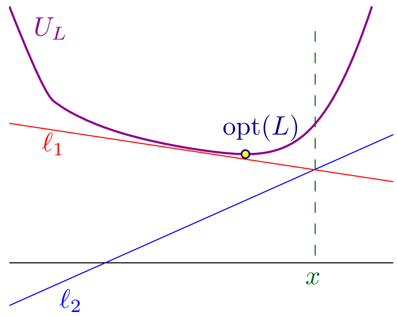

Figure 6.6: Illustration of the proof of Lemma 6.2.5.

Proof: Symmetric argument to the one used in the proof of Lemma 6.2.4. ∎

Observation 6.2.6. *The point $p = \mathrm{opt}(L)$ is a vertex formed by the intersection of two lines of L. Indeed, since none of the lines of L are horizontal, if p was in the middle of a line, then we could move it and improve the value of the solution.*

Lemma 6.2.7 (Prune). *Given a set L of n lines, one can compute, in linear time, either:*

(A) A set $L' \subseteq L$ such that $\mathrm{opt}(L) = \mathrm{opt}(L')$, and $|L'| \leq (7/8)\,|L|$.

(B) A value x such that $x^*(L) = x$.

Proof: If $|L| = n = O(1)$ then one can compute $\mathrm{opt}(L)$ by brute force. Indeed, compute all the $\binom{n}{2}$ vertices induced by L, and for each one of them check if they define the optimal solution using the algorithm of Lemma 6.2.3. This takes $O(1)$ time, as desired.

Otherwise, pair the lines of L in $N = \lfloor n/2 \rfloor$ pairs ℓ_i, ℓ'_i. For each pair, let x_i be the x-coordinate of the vertex $\ell_i \cap \ell'_i$. Compute, in linear time, using median selection, the median value z of $x_1, \ldots, x_N$. For the sake of simplicity of exposition assume that $x_i < z$, for $i = 1, \ldots, N/2 - 1$, and $x_i > z$, for $i = N/2 + 1, \ldots, N$ (otherwise, reorder the lines and the values so that it happens).

Using the algorithm of Lemma 6.2.3 decide which of the following happens:

(I) $z = x^*$: we found the optimal solution, and we are done.

(II) $z < x^*$. But then $x_i < z < x^*$, for $i = 1, \ldots, N/2 - 1$, By Lemma 6.2.4, either ℓ_i or ℓ'_i can be dropped without effecting the optimal solution, and which one can be dropped can be decided in $O(1)$ time. In particular, let L' be the set of lines after we drop a line from each such pair. We have that $\mathrm{opt}(L') = \mathrm{opt}(L)$, and $|L'| = n - (N/2 - 1) \leq (7/8)n$.

(III) $z > x^*$. This case is handled symmetrically, using Lemma 6.2.5. ∎

Theorem 6.2.8. *Given a set L of n lines in the plane, one can compute the lowest point that is above all the lines of L (i.e., $\mathrm{opt}(L)$) in linear time.*

Proof: The algorithm repeatedly apply the pruning algorithm of Lemma 6.2.7. Clearly, by the above, this algorithm computes $\mathrm{opt}(L)$ as desired.

In the ith iteration of this algorithm, if the set of lines has n_i lines, then this iteration takes $O(n_i)$ time. However, $n_i \leq (7/8)^i n$. In particular, the overall running time of the algorithm is

$$O\left(\sum_{i=0}^{\infty} (7/8)^i n\right) = O(n). \qquad \blacksquare$$

6.3. Bottleneck edge in MST

Given an undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, with n vertices and m edges, and with weights $\omega(\cdot)$ on the edges. Consider the problem of computing the longest edge in the MST of G. One can of course compute the MST, and then compute the longest edge. However, currently no deterministic algorithm for MST is known that runs in linear time (i.e., $O(n + m)$). It turns out that there is a simple elegant algorithm that achieves linear time.

We assume that the edges of the graph G has all unique weights, and the longest edge in the MST of G is its ***bottleneck*** edge, denoted by $\ell_{\mathsf{MST}}(\mathsf{G})$.

We assume as usually that the weights of the edges are all distinct.

6.3.1. A fast decider

Lemma 6.3.1. *Given a graph* G *as above with* n *vertices and* m *edges, and a real number* τ*, one can decide, in* $O(n + m)$ *time, if*

(i) $\ell_{MST}(\mathsf{G}) < \tau$*,*

(ii) $\ell_{MST}(\mathsf{G}) = \tau$*, or*

(iii) $\ell_{MST}(\mathsf{G}) > \tau$*.*

Proof: Compute the graph $\mathsf{G}_{<\tau} = (\mathsf{V}(\mathsf{G}), \{uv \in \mathsf{E}(\mathsf{G}) \mid \omega(uv) < \tau\})$. This takes $O(n+m)$ time. If $\mathsf{G}_{<\tau}$ has a single connected component, then $\ell_{\mathsf{MST}}(\mathsf{G}) > \tau$. This can be checked using BFS or DFS in $O(n + m)$ time.

Next, compute the graph $\mathsf{G}_{\leq\tau} = (\mathsf{V}(\mathsf{G}), \{uv \in \mathsf{E}(\mathsf{G}) \mid \omega(uv) < \tau\})$. This takes $O(n + m)$ time. If $\mathsf{G}_{\neq\tau}$ has a single connected component, then $\ell_{\mathsf{MST}}(\mathsf{G}) = \tau$. This can be checked using BFS or DFS in $O(n+m)$ time.

Otherwise, it must be that $\ell_{\mathsf{MST}}(\mathsf{G}) > \tau$. ■

A naive algorithm. One can now sort the edges of G by their weights, and perform a binary search over their weights, calling the decider described above. Clearly, this would compute the bottleneck weight (and thus the edge) in $O((n + m) \log m)$ time.

6.3.2. A search and prune algorithm

The algorithm first verifies that the input graph is connected. If not, it immediately rejects it.

Otherwise, the algorithm would compute the edge realizing the median edge weight in G. This can be done in $O(m)$ time, and let e be this edge. Calling the decider we decide how $\omega(e)$ relates to the weight of the bottleneck edge in the MST. There are three possibilities:

(I) $\ell_{\mathsf{MST}}(\mathsf{G}) < \tau$. None of the edges longer than τ can appear in the MST. We might as well throw them all away. Let G'' be the resulting graph. This graph has at most $m/2$ edges, has the same bottleneck edge is G, and computing it takes $O(n + m)$ time.

(II) $\ell_{\mathsf{MST}}(\mathsf{G}) > \tau$: The algorithm computes the graph $\mathsf{G}_{\leq\tau} = (\mathsf{V}(\mathsf{G}), \{uv \in \mathsf{E}(\mathsf{G}) \mid \omega(uv) < \tau\})$. We know that each connected component of this graph would be spanned by a tree in the Kruskal algorithm for computing MST after inserting all the edges of weights smaller than τ. As such, we collapse each connected component of $\mathsf{G}_{\leq\tau}$ into a single vertex. An edge $uv \in \mathsf{E}(\mathsf{G})$ connecting two different connected components of $\mathsf{G}_{\leq\tau}$ is added to the graph, as connecting the two connected components. Note that we might have parallel edges, but it is straightforward to use bucket sort to sort the edges, so that all the edges connecting the same connected components are grouped together. We then keep only the cheapest edge. Let G' be the resulting graph. Importantly, the bottleneck edge in G and G' are the same, and computing G' took $O(n+m)$ time. Furthermore, G' has at most $m/2$ edges.

(III) $\ell_{\mathsf{MST}}(\mathsf{G}) = \tau$: We are done. Yey!

If the algorithm is not done, it continues recursively on the G' or G'' (depending on the case). The key observation is that in either case, these two graphs are connected, and thus the number of edges dominate the number of vertices. We thus have the recurrence

$$T(m) = O(m) + T(m/2),$$

and the solution to this recurrence is $O(m)$.

Theorem 6.3.2. *Given an undirected graph* $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, *with* n *vertices and* m *edges, and with weights* $\omega(\cdot)$ *on the edges. The bottleneck edge of the* MST *can be computed in* $O(n+m)$ *time.*

6.4. Bibliographical notes

The beautiful median of medians algorithm of Section 6.1 is from Blum *et al.* [BFP+73] (there are four Turing award winners among the authors of this paper)..

The algorithm presented in Section 6.2 is a simplification of the work of Megiddo [Meg84]. Megiddo solved the much harder problem of solving linear programming in constant dimension in linear time, The algorithm presented is essentially the core of his basic algorithm.

Bibliography

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Sys. Sci.*, 7(4):448–461, 1973.

[Meg84] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. Assoc. Comput. Mach.*, 31:114–127, 1984.