UPPSALA UNIVERSITET

Algorithms and Datastructures Assignment 1

Alexander Sundquist, Pontus Björklid, Vilmer Olin & Erik Hennings

19 september 2022

1 Asymptotic Notation

1.1

The following functions are ordered in ascending order. In other words, $g_1 = \mathcal{O}(g_2), g_2 = \mathcal{O}(g_3), ..., g_9 = \mathcal{O}(g_{10})$.

$$
\begin{aligned}
g_1 &= log(2)^n \\
g_2 &= log(n)^2 \\
g_3 &= log(2^n) \\
g_4 &= 10^{1000} * n \\
g_5 &= n * log(n) \\
g_6 &= (n + log(n))^2 \\
g_7 &= (n + log(n))^2 = n^2 + 2(n * log(n)) + (log(n))^2 \\
g_8 &= n^4 \\
g_9 &= 2^{2n} \\
g_{10} &= n!
\end{aligned}
$$

1.2

a) False

b) False

c) True

d) True

1.3

The largest problem size computer A and B can solve respectively for the given algorithm.

Algorithm	A	B
$log(n)$	2^{9*10^7}	$2^{9*10^{10}}$
$n * log(n)$	$4.097 * 10^6$	$2.8648 * 10^9$
n^2	$3 * \sqrt{10^7}$	$3 * 10^5$
2^n	$log(9 * 10^7)$	$log(9 * 10^{10})$

From the above table we conclude the following: the difference between the size of the problems that the two computers can calculate for a given algorithm is noticeably larger for faster algorithms. Therefore, improved hardware has a greater impact on algorithms with lower order growth.

Please only consider upgrading your computer once you have found a faster algorithm.

2 Algorithm Analysis

2.1

A $\mathcal{O}(n \log n)$-time algorithm that determines whether or not there exists a pair of elements $x, y \in S$ such that $x - y = 1$ could be constructed by firstly converting the set to an array, and applying merge sort to the array.

Then the algorithm could loop the sorted array, comparing every element to the next element in the array, returning true if the next element is one larger.

```
**for** i ← 1 to A.length
    **if** $A[i] + 1 = A[i+1]$
    **return** true
**return** false
```

Hence, the upper-bound complexity of the suggested algorithm becomes the upper-bound complexity of merge sort in addition to the time complexity of the conversions from a set to an array and the complexity of the for-loop. The merge sort complexity is given as $\mathcal{O}(n \log n)$ and both of the added operations have linear time complexity $\mathcal{O}(n)$. Since $\mathcal{O}(n \log n)$ has a higher order of growth than the linear operations that were added, the overall time complexity of the algorithm is not affected.

2.2

a) Given the input size, n, of an array, we know that for an iteration of the algorithm, the size of the array is cut in to two sub arrays of half the original size. The sub arrays are then divided in halves and the procedure is repeated until the size of the sub arrays are of size one. From this reasoning the following relation can be derived between the input size and the number of iterations:

n = input size
i = iteration
$1 = \frac{n}{2^i} \Rightarrow n = 2^i \Rightarrow i = log_2(n)$

This means that the maximal number of iterations needed for binary search to find a key for arrays of size 1024 and 2^{40} can be calculated as follows:

$log_2(1024) = 10$
$log_2(2^{40}) = 40$

b) From a recursive tree of binary search the time cost of each level can be seen as a constant, c. There will also be $\log(n) + 1$ levels. From this the total run time, $T(n)$, and time complexity upper bound, $\mathcal{O}$, of binary search can be described as following:

$$T(n) = c(\log(n) + 1) = c\log(n) + c = \mathcal{O}(log(n))$$

c) We strive to show that the upper bound complexity of the modified algorithm is $\mathcal{O}(log(n))$. We show this with the second case of the master theorem, which is defined as follows for the generic function $T(n) = aT(\frac{n}{b}) + f(n)$:

$$if : f(n) = \Theta(n^{log_{b]}(a)}) \Rightarrow then : T(n) = \Theta(n^{log_{b]}(a)} * log(n)$$

In our case: for $n < 1, T(n) = \Theta(1)$ and for $n \geq 1, T(n) = T(n(1-q)) + \Theta(1)$
Which implies:

- $a = 1$
- $f(n) = \mathcal{O}(1) = c$
- $\frac{1}{b} = (1-q) \Rightarrow b = \frac{1}{1-q}$

This gives the following expression for $f(n)$:

$$f(n) = \Theta(n^{log_{\frac{1}{(1-q)}}(1)})$$

Since $b > 1$, for $0 < q < \frac{1}{2}$, and $a = 1$, $f(n)$ can be expressed as:

$$f(n) = \Theta(n^0) = \Theta(1) = c$$

In accordance with the second case of the master theorem this implies the following:

$$T(n) = \Theta(n^{log_{\frac{1}{(1-q)}}(1)} * log(n)) \Rightarrow T(n) = \Theta(n^0 * log(n)) \Rightarrow T(n) = \Theta(log(n))$$

The fact that T(n) is tight bound by $log(n)$ implies that $T(n) = \mathcal{O}(log(n))$. Quod erat demonstrandum.

2.3

1. Best case:

Line	Cost	Times
1	c_1	1
2	c_2	1
3	c_3	1
4	c_4	1
5	c_5	1
6	c_6	$n-1$
7	c_7	0
8	c_8	0
9	c_9	$n-1$
10	c_{10}	1
11	c_{11}	1
12	c_{12}	1

The total run time, $T(n)$, for the best case can be written as the following:

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 * (n-1) + c_9 * (n-1) + c_{10} + c_{11} + c_{12} \\ &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 n - c_6 + c_9 n - c_9 + c_{10} + c_{11} + c_{12} \\ &= (c_6 + c_9)n + c_1 + c_2 + c_3 + c_4 + c_5 - c_6 - c_9 + c_{10} + c_{11} + c_{12} \\ &= bn + c \\ &\Rightarrow T(n) = \mathcal{O}(n) \end{aligned}$$

Worst case:

Line	Cost	Times
1	c_1	1
2	c_2	n
3	c_3	n
4	c_4	n
5	c_5	$n-1$
6	c_6	$n(n-1)$
7	c_7	$\sum_{i=1}^{n}(i-1)$
8	c_8	$\sum_{i=1}^{n}(i-1)$
9	c_9	$n(n-1)$
10	c_{10}	$n-1$
11	c_{11}	n
12	c_{12}	1

The summation, $\sum_{i=1}^{n}(i-1)$, in line 7 and 8 can be rewritten as:

$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$$

The total run time, $T(n)$, of the worst case can be written accordingly:

$$
\begin{aligned}
T(n) &= c_1 + c_2 n + c_3 n + c_4 n + c_5(n-1) + c_6 n(n-1) + c_7 \frac{n(n-1)}{2} + c_8 \frac{n(n-1)}{2} + \\
&+ c_9 n(n-1) + c_{10}(n-1) + c_{11} n + c_{12} \\
&= (c_6 + c_9) n(n-1) + (c_2 + c_3 + c_4 + c_{11}) n + (c_5 + c_{10})(n-1) + \\
&+ (c_7 + c_8)(\frac{n(n-1)}{2}) + (c_1 + c_{12}) \\
&= (c_6 + \frac{c_7}{2} + \frac{c_8}{2} + c_9) n^2 + (c_2 + c_3 + c_4 + c_5 + c_6 - \frac{c_7}{2} - \frac{c_8}{2} + c_9 + c_{10} + c_{11}) n \\
&+ (c_1 + c_{12}) \\
&= an^2 + bn + c \\
&\Rightarrow T(n) = \mathcal{O}(n^2)
\end{aligned}
$$

2. After every **repeat** iteration the smallest element, that has not yet been sorted, in the array is moved down until it meets a smaller already sorted element. For each **repeat** iteration the not yet sorted, smallest element will find its final spot. It is therefore unnecessary to keep checking the elements to the left of the element once we have reached the sorted elements. We could resolve this minor inconvenience by simply making the base element one value larger for each repeat iteration. The optimisation could be implemented by moving the base assignment outside the **repeat** loop and adding an incremental addition to the base after line 10 instead.

   With the described implementation, the only significant change in line run time is for the if-statement. The if-statement will run $(\sum_{i=1}^{n}(i-1)) + 1 = \frac{n(n-1)}{2} + 1$ times, in the worst case. Because, the total run time is calculated as a summation of all the line run times, it is clear that the upper bound complexity of the total run time, $T(n)$, will remain the same as before the optimization. Therefore, $T(n) = \mathcal{O}(n^2)$.

3 Bonus Question

4 Work Distribution

In brief, the work distribution looks as follows; We started by discussing every question as a group. Once we had come up with a solution that everyone was satisfied with we wrote down the answer in our notes. Afterwards we wrote our answers in Latex.