

Algorithmic Design  
- Homeworks -

Claudia Dorigo

AA 2019-2020

# Strassen

## Exercise 1

Generalize the implementation of the Strassen's algorithm to deal with non-square matrices.

## Exercise 2

Improve the implementation of the Strassen's algorithm by reducing the memory allocations and test the effects on the execution time.

### Solution:

```
TEST ALGORITHM ON RECTANGULAR MATRICES:
Using A=521x1724, B=1724x5173

Strassen algorithm: 4.788188
Naive algorithm: 44.684424
Same result: 1

TEST ALGORITHM ON SQUARE MATRICES:
```

	Strassen's Alg opt	Strassen's Alg	Naive Alg.	Same result
1	0.000001	0.000001	0.000000	1
2	0.000000	0.000000	0.000000	1
4	0.000001	0.000000	0.000000	1
8	0.000002	0.000001	0.000001	1
16	0.000008	0.000005	0.000004	1
32	0.000039	0.000033	0.000035	1
64	0.000286	0.000259	0.000283	1
128	0.002150	0.001988	0.002187	1
256	0.016535	0.017962	0.118002	1
512	0.100892	0.094344	1.047641	1
1024	0.668273	0.649684	9.264549	1
2048	4.682649	4.569671	77.603362	1
4096	37.319078	34.310755	720.452319	1

From the main output I can see that the Strassen's algorithm generalized for rectangular matrices works and it's faster than the naive algorithm for the chosen sizes.

I've also tested the optimized version (the one using just 2 S matrices instead of 10) and I can see that the execution time it's slightly greater than the one of non-optimized Strassen's algorithm but it's sill way faster than naive algorithm.

# Binary heaps 1

## Exercise1

Implement the array-based representation of binary heap together with the functions HEAP\_MIN, REMOVE\_MIN, HEAPIFY, BUILD\_HEAP, DECREASE\_KEY and INSERT\_VALUE.

## Exercise2

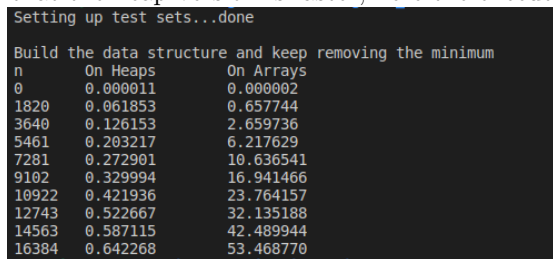
Implement an iterative version of HEAPIFY.

## Exercise3

Test the implementation on a set of instances of the problem and evaluate the execution time.

### Solution:

Using different sizes build the heap and keep removing the minimum until it's empty. Compare the results with the same operations done on an array. I can see that the heap version is faster, here the execution times:



```
Setting up test sets...done
Build the data structure and keep removing the minimum
n      On Heaps      On Arrays
0      0.000011      0.000002
1820   0.061853      0.657744
3640   0.126153      2.659736
5461   0.203217      6.217629
7281   0.272901      10.636541
9102   0.329994      16.941466
10922  0.421936      23.764157
12743  0.522667      32.135188
14563  0.587115      42.489944
16384  0.642268      53.468770
```

## Exercise 4

Show that, with the array representation, the leaves of a binary heap containing  $n$  nodes are indexed by  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ .

### Solution:

An heap containing  $n$  elements can be represented as an array of length  $n$ . In the array representation when a node in position  $i$  has a right child it will be in position  $2i + 1$  while when it has a left child it will be in position  $2i$ . This means that each element in position  $i \leq \lfloor \frac{n}{2} \rfloor$  has at least a left child in position  $2i$ , so it's not a leaf node. While each element in position  $i > \lfloor \frac{n}{2} \rfloor$  has no left child since  $2i > n$ . If a node has no left child it also has no right child (heap topology) so it's a leaf.

## Exercise 5

Show that the worst-case running time of HEAPIFY on a binary heap of size  $n$  is  $\Omega(\log n)$ . (Hint: For a heap with  $n$  nodes, give node values that can cause HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

### Solution:

Assume that we are dealing with a min-heap (symmetric for max-heaps) and the values of the nodes of the left-most branch are (starting from the root):  $h, 0, 1, 2, \dots, h-1$  where  $h = \log n$  is the height of the heap. In this case the heap property doesn't hold in the root because the child node has value 0 which is smaller than the value  $h$  of the parent node. Let's assume also that this is the only node of the heap in which the heap property doesn't hold. Once the heap property is fixed in the root (swapping nodes with values 0 and  $h$ ) the problem is pushed down to the next level: now I have the parent node having value  $h$  while the child node has value 1. To fix the heap property in the whole branch I need to perform  $h$  swaps  $(0, 1, \dots, h-1)$ . So the worst-case running time is  $\Omega(h) = \Omega(\log n)$ . It's not  $\Theta(\log n)$  because the heap property may not hold also in other nodes of the heap and not only on the root as we've assumed.

## Exercise 6

Show that there are at most  $\lceil n/2^{h+1} \rceil$  nodes of height  $h$  in any  $n$ -element binary heap.

### Solution:

Let's call  $n(i)$  the number of nodes at height  $i$ . By induction:

- Base case: root node has height  $h = \lfloor \log_2 n \rfloor$

$$n(\lfloor \log_2 n \rfloor) = \lceil \frac{n}{2^{\lfloor \log_2 n \rfloor + 1}} \rceil = 1$$

Indeed in the denominator:  $2^{\lfloor \log_2 n \rfloor + 1} > 2^{\log_2 n} = n$  because  $\lfloor x \rfloor + 1 > x$ . The fraction is smaller than 1 because the denominator is greater than  $n$ . So the ceiling is 1 and this rule holds in the root.

- Inductive step: given the formula for an height of  $i$ , let's see what happens at height  $i-1$

So assuming  $n(i) = \lceil \frac{n}{2^{i+1}} \rceil$ :

$$n(i-1) \leq 2n(i) = 2 \lceil \frac{n}{2^{i+1}} \rceil \leq \lceil \frac{n}{2^{(i-1)+1}} \rceil$$

where first  $\leq$  is because each node at level  $i$  can have at most two children and the second  $\leq$  is because  $2 \lceil x \rceil \leq \lceil 2x \rceil$ . Valid until  $h=0$ .

## Binary heaps 2

### Exercise 1

By modifying the code written during the last lessons provide an array-based implementation of binary heaps which avoids to swap the elements in the array A.

(Hint: use two arrays, *key\_pos* and *rev\_pos*, of natural numbers reporting the position of the key of a node and the node corresponding to a given position, respectively)

### Exercise 2

Consider next algorithm:

```
1 def Ex2(A)
2   D <- build(A)
3   while !is_empty(D) do
4     extract_min(D)
5   end while
6 end def
```

where A is an array. Compute the time-complexity of the algorithm when:

- build, is\_empty  $\in \Theta(1)$ , extract\_min  $\in \Theta(|D|)$ ;
- build  $\in \Theta(|A|)$ , is\_empty  $\in \Theta(1)$ , extract\_min  $\in O(\log|D|)$ ;

### Solution:

The while loop is performed  $|D|$  times.

In the first case the complexity is:

$$T(|D|) = \Theta(1) + \sum_{i=1}^{|D|} \Theta(i) = \Theta(|D|^2)$$

while in the second case:

$$T(|D|) = \Theta(|A|) + \sum_{i=1}^{|D|} O(\log|D|) = \Theta(|A|) + O(|D|\log|D|) = O(|A| + |D|\log|D|)$$

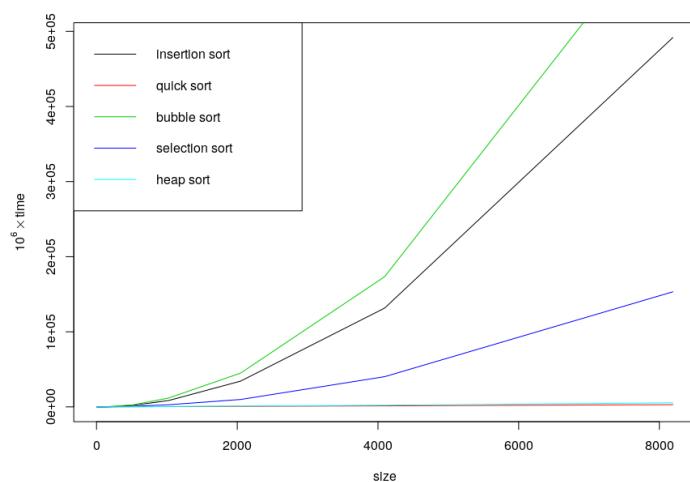
# Sorting 1

## Exercise 1

Implement INSERTION\_SORT, QUICK\_SORT, BUBBLE\_SORT, SELECTION\_SORT and HEAP\_SORT.

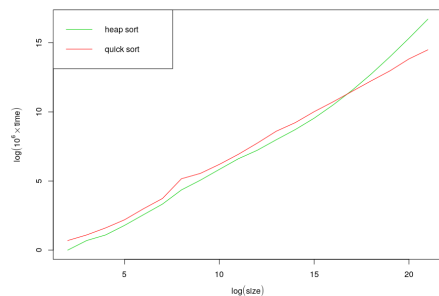
## Exercise 2

For each of the implemented algorithm, draw a curve to represent the relation between the input size and the execution-time.



I can distinguish the algorithms with complexity  $\Theta(n^2)$  in the random case: bubble sort is the slowest one, followed by insertion sort and eventually selection sort.

But in this graph I cannot understand very well the behaviour of heap sort and quick sort so I plot the times of this two algorithms (I have times for sizes up to  $2^{20}$ ): quick sort is slightly slower than heap sort until  $size \simeq 2^{18}$



### Exercise 3

Argue about the following statements and answer the questions:

- (a) HEAP SORT on an array A whose length is  $n$  takes time  $O(n)$ .

FALSE. I first need to build the heap which takes  $\Theta(n)$  and then to keep extracting the minimum ( $O(\log n)$ ) until the heap is empty. So the overall complexity of HEAP SORT is:

$$T_H(n) = \Theta(n) + \sum_{i=1}^n O(\log i) = O(n \log n)$$

- (b) HEAP SORT on an array A whose length is  $n$  takes time  $\Omega(n)$ .

TRUE. Since the complexity of building the heap is  $\Theta(n)$  and extracting the minimum is  $\Omega(1)$  (no need to call heapify in the best case), then HEAP SORT is  $\Omega(n)$ .

- (c) What is the worst case complexity for HEAP SORT?

In the worst case the complexity is  $\Theta(n \log n)$  because in any case I need to build the heap ( $\Theta(n)$ ) and extract the minimum, which in the worst case takes time  $\Theta(\log n)$ . So:

$$T_H(n) = \Theta(n) + \sum_{i=1}^n \Theta(\log i) = \Theta(n \log n)$$

- (d) QUICK SORT on an array A whose length is  $n$  takes time  $O(n^3)$ .

TRUE. QUICK SORT takes time  $\Theta(n \log n)$  in the best and in the average cases while it takes time  $O(n^2)$  in the worst case. So being  $O(n^2)$  it's also  $O(n^3)$  for the properties of  $O(\cdot)$ .

- (e) What is the complexity of QUICK SORT?

The complexity can be expressed with the following recursive equation:

$$T_Q(|A|) = T_Q(|S|) + T_Q(|A| - |S| - 1) + \Theta(|A|)$$

where  $\Theta(|A|)$  is the complexity of partitioning the array and  $T_Q(|S|)$  and  $T_Q(|A| - |S| - 1)$  are the costs for the recursive calls on the subarrays containing values smaller and greater than the pivot respectively. The solution of this recursive equation depends on the choice of the pivot:

- in the worst case —S— or —G— are 0 and so

$$T_Q(|A|) = \Theta(1) + T_Q(|A| - 1) + \Theta(|A|) = O(|A|^2)$$

- in the best case (balanced partition) I reach complexity  $\Theta(|A| \log |A|)$  (and also in average case)

So we can conclude that in general QUICK SORT complexity belongs to  $\Omega(|A| \log |A|)$  and  $O(|A|^2)$ .

(f) BUBBLE SORT on an array A whose length is  $n$  takes time  $\Omega(n)$ .

TRUE. BUBBLE SORT takes time  $\Theta(n^2)$ . This means that it belongs to  $\Omega(n^2)$  and so also to  $\Omega(n)$  because of  $\Omega(\cdot)$  properties.

(g) What is the complexity of BUBBLE SORT?

BUBBLE SORT requires to compare the last element of the array with any previous element and swap them if I find a greater one. Once it scans all the array it does the same on the subarray without the last element until I reach an array of size 1.

The swap takes time  $\Theta(1)$  so the complexity of BUBBLE SORT is:

$$T_B(|A|) = \sum_{i=1}^{|A|} \sum_{j=1}^{i-1} \Theta(1) = \sum_{i=1}^{|A|} \Theta(i) = \Theta(|A|^2)$$

This algorithm can be optimized in such a way to stop when the inner loop performs no swaps. In this case the best case (already sorted array) takes time  $\Omega(n)$ .

## Exercise 4

Solve the following recursive equation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 32 \\ 3 \cdot T(\frac{n}{4}) + \Theta(n^{3/2}) & \text{otherwise} \end{cases}$$

### Solution:

Let's try to solve this recursive equation using recursive tree. The cost of each node is  $cn^{3/2}$  (a representative of  $\Theta(n^{3/2})$ ). Each node in the tree will have 3 children and at a generic level  $i$  we have  $3^i$  nodes working on size  $\frac{n}{4^i}$ .

We can use this information to find out the height of tree: we reach the base case when  $n = 32$ , so:

$$\frac{n}{4^h} = 32 \iff \frac{n}{2^5} = 2^{2h} \iff \log_2 n - 5 = 2h \iff h = \frac{\log_2 n - 5}{2}$$

The costs at level  $i$  sum up to  $3^i c \left(\frac{n}{4^i}\right)^{3/2}$ . The number of nodes in the last level is  $3^{\frac{\log_2 n - 5}{2}} = n^{\frac{\log_2 3}{2}} 3^{-(5/2)}$ . So we can compute the cost at the last level which is  $\Theta(n^{\frac{\log_2 3}{2}} 3^{-(5/2)}) = \Theta(n^{\frac{\log_2 3}{2}})$



Now I can compute the overall complexity as:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\frac{\log_2 n - 5}{2} - 1} 3^i c \left( \frac{n}{4^i} \right)^{3/2} + \Theta(n^{\frac{\log_2 3}{2}}) = cn^{3/2} \sum_{i=0}^{\frac{\log_2 n - 5}{2} - 1} \left( \frac{3}{8} \right)^i + \Theta(n^{\frac{\log_2 3}{2}}) \\
&\leq cn^{3/2} \sum_{i=0}^{+\infty} \left( \frac{3}{8} \right)^i + \Theta(n^{\frac{\log_2 3}{2}}) = cn^{3/2} \frac{1}{1 - (3/8)} + \Theta(n^{\frac{\log_2 3}{2}}) \\
&\frac{8}{5} cn^{3/2} + \Theta(n^{\frac{\log_2 3}{2}}) \in O(n^{3/2})
\end{aligned} \tag{1}$$

## Sorting 2

### Exercise 1

Generalize the SELECT algorithm to deal also with repeated values and prove that it still belongs to  $O(n)$ .

### Solution:

Use three way partition instead of partition, here the pseudocode:

```
1 def select_generalized(A, i, l=1, r=|A|):
2   m <- select_pivot(A, l, r)
3   first, last <- three_partition(A, l, r, m)
4
5   if (i<first):
6     return select_generalized(A, i, l, first-1)
7   elseif (first<i<last):
8     return i
9   else:
10    return select_generalized(A, i-last, last+1, r) + last
11  end if
12 end def
13
14 def three_partition(A, l, r, p)
15   swap(A, l, p)
16   (p, l) <- (l, l+1)
17
18   while (l<=r):
19     if (A[l]==A[p]) :
20       p <- p+1
21       swap(A, l, p)
22       l <- l+1
23     else if (A[l]<A[p]):
24       l <- l+1
25     else:
26       swap(A, l, r)
27       r <- r-1
28     end if
29   end while
30
31   for t=1,2,...,p
32     swap(A, t, i-t)
33   end for
34   return (i-p, i-1)
35
36 end def
```

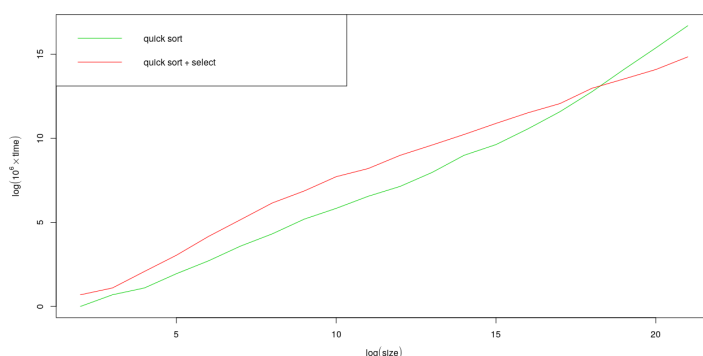
The complexity of three\_partition is  $\Theta(n)$  because at each iteration of the while loop I increase l or decrease r so I perform n iterations. (assuming that the number of copies of p is not relevant w.r.t.  $n$ )

Since the complexity of three\_partition is the same of partition, also the complexity of select\_generalized it's the same of select:  $O(n)$

## Exercise 2

- Implement the SELECT algorithm of Ex1
- Implement a variant of QUICK SORT algorithm using above-mentioned SELECT to identify the best pivot for partitioning
- Draw a curve to represent the relation between the input size and the execution-time of the two variants of QUICK SORT and discuss about their complexity.

## Solution:



The complexity of the quick sort algorithm depends on the choice of the pivot used to partition the array. In the worst case the selected pivot is always the first or last element of the array and the complexity is  $O(n^2)$ . In the best case the pivot is always the median of the array which gives  $|G| = |S| = \frac{|A|}{2}$  (but in general it works also for other fixed ratios). In this case the complexity is  $\Theta(n \log n)$ . So in the average case of quick sort I will have sometimes a good pivot, sometimes a bad one, which leads us to the complexity  $\Theta(n \log n)$ .

The quick sort + select algorithm has the same complexity of quicksort but its improvement is that to choose the pivot in such a way that I will not end up in the worst case in any recursive call of the algorithm. In order to do this we introduce some overhead and so we can see from the plot that only for a size equal to  $2^{19}$  quick sort + select outperforms quicksort.

## Exercise 3

In the algorithm SELECT, the input elements are divided into chunks of 5. Will the algorithm work in linear time if they are divided into chunks of 7? What about chunks of 3?

### Solution:

- if I divide into chunks of 7 we have that the maximum size of S is equal to  $\frac{10}{14}n + 8$  and so the complexity is

$$T_S(n) = T_S\left(\left\lceil \frac{n}{7} \right\rceil\right) + T_S\left(\frac{10}{14}n + 8\right) + \Theta(n);$$

Let's guess  $T_S(n) \in O(n)$ , we select a representative  $cn$  of  $O(n)$  and a representative  $c'n$  of  $\Theta(n)$ . Now assuming that  $T_S(n) \leq cm \ \forall m < n$  we want to prove that this holds for  $n$ :  $T_S(n) \leq cn$ .

Let's substitute the representatives in the recursive equation assuming  $n > 7$  and  $\frac{10}{14}n + 8 < n$  so  $n > 28$ :

$$\begin{aligned} T_S(n) &\leq c \cdot \left\lceil \frac{n}{7} \right\rceil + c \cdot \left(\frac{10}{14}n + 8\right) + c'n \leq c\left(\frac{n}{7} + 1\right) + c \cdot \left(\frac{10}{14}n + 8\right) + c'n \\ &\leq c\frac{12}{14}n + 9c + c'n \leq c\frac{12}{14}n + 9c + \frac{1}{28}cn = \frac{25}{28}cn + 9c \leq cn \end{aligned} \quad (2)$$

when  $28c' \leq c$  and  $n > 84$ . So by induction  $T_S(n) \in O(n)$ .

- if I divide into chunks of 3 we have that the maximum size of S is equal to  $\frac{4}{6}n + 4$  and so the complexity is

$$T_S(n) = T_S\left(\left\lceil \frac{n}{3} \right\rceil\right) + T_S\left(\frac{4}{6}n + 4\right) + \Theta(n);$$

Repeating the same reason we have that for  $n > 3$  and  $\frac{4}{6}n + 4 < n$  (which means  $n \geq 12$ ):

$$\begin{aligned} T_S(n) &\leq c \cdot \left\lceil \frac{n}{3} \right\rceil + c \cdot \left(\frac{4}{6}n + 4\right) + c'n \leq c\left(\frac{n}{3} + 1\right) + c \cdot \left(\frac{4}{6}n + 4\right) + c'n \\ &\leq c\frac{6}{6}n + 5c + c'n \leq cn + 5c + \frac{1}{12}cn = \frac{13}{12}cn + 5c > cn \end{aligned} \quad (3)$$

when  $12c' \leq c$ . I can see that I cannot reach the conclusion that  $T_S(n) \leq cn$ , so in this case the algorithm is no more linear.

### Exercise 4

Suppose that you have a "black-box" worst-case linear-time subroutine to get the position in A of the value that would be in position  $n/2$  if A was sorted. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary position  $i$ .

## Solution:

Let's call `find_median` this black-box subroutine and `select_index` the proposed function. Pseudocode:

```

1 def select_index(A,l=1,r=|A|,i):
2     m <- find_median(A,l,r)
3     p <- partition(A,l,r,m)
4
5     if (p==i):
6         return p
7     else if (i<p):
8         select_index(A,l,p-1,i)
9     else:
10        select_index(A,p+1,r,i-p)
11 end def

```

We've already seen that this algorithm takes linear time  $O(n)$  if `find_median` is linear.

## Exercise 5

Solve the following recursive equations by using both the recursion tree and the substitution method:

- $T_1(n) = 2 \cdot T_1(\frac{n}{2}) + O(n)$
- $T_2(n) = T_2(\lceil \frac{n}{2} \rceil) + T_2(\lfloor \frac{n}{2} \rfloor) + \Theta(1)$
- $T_3(n) = 3 \cdot T_3(\frac{n}{2}) + O(n)$
- $T_4(n) = 7 \cdot T_4(\frac{n}{2}) + \Theta(n^2)$

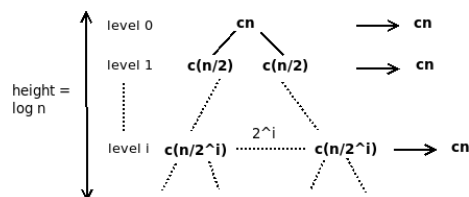
## Solution:

### Recursion tree method

1. In the figure we can see the recursive tree representing the first recursive equation. The tree has height  $h = \log_2 n$  and the maximum costs at each level sum up to  $cn$ . So the total cost is at most:

$$T_1(n) \leq \sum_{i=0}^{\log_2 n} cn = cn \cdot \log n$$

So we can say that  $T_1(n) \in O(n \log_2 n)$



2. in this case the recursive tree is again a binary tree but branches may have different sizes. If  $n$  is a power of 2 height will be  $\log_2 n$  but if it's not the case there exist for sure a power of 2 between  $\frac{n}{2}$  and  $n$  and also one between  $n$  and  $2n$ .

So the length of the left-most branch (the one applying always ceiling) is for sure  $h \leq \log_2 2n$ , while the length of the left-most branch (the one applying always floor) is for sure  $h \geq \log_2 \frac{n}{2}$

Since the last complete level is at height  $h \geq \log_2 \frac{n}{2}$  I can compute an upperbound of  $T_2(n)$  choosing  $c$  as representative of  $\Theta(1)$  and knowing that at level  $i$  there are  $2^i$  nodes:

$$T_2(n) \geq \sum_{i=0}^{\log_2 \frac{n}{2}} 2^i = c \left( 2^{\log_2 \frac{n}{2} + 1} - 1 \right) = c \left( 2^{\log_2 n} - 1 \right) = cn - c$$

so  $T_2(n) \in \Omega(n)$

I can compute also the upperbound:

$$T_2(n) \leq \sum_{i=0}^{\log_2 2n} 2^i = c \left( 2^{\log_2 2n + 1} - 1 \right) = c(4n - 1) = 4cn - c$$

so  $T_2(n) \in O(n)$ .

We can conclude:  $T_2(n) \in \Omega(n)$  and  $T_2(n) \in O(n) \implies T_2(n) \in \Theta(n)$ .

3. The height of the tree is  $h = \log_3 n$  and the maximum costs at the generic level  $i$  sum up to  $cn \cdot \left(\frac{3}{2}\right)^i$ . So the overall cost is at most:

$$T_3(n) \leq \sum_{i=0}^{\log_2 n} cn \cdot \left(\frac{3}{2}\right)^i = cn \cdot \frac{1 - (3/2)^{\log_2 n + 1}}{1 - 3/2} = 2 \cdot cn \left( (3/2)^{\log_2 n + 1} - 1 \right)$$

This means that  $T_3(n) \in O(n^{\log_2 3})$

4. The recursive tree of this recursive equation is similar to the previous one but the cost in each node is exactly  $cn^2$  and each node has 7 children. So the costs at the generic level  $i$  sum up to  $7^i c \left(\frac{n}{2^i}\right)^2$ . The height of the tree is  $h = \log_2 n$  so the complexity is:

$$T_4(n) \leq \sum_{i=0}^{\log_2 n} 7^i c \left(\frac{n}{2^i}\right)^2 = cn^2 \sum_{i=0}^{\log_2 n} \left(\frac{7}{4}\right)^i = cn^2 \cdot \frac{1 - (7/4)^{\log_2 n + 1}}{1 - 7/4} = \frac{4}{3} cn^2 \cdot \left( (7/4)^{\log_2 n + 1} - 1 \right)$$

I can conclude that  $T_4(n) \in O(n^{\log_2 7})$

### Substitution method

1. Assuming that  $T_1(m) \in O(m \log m) \forall m < n$ , let's substitute some representatives in the recursive equation (so  $n > 2$ ):  $cn \log n$  is a representative for  $O(n \log n)$  and  $c'n$  is a representative for  $O(n)$ .

$$T_1(n) \leq 2c \frac{n}{2} \log \frac{n}{2} + c'n = cn(\log n - 2) + c'n = cn \log n - cn \log 2 + c'n \leq cn \log n$$

only when  $-cn \log 2 + c'n \geq 0$ . So if  $c' \geq c \log 2$  we have by induction that  $T_1(n) \in O(n \log n)$

2. Assuming that  $T_2(m) \in \Theta(m) \forall m < n$  ( $n > 2$ ), we want to prove that  $T_2(n) \in \Theta(n)$  holds. First we will prove  $T_2(n) \in O(n)$  and then  $T_2(n) \in \Omega(n)$ .

Let's substitute  $cn$  as representative of  $O(n)$  and  $c'$  as representative of  $\Theta(1)$ :

$$T_2(n) \leq c \left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor + c' \leq cn + c'$$

I can't conclude anything here so I change the representative:  $cn - c''$

$$T_2(n) \leq c \left\lceil \frac{n}{2} \right\rceil - c'' + \left\lfloor \frac{n}{2} \right\rfloor - c'' + c' \leq cn - 2c'' + c' \leq cn - c''$$

when  $c' < c''$ . So  $T_2(n) \in O(n)$ .

Now we choose  $cn$  as representative of  $\Omega(n)$  and  $c'$  as representative of  $\Theta(1)$ :

$$T_2(n) \geq c \left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor + c' \geq cn + c' \geq cn$$

so  $T_2(n) \in \Omega(n)$ . In conclusion  $T_2(n) \in O(n)$  and  $T_2(n) \in \Omega(n) \implies T_2(n) \in \Theta(n)$ .

3. Assuming that  $T_3(m) \in O(m^{\log_2 3}) \forall m < n$  (still  $n > 2$ ), we want to prove that  $T_3(n) \in O(n^{\log_2 3})$  holds. We can substitute in the recursive equation a representative for  $O(n^{\log_2 3}) = cn^{\log_2 3}$  and a representative for  $O(n) = c'n$ :

$$T_3(n) \leq 3c \left( \frac{n}{2} \right)^{\log_2 3} + c'n = 3c \frac{n^{\log_2 3}}{3} + c'n = cn^{\log_2 3} + c'n$$

I've obtained a representative of  $O(cn^{\log_2 3})$  but not the one I started with. So I can try to choose a representative with this form  $cn^{\log_2 3} - c''n$  and recompute the complexity:

$$T_3(n) \leq 3c \left( \frac{n}{2} \right)^{\log_2 3} - 3c'' \frac{n}{2} + c'n = cn^{\log_2 3} - \left( \frac{3}{2}c'' - c' \right)n$$

If I choose  $c'$  and  $c''$  such that  $c' \leq \frac{3}{2}c''$  I can conclude by induction that  $T_3(n) \in O(n^{\log_2 3})$

4. Assuming that  $T_4(m) \in O(m^{\log_2 7}) \forall m < n$  (still  $n > 2$ ), we want to prove that  $T_4(n) \in O(n^{\log_2 7})$  holds. We can take  $cn^{\log_2 7}$  as representative of  $O(n^{\log_2 7})$ ,  $c'n^2$  as representative of  $\Theta(n^2)$  and substitute them in the recursive equation:

$$T_4(n) \leq 7c\left(\frac{n}{2}\right)^{\log_2 7} + c'\left(\frac{n}{2}\right)^2 = cn^{\log_2 7} + \frac{1}{4}c'n^2$$

As above I want to change the representative for  $O(n^{\log_2 7})$ , this time I choose  $cn^{\log_2 7} - c''n^2$ :

$$T_4(n) \leq 7c\left(\frac{n}{2}\right)^{\log_2 7} - 7c''\left(\frac{n}{2}\right)^2 + c'\left(\frac{n}{2}\right)^2 = cn^{\log_2 7} - \frac{1}{4}(7c'' - c')n^2$$

So if I choose  $c'' > \frac{1}{7}c'$  I can conclude that  $T_4(n) \in O(n^{\log_2 7})$



## Weighted Graphs

### Exercise 1

Implement the array-based version of the Dijkstra's algorithm.

### Exercise 2

Implement the binary heap-based version of the Dijkstra's algorithm by using the library `binheap` that was developed during Lesson 6, Lesson 7 and Lesson 8.

### Exercise 3

Test the implementation on a set of instances of the problem and compare their execution times.

### Solution:

Here the execution times of both versions of Dijkstra algorithm tested on different sizes:  $1, 2, \dots, 2^{12}$ . I can see that the heap version is faster.

n	array_dijkstra	heap_dijkstra
1	0.000000	0.000001
2	0.000002	0.000001
4	0.000002	0.000001
8	0.000003	0.000003
16	0.000008	0.000009
32	0.000028	0.000023
64	0.000043	0.000042
128	0.000113	0.000104
256	0.000452	0.000580
512	0.001451	0.001230
1024	0.005580	0.004573
2048	0.021859	0.017379
4096	0.086914	0.068897