

# COMP 251: Algorithms & Data Structures

Owen Lewis

Winter 2018

# Contents

<b>1</b>	<b>Overview of Graph Theory</b>	<b>2</b>
1.1	Definitions . . . . .	2
1.2	Some Theorems for Undirected Graphs . . . . .	2
1.3	““Data Structures”” for Representing Graphs . . . . .	4
1.3.1	Adjacency Matrices . . . . .	4
1.3.2	Adjacency Lists . . . . .	4
1.3.3	Adjacency Matrices vs. Adjacency Lists . . . . .	5
<b>2</b>	<b>Divide &amp; Conquer</b>	<b>6</b>
2.1	The Master Theorem . . . . .	6
2.2	MergeSort . . . . .	8
2.3	Binary Search . . . . .	9
2.4	Fast Multiplication . . . . .	10
2.5	Fast Matrix Multiplication . . . . .	11
2.6	Fast Exponentiation . . . . .	12
2.7	The Selection Problem . . . . .	12
2.8	The Closest Pair of Points in a Plane . . . . .	14
<b>3</b>	<b>Graph Algorithms</b>	<b>15</b>
<b>4</b>	<b>Greedy Algorithms</b>	<b>16</b>
<b>5</b>	<b>Dynamic Programming</b>	<b>17</b>
<b>6</b>	<b>Network Flows</b>	<b>18</b>
<b>7</b>	<b>Data Structures</b>	<b>19</b>

# 1 Overview of Graph Theory

## 1.1 Definitions

A graph  $G = (V, E)$  is a set  $V$  of vertices (a.k.a. nodes) and a set  $E$  of edges (denoting vertex pairs). We set  $n = |V|$ , and  $m = |E|$ . A graph is said to be *undirected* when for any edge  $(u, v) \in E$  there exists an edge  $(v, u) \in E$  for some nodes  $u$ , and  $v$ . A graph is said to be *directed* if it is not undirected. In other words, the edge set of a directed graph consists of ordered pairs where the edge set of an undirected graph consists of unordered pairs.

A *walk* is a set of vertices  $\{v_0, v_1, \dots, v_\ell\}$  such that  $(v_i, v_{i+1}) \in E, \forall 0 \leq i \leq \ell$ . A walk where  $v_0 = v_\ell$  is said to be a *circuit* or a *closed walk*. A circuit where every edge in the graph is used exactly once is known as an *Eulerian circuit*. A *cycle* is a walk  $\{v_0, v_1, \dots, v_\ell\}$  such that every vertex is distinct except  $v_0 = v_\ell$ . A cycle where every vertex of the graph is used exactly once is known as a *Hamiltonian cycle*. A walk where every vertex is distinct is said to be a *path*.

A graph is said to be *connected* if for each  $u, v \in V$  there exists a walk from  $u$  to  $v$ . A graph is said to be *disconnected* if it is not connected. Each connected subgraph of a graph is called a *component*. A connected graph therefore has exactly one component.

A connected component with no cycles is called a *tree*. A graph whose components are all trees is said to be a *forest*. A tree is said to be *spanning* if it contains every vertex in the graph. A vertex in a tree with at most one neighbour is called a *leaf*.

A *matching* is a set of vertex-disjoint edges i.e. each edge is incident to at most one other edge in a matching. A matching is said to be *perfect* if every vertex is incident to exactly one edge in the matching.

A *clique* is a set of pairwise adjacent vertices. In *independent set* (a.k.a. a *stable set*) is set of pairwise non-adjacent vertices.

A *bipartite graph* is a graph such that the vertex set  $V$  can be partitioned as  $V = X \cup Y$  where each edge has one node in  $X$  and the other node in  $Y$ . Note that  $X$  and  $Y$  are necessarily independent sets.

## 1.2 Some Theorems for Undirected Graphs

**Theorem:** (Handshaking Lemma) Let  $G = (V, E)$  be an undirected graph, let  $\Gamma(v) := \{u : (u, v) \in E\}$  be the set of neighbours of a node  $v$ , and let the *degree*  $\deg(v)$  of a vertex  $v$  equal the cardinality of  $\Gamma(v)$ . Then there are an even number of vertices with odd degree.

**Proof:** First note that since we're double-counting the number of pairs where

$(v, e)$  is an edge incident to  $v$

$$2 \cdot |E| = \sum_{v \in V} \deg(v)$$

Since the degree of a vertex is either even or odd, we can partition  $V$  into a set of odd-degree vertices  $\mathcal{O}$ , and a set of even-degree vertices  $\mathcal{E}$ . This gives us

$$\sum_{v \in V} \deg(v) = \sum_{v \in \mathcal{O}} \deg(v) + \sum_{v \in \mathcal{E}} \deg(v)$$

which implies

$$\sum_{v \in \mathcal{O}} \deg(v) = 2 \cdot |E| - \sum_{v \in \mathcal{E}} \deg(v)$$

since both the  $2 \cdot |E|$  term is even (obvious) and the  $\sum_{v \in \mathcal{E}} \deg(v)$  term is even (sum of even numbers) then the  $\sum_{v \in \mathcal{O}} \deg(v)$  term must also be even.  $\square$

**Theorem:** (Euler's Theorem) If  $G$  is an undirected graph then  $G$  contains an Eulerian circuit if and only if every vertex has even degree.

**Proof:** Easy proof by induction

**Lemma:** A tree  $T$  with  $n \geq 2$  vertices has at least one leaf vertex.

**Proof:** Trees are connected so there exists no vertices with degree 0 when  $n \geq 2$ . Suppose each vertex has degree of at least 2. Then consider the longest path  $P \subseteq T$ ,  $P = \{v_1, v_2, \dots, v_{\ell-1}, v_{\ell}\}$ . Since  $\deg(v_{\ell}) \geq 2$ ,  $\exists$  a neighbour (of  $v_{\ell}$ )  $x \in P$  with  $x \neq v_{\ell-1}$ . If  $x = v_{\ell+1}$  then  $P$  is not the longest path, a contradiction. Therefore, for  $P$  to be the longest path,  $x$  must be somewhere else in  $P$ , but this creates a cycle, another contradiction. Thus there must exist at least one node  $v$  such that  $0 < \deg(v) < 2$  – a leaf.  $\square$

**Theorem:** A tree with  $n$  vertices has exactly  $n - 1$  edges.

**Proof:** Simple proof by induction.

*Base case:* A tree with one vertex trivially has 0 edges.

*Induction Hypothesis:* Assume any tree with  $n - 1$  vertices has  $n - 2$  edges.

*Inductive Step:* Take a tree with  $n \geq 2$  vertices. By the previous lemma this tree contains a leaf vertex  $v$ . This implies that  $T \setminus \{v\}$  is a tree with  $n - 1$  vertices and by the induction hypothesis  $T \setminus \{v\}$  is a tree with  $n - 2$  edges, which implies that  $T$  is a tree with  $n - 1$  edges.  $\square$

**Theorem:** (Hall's Theorem) Let  $G = (X \cup Y, E)$  with  $|X| = |Y|$  be a bipartite graph.  $G$  contains a perfect matching if and only if  $\forall B \subseteq X$ ,  $|\Gamma(B)| \geq |B|$  (Hall's condition).

**Proof:** Firstly, the  $(\Rightarrow)$  direction is fairly obvious. If  $B \subseteq X$  with  $|\Gamma(B)| < |B|$  then the graph can't have a perfect matching. The  $(\Leftarrow)$  direction is a bit trickier. Suppose Hall's condition is satisfied. Then, take the maximum cardinality

matching  $M$  is the graph. If  $M$  is perfect then we are done. Otherwise there must exist an unmatched vertex  $b_0$ .

- Since Hall's condition holds, we have  $|\Gamma(\{b_0\})| \geq |\{b_0\}| = 1$  so  $b_0$  must have at least one neighbour  $s_0$ .
- Suppose  $s_0$  is matched in  $M$  to  $b_1$ .
- Since Hall's condition holds, we have  $|\Gamma(\{b_0, b_1\})| \geq |\{b_0, b_1\}| = 2$  so  $\{b_0, b_1\}$  must have at least one neighbour  $s_1 \neq s_0$ .
- Suppose  $s_1$  is matched in  $M$  to  $b_2$ .
- Since Hall's condition holds, we have  $|\Gamma(\{b_0, b_1, b_2\})| \geq |\{b_0, b_1, b_2\}| = 3$  so  $\{b_0, b_1, b_2\}$  must have at least one neighbour  $s_2 \notin \{s_0, s_1\}$ .
- ...

we repeat this argument as long as we can. Since the graph contains a finite number of vertices this process must terminate, but it can only terminate when we reach an unmatched node  $s_k$ . Using the edges we've formed in  $M$  we can create a path  $P$  from  $b_0$  to  $s_k$  that alternates between using non-matching edges and using matching edges. Swapping the matching edges with the non-matching edges gives us one more matching edge (as we have an odd number of edges.) This is still a valid matching as the internal nodes of  $P$  are still incident to exactly one matching edge. Also, the end nodes,  $b_0$  and  $s_k$  were previously unmatched but are now incident to exactly one edge in the new matching. Thus  $M$  isn't the maximum capacity matching – a contradiction.  $\square$

### 1.3 “““Data Structures””” for Representing Graphs

#### 1.3.1 Adjacency Matrices

For a graph, an *adjacency matrix*  $M$  is a matrix such that

1. There is a row for each vertex
2. There is a column for each vertex
3. The  $ij$  – th entry is defined as  $M_{ij} = \begin{cases} 1, (i, j) \in E \\ 0, (i, j) \notin E \end{cases}$

Note that in an undirected graph the matrix is symmetric around the diagonal because  $(i, j) \equiv (j, i)$ . Of course this is not necessarily true of directed graphs.

#### 1.3.2 Adjacency Lists

An *adjacency list* of an undirected graph is such that for each vertex  $v$  of  $V$  we store a list of its neighbours. For a directed graph we have two lists: one in which we store the in-neighbours of  $v$  and one in which we store the out-neighbours of  $v$ .

### 1.3.3 Adjacency Matrices vs. Adjacency Lists

The main difference between the two is the amount of storage required to implement them.

- An adjacency matrix requires we store  $\Theta(n^2)$  numbers
- An adjacency list requires we store  $\Theta(m)$  numbers

In any graph  $m = O(n^2)$ . This means that for a sparse graph adjacency lists are highly favourable in terms of space complexity.

Verifying whether an edge exists, however, is much faster in an adjacency matrix – when using the array representation of a matrix it takes  $O(1)$  time, where verifying the existence of an edge takes  $O(\log n)$  time for an ordered adjacency list (using binary search), and  $O(n)$  time if the adjacency list is not ordered (using sequential search).

## 2 Divide & Conquer

A *divide and conquer* algorithm ideally breaks up a problem of size  $n$  into smaller sub-problems such that:

- There are exactly  $a$  sub-problems
- Each sub-problem has a size of at most  $\frac{1}{b} \cdot n$
- Once solved, the solutions to the sub-problems must be combined in  $O(n^d)$  time to produce a solution to the original problem

Therefore the time-complexity of a divide and conquer algorithm satisfies a recurrence relation given by

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

### 2.1 The Master Theorem

**Lemma 1:**  $\sum_{k=0}^{\ell} \tau^k = \frac{1 - \tau^{\ell+1}}{1 - \tau}$ , for any  $\tau \neq 1$ .

**Proof:**

$$\begin{aligned} (1 - \tau) \sum_{k=0}^{\ell} \tau^k &= \sum_{k=0}^{\ell} \tau^k - \sum_{k=1}^{\ell+1} \tau^k \\ &= \tau^0 - \tau^{\ell+1} \\ &= 1 - \tau^{\ell+1} \\ \sum_{k=0}^{\ell} \tau^k &= \frac{1 - \tau^{\ell+1}}{1 - \tau} \end{aligned}$$

□

**Lemma 2:**  $x^{\log_b y} = y^{\log_b x}$  for any base  $b \in \mathbb{R}$ .

**Proof:** From the power rule of logarithms we have

$$\log_b x \cdot \log_b y = \log_b (y^{\log_b x})$$

similarly, we have

$$\log_b x \cdot \log_b y = \log_b (x^{\log_b y})$$

therefore

$$\log_b (x^{\log_b y}) = \log_b (y^{\log_b x})$$

thus

$$x^{\log_b y} = y^{\log_b x}$$

□

**Theorem:** (The Master Theorem) If a recurrence relation is of the form  $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$ , for constants  $a > 0$ ,  $b > 1$ , and  $d \geq 0$ , then

$$T(n) = \begin{cases} O(n^d) & , \text{ if } a < b^d \text{ [Case I]} \\ O(n^d \cdot \log n) & , \text{ if } a = b^d \text{ [Case II]} \\ O(n^{\log_b a}) & , \text{ if } a > b^d \text{ [Case III]} \end{cases}$$

**Proof:** By adding dummy numbers, we may assume that  $n$  is a power of  $b$ , i.e.  $n = b^\ell$ , for some  $\ell \in \mathbb{N}_0$ . Then

$$T(n) = n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^\ell\left(\frac{n}{b^\ell}\right)^d$$

simplifying, we get

$$T(n) = n^d \left( 1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^\ell \right)$$

we now have three cases:

Case I:  $\frac{a}{b^d} < 1$

Set  $\tau := \frac{a}{b^d}$  then we have

$$T(n) = n^d \sum_{k=0}^{\ell} \tau^k$$

applying lemma 1 we get

$$T(n) = n^d \left( \frac{1 - \tau^{\ell+1}}{1 - \tau} \right) \leq n^d \left( \frac{1}{1 - \tau} \right)$$

but  $\frac{1}{1 - \tau}$  is a constant so  $T(n) \leq n^d$  and thus

$$T(n) = O(n^d)$$

Case II:  $\frac{a}{b^d} = 1$

Then we have that

$$T(n) = n^d (1 + 1 + 1^2 + \cdots + 1^\ell) = n^d (\ell + 1)$$

but  $n = b^\ell \Rightarrow \ell = \log_b n$ , thus

$$T(n) = O(n^d \cdot \log n)$$



Case III:  $\frac{a}{b^d} > 1$

Set  $\tau := \frac{a}{b^d}$  then we have

$$T(n) = n^d \sum_{k=0}^{\ell} \tau^k$$

applying lemma 1 we get

$$T(n) = n^d \left( \frac{\tau^{\ell+1} - 1}{\tau - 1} \right) \leq n^d \left( \frac{\tau^{\ell+1}}{\tau - 1} \right)$$

but since  $\tau - 1$  is a constant we get

$$\begin{aligned} T(n) &= n^d O(\tau^{\ell+1}) \\ &= O(n^d \tau^{\ell+1}) \\ &= O(n^d \tau^{\ell}) \\ &= O\left(\left(\frac{a}{b^d}\right)^{\ell} n^d\right) \\ &= O\left(\left(\frac{n}{b^{\ell}}\right)^d a^{\ell}\right) \end{aligned}$$

but  $n = b^{\ell}$  so

$$T(n) = O(a^{\ell})$$

and  $\ell = \log_b a$  so

$$T(n) = O(a^{\log_b n})$$

and applying lemma 2 gives

$$T(n) = O(n^{\log_b a})$$

This completes the proof. □

## 2.2 MergeSort

MergeSort is an algorithm to sort  $n$  numbers into non-decreasing order.

---

```

MergeSort( $x_1, x_2, \dots, x_n$ )
  if  $n = 1$ 
    return  $x_1$ 
  else
    return Merge(MergeSort( $x_1, \dots, x_{\lfloor \frac{n}{2} \rfloor}$ ), MergeSort( $x_{\lfloor \frac{n}{2} \rfloor + 1}, \dots, x_n$ ))

```

---

Where the Merge function is a linear time algorithm combining two sorted lists (by comparing the first element in each list and moving the smaller element of

the two into our new list.)

### **MergeSort is correct!**

- It calls itself on smaller instances until the division process terminates when it reaches a base case where each list has size 1
- MergeSort works trivially on the base cases
- This sort of strong induction-type proof works for just about all divide and conquer algorithms

### **MergeSort is efficient!**

- The recurrence relation that we can construct to model the running time of MergeSort is given by

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

as we break the problem into two sub-problems of size  $\frac{n}{2}$ , plus a linear time sub-routine to combine the solutions (Merge.)

- This recurrence relation is in the proper form to use the Master Theorem and indeed we're in its Case II.
- By the master theorem the running time of MergeSort is  $O(n \cdot \log n)$ .
- The running time can also be proved by unwinding the recurrence.

## **2.3 Binary Search**

Binary Search is an algorithm to search for whether a key  $k$  exists in a given sorted list.

---

```
BinarySearch( $a_1, a_2, a_n; k$ )
  while  $n > 0$ 
    if  $a_{\lfloor \frac{n}{2} \rfloor} = k$ 
      return true
    if  $a_{\lfloor \frac{n}{2} \rfloor} > k$ 
      return BinarySearch( $a_1, \dots, a_{\lfloor \frac{n}{2} \rfloor - 1}; k$ )
    if  $a_{\lfloor \frac{n}{2} \rfloor} < k$ 
      return BinarySearch( $a_{\lfloor \frac{n}{2} \rfloor + 1}, \dots, a_n; k$ )
  return false
```

---

### **Binary Search works!**

- It works for the same strong induction argument as MergeSort

### Binary Search is efficient!

- The recurrence relation that we can construct to model the running time of Binary Search is given by

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

as we break the problem into one sub-problem of size  $\frac{n}{2}$ .

- This recurrence relation is case II of the master theorem.
- By the master theorem the running time of MergeSort is  $O(\log n)$ .
- Again, the running time can also be proved by unwinding the recurrence.

### 2.4 Fast Multiplication

In COMP 250 we found an  $O(n^2)$  algorithm to multiply 2  $n$ -digit numbers (in any base). Let's try to do better using divide and conquer. Consider the example:

Let  $\mathbf{x} := x_n x_{n-1} \dots x_{\frac{n}{2}+1} x_{\frac{n}{2}} \dots x_2 x_1$

Let  $\mathbf{y} := y_n y_{n-1} \dots y_{\frac{n}{2}+1} y_{\frac{n}{2}} \dots y_2 y_1$

Then  $\mathbf{x} = 10^{\frac{n}{2}} \mathbf{x}_L + \mathbf{x}_R$ , where  $\mathbf{x}_L = x_n \dots x_{\frac{n}{2}+1}$ , and  $\mathbf{x}_R = x_{\frac{n}{2}} \dots x_1$

Similarly  $\mathbf{y} = 10^{\frac{n}{2}} \mathbf{y}_L + \mathbf{y}_R$ , where  $\mathbf{y}_L = y_n \dots y_{\frac{n}{2}+1}$ , and  $\mathbf{y}_R = y_{\frac{n}{2}} \dots y_1$

Thus

$$\begin{aligned} \mathbf{x} \cdot \mathbf{y} &= (10^{\frac{n}{2}} \mathbf{x}_L + \mathbf{x}_R)(10^{\frac{n}{2}} \mathbf{y}_L + \mathbf{y}_R) \\ &= 10^n \cdot \mathbf{x}_L \mathbf{y}_L + 10^{\frac{n}{2}} (\mathbf{x}_L \mathbf{y}_R + \mathbf{x}_R \mathbf{y}_L) + \mathbf{x}_R \mathbf{y}_R \end{aligned}$$

So we've found a divide and conquer algorithm to multiply two  $n$ -digit numbers. Here we're breaking the problem down into four sub-problems of size  $\frac{n}{2}$ , with a few additions thrown in there (can be done in linear time). Therefore, the recurrence for this algorithm is given by

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)$$

then, by the master theorem, the running time of this algorithm is  $O(n^2)$ . Hmm. This isn't better. Well thanks to our good friend Carl "G-Money" Gauss, all is not lost. Gauss was studying the product of complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

which seemingly involved 4 multiplications, when he noticed that actually we can do it with only 3. Note that

$$(bc + ad) = (a + b)(c + d)$$

Now we can use this exact same trick but we'll replace the  $i$  with  $10^{\frac{n}{2}}$ . This gives

$$(\mathbf{x}_L \mathbf{y}_R + \mathbf{x}_R \mathbf{y}_L) = (\mathbf{x}_R + \mathbf{x}_L)(\mathbf{y}_R + \mathbf{y}_L) - \mathbf{x}_R \mathbf{y}_R - \mathbf{x}_L \mathbf{y}_L$$

So now we can break our problem into only 3 sub-problems! Our new recurrence is

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n)$$

so finally, by the master theorem,  $T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$

## 2.5 Fast Matrix Multiplication

Let's try to efficiently multiply two  $n \times n$  matrices. Let

$$X := \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \text{ and } Y := \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

where  $X$ , and  $Y$  are two  $n \times n$  matrices and  $A, B, \dots, H$  are eight  $\frac{n}{2} \times \frac{n}{2}$  matrices. Then their product  $Z$  is given by

$$Z = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Therefore multiplying two  $n \times n$  matrices involves eight products of  $\frac{n}{2} \times \frac{n}{2}$  matrices, and  $n^2$  additions. Therefore the recurrence relation is given by

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

By the master theorem, this then takes  $O(n^3)$  time. That's ok but again, we can do better. To do better however, we need a little algebraic trick. To do this, we need to define 7 matrices,  $S_1, \dots, S_7$  given by

$$\begin{aligned} S_1 &= (B - D) \cdot (G + H) & S_2 &= (A + D) \cdot (E + H) \\ S_3 &= (A - C) \cdot (E + F) & S_4 &= (A + B) \cdot H \\ S_5 &= A \cdot (F - H) & S_6 &= D \cdot (G - E) \\ S_7 &= (C + D) \cdot E \end{aligned}$$

then the product of  $X$ , and  $Y$  is given by

$$Z = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$$

This means that we only need to do 7 multiplications! Therefore our new recurrence is

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

and thus by the master theorem  $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$

## 2.6 Fast Exponentiation

Consider the following algorithm to quickly calculate  $x^n$ :

---

```
FastExp( $x, n$ )
  if  $n = 1$ 
    return  $x_1$ 
  else if  $n$  is even
    return FastExp( $x, \lfloor \frac{n}{2} \rfloor$ )2
  else if  $n$  is odd
    return  $x \cdot$ FastExp( $x, \lfloor \frac{n}{2} \rfloor$ )2
```

---

Assuming that  $n$  is a power of two, we get that recurrence is given by

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

and thus by the master theorem this algorithm has time complexity of  $O(\log n)$ .

## 2.7 The Selection Problem

Suppose we're given a list of number and we want to find the  $k^{th}$  smallest number in that list. We can use the following divide and conquer algorithm:

---

```
Select( $\mathcal{S}, k$ )
  if  $|\mathcal{S}| = 1$ 
    return  $x_1$ 
  else
    set  $\mathcal{S}_L := \{x_i \in \mathcal{S} : x_i < x_1\}$ 
    set  $\mathcal{S}_R := \{x_i \in \mathcal{S} : x_i \geq x_1\}$ 

    if  $|\mathcal{S}_L| = k - 1$ 
      return  $x_1$ 
    else if  $|\mathcal{S}_L| > k - 1$ 
      return Select( $\mathcal{S}_L, k$ )
    else if  $|\mathcal{S}_L| < k - 1$ 
      return Select( $\mathcal{S}_R, k - 1 - |\mathcal{S}_L|$ )
```

---

Well.. this actually sucks because it runs in  $\Omega(n^2)$  time. Note that it would actually be faster to just sort the list and then pull the  $k^{th}$  element (this can of course be done in  $O(n \cdot \log n)$  time.) One idea that we can exploit to improve this is to use a random pivot rather than deterministically selecting  $x_1$  as a pivot.

We will define a *good pivot* to be such neither  $\mathcal{S}_L$ , nor  $\mathcal{S}_R$  contain more than  $\frac{3}{4}$  of  $\mathcal{S}$ , and a *bad pivot* to be otherwise. Note that by this definition the probability that a pivot will be either good or bad (respectively) is 0.5.

For randomized algorithms we're always interested in the *expected runtime* of the algorithm rather than the worst case runtime. The recurrence relation,  $\bar{T}(n)$ , for the expected runtime for the randomized version of the selection is then given by

$$\bar{T}(n) \leq \frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right) + \frac{1}{2} \cdot \bar{T}(n) + O(n)$$

where the  $\frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right)$  term represents the recursive call given a good pivot, and the  $\frac{1}{2} \cdot \bar{T}(n)$  term represents the recursive call on a bad pivot. It's less than or equal to because we will only ever use one of those terms at once but not both. We can clean this recurrence up a bit to find the expected runtime.

$$\begin{aligned} \bar{T}(n) &\leq \frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right) + \frac{1}{2} \cdot \bar{T}(n) + O(n) \\ \frac{1}{2} \cdot \bar{T}(n) &\leq \frac{1}{2} \cdot \bar{T}\left(\frac{3n}{4}\right) + O(n) \\ \bar{T}(n) &\leq \bar{T}\left(\frac{3n}{4}\right) + O(n) \end{aligned}$$

Applying the master theorem to that recurrence gives  $\bar{T}(n) = O(n)$

This is good but let's try to do better, i.e. let's see if we can find a deterministic linear time algorithm. It turns out we can but it requires a little trick called the *median of medians*.

It's best to illustrate this concept with an example. Suppose we have a list,  $\mathcal{S} = \{x_1, x_2, \dots, x_n\}$ , from which we want to find the median of medians. To do this we partition the list into  $\left\lceil \frac{n}{5} \right\rceil$  groups of cardinality 5. Denote each such group by  $G_1, \dots, G_{\lceil \frac{n}{5} \rceil}$ . Then we'll sort each group and let  $z_i$  be the median of each group  $G_i$ . Let  $m$  be the median of the set  $\mathcal{Z} = \{z_1, \dots, z_{\lceil \frac{n}{5} \rceil}\}$ . Predictably, we call  $m$  the *median of medians*. This is significant because  $m$  will always be a good pivot. This is because there will always be at least  $\frac{3n}{10} - 1$  numbers in  $\mathcal{S}$  less than  $m$ , and  $\frac{3n}{10} - 1$  numbers in  $\mathcal{S}$  at least as large as  $m$ . Therefore  $|\mathcal{S}_R| \leq \frac{7n}{10}$ , and  $|\mathcal{S}_L| \leq \frac{7n}{10}$ .

Now that we have a deterministic way to guarantee we have a good pivot, all that's left is to bake it into a selection algorithm. Luckily, our algorithm won't vary too much since all we have to do is add a few steps at the beginning.

---

```

DetSelect( $\mathcal{S}, k$ )
  if  $|\mathcal{S}| = 1$ 
    return  $x_1$ 
  else
    partition  $\mathcal{S}$  into  $\lceil \frac{n}{5} \rceil$  groups of 5
    for  $j$  in range  $(1, \lceil \frac{n}{5} \rceil)$ 
      let  $z_j$  be the median of the group  $G_j$ .
    let  $\mathcal{Z} = \{z_1, \dots, z_{\lceil \frac{n}{5} \rceil}\}$ 
    set  $m := \text{DetSelect}(\mathcal{Z}, \lceil \frac{n}{10} \rceil)$ 

    set  $\mathcal{S}_L := \{x_i \in \mathcal{S} : x_i < m\}$ 
    set  $\mathcal{S}_R := \{x_i \in \mathcal{S} : x_i \geq m\}$ 

    if  $|\mathcal{S}_L| = k - 1$ 
      return  $m$ 
    if  $|\mathcal{S}_L| > k - 1$ 
      return DetSelect( $\mathcal{S}_L, k$ )
    if  $|\mathcal{S}_L| < k - 1$ 
      return DetSelect( $\mathcal{S}_R, k - 1 - |\mathcal{S}_L|$ )

```

---

The recursive formula for the running time of this algorithm is

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n)$$

where the  $T\left(\frac{n}{5}\right)$  term is the time to find the median of medians, the  $T\left(\frac{7n}{10}\right)$  term is pivoting based on the median of medians. Unfortunately, this doesn't work with the master theorem and we can't simplify it down to a form that does (unlike with the randomized algorithm, we need both of the  $T(\dots)$  terms.) Since it doesn't work with the master theorem, we need to use the recursion tree method. From the recursion tree method,  $T(n) = O(n)$ . So there it is.. a linear time deterministic algorithm to select the  $k^{th}$  smallest element from a list!

## 2.8 The Closest Pair of Points in a Plane

### 3 Graph Algorithms



## 4 Greedy Algorithms

## 5 Dynamic Programming

## 6 Network Flows

## 7 Data Structures