Name: Guoqing Yu

Problem #1

1. A heap is a complete binary tree, a binary tree in which every level, except possibly the last, is completely filled, and all nodes on the last level are as far left as possible. It can have between 1 and 2*h* nodes on the last level *h*.

Therefore, while *h* = 7, the last level of a binary tree can contain 1 ~ 27 nodes, and there must be 1 + 2 + 22 + … + 26 = 27 – 1 nodes on the higher levels.

So, in the minimum case, there are 1 + (27 – 1) = 128 nodes.

In the maximum case, there are 27 + (27 – 1) = 255 nodes.

1. A diagram of numbers and circles

   Description automatically generatedThe initial ordering is [26, 15, 10, 35, 29, 17, 22], as represented by the tree below:

To start the BUILD-MAX-HEAP procedure, we know heap\_size = 7, so i loops from 3 to 1.

While i = 3: MAX-HEAPIFY on element 10. Let 10 compare with its children. Since 22 is the largest element, elements 10 and 22 are swapped. The current ordering is [26, 15, 22, 35, 29, 17, 10]. Then, MAX\_HEAPIFY on element 10 yields no change.

A diagram of numbers and circles

Description automatically generated

While i = 2: MAX\_HEAPIFY on element 15. Let 15 compare with its children. Since 35 is the largest element, elements 15 and 35 are swapped. The current ordering is [26, 35, 22, 15, 29, 17, 10]. Then, MAX\_HEAPIFY on element 15 yields no change.

A diagram of numbers and circles

Description automatically generated

A diagram of numbers and circles

Description automatically generatedWhile i = 1, MAX\_HEAPIFY on element 26. Let 26 compare with its children. Since 35 is the largest element, elements 26 and 35 are swapped. The current ordering is [35, 26, 22, 15, 29, 17, 10].

Then, MAX\_HEAPIFY on element 26. Let 26 compare with its children. Since 29 is the largest element, elements 26 and 29 are swapped. The current ordering is [35, 29, 22, 15, 26, 17, 10]. Then, MAX\_HEAPIFY on element 26 yields no change.

A diagram of numbers and circles

Description automatically generated

Problem #2

1. *A* = [17, 10, 25, 35, 29, 15, 22]

* To start with, we implement partitioning on subarray *A*[1…7] (*p* = 1, *r* = 7):
  + Pick the last element *A*[7] as the pivot: *x* = 22
  + Set *i* as 0 and traverse *j* from 1 to 6:
  + *i* = 0, *j* = 1 => *A*[1] = 17 < 22 => *i* increments to 1, exchange *A*[1] with *A*[1] yields no change.
  + *i* = 1, *j* = 2 => *A*[2] = 10 < 22 => *i* increments to 2, exchange *A*[2] with *A*[2] yields no change.
  + *i* = 2, *j* = 3 => *A*[3] = 25 > 22
  + *i* = 2, *j* = 4 => *A*[4] = 35 > 22
  + *i* = 2, *j* = 5 => *A*[5] = 29 > 22
  + *i* = 2, *j* = 6 => *A*[6] = 15 < 22 => *i* increments to 3, exchange *A*[3] with *A*[6] to get: [17, 10, 15, 35, 29, 25, 22]
  + Swap *A*[4] with *A*[7] to get: [17, 10, 15, 22, 29, 25, 35]
  + Return 4.
* Partition on *A*[1…3] (*p* = 1, *r* = 3):
  + Pick the last element *A*[3] as the pivot: *x* = 15
  + Set *i* as 0 and traverse *j* from 1 to 2:
  + *i* = 0, *j* = 1 => *A*[1] = 17 > 15
  + *i* = 0, *j* = 2 => *A*[2] = 10 < 15 => *i* increments to 1, exchange *A*[1] with *A*[2] to get: [10, 17, 15, 22, 29, 25, 35]
  + Swap *A*[2] with *A*[3] to get: [10, 15, 17, 22, 29, 25, 35]
  + Return 2
* Partition on *A*[1…1] is base case.
* Partition on *A*[3…3] is also base case.
* Partition on *A*[5…7] (*p* = 5, *r* = 7):
  + Pick the last element *A*[7] as the pivot: *x* = 35
  + Set *i* as 4 and traverse *j* from 5 to 6:
  + *i* = 4, *j* = 5 => *A*[5] = 29 < 35 => *i* increments to 5, exchange *A*[5] with *A*[5] yields no change
  + *i* = 5, *j* = 6 => *A*[6] = 25 < 35 => *i* increments to 6, exchange *A*[6] with *A*[6] yields no change
  + Swap *A*[7] with *A*[7] yields no change.
* Partition on *A*[5…6] (*p* = 5, *r* = 6):
  + Pick the last element *A*[6] as the pivot: *x* = 25
  + *i* = 4, *j* = 5 => *A*[5] = 29 > 25
  + Swap *A*[5] and *A*[6] to get: [10, 15, 17, 22, 25, 29, 35]

1. *A* = [10, 15, 20, 25, 30, 35, 40]

In the first partition of the entire array of length 7, we need *n* = 6 total comparisons. Since all elements on the left side of the pivot are smaller than x, none of elements are swapped (*i* and *j* always increment concurrently and swapping them yields no difference). The returned value of the first partition is 7.

Then, we move on to partition the subarray *A*[1…6]. Similarly, we need 5 comparisons for a subarray of length 6, with a returned value of 6 because none of elements are swapped.

Then, we move on to partition the subarray *A*[1…5], which we need 4 comparisons.

…

Finally, we partition the subarray *A*[1…2], where only 1 comparison is needed.

As a result, we need 6 + 5 + 4 + … + 1 = 21 total comparisons for quicksort on array *A*.

1. In the first partition of the entire array of length *n*, we need *n* – 1 total comparisons.

Suppose the returned value is *q*. Then, we need to partition the subarrays *A*[1…*q* – 1] and *A*[*q* + 1…*n*] respectively.

*A*[1…*q* – 1] requires *q* – 2 comparisons if *q* – 1 ≥ 1 (*q* ≥ 2), or no comparison if *q* < 2.

[*q* + 1…n] requires *n* – *q* – 1 comparisons if *q* + 1 ≤ *n* (*q* ≤ *n* – 1), or no comparison if *q* > *n* – 1.

Therefore, if *q* < 2, there should be *n* – *q* – 1 comparisons for the two subarrays.

If 2 ≤ *q* ≤ *n* – 1, there should be (*q* – 2) + (*n* – *q* – 1) = *n* – 3 comparisons.

If *q* > *n* – 1, there should be *q* – 2 comparisons.

Clearly, *n* – *q* – 1 > *n* – 3 if *q* < 2. *q* – 2 > *n* – 3 if *q* > *n* – 1.

To sum up, a returned value *q* within [2, *n* – 1] would offer the least number of comparisons (i.e., *n* – 3) for the next round.

So, for an array of length 7, the first partition should conduct 6 comparisons and return a value within [2, 6] to minimize the number of future comparisons.

*A*[1…*q* – 1] and *A*[*q* + 1…7] should have 7 – 3 = 4 comparisons in total.

For partitioning *A*[1…*q* – 1], we expect a returned value within [2, *q* – 2] (thus, *q* ≥ 4) to gain the least number of comparisons (i.e., *q* – 4) for next level.

For partitioning *A*[*q* + 1…7], we expect a returned value within [*q* + 2, 6] (thus, *q* ≤ 4) to gain the least number of comparisons (i.e., 4 – q) for next level.

Therefore, picking *q* = 4 would achieve the minimum number of comparisons, which is 6 + 4 + (*q* – 4) + (4 – *q*) = 10.

The exact process is: First, partitioning on *A* requires 6 comparisons and returns 4, which divides *A* into *A*[1…3] and *A*[5…7]. Then, partitioning on *A*[1…3] requires 2 comparisons and returns 2. Partitioning on *A*[5…7] also requires 2 comparisons and returns 6. No more comparison is needed. The minimum number of total comparisons is 10.

1. To get the minimum number of total comparisons:

Since *n* = 2*k* – 1, the first partition on the entire array would require 2*k* – 2 comparisons.

As shown in (c), the optimal way for the next level partitioning would be to choose the middle point of the array as the return value.

Therefore, the array is further divided into two subarrays of length .

Similarly, each of these two subarrays require 2k-1 – 2 comparisons, which sums upto 2(2*k*-1 – 2) for Round 2.

These two subarrays are further divided into 4 subarrays of length in Round 3.

…

We can conclude that the minimum number of total comparisons is equal to:

.

For all

So .

Therefore, we can pick *c* = 2 and *n*0 = 1 to show that the running time of Quicksort is in the best case.

To get the maximum number of total comparisons:

Suppose the array is already sorted like (b), then in each round we always have to handle a subarray whose length is one element shorter than the parent subarray (e.g., [1, 2, 3, 4, 5] => [1, 2, 3, 4] => [1, 2, 3] …).

Since each round requires *n* – 1 comparisons (Here, *n* refers to the length of subarray), the maximum number of total comparisons is equal to:

.

For all

So .

Therefore, we can pick *c* = and *n*0 = 1 to show that the running time of Quicksort is in the worst case.

Problem #3

1. The algorithm to solve this problem in 7 matches is:

In Round 1, we divide 8 students into 4 pairs, and let each pair have a chess match, creating 4 matches so far (i.e., A vs B, C vs D, E vs F, G vs H).

In Round 2, we have 4 winners from Round 1, and we divide them into 2 pairs and create 2 more matches.

In Round 3, we have 2 winners from Round 2 and only one more match suffices to find the best player.

The minimum number of matches is 4 + 2 + 1 = 7.

It’s not possible to solve this problem in 6 or fewer matches because:

In order to ensure that the highest-rated player is determined, we need to have enough matches to eliminate the rest of the players, which has a number of 7. If there are 6 or fewer matches, we can only eliminate 6 or fewer players, making us unable to identify the highest-rated player.

1. In terms of the ranking of 8 players, there are 8! permutations.

Consider all the permutations are included in a balanced decision tree, where each leaf node represents a permutation.

According to Problem 1(a), a binary tree of height *h* can have at most 2*h* nodes on the last level.

Therefore, the height of this decision tree should be larger than so that all permutations are considered, while the smallest integer bigger than is 16.

It represents that in order to reach any leaf node of this decision tree, there should be at least 16 comparisons, so it’s impossible to solve this problem in 15 or fewer comparisons.

1. The algorithm to solve this problem in 7 matches is:

In Round 1, we divide 8 students into 4 pairs, and let each pair have a chess match, creating 4 matches so far (i.e., A vs B, C vs D, E vs F, G vs H).

In Round 2, we have 4 winners from Round 1, and we divide them into 2 pairs and create 2 more matches. We can also determine 4 losers from Round 1, and similarly we divide them into 2 pairs and create 2 more matches.

In Round 3, we have 2 winners from Round 2 and only one more match suffices to find the best player. We also have 2 losers from Round 2, and it also takes one more match to find the lowest-rated player.

The minimum number of matches is 4 + 2 + 2 + 1 + 1 = 10.

It’s not possible to solve this problem in 9 or fewer matches because:

By using a tournament-style approach, our algorithm in (a) is guaranteed to find the highest-rated player in 7 matches. However, this problem also requires us to find the lowest-rated player. If we adopt a similar approach to (a) and eliminate the lowest-rated player from the beginning, it will require another 7 matches, which adds up to 14 matches in total. However, note that there will be repeated matches in Round 1, because either finding the highest-rated player or finding the lowest-rated player needs 4 matches in Round 1 to eliminate half of them, and it can precisely tell us what the other four players are. Therefore, we can reduce 14 matches to 10 matches in this algorithm. There is no other way to further reduce the number of total matches, because in the following rounds, finding the highest-rated player in a group of 4, or finding the lowest-rated player in another group of 4 (Note that these two groups don’t overlap anymore) both requires at least 3 matches to eliminate the rest of the candidates, so the total number of matches is 10 and it’s impossible to solve the problem using fewer matches.

1. The algorithm to solve this problem in 9 matches is:

First, we follow the algorithm given in (a) to find the best player using 7 matches. Suppose Player 1 is the best player and tournament results are illustrated by the graph below:

A diagram of a network

Description automatically generatedThen, we can find that only Players 2, 3, and 5 lost to Player 1, and all the other players proved to be less strong than them, therefore impossible to be the second-best player. Then, to find the best player among Player 2, 3, and 5, two more matches are needed (Choose any two to have the first match and let the winner compete with the third player).

Therefore, it is guaranteed to solve this problem in 7 + 2 = 9 matches.

It’s not possible to solve this problem in 8 or fewer matches because:

By using a tournament-style approach, our algorithm in (a) is guaranteed to find the highest-rated player in 7 matches. However, this problem also requires us to find the second highest-rated player. If we are using 8 or fewer matches, it only allows us at most one more match to find the second-best player. As discussed previously, there are 3 candidates for the second-best player, so one more match is not enough to eliminate two of them or find the highest rated among them. Two more matches are necessary to guarantee the second highest-rated player, which is why the algorithm above requires 9 total matches, instead of 8 or fewer.

Problem #4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Lomuto quicksort | Hoare quicksort | Heapsort |
| [2, 8, 7, 1, 3, 5, 6, 4] | Total comparisons | 15 | 16 | 25 |
| Total swaps | 14 | 8 | 18 |
| Running time \* 1000 | 17.17 | 14.07 | 16.93 |
| [1, 2, 3, 4, 5, 6, 7, 8] | Total comparisons | 28 | 28 | 27 |
| Total swaps | 35 | 0 | 22 |
| Running time \* 1000 | 5.01 | 2.86 | 6.91 |
| [8, 7, 6, 5, 4, 3, 2, 1] | Total comparisons | 28 | 24 | 24 |
| Total swaps | 19 | 4 | 16 |
| Running time \* 1000 | 5.01 | 4.05 | 5.25 |
| [8, 5, 3, 4, 2, 6, 1, 7] | Total comparisons | 19 | 23 | 25 |
| Total swaps | 14 | 3 | 19 |
| Running time \* 1000 | 2.86 | 3.10 | 5.01 |
| random integer array of size 100,000 | Total comparisons | 6315451 | 952411 | 3018773 |
| Total swaps | 715053 | 548397 | 1574147 |
| Running time \* 1000 | 411780.12 | 142091.04 | 385272.03 |

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated