University of Helsinki
Department of Computer Science

582487
Data Compression Techniques

Lecture 10: Fun with Rank and Select

Simon J. Puglisi
(puglisi@cs.helsinki.fi)
Outline

• Some words about succinct data structures

• Rank and select on bit vectors (binary strings)
  – A classic succinct space solution

• Succinct representations of trees
  – Level order representation

• Mixing rank and select with Integer Codes
Succinct Data Structures...
Succinct Data Structures

- Succinct data structure
  = succinct representation of data + a succinct index

- (usually static)

- High-level goal: reduce space so the data structure might fit in RAM and therefore be faster to use

- Examples
  - Sets
  - Trees, graphs
  - Strings
  - Permutations, functions
Succinct Representation

• A representation of data whose size (roughly) matches the information-theoretic lower bound

• If the input is taken from $L$ distinct possible inputs, then its information-theoretic lower bound is $\lceil \log L \rceil$ bits
  - To be considered succinct a data structure must use: $\lceil \log L \rceil + o(\log L)$ bits

• Example: a lower bound for a set $S$, subset of $\{1,2,\ldots,n\}$
  - $\log(2^n) = n$ bits
  - $n = 3$ we have 8 distinct sets… so d.s. will need at least 3 bits
    $\emptyset$
    $\{1\} \quad \{2\} \quad \{3\}$
    $\{1,2\} \quad \{1,3\} \quad \{2,3\}$
    $\{1,2,3\}$
Succinct Representation

• Another example: ordered trees of \( n \) nodes
  – \( \log((1/2n) \cdot \binom{2n}{n}) = 2n - \Theta(\log n) \) bits
  – For \( n = 4 \) we have 5 distinct ordered trees
• For any data there exists a succinct representation
  – Enumerate all $L$ items in some order, and assign codes
  – These codes will require $\lceil\log L\rceil$ bits each

– What is not clear is if the representation also supports efficient queries: query time depends on the succinct representation chosen
• Auxiliary data structure to support queries on the succinct representation

• Size: $o(\log L)$ bits

• The index should allow queries/operations on the succinct representation in (almost) the same time complexity as using a conventional data structure
  – This is the aim anyway

• Computational model is the word RAM
  – Assume word length $w = \log \log L$
  – (this is the same pointer size as conventional data structures)
  – read/write $w$ bits of memory in $O(1)$ time
  – arithmetic/logical operations on $w$ bit numbers take $O(1)$ time
  – $+,-,\times,/,\log,\&,|,!,>>,<<$
• The ability to answer *rank* and *select* queries over bit vectors (binary strings, bit arrays) is essential for implementing succinct data structures

• Given a binary string B[1..n]
  – rank<sub>B</sub>(i) returns the number of 1 bits in B[1..i]
  – select<sub>B</sub>(i) returns the position of the i<sup>th</sup> 1 bit in B
Supporting rank and select...
Naïve rank

• To answer rank(i) scan B[1..i] and count 1-bits

• Simple but slow
  - $O(i)$ time = $O(n)$ time in the worst case

• How can we do better?
  - After all, what are we?
(Slightly) Less naïve rank

- Store an table $A[1..n]$, containing the rank answers

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$A$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

- $A[i] = \text{rank}(i)$
  - Now rank($i$) takes constant time - just an array lookup!

- Drawback:
  - $A$ requires $n \log n$ bits - $\log n$ times the size of $B$ - not succinct!
  - We’d like a solution with $O(1)$ queries and $o(n)$ extra space...
We want $O(1)$ queries with $o(n)$ extra bits...

- General approach will be to precompute some tables

- Each table stores part of the answer to every query
  - For any given query, we can extract needed parts in $O(1)$ time
  - The total size of the tables is $o(n)$ bits

```
B
1 0 0 1 0 1 1 1 0 1 0 0 1 0 1 0
```

- Premise:
  - Can read $O(\log n)$ bits into an integer in range $1..n$ in $O(1)$ time
  - However, to inspect each of those bits take $O(\log n)$ time
- Divide B into superblocks of size \( s = \log_2 n / 2 = 4 \times 4 / 2 = 8 \)

- Build a small table \( R_s \) containing ranks for only some positions

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
\text{B} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\text{R}_s & 0 & & & & & & & & & & & & & & & 1 \\
\end{array}
\]

- Store in \( R_s[j] = \text{rank}_B(j \times s) \), for all \( 0 \leq j < n / s \)
• Divide each superblock into blocks of size $b = \log n/2 = 2$

• Build a table $R_b$ which contains the rank from the start of each block to the start of its superblock

<table>
<thead>
<tr>
<th>B</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R_s</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R_b</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Store $R_b[k/b] = \text{rank}_B(k*s) - \text{rank}_B(j*s)$, for all $0 \leq k < n/b$
What we have so far (tables $R_s$ and $R_b$) almost gets us the answer we’re after

- $\text{rank}_B(i) \approx R_s[i/s] + R_b[i/b]$
  - Just need to answer in-block queries in $O(1)$ time
Tables: Resolving in-block queries

- Solution? Use another table!

- Blocks have size $b = \log_2 n/2$
  - There are $2^b$ such blocks possible
  - In each block there are $b$ possible rank queries
  - Each answer (relative to the block) is in the range $1..b$

<table>
<thead>
<tr>
<th>Type</th>
<th>0</th>
<th>1</th>
<th>rank(0)</th>
<th>rank(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Final Data Structure

Type 0 1 rank(0) rank(1)
0 0 0 0 0
1 0 1 0 1
2 1 0 1 1
3 1 1 1 2
Blocks have size \( b = \log_2 n/2 \)
- There are \( 2^b \) such blocks possible
- In each block there are \( b \) possible rank queries
- Each answer (relative to the block) is in the range 1..\( b \)

\[ R_p \]

<table>
<thead>
<tr>
<th>Type</th>
<th>0</th>
<th>1</th>
<th>rank(0)</th>
<th>rank(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Therefore size of \( R_p \), the in-block data structure is
- \( 2^b \times b \times \log b = n^{1/2} \times \log n \times \log \log n / 2 \) bits = \( o(n) \) bits
Summing up sizes...

- The size of $R_s$, the superblock data structure is
  - $2n/\log^2 n$ superblocks, each of size $\log n$ bits
  - $(n/\log^2 n) \cdot \log n = 2n/\log n$ bits = $o(n)$ bits

- The size of $R_b$, the block data structure is
  - $2n/\log n$ blocks, each of size $\log \log n$ bits
  - $2n \log \log n / \log n$ bits = $o(n)$ bits

- $R_s + R_b + R_p = o(n)$ extra bits for $O(1)$ time rank queries
  - It is possible to construct this data structure in $O(n)$ time
Variations

• Just store $R_s$ + use manual counting within superblocks
  - Saves space for $R_b$ and $R_p$, takes time $O(\log^2 n)$ per query

• Store $R_s$ and $R_b$ + use manual counting within blocks
  - Saves only space for $R_p$, takes time $O(\log n)$ per query

• Use different superblock & block sizes
  - No more theoretical guarantees, but...
  - Perhaps faster in practice: blocks that are multiples of word sizes (32-bits) can be faster to handle
Summary of rank

• Rank index takes $O(n \log \log n / \log n) = o(n)$ bits so we use $n + o(n)$ overall and can answer queries in $O(1)$ time

• While it is sublinear, we’d still like the $o(n)$ term to be small
  – Best is by Patrascu: $O(n / \log^k n)$ bits, $O(k)$ time queries

• Dynamic solutions exist
  – Queries no longer constant: $O(\log n / \log \log n)$ time (Raman et al.)
We can use our solution to rank to get a (fairly) efficient solution to select(i), with this observation:

- If \( \text{rank}(n/2) > i \), then the \( i^{\text{th}} \) 1-bit is in \( B[1..n/2] \)
  - Otherwise it is in \( B[n/2+1..n] \)

\[
\text{select}_B(3)
\]

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
\]
- Applying this idea recursively to arrive at select(i)
  - $O(\log_2 n)$ time, $o(n)$ space

- $O(1)$ time, $o(n)$ space solutions for select also exist
  - Slightly more complicated than $O(1)$ rank
  - (Munro and Clark)

- Similar variations as we discussed with rank (trading space for query time) are also possible
Succinct Binary Tries...
• Each node in a binary trie can have up to two children: left and/or right or none at all.

• A conventional pointer based representation would store a pair of pointers at each node.

• Each pointer would take 32 (or these days 64) bits, so for an n node trie we’d use at least 32n bits of memory.
• We’ll look at just one succinct data structure for binary tries:
  – Level-order representation, but…
  – …there are others, with different advantages/disadvantages

• First thing we want to know is: how many binary tries are there?
  – In fact there are $C_n = (2n \text{ choose } n)/(n+1) \approx 4^n$ such tries
  – So we’re aiming to use $\log(C_n) + o(\log(C_n)) = 2n + o(n)$ bits
  – (Significantly less than the conventional representation)

• And we’d like to support standard navigation operations:
  – left child, right child and parent
Level-order representation

- For each node in level order:
  - Write 0/1 for whether it has a left child
  - Write 0/1 for whether it has a right child
• For each node in level order:
  – Write 0/1 for whether it has a left child
  – Write 0/1 for whether it has a right child
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child
• For each node in level order:
  – Write 0/1 for whether it has a left child
  – Write 0/1 for whether it has a right child
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child

Level-order representation

```
1 1 0
B C
```
Level-order representation

- For each node in level order:
  - Write 0/1 for whether it has a left child
  - Write 0/1 for whether it has a right child

```
1 1 0 1
B C D
```
Level-order representation

- For each node in level order:
  - Write 0/1 for whether it has a left child
  - Write 0/1 for whether it has a right child

```
1 1 0 1
B C D
```
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child
• For each node in level order:
  – Write 0/1 for whether it has a left child
  – Write 0/1 for whether it has a right child

Level-order representation

A

B

D

G

C

E

F

1 1 0 1 1 1
B C D E F
For each node in level order:
  - Write 0/1 for whether it has a left child
  - Write 0/1 for whether it has a right child
Level-order representation

- For each node in level order:
  - Write 0/1 for whether it has a left child
  - Write 0/1 for whether it has a right child
Level-order representation

- For each node in level order:
  - Write 0/1 for whether it has a left child
  - Write 0/1 for whether it has a right child

1 1 0 1 1 1 0 1
B C D E F G
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child

Level-order representation

A
B
D
G
C
E
F

1 1 0 1 1 1 0 1 0 0
B C D E F G
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child

1 1 0 1 1 1 0 1 0 0
B C D E F G
Level-order representation

- For each node in level order:
  - Write 0/1 for whether it has a left child
  - Write 0/1 for whether it has a right child

```
  1 1 0 1 1 1 0 1 0 0 0 0
B C D E F G
```
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child

```
1 1 0 1 1 1 0 1 0 0 0 0
B C D E F G
```
For each node in level order:
- Write 0/1 for whether it has a left child
- Write 0/1 for whether it has a right child
Level-order representation

• For each node in level order:
  – Write 0/1 for whether it has a left child
  – Write 0/1 for whether it has a right child

```
1 1 0 1 1 1 1 0 1 0 0 0 0 0
B C D E F G
```

2n bits
Level-order representation

- **Equivalently:** append external (●) node for each missing child
- For each node in level order:
  - Write 0 if external, else write 1 if internal

```
1 1 0 1 1 1 0 1 0 0 0 0 0 0
B C D E F G
```
Level-order representation

- Equivalently: append external (◯) node for each missing child
- For each node in level order:
  - Write 0 if external, else write 1 if internal
Level-order representation

- Equivalently: append external (○) node for each missing child
- For each node in level order:
  - Write 0 if external, else write 1 if internal
Level-order representation

- Equivalently: append external (●) node for each missing child
- For each node in level order:
  - Write 0 if external, else write 1 if internal

```
1 1 0 1 1 1 0 1 0 0 0 0 0 0
A B C • D E F • G • • • • • • •
```
Level-order representation

- Equivalently: append external (○) node for each missing child
- For each node in level order:
  - Write 0 if external, else write 1 if internal

(1) 1 1 0 1 1 1 1 0 1 0 0 0 0 0 0

A B C • D E F • G • • • • • • • • • •
Level-order representation

- Equivalently: append external (◯) node for each missing child
- For each node in level order:
  - Write 0 if external, else write 1 if internal

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(1) 1 1 0 1 1 1 0 1 0 0 0 0 0 0 0
A B C • D E F • G • • • • • • • • • • ➤2n+1 bits
• Claim: left and right children of the $i^{th}$ internal node are at positions $2i$ and $2i + 1$. 
• Claim: left and right children of the $i^{th}$ internal node are at positions $2i$ and $2i + 1$. 
• Claim: left and right children of the $i^{th}$ internal node are at positions $2i$ and $2i + 1$.

• Easy to prove by induction on $i$…

• The $i^{th}$ internal node’s children must be just after the $(i-1)^{st}$ internal node’s children (because external nodes have no children)…
• So… left and right children of the $i^{th}$ internal node are at positions $2i$ and $2i + 1$, but how do we know our internal node number?
left-child(i) = 2\cdot rank_1(i)
right-child(i) = 2\cdot rank_1(i)+1
parent(i) = select_1(i/2)
Succinct binary trie representation

- The level-order representation of the binary trie captures the trie shape in $2n$ bits, which matches the information theoretic minimum.

- In order to be able to navigate the tree we build data structures (for rank and select) that add $o(n)$ bits to the level-order representation.

- This means the data structure is succinct: $2n + o(n)$ bits.

- Parent and child operations take constant time (just like they would in a conventional representation).
  - More complicated operations (e.g. subtree size) can be achieved with different, slightly more complex succinct representations.
Integer codes redux...
Remember integer codes?

• Three classic flavors
  – Unary
  – Elias codes (gamma, delta)
  – Golomb codes (Rice, general)

• Three modern flavours
  – Interpolative binary codes
  – Variable-byte codes
  – Word-aligned binary codes (simple, relative, carryover)
• Integer codes work because they are “self delimiting”: if we are at the start of a code word we are able to tell where the code word ends

• E.g. \( \gamma \) codes
  
  0011010000011010110111
• Integer codes work because they are “self delimiting”: if we are at the start of a code word we are able to tell where the code word ends

• E.g. $\gamma$ codes

0011010000011010110111


• So, given an encoded array $A[0..n]$, we’re able to decode it from start to finish. We decode $A[0]$, then $A[1]$, et c.

• Accessing the $i$th element of the array takes $O(i)$ time
  – This is much slower than the original uncompressed array, where access takes $O(1)$ time
  – Is it possible to get compression and fast access?

• Yes. With the help of rank and select…
Random access to $\gamma$ encoded integers

- As we’ve just remembered, an Elias $\gamma$ code is composed of two parts: a selector and an offset.
  - The selector (in unary) tells us how long (in bits) the offset is.

- The trick for random access is to separate these parts out…
Random access to γ encoded integers…

• Turn the encoded stream of integers into two bit vectors
  0001101000000110101100111

  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
  Selectors: 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
  Offsets:  1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |

  Preprocess selector’s bit vector for rank/select queries

• Locate the end of the selector of the ith integer as \( k = \text{select}(i) \)
• Locate it’s start at \( j = \text{select}(\text{rank}(k-1))+1 \)
• We know the length of the offset! Where is the offset?
• It’s at Offsets\([j..k]\)
Quick analysis

• Because rank and select take constant time, access to the $i$th Elias $\gamma$ encoded integer takes constant time

• If the encoded array of integers is of size $n$ bits, by preprocessing the selectors bit vector for rank and select we add $o(n)$ bits to the overall size. Total: $n + o(n)$ bits.
Random access to VBYTE encoded integers

• Dedicate 1 bit in each byte we output (high bit) to be a continuation bit $c$.
  – 00000000

• If the int $G$ fits within 7 bits, binary-encode it in the 7 available bits and set $c = 0$.
  – Eg. Integer 29 = 11101 fits in 7 bits so we output: 0011101
  – Eg. integer 117 = 1110101 fits in 7 bits so we output: 0110101

• Else: set $c = 1$, encode lower-order 7 bits and then use additional bytes to encode the higher order bits using same algorithm.
  – Eg. integer 767 = 1011100101 > 7 bits so:
    – Put lower 7 bits (100101) in first byte and set the $c$ bit: 11100101
    – We now have the bits 101 to deal with, < 7 bits so output 00000101

• As with $\gamma$ codes, the trick for getting random access is to rearrange the parts of the code
Random access to VBYTE encoded integers...

<table>
<thead>
<tr>
<th>integers</th>
<th>824</th>
<th>5</th>
<th>7</th>
<th>767</th>
<th>117</th>
</tr>
</thead>
<tbody>
<tr>
<td>in binary</td>
<td>1100111000</td>
<td>101</td>
<td>111</td>
<td>1011100101</td>
<td>1110101</td>
</tr>
<tr>
<td>VB codes</td>
<td>10111000</td>
<td>00000101</td>
<td>00000111</td>
<td>11100101</td>
<td>01110101</td>
</tr>
<tr>
<td></td>
<td>00000110</td>
<td></td>
<td></td>
<td>00000101</td>
<td></td>
</tr>
</tbody>
</table>

- Separate the bytes out into *layers*
  - First layer is an array of bytes containing the first byte of each code word
  - Second layer contains the second byte of each code word that has length two bytes or more
  - Third layer… et c.

- For each layer make a bit vector of the continuation bits
Random access to VBYTE encoded integers...

<table>
<thead>
<tr>
<th>Layer 1</th>
<th>Layer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VB codes</td>
<td>1011000 0000101 0000111 11100101 01110101 0000110 0000101 0000111 1100101 1110101</td>
</tr>
</tbody>
</table>

| Bit vector | 1 0 0 1 0 |
| Bytes      | 0111000 0000101 0000111 1100101 1110101 |

| Bit vector | 0 0 |
| Bytes      | 0000110 0000101 |

- Accessing the first byte of the $i^{th}$ integer is easy
  - If the continuation bit (in the bit vector) is 1, second byte is at position $\text{rank}(i)-1$ in Layer 2, and so on…
Word-aligned codes? Interpolative codes?

• What about some of the other integer codes we looked at – can they be made to support fast random access?

• See this week’s exercise sheet.
Summary

• Today we’ve had a taste of succinct data structures
  – A succinct tree representation
  – A compressed array
  – In both cases, rank and select support over bit vectors was key

• Succinct and compressed data structures are relatively new and very active research areas... lots of cool discoveries to come. A good area for a Master’s project...
<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>13/01</td>
<td>Shannon’s Theorem</td>
</tr>
<tr>
<td>15/01</td>
<td>Huffman Coding</td>
</tr>
<tr>
<td>20/01</td>
<td>Integer Codes I</td>
</tr>
<tr>
<td>22/01</td>
<td>Integer Codes II</td>
</tr>
<tr>
<td>27/01</td>
<td>Dynamic Prefix Coding</td>
</tr>
<tr>
<td>29/01</td>
<td>Arithmetic Coding</td>
</tr>
<tr>
<td>03/02</td>
<td>Dictionary Compression</td>
</tr>
<tr>
<td>05/02</td>
<td>No Lecture</td>
</tr>
<tr>
<td>10/02</td>
<td>Burrows-Wheeler Transform</td>
</tr>
<tr>
<td>12/02</td>
<td>Wavelet Trees</td>
</tr>
<tr>
<td>17/02</td>
<td>BWT Tutorial</td>
</tr>
<tr>
<td>19/02</td>
<td>Succinct Trees</td>
</tr>
<tr>
<td>24/02</td>
<td>Quadtrees, RLZ revisited, project topics</td>
</tr>
<tr>
<td>27/02</td>
<td>Mini Review</td>
</tr>
</tbody>
</table>
End