1. (a) A naive implementation requires around $2n$ comparisons. The following procedure is better. We will do comparison only between adjacent pairs (so $n/2$ comparisons). The larger numbers go to the next round where the same procedure is repeated. Thus in total $\sum_{i=1}^{\log n} n/2^i = n - 1$ comparisons we can find the largest element. Now the second largest element must have been compared directly with the largest element. So it is only one of $\log n$ possible elements (which we can keep track of), and at the end we can, in $\log n - 1$ comparisons, find the largest among these, thus giving us the second largest element. The number of comparisons is $n + \log n - 1$.

(b) The following is from the DPV textbook: Tree edges are part of the DFS traversal. Forward edges go from a node to a non-child descendant. Back edges lead to an ancestor in the DFS tree. Finally, Cross edges lead to nodes that have already been completely explored.

![Figure 1: The types of edges when the graph is traversed in the order 1, 2, 3, 4.](image)

2. (a) Merge the first $k$ elements of $A$ and $B$ and pick the $k$th element in the merged array. This is an $O(k)$ algorithm.

(b) Compare $A[k/2]$ and $B[k/2]$. If they are equal then they are the $k$th smallest. If $A[k/2] > B[k/2]$, then we recurse on $A[1 \ldots k/2]$ and $B[k/2 + 1 \ldots k]$. This is because the $k$th smallest cannot lie in the range $A[k/2 + 1 \ldots k]$, since they are larger than at least $k$ elements. Neither can it lie in the range $B[1 \ldots k/2]$, since they are smaller than at least $k$ elements. By a similar argument, if $A[k/2] < B[k/2]$, then we recurse on $A[k/2 + 1 \ldots k]$ and $B[1 \ldots k/2]$.

3. As in the hint, we will randomly pick a nut and partition the bolts according to this nut. On average, since the nut was chosen randomly, the partitioning procedure will produce two parts of almost equal size. After $n$ comparisons, we have found the matching bolt for this nut. This matching bolt is also uniformly random among bolts and can now be used to partition the nuts. Now we recurse on the left part of the nuts and bolts and on the right parts. By an analysis similar to that of randomized quicksort, we will take $O(n \log n)$ comparisons in expectation.

4. First we will convert tree $T$ into a rooted tree with an arbitrary root $r$. Every node $u$ now knows its children $C(u)$ in the rooted tree. This can be done in linear time.
Let us define the quantity $M(u)$ as the maximum weight matching in the subtree rooted at node $u$. Then the following recurrence calculates the answer when run on $M(r)$:

$$M(u) = \begin{cases} 
\max \left\{ \sum_{v \in C(u)} M(v), \max_{v \in C(u)} \left( w(u, v) + \sum_{w \in C(v)} M(w) + \sum_{v' \in C(u) \setminus \{v\}} M(v') \right) \right\} & \text{if } C(u) \neq \phi, \\
0 & \text{otherwise.}
\end{cases}$$

In words, we have two choices to calculate $M(u)$, either not to select any edge from $u$ or to select an edge to one child as being in the matching. The running time is linear in the number of vertices.

5. Let us try the second approach. We start with calculating the min-cut with constant probability. This takes $O(n^4)$ time. The second smallest min-cut has to differ from the smallest min-cut in at least one edge, say $e$. So if we contract $e$ and recalculate the min-cut (now on a graph with $n - 1$ vertices), we will encounter the second-smallest min-cut with constant probability. Since we don’t know which edge differs, we iterate over every edge and take the min-cut. The total time taken will be $O(n^5)$. 