Foundations of Computing Assignment 4: Algorithms

Agata Borkowska, UID: 1690550

February 12, 2017

1

First note that in an hour we can perform $60 \cdot 60 \cdot 987 = 3553200$ simple operations.

1.1 $f(n) = n$

max. n = 3553200

1.2 $f(n) = n^3$

max. n = $\sqrt[3]{3553200} = 152.59 \approx 152$ (rounded down)

1.3 $f(n) = 5 \cdot log_2(n) + 6$

max. n will be such that:
$5 \cdot log_2(n) + 6 \leq 3553200$
$\iff 5 \cdot log_2(n) \leq 3553194$
$\iff log_2(n) \leq 710638.8$
$\iff n \leq 2^{710638.8}$
$\iff n \leq 3.93444 \cdot 10^{213923}$

1.4 $f(n) = 2n \times log_2(n)$

max. n will be such that:
$2n \cdot log_2(n) \leq 3553200$
$\iff n \cdot ln(n) \leq 1776600 \cdot ln(2)$
$\iff n \leq e^{W(1776600ln(2))}$
Where W is the product-log function, which gives the solution for w in $z = we^w$. More specifically solutions for $xlog_b(x) = a$ are of the form $x = e^{W(aln(b))}$ [1]

Therefore $n \leq 106389$.

1.5 $f(n) = 2^n$

.

max. n will be such that:
$2^n \leq 3553200$
$\iff n \leq log_2(3553200) = 21.76$
Therefore n can be at most 21.

[1] Source: https://en.wikipedia.org/wiki/Lambert_W_function#Example_5

2

Recall definitions:

- A function $f(n)$ is $O(g(n))$ if there exists a constant $c > 0$ such that $f(n) \leq cg(n)$ for all n sufficiently large.
- A function $f(n)$ is $\Omega(g(n))$ if there exists a constant $c > 0$ such that $f(n) \geq cg(n)$ for all n sufficiently large.
- a function $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n sufficiently large.

2.1 Claim: $f(n) = 12n^3 - 7n + 6$ is $O(n^3)$

True. For $n > 0$, $-7n + 6 < 0$ so $f(n) < 12n^3$, where $12n^3$ is $cg(n)$, with $c = 12$ and $g(n) = n^3$. Therefore it is $O(n^3)$.

2.2 Claim: $f(n) = 8n^2 + 4$ is $O(n^2)$

True. For $n > 2$, $8n^2 + 4 < 9n^2$, so take $c = 9$, $g = n^2$, and thus $f(n)$ is $O(n^2)$.

2.3 Claim: $f(n) = 3n^4 - 2n^2 + 3$ is $\Omega(n^2)$.

True. For $n > 1, (n^4 - n^2) > 0$, so f(n) can be rewritten as: $f(n) = n^4 + 2(n^4 - n^2) + 3 > n^4 > n^2$, so $f(n)$ is $\Omega(n^2)$.

2.4 Claim: $f(n) = 3n^4 - 2n^2 + 3$ is $O(n^2)$.

False. We've shown above that n^2 is a lower bound on f(n), and so is n^4. Observe that for all $c \in \mathbb{N}$, there is a $k > 0$ such that for all $n > k$, $n^4 > cn^2$. Therefore, $f(n) > n^4 > cn^2$, so n^2 cannot be an upper bound on f(n).

2.5 Claim: $18n + 19$ is $\Theta(n)$

True. Take $c_1 = 10, c_2 = 100$, then $10n < 18n + 19 < 100n$ for all $n > 0$.

3

Recall: Master Theorem applies to recurrence relations of the form $T(n) = aT(\frac{n}{b}) + f(n)$, where:

- a and b are constants
- f(n) is a positive polynomial
- recursive calls have the same problem size

(Following statements in the lecture notes.) Therefore:

3.1 $4T(\frac{n}{2}) + 16n^2$

$a = 4$, $b = 2$, $f(n) = 16n^2$. The conditions of Master Theorem are satisfied, and we can apply it.

First note that $f(n) = 16n^2 \in \Theta(n^2 \cdot log^0(n))$, so $c = 2$, and $n^{log_b(a)} = n^{log_2(4)} = n^2$.
Thus $c = log_b(a)$, and we can apply the 2nd case of the theorem, to get:

$$T(n) = \Theta(n^2 \cdot log(n))$$

3.2 $8T(\frac{n}{2}) + 160n$

$a = 8$, $b = 2$, $f(n) = 160n$, and the conditions of Master Theorem are satisfied, so we can apply it.
$f(n) \in O(n^1)$, so $c = 1$.
$log^2(8) = 3 > c$, so we're in the 1st case of the theorem, and the term $n^{log_b(a)}$ is dominant, so we get:

$$T(n) = \Theta(n^3)$$

3.3 $T(n) = 2^n \cdot T(\frac{n}{2}) + n$

$a = 2^n$ isn't constant, so we cannot apply Master Theorem.

3.4 $T(n) = 9T(\frac{n}{3}) + 97n^3$

$a = 9$, $b = 3$, and $f(n) = 97n^3 = \Theta(n^3)$. The conditions of the Master Theorem are satisfied (it's worth noting that n is positive-valued, therefore n^3 is also positive).

$n^{log_b(a)} = n^{log_3(9)} = n^2$, so we're in the 3rd case of the Master Theorem. Thus:

$$T(n) = \Theta(n^3)$$

3.5 $T(n) = T(\frac{2n}{3}) + 1$

$a = 1$, $b = \frac{3}{2}$, and $f(n) = 1$, so $n^{log_b(a)} = n^{log_{3/2}(1)} = n^0 = 1$, so the 2nd case applies, with $f(n) = \Theta(1) = \Theta(n^{log_b(a)})$. Thus we get:

$$T(n) = \Theta(log(n))$$

4

Since all of the examples have unsorted integers 1-6, we could use **counting sort**[2], which runs in $O(n)$, by creating an array of size 6 and inserting the elements into the correct places in this array, with that prior knowledge about their values.

Otherwise[3]:

4.1 1, 2, 3, 4, 6, 5

Use **insertion sort**, because it will terminate after scanning through the input once, and not making any swaps. Alternatively, **augmented bubble sort** - with the augmentation being a counter on the number of swaps performed in each pass, and the algorithm terminating when the counter is 0 after a pass.

4.2 6, 2, 3, 4, 5, 1

Cocktail shaker sort[4] which is a bi-directional bubble sort. It will first move 6 to the end, then start from the end, and move 1 to the beginning. Again, we want it to terminate after the pass where no swaps are made. Similarly to bubble sort, it's done in place.

[2] T. Cormen et al., "Introduction to Algorithms", 3rd ed., Cambridge, MA: MIT Press, 2009, ch. 8, pp. 194-196 - aka CLRS
[3] https://en.wikipedia.org/wiki/Sorting_algorithm was of great help
[4] https://en.wikipedia.org/wiki/Cocktail_shaker_sort

4.3 6, 5, 4, 3, 2, 1

Heap sort will be the best choice here, because the list is in reverse order, so initialising the algorithm will create a max-heap, and no swaps will be necessary. The time-complexity will be $O(n \cdot log(n))$, because the heapify operation is $O(n)$ and repeatedly extracting max is $O(n\dot{l}og(n))$.

While merge sort shares this property, heap sort can be done in place, while merge sort space complexity is $O(n)$.

We do NOT want to use quicksort, as this is the exemplar worst case for it, and it would run in $O(n^2)$.

4.4 1, 5, 4, 6, 3, 2

There is no pattern to these numbers, so anything with a good average-case running time will be similarly appropriate. Therefore, my go-to algorithm would again be **heap sort**, because of in-place sorting and $O(n \cdot log(n))$ time complexity.

4.5 ((9,3), (2,5), (2,4), (2,7), (1,4), (5,7))

We need a stable algorithm, so our choice is limited to bubble sort, insertion sort, and merge sort. Of those three, the first two are done in place, so we can exclude merge sort from further consideration.

If the data is given as an array, I would pick **bubble sort**, which will require 8 swaps until the list is sorted (again assuming that the algorithm is augmented to recognize when no swaps are performed).

If the data is given as a linked list, **insertion sort** will be slightly faster, because inserting an element into the sorted list will require only updating pointers, while if it was an array, to achieve sorting in place, we would have to propagate swaps, and it would be a time-consuming procedure, not better than bubble sort.

5

Stage	Current V	A	B	C	D	E	F	Visited set
0	A	- 0	- ∞	- ∞	- ∞	- ∞	- ∞	{}
1	A	- 0	A 19	- ∞	- ∞	- ∞	- ∞	{}
2	A	- 0	A 19	A 12	- ∞	- ∞	- ∞	{}
3	B	- 0	A 19	A 12	B 23	- ∞	- ∞	{A}
4	B	- 0	A 19	A 12	B 23	B 32	- ∞	{A}
5	C	- 0	A 19	A 12	B 23	C 28	- ∞	{A,B}
6	D	- 0	A 19	A 12	B 23	D 26	- ∞	{A,B, C}
7	E	- 0	A 19	A 12	B 23	D 26	F 27	{A,B,C,D}
8	F	- 0	A 19	A 12	B 23	D 26	F 27	{A,B,C,D,E}

Table 1: Dijkstra's algorithm performed on the given graph

Short explanation: I find it helpful to keep track of "visited" nodes. When all neighbours of a current node have been considered, it is then added to the "visited" set. After the 8th iteration, F is added to this set, which now includes all nodes, and thus the algorithm terminates.

6

As before, I am keeping track of the visited nodes. As this is a directed graph, it takes a few iterations between considering a node and considering all its neighbours. For example, A can be marked as visited only once we have considered C, which is the last node picked by the algorithm.

In the case of a directed graph, visited nodes are marked when all incoming edges incident to them have been considered.

Stage	Current V	A	B	C	D	E	F	Visited
0	A	- 0	- ∞	- ∞	- ∞	- ∞	- ∞	{}
1	A	- 0	A 19	- ∞	- ∞	- ∞	- ∞	{}
2	B	- 0	A 19	- ∞	- ∞	B 32	- ∞	{}
3	E	- 0	A 19	- ∞	E 35	B 32	- ∞	{}
4	E	- 0	A 19	- ∞	E 35	B 32	E 33	{D}
5	E	- 0	A 19	E 48	E 35	B 32	E 33	{D,F}
6	D	- 0	A 19	E 48	E 35	B 32	E 33	{D,F,C}
7	F	- 0	A 19	E 48	E 35	B 32	E 33	{D,F,C,B}
8	C	- 0	A 19	E 48	E 35	B 32	E 33	{D,F,C,B,E}

Table 2: Dijkstra's algorithm performed on the given graph

7

7.1

Stage	V	E
0	{A}	{}
1	{A,C}	{(A,C)}
2	{A,C,E}	{(A,C), (C,E)}
3	{A,C,E,F}	{(A,C), (C,E), (E,F)}
4	{A,C,E,F,D}	{(A,C), (C,E), (E,F), (E,D)}
5	{A,C,E,F,D,B}	{(A,C), (C,E), (E,F), (E,D), (E,B)}

Table 3: Prim's algorithm performed on the given graph

The weight of the MST is $8 + 5 + 1 + 3 + 6 + 32 = 55$

7.2

Stage	Edges	Components	E
0	AB = 11, AC = 8, BD = 19, BE = 6, CE = 5, CG = 32, DE = 3, EF = 1	{(A), (B), (C), (D), (E), (F),(G)}	{}
1	AB = 11, AC = 8, BD = 19, BE = 6, CE = 5, CG = 32, DE = 3	{(A), (B), (C), (D), (E, F), (G)}	{(E, F)}
2	AB = 11, AC = 8, BD = 19, BE = 6, CE = 5, CG = 32	{(A), (B), (C), (D, E, F), (G)}	{(E, F), (D, E)}
3	AB = 11, AC = 8, BD = 19, BE = 6, CG = 32	{(A), (B), (C, D, E, F), (G)}	{(E, F), (D, E), (C,E)}
4	AB = 11, AC = 8, BD = 19, CG = 32	{(A), (B, C, D, E, F), (G)}	{(E, F), (D, E), (C,E), (B,E)}
5	AB = 11, BD = 19, CG = 32	{(A, B, C, D, E, F), (G)}	{(E, F), (D, E), (C,E), (B,E), (A,C)}
6	BD = 19, CG = 32	{(A, B, C, D, E, F), (G)}	{(E, F), (D, E), (C,E), (B,E), (A,C)}
7	CG = 32	{(A, B, C, D, E, F), (G)}	{(E, F), (D, E), (C,E), (B,E), (A,C)}
8		{(A, B, C, D, E, F, G)}	{(E, F), (D, E), (C,E), (B,E), (A,C), (C,G)}

Table 4: Kruskal's algorithm performed on the given graph

Weight of the MST is: $1 + 3 + 5 + 6 + 8 + 32 = 55$.