

**Technical Report: Patients or Patience?**

**Authors: Ryan Westwood**

**Date: 13/05/2020**

**Wordcount: 1326**

**Page count: 12**

**Confidential: No**

Contents

Executive summary 2

Introduction 3

Theory 3

Implementation 4

Conclusions 6

References 7

Appendix A 7

Executive Summary

This report will be on simulating an A&E room with varying patient conditions, some more severe than others. Occasionally the hospital may have patients that are waiting for longer than 4 hours and this will result in a £10,000 fine.

Through running the simulation, users will be able to change the variables to effectively simulate different times of day. By changing how many patients can be admitted and discharged every 10 minutes the simulation will be able to forecast the number of fines the hospital may incur and how long the queue could become.

The simulation in this report will be able to identify which patients are waiting the longest and in turn which patients are making the hospital incur fines. With this the information the hospital can better weight the patient’s priority to lessen the number of fines or find other ways to improve treating patients, such as building more patient rooms or treating patients faster.

Introduction

The *Patients or Patience?* case study requires a simulation of an A&E department with a range of incoming patients with varying conditions. These patients must then be effectively prioritized so the most severely hurt patient is attended to first. All the patients will have an entry priority and a time priority which will increase the patient’s total priority over time. However, if a patient has been waiting 4 hours or more the hospital will receive a £10,000 fine.

The program has 2 requirements:

* The patient with the highest priority should be seen to first
* 10 unique patient conditions with weights already associated with them

The program has 3 variables:

* How many people are already in the A&E room?
* How many people arrive every 10 minutes?
* How many people leave every 10 minutes?

Theory

For the *Patients or Patience?* case study. Custom classes will be used as this will allow for great control over what the objects will be able to store and process. The use of custom classes will allow for data structures and algorithms that are efficient at working with the data necessary to simulate a hospital A&E room.

Nodes will be used to hold the patient’s information as nodes can store multiple different data types and allow for functionality such as an update over time function. The patients will store; name, injury, entry priority, time priority, total priority, wait time and ID. While holding functions such as; print patient information and update. Which will print out a formatted string containing all the patient’s information and update which will update the patient’s priority and add 10 minutes to their current wait time.

A queue will be used to simulate the waiting in the A&E room. The queue will need functions such as enqueue, dequeue and sort along with holding multiple data types to correctly keep statistics of how the hospital is performing. The queue class will use lists to hold the Node objects as it will be easy to add/remove items using the lists built-in functionality. Such as, .Append() and .Clear().

The sorting algorithm used is merge sort as its average time complexity is 0(n log(n)). This is important as the list will need to be sorted any time patients are added to the queue or updated. With small datasets such as 100 patients, the user cannot discern the difference between the sorting algorithm in use. However, as soon as the patients currently in queue exceed 10,000. Which would not be realistic for this simulation as most A&E rooms cannot hold that amount of people. The time to sort would become noticeable to the user as it could mean seconds or even minutes between steps.

Implementation

In the theory section the different ways of storing and sorting data were explored. This simulation will be using custom classes for the Queue and Nodes. This allows for bespoke functionality of the system.

The node class will be created to store all the information the patient requires. This class will also have functionality for updating the patient’s priority so that stepping through the simulation is possible. Along with a print function that will print a formatted string that contains all the patient’s data.

The queue class will hold all the node objects along with functionality. These functions include enqueue, dequeue, print, sort and set criteria which is used to set up the variables for the simulation, which controls how many patients are already waiting and how many to enqueue and dequeue per step.

Choosing the correct sorting algorithm greatly changes how efficient the program is. Insertion sort is useful as its best-case scenario is only Ω (n). This is good for if you wanted to change the simulation to be less random and simulate an accident where an influx of patients with the same injury arrive. However, on average insertion sort is Θ (n^2) which makes it very inefficient. This becomes a lot more noticeable on bigger data sets.

Merge sort is an alternative to insertion sort as on average as the time complexity is Θ (n log(n)). This means on average; merge sort should be faster than insertion sort Θ (n^2) when all incoming patients are random. However, the best-case scenario for merge sort is Ω (n log(n)) which is slower than insertion sorts Ω (n)

Another alternative algorithm is tim sort. Which takes the advantages of both insertion and merge sort and combines them into one. This means that tim sort will have the same best case as insertion sort Ω (n)and the same average case of merge sort Θ (n log(n)).

Table : Static analysis of the different sorting algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Time Complexity | | |
|  | Best (Ω) | Average (Θ) | Worst (O) |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) |
| Merge Sort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) |
| Tim Sort | Ω(n) | Θ(n log(n)) | O(n log(n)) |

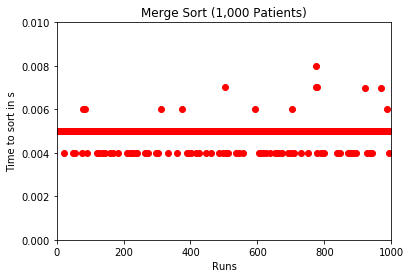


Figure 1: Merge sort on 1,000 patients. Average time: 4957490.9ns

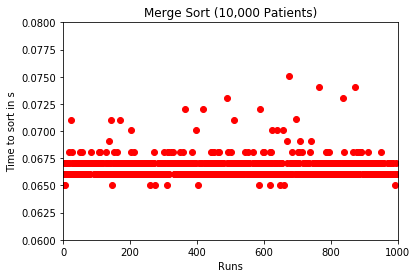


Figure 2: Merge sort on 10,000 patients. Average time: 66633569.4ns

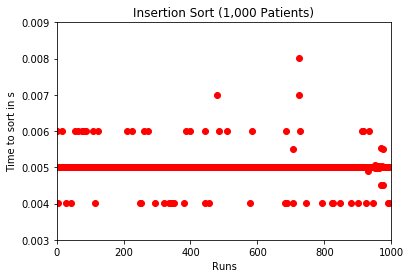


Figure 3: Insertion sort on 1,000 patients. Average time: 5008673.0ns

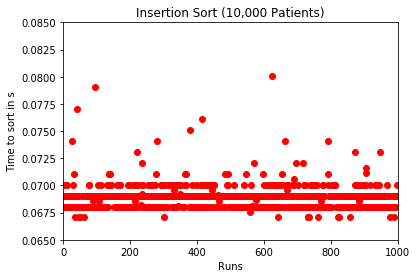


Figure 4: Insertion sort on 10,000 patients. Average time: 68906051.9ns

Table : Dynamic analysis of the different sorting algorithms.

|  |  |  |
| --- | --- | --- |
| Algorithm | Average Completion (ns) | |
|  | 1,000 Patients | 10,000 Patients |
| Insertion Sort | 5008673.0 | 68906051.9 |
| Merge Sort | 4957490.9 | 66633569.4 |
| Tim Sort | 208187.6 | 2220097.9 |

The entire program will be wrapped in a while loop which will wait for an enter key press before iterating. This is useful as it will allow the user to see the patients currently waiting in the queue and the patients that were dequeued during that step of the simulation. Along with stats in an easier to read form with current number of patients, patients that have been seen and the hospitals current fine.

Conclusions

In conclusion, the program can be useful for simulating multiple different variations of an A&E department. The program could be further developed by changing the patient weightings to reduce the number of fines the hospital receives. The variables the user can change could also become more in depth by having them set more criteria than just incoming and outgoing per step. This could mean more patients arrive during peak times or during accidents and less night-time.

This program could also be improved to be more efficient and data driven. For example, Python’s built in list .sort() method uses tim sort which combines the best of insertion and merge sort which would make the sorting of patients quicker. This could also be aided by not giving the user visual feedback per step(see Figure 7) rather just statistics at the end of the simulation with all the necessary information.

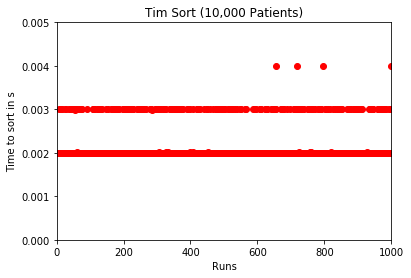


Figure 6: Tim sort on 10,000 patients. Average time: 2220097.9ns

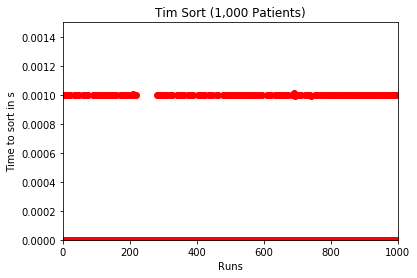


Figure 5: Tim sort on 1,000 patients. Average time: 208187.6ns

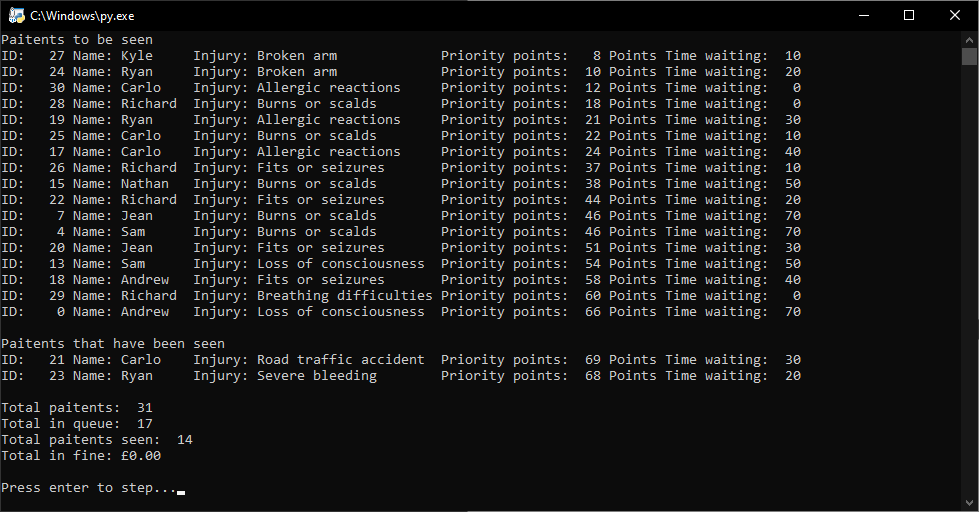


Figure 7: Visual feedback per step.

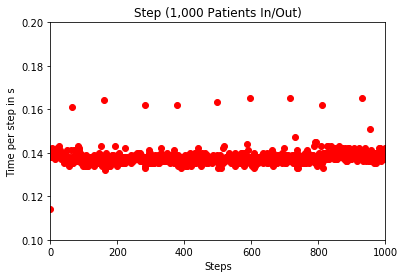


Figure 8: Average time to complete per step

References

Marini, J., 2020. *Programming Foundations: Algorithms*. [image] Available at: <https://www.linkedin.com/learning/programming-foundations-algorithms/algorithms-power-the-world?u=42288921> [Accessed 15 April 2020].

Appendix A

The following is the Python code for the A&E simulation.

1. """
2. Created on Sat Apr 25 17:28:14 2020
4. @author: RyanW
5. """
7. **import** locale
8. locale.setlocale( locale.LC\_ALL, '' )
10. **import** random
12. **import** os
13. clear = **lambda**: os.system('cls')
15. fine = 0
17. **class** Queue:
18. **def** \_\_init\_\_(self):
19. self.items = []
20. self.outgoingPaitents = []
21. self.id = 0
22. self.seen = 0
23. self.condition = 0
24. self.incomingTime = 0
25. self.incomingTimeCur = 0
26. self.outgoingTime = 0
27. self.outgoingTimeCur = 0
29. **def** setCriteria(self, freqIncoming, freqOutgoing):
30. self.incomingTime = freqIncoming
31. self.outgoingTime = freqOutgoing
33. **def** enqueue(self, noOfPaitents):
34. name = ['Ryan', 'Carlo', 'Richard', 'Nathan', 'Sam', 'Kyle', 'Andrew', 'Jean']
35. **for** i **in** range(noOfPaitents):
36. temp = random.randint(0,9)
37. **if** temp == 9:
38. self.items.append(LossOfConsciousness(random.choice(name), self.id))
39. **if** temp == 8:
40. self.items.append(FitsOrSeizures(random.choice(name), self.id))
41. **if** temp == 7:
42. self.items.append(ChestPain(random.choice(name), self.id))
43. **if** temp == 6:
44. self.items.append(BreathingDifficulties(random.choice(name), self.id))
45. **if** temp == 5:
46. self.items.append(SevereBleeding(random.choice(name), self.id))
47. **if** temp == 4:
48. self.items.append(AllergicReactions(random.choice(name), self.id))
49. **if** temp == 3:
50. self.items.append(BurnsOrScalds(random.choice(name), self.id))
51. **if** temp == 2:
52. self.items.append(Stroke(random.choice(name), self.id))
53. **if** temp == 1:
54. self.items.append(RoadTrafficAccident(random.choice(name), self.id))
55. **if** temp == 0:
56. self.items.append(BrokenArm(random.choice(name), self.id))
57. self.id += 1
59. **def** dequeue(self, val):
60. self.outgoingPaitents.clear()
61. **for** i **in** range(val):
62. n = len(self.items) - 1
63. self.outgoingPaitents.append(self.items[n])
64. **del** self.items[n]
65. self.seen += 1
67. **def** update(self):
68. **for** x **in** self.items:
69. x.update()
71. **def** printQueue(self):
72. **if** len(self.items) == 0:
73. **print**(f'Nobody is in the queue!')
74. **else**:
75. **print**(f'Paitents to be seen')
76. **for** x **in** self.items:
77. x.stat()
78. **print**()
79. **print**(f'Paitents that have been seen')
80. **for** x **in** self.outgoingPaitents:
81. x.stat()
83. **def** mergesort(self, dataset):
84. **if** len(dataset) > 1:
85. mid = len(dataset) // 2
86. leftarr = dataset[:mid]
87. rightarr = dataset[mid:]
88. self.mergesort(leftarr)
89. self.mergesort(rightarr)
90. i=0
91. j=0
92. k=0
93. **while** i < len(leftarr) **and** j < len(rightarr):
94. **if** leftarr[i].Priority() < rightarr[j].Priority():
95. dataset[k] = leftarr[i]
96. i += 1
97. **else**:
98. dataset[k] = rightarr[j]
99. j += 1
100. k += 1
101. **while** i < len(leftarr):
102. dataset[k] = leftarr[i]
103. i += 1
104. k += 1
105. **while** j < len(rightarr):
106. dataset[k] = rightarr[j]
107. j += 1
108. k += 1
110. **def** sort(self):
111. self.mergesort(self.items)
113. **def** getSize(self):
114. **return** len(self.items)
116. **def** getID(self):
117. **return** self.id
119. **def** getSeen(self):
120. **return** self.seen
122. **def** getFine(self):
123. **return** self.fine
125. **class** Node:
126. **def** \_\_init\_\_(self, injury, entry, timePriority, name, num):
127. self.name = name
128. self.injury = injury
129. self.entryPriority = entry
130. self.timePriority = timePriority
131. self.totalPriority = entry
132. self.waitTime = 0
133. self.num = num
135. **def** update(self):
136. **global** fine
137. self.waitTime += 10
138. self.totalPriority += self.timePriority
139. **if** self.waitTime == 240:
140. fine += 10000
142. **def** stat(self):
143. **print**(f"ID: {self.num:4} Name: {self.name:8} Injury: {self.injury:22} Priority points: {self.totalPriority:3} Points Time waiting: {self.waitTime:3}")
145. **def** Priority(self, t = None):
146. **if** t == 0: self.totalPriority = 0
147. **if** t: self.totalPriority = t
148. **return** self.totalPriority
150. **def** WaitTime(self, t = None):
151. **if** t == 0: self.waitTime = 0
152. **if** t: self.waitTime = t
153. **return** self.waitTime
155. **def** Name(self, t = None):
156. **if** t: self.name = t
157. **return** self.name
159. **def** getID(self):
160. **return** self.num
162. **def** getInjury(self):
163. **return** self.injury

166. **class** LossOfConsciousness(Node):
167. **def** \_\_init\_\_(self, name, num):
168. super(LossOfConsciousness,self).\_\_init\_\_("Loss of consciousness", 24, 6, name, num)
170. **class** FitsOrSeizures(Node):
171. **def** \_\_init\_\_(self, name, num):
172. super(FitsOrSeizures,self).\_\_init\_\_("Fits or seizures", 30, 7, name, num)
174. **class** ChestPain(Node):
175. **def** \_\_init\_\_(self, name, num):
176. super(ChestPain,self).\_\_init\_\_("Chest Pain", 42, 8, name, num)
178. **class** BreathingDifficulties(Node):
179. **def** \_\_init\_\_(self, name, num):
180. super(BreathingDifficulties,self).\_\_init\_\_("Breathing difficulties", 60, 10, name, num)
182. **class** SevereBleeding(Node):
183. **def** \_\_init\_\_(self, name, num):
184. super(SevereBleeding,self).\_\_init\_\_("Severe bleeding", 48, 10, name, num)
186. **class** AllergicReactions(Node):
187. **def** \_\_init\_\_(self, name, num):
188. super(AllergicReactions,self).\_\_init\_\_("Allergic reactions", 12, 3, name, num)
190. **class** BurnsOrScalds(Node):
191. **def** \_\_init\_\_(self, name, num):
192. super(BurnsOrScalds,self).\_\_init\_\_("Burns or scalds", 18, 4, name, num)
194. **class** Stroke(Node):
195. **def** \_\_init\_\_(self, name, num):
196. super(Stroke,self).\_\_init\_\_("Stroke", 36, 2, name, num)
198. **class** RoadTrafficAccident(Node):
199. **def** \_\_init\_\_(self, name, num):
200. super(RoadTrafficAccident,self).\_\_init\_\_("Road traffic accident", 54, 5, name, num)
202. **class** BrokenArm(Node):
203. **def** \_\_init\_\_(self, name, num):
204. super(BrokenArm,self).\_\_init\_\_("Broken arm", 6, 2, name, num)

2. **if** \_\_name\_\_ == '\_\_main\_\_':
3. root = Queue()
4. noOfPaitents = int(input(f'Number of paitents: '))
5. intake = int(input(f'Incoming: '))
6. outgoing = int(input(f'Outgoing: '))
7. root.enqueue(noOfPaitents)
8. firstRun = True
9. **while** 1:
10. clear()
11. **if** firstRun == False:
12. root.enqueue(intake)
13. root.sort()
14. root.dequeue(outgoing)
15. root.printQueue()
16. **print**()
17. **print**(f'Total paitents: {root.getID():3}')
18. **print**(f'Total in queue: {root.getSize():3} ')
19. **print**(f'Total paitents seen: {root.getSeen():3}')
20. **print**(f'Total in fine: {locale.currency( fine, grouping=True )} ')
21. **print**()
22. input(f'Press enter to step...')
23. root.update()
24. firstRun = False