ListT5: Listwise Reranking with Fusion-in-Decoder Improves Zero-shot Retrieval

Soyoung Yoon[1*] **Eunbi Choi**[2] **Jiyeon Kim**[3] **Hyeongu Yun**[2]
Yireun Kim[2] **Seung-won Hwang**[1†]
[1]Seoul National University [2]LG AI Research [3]KAIST AI
{soyoung.yoon, seungwonh}@snu.ac.kr

Abstract

We propose ListT5, a novel reranking approach based on Fusion-in-Decoder (FiD) that handles multiple candidate passages at both train and inference time. We also introduce an efficient inference framework for listwise ranking based on m-ary tournament sort with output caching. We evaluate and compare our model on the BEIR benchmark for zero-shot retrieval task, demonstrating that ListT5 (1) outperforms the state-of-the-art RankT5 baseline with a notable +1.3 gain in the average NDCG@10 score, (2) has an efficiency comparable to pointwise ranking models and surpasses the efficiency of previous listwise ranking models, and (3) overcomes the lost-in-the-middle problem of previous listwise rerankers. Our code, model checkpoints, and the evaluation framework are fully open-sourced at https://github.com/soyoung97/ListT5.

1 Introduction

Recent advancements on neural information retrievers have made significant progress in their semantic search capabilities. However, they still struggle in zero-shot or out-domain tasks where statistical retrievers such as BM25 (Robertson and Zaragoza, 2009) often outperform, a crucial challenge since the setting is closely related to real-world scenarios.

Until now, the field of zero-shot reranking has been generally driven by cross-encoder models (Rosa et al., 2022) such as MonoT5 (Nogueira et al., 2020) or RankT5 (Zhuang et al., 2022). These models rely on *pointwise* reranking of each passage, and thus lacks the ability to compare between passages relatively at inference time. This could lead to a suboptimal solution in the task of reranking, where discrimination and ordering between passages are crucial.

Recently, *listwise* reranking models (Ma et al., 2023; Sun et al., 2023b,a), which evaluates multiple passages together, is gaining attention for its effectiveness in zero-shot retrieval. Listwise rerankers can condition on and compare multiple passages to calibrate the relevance scores better, and can reduce the inaccuracy of zero-shot predictions arising from domain shift, as theoretically supported by Xian et al. (2023), and empirically evidenced by a line of works such as Ma et al. (2023); Sun et al. (2023a).

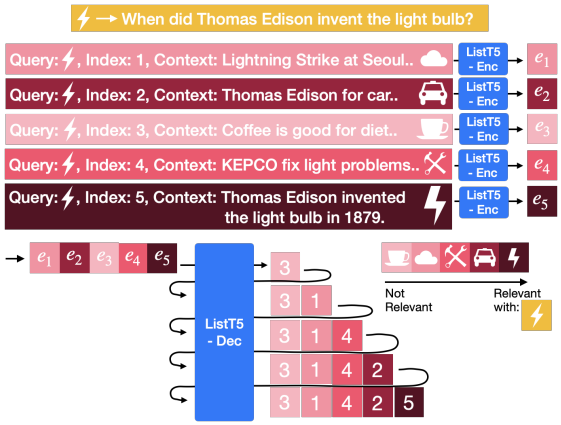

Figure 1: Operating unit of ListT5. ListT5 jointly considers multiple (5) candidate passages at once using FiD, each concatenated with the query and an identifier. The output is an ordered list of the identifiers (numbers) where the most relevant passage comes at the *last*.

However, existing approaches that attempt listwise reranking have limitations. DuoT5 (Pradeep et al., 2021) implements pairwise reranking, a form of listwise reranking as pair, which incurs quadratic time complexity to the number of candidate passages. On the other hand, listwise reranking using Large Language Models (LLMs) (Pradeep et al., 2023a; Sun et al., 2023b,a; Ma et al., 2023) sacrifice efficiency in another aspect due to large parametric model size, and also face the "Lost in the middle" problem (Liu et al., 2023; Tang et al., 2023).

In this work, we present ListT5 (Fig. 1), a novel listwise approach for zero-shot reranking that overcomes the aforementioned limitations. ListT5 jointly considers the relevancy of multiple candi-

*Work done during an internship at LG AI Research.
†Corresponding author.

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 2287–2308

date passages at both training and inference time based on Fusion-in-Decoder (FiD) (Izacard and Grave, 2021) architecture. It outputs a sorted list of input passages in the *increasing* order of relevance, and employs a novel *tournament tree* structure at inference time for efficient sorting. With these, LISTT5 provides the following contributions:

(1) Computational Efficiency. LISTT5 improves efficiency over previous methods in two aspects. First, for reranking top-k passages given n candidates, LISTT5 achieves $O(n + k \log n)$ asymptotic cost, lower than $O(n^2)$ of pairwise methods and comparable to $O(n)$ of pointwise methods (Sec. 3.2). We empirically demonstrate such efficiency through FLOPs analysis (Sec. 4.4). Second, while previous listwise reranking methods relied on LLMs to process long listwise inputs, we show that adopting FiD removes the dependency on LLMs and allows us to perform listwise reranking with smaller and more parameter efficient architectures, such as T5-base (Tab. 3).

(2) Robustness to Positional Bias. In addition to efficiency, we show that LISTT5 overcomes the "Lost in the middle" problem commonly encountered in LLM-based listwise rerankers (Liu et al., 2023) which tend to be positionally biased to passages presented in the first and last parts of the listwise input (Sec. 4.5) and are more robust to the change in initial ordering of the passage (Sec. 4.6). We attribute the robustness of LISTT5 to such bias to the nature of FiD, which distinguishes each input passage with identifiers, instead of positions as in LLMs.

(3) Zero-shot Performance. On top of efficiency and robustness, listwise reranking with LISTT5 demonstrates strong performance in zero-shot retrieval, compared to not only the pointwise and pairwise methods, but also listwise methods based on LLMs that are argued to be specialized for zero-shot (Pradeep et al., 2023a,b). Specifically, on the BEIR benchmark for zero-shot retrieval, our approach using T5-3B surpasses state-of-the-art methods including pointwise RankT5 (+1.4 gain in NDCG@10) (Tab. 2) and listwise RankZephyr (+1.4 gain in NDCG@10) (Tab. 3). We further perform a comprehensive ablation study and find that our key components, such as tournament sort (App. J) and generating in the increasing order of relevance, benefit zero-shot retrieval (Sec. 4.7).

2 Related Work

2.1 Generative Models for Reranking

In the reranking scenario, rather than dual encoder models (Karpukhin et al., 2020) which separately encode query and passage information, models that see query and passage information jointly at inference time (Reimers and Gurevych, 2019; Nogueira et al., 2020) are shown to be effective for zero-shot retrieval (Rosa et al., 2022). Among those, formulating reranking as sequence generation, such as conducting listwise sorting (Ma et al., 2023; Sun et al., 2023b; Pradeep et al., 2023a) or generating rationales (Ferraretto et al., 2023), has shown an advantage in application to zero-shot retrieval by leveraging the language model's auto-regressive generation capabilities. Specifically, a series of studies that use the encoder-decoder architecture of T5 (Sec. 2.2), and applying zero-shot reranking with LLMs (Sec. 2.3), or viewing reranking as autoregressive text generation problem (Wang et al., 2024) has been successful.

2.2 Listwise Reranking for T5

MonoT5 (Nogueira et al., 2020) leaverages predefined token probabilities (i.e., true/false) as relevance scores at inference time. RankT5 (Zhuang et al., 2022) introduces listwise training loss but both MonoT5 and RankT5 performs pointwise reranking at inference time. Built on top of MonoT5, DuoT5 (Pradeep et al., 2021), implements pairwise reranking, showing superior performance. However, DuoT5 only sees two passages at each prediction, and is inefficient as it has to run prediction on every pair of candidate passages (thus requiring $n^2 - n$ predictions to rerank n passages).

2.3 Listwise Reranking with LLMs

A line of work (Sun et al., 2023b; Pradeep et al., 2023a; Ma et al., 2023; Pradeep et al., 2023b; Qin et al., 2023) implements listwise reranking to LLMs, usually in a format of taking multiple (20) passages as input with the sliding window approach. However, this format presents a challenge due to the monolithic and lengthy input size. The model must process all tokens at once, limiting its applicability to LLMs trained with large context lengths. The lengthy input size also leads to the "lost in the middle" problem (Liu et al., 2023) for LLMs, exhibiting strong positional bias to the information in the first and last parts of long inputs, failing to comprehend relevant information

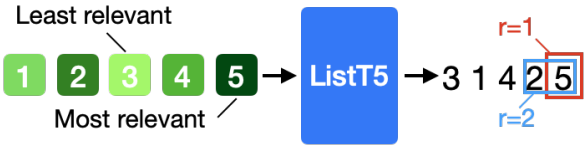

Figure 2: Two variants of ListT5, (r=1) and (r=2). The underlying model is identical and only the inference method varies. (r=1) keeps only the top 1 relevant index from the output, and (r=2) keeps top 2 relevant indices.

in the middle. This problem is also prevalent in listwise reranking (Tang et al., 2023). Furthermore, this approach comes at the cost of sacrificing computational efficiency in the already computation-intensive reranking scenarios.

2.4 Listwise Reranking with m-ary Units

Ai et al. (2019) is in a similar spirit to ours, as they define a basic ranking unit on m elements and extends it to rank the list of $n(\gg m)$ elements. However, while they consider an intractable number of all $m!$ orderings of all m-sized subsets and then employ Monte Carlo sampling for approximation, we use m-ary tournament sort algorithm which is complete and efficient by construction (Sec. 3.2).

3 Methods

Our method ListT5 for listwise reranking has two components: **(1)** the basic operating unit that takes a fixed number of $\boldsymbol{m}$ passages and ranks the top-$\boldsymbol{r}$ based on FiD, and **(2)** an extension of this basic unit that takes full $\boldsymbol{n}$ passages and ranks the top-$\boldsymbol{k}$ based on tournament sort. We describe the basic unit in Sec. 3.1 and its extension in Sec. 3.2.

3.1 Basic Operating Unit ($m \rightarrow r$)

The basic operating unit of ListT5 processes a fixed number of m input passages and outputs top-r passages ($m \rightarrow r$), as in Fig. 2. We develop the basic unit upon the Fusion-in-Decoder (FiD) architecture (Izacard and Grave, 2021), which modifies the Encoder-Decoder structure of T5 (Raffel et al., 2020) to process listwise inputs efficiently. Specifically, given a query $\mathbf{q}$ and a list of m candidate passages $[\mathbf{p}_1, ..., \mathbf{p}_m]$, our basic unit concatenates each passage $\mathbf{p}_i$ with its index identifier i and the query $\mathbf{q}$ in the following format, and then feeds them to Encoder to obtain their full sequence token embedding $\mathbf{h}_i$:

$$\mathbf{h}_i = \text{Enc}(\text{"Question: } \mathbf{q}\text{, Index: } i\text{, Context: } \mathbf{p}_i\text{"}) \quad (1)$$

In Eq. (1), notice that each passage $\mathbf{p}_i$ is encoded separately. Listwise reasoning upon multiple passages $\mathbf{p}_1, ..., \mathbf{p}_m$ is handled by the decoder. Specifically, the encoded representations $[\mathbf{h}_1, ..., \mathbf{h}_m]$ are concatenated to a single sequence and fed to the decoder, which performs listwise ranking and autoregressively generates an ordered sequence of the passage indices (identifiers) $[i'_1, ..., i'_m]$ in the *increasing* order of relevance:

$$[i'_1, ..., i'_m] = \text{Dec}(\text{concat}[\mathbf{h}_1, ..., \mathbf{h}_m]),$$
$$\text{where } \text{rel}(\mathbf{q}, \mathbf{p}_{i'_1}) < ... < \text{rel}(\mathbf{q}, \mathbf{p}_{i'_m}) \quad (2)$$

Importantly, in Eq. (2), notice that the decoder generates output passages from the *least* relevant one i'_1 to the *most* relevant one i'_m. This is different from previous listwise rerankers (Sec. 2.3), which generate from the most relevant passage index. Our rationale is that generating from the least relevant to the most relevant can be beneficial, as it may act a reasoning chain that progressively eliminates irrelevant passages to deduce the most relevant passages. One can imagine solving a confusing multiple-choice question; crossing out the options that are definitely not the answer until the last remaining option serves as a good strategy.

After the decoder ranks all m input passages, we decide the number r of relevant passages we will keep as the output of the basic operating unit. Namely, **ListT5 (r = 1)** keeps one, so only $[\mathbf{p}_{i'_m}]$ is selected as the output. **ListT5 (r = 2)** keeps two, selecting $[\mathbf{p}_{i'_m}, \mathbf{p}_{i'_{m-1}}]$ as the output. Larger choices of r are possible in principle, but since we have observed saturation of performance for $r > 2$ in early trials, we only use the r = 1 and r = 2 variants of the basic unit in our experiments. For m, we set its value to 5 in our experiments. Ablations on the choice of m can be found at App. B.1.

3.2 Extension with Tournament Sort ($n \rightarrow k$)

While the basic operating unit of ListT5 (Sec. 3.1) reranks a fixed number of $m\,(= 5)$ passages, our end goal is to rerank k passages given a much larger number of $n\,(\gg m)$ candidate passages. This necessitates an algorithm to extend the basic unit ($m \rightarrow r$) to full reranking ($n \rightarrow k$). For the similar purpose, previous listwise rerankers (Sec. 2.3) has mostly used the sliding window algorithm. In these, an operating unit $m \rightarrow r$, typically an LLM, is slided over the n candidate passages, and at each step m passages are locally reranked and reordered. After the sliding is done, the top k passages are used as the output of global reranking.

However, we observe that sliding window has a drawback. Since a window of size m can only "cache" up to m passages, full reranking $n \rightarrow k$ is bound to be inaccurate when k is set to be $> m$. In such cases, it is necessary to run the whole sliding over the entire n passages multiple times (*e.g.*, $\lceil \frac{k}{m} \rceil$ times). Each sliding run would cost $\mathcal{O}(n)$ asymptotically, which can be computationally demanding given $n \gg m$. Therefore, for ListT5, we opt into developing a novel algorithm that does not require multiple evaluations over the entire n passages.

Our inference algorithm for ListT5 that extends the basic unit ($m \rightarrow r$) to full reranking ($n \rightarrow k$) is inspired by the *tournament sort* algorithm (McLuckie and Barber, 1986) (App. C.1). Given n inputs, tournament sort ranks top-k by **(1)** constructing a binary tournament tree of n inputs where the root is the top-1 element, and **(2)** running k iterations of extracting the root and re-doing the tournament with the remaining inputs. Here, a crucial property is that once the tournament tree has been constructed, re-doing the tournament only requires recomputing a *single* path that traverses from a leaf to the root. As a result, most of the nodes and edges can be cached and reused over the k iterations, and each iteration only costs $\mathcal{O}(\log_2 n)$ asymptotically. Unlike sliding window, multiple evaluations over the entire n inputs is avoided.

Based on tournament sort, our inference algorithm $n \rightarrow k$ for ListT5 is defined by **(1)** extending from binary to m-ary tournament trees, and **(2)** invoking the basic unit $m \rightarrow r$ (Sec. 3.1) at each node of the tournament tree. Specifically, given an arbitrary number of n passages, we perform a bottom-up comparison using the basic unit to construct an m-ary tournament tree. Then, we run k iterations of extracting the top-1 passage at the root and re-doing the tournament by recomputing a single path from the replaced leaf to root. In this way, the model can rank top-k passages from an arbitrary number of n passages while inheriting the efficiency of tournament sort. Fig. 3 illustrates our inference algorithm using ListT5 (r = 1) as the basic unit. Further details can be found at App. C.

Asymptotic Complexity. We now present an asymptotic analysis of the cost of listwise reranking $n \rightarrow k$ with ListT5 using a basic unit $m \rightarrow r$. At the first iteration of tournament sort, we need to construct the full tournament tree, which amounts to $n(\frac{1}{m} + \frac{r}{m^2} + \frac{r^2}{m^3} + ...) = \frac{n}{m}\frac{m}{m-r}$ evaluations which is $\mathcal{O}(n)$. In each iteration afterwards, we only need to recompute a single path from the leaf to root to compute the top-1, resulting in an asymptotic cost of $\mathcal{O}(\log_m n)$. With k iterations, ListT5 achieves $\mathcal{O}(n + k \log_m n)$ asymptotic cost for reranking top-k from n candidate passages. If we compare this complexity with others (Tab. 1), we can see that ListT5 is more efficient than pairwise reranking models of $\mathcal{O}(n^2)$ (Pradeep et al., 2021), and comparable to pointwise models of $\mathcal{O}(n)$ (Nogueira et al., 2020; Zhuang et al., 2022).

Method Name	Reranking Method	Complexity
MonoT5 (Nogueira et al., 2020), RankT5 (Zhuang et al., 2022)	Pointwise	$\mathcal{O}(n)$
DuoT5 (Pradeep et al., 2021)	Pairwise	$\mathcal{O}(n^2)$
ListT5 (Ours)	Listwise	$\mathcal{O}(n + k \log n)$

Table 1: Comparison of the computational complexity of different ranking methods on obtaining top-k passages from n candidate passages. The base of the log function is m. ListT5 achieves $\mathcal{O}(n + k \log n)$, or $\mathcal{O}(n + n \log n)$ when $n = k$, which is way better than pairwise models, competitive with pointwise models, and is in fact the best possible (worst-case asymptotic) complexity for sorting algorithms based on comparisons (Ren et al., 2018)[1].

4 Experiments

4.1 Overview

In this section, after explaining the training and evaluation details, we present three main results in the following order: (1) General performance and efficiency comparison of our models with respect to pointwise and listwise ranking models at Sec. 4.4, (2) Analyzing the lost-in-the-middle problem by measuring the robustness to positional bias between ListT5 and listwise ranking models at Sec. 4.5, and (3) Ablation experiments on the design choice at Sec. 4.7. All results are based on the T5-base model except explicitly mentioning 3B.

4.2 Training

To train ListT5, we use the official *train* subset of the MS MARCO (Bajaj et al., 2018) passage ranking dataset.[2] The dataset consists of 532,761 distinct queries and 8.8M passages, with binary annotations of relevancy. Since the ordering of rel-

[1] https://en.wikipedia.org/wiki/Comparison_sort

[2] https://microsoft.github.io/msmarco/Datasets

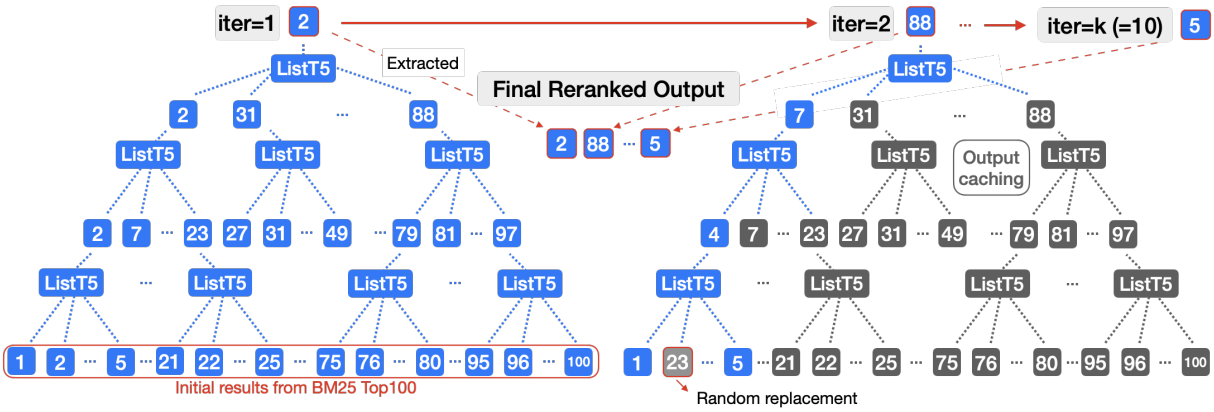

Figure 3: Illustration of our inference framework using m-ary tournament sort, with ListT5 ($r = 1$) as the basic unit. Given n candidates, we can order top-k most relevant passages in $O(n + k \log n)$ asymptotic complexity. We can use either ($r = 1$) or ($r = 2$) for the basic unit, but the uppermost unit always outputs 1 ($r = 1$). We fix m to 5 in our experiments. Full illustration at Appendix Fig.7.

	BM25 Top-100							BM25 Top-1000		
	Initial	MonoT5 -base	RankT5 -base	ListT5 -base (r=2)	MonoT5 -3B	RankT5 -3B	ListT5 -3B (r=2)	MonoT5 -base	RankT5 -base	ListT5 -base (r=2)
TREC-COVID	59.5	**78.3**	77.7	**78.3**	79.8	81.7	**84.7**	78.3	79.1	**82.1**
NFCorpus	32.2	**35.7**	35.1	35.6	37.3	**37.4**	37.7	**36.1**	35.3	**36.1**
BioASQ	52.2	55.3	**58.2**	56.4	57.5	**58.3**	**58.3**	52.6	**57.6**	55.0
NQ	30.5	52.1	**53.2**	53.1	56.4	**57.8**	56.2	55.9	**57.6**	57.5
HotpotQA	63.3	71.2	**72.8**	72.6	74.3	74.8	**75.6**	70.9	**73.8**	73.6
FiQA-2018	23.6	39.2	39.2	**39.6**	**46.0**	45.2	45.1	41.2	41.1	**41.8**
Signal-1M (RT)	33.0	32.0	30.8	**33.5**	32.2	31.9	**33.8**	29.3	28.6	**30.9**
TREC-NEWS	39.5	48.0	45.4	48.5	48.3	49.5	**53.2**	47.8	45.9	**50.9**
Robust04	40.7	53.4	**54.3**	52.1	**58.5**	58.3	57.8	55.4	**57.2**	54.7
Arguana	40.8	34.4	35.5	**48.9**	46.8	37.4	**50.6**	24.2	26.6	**46.9**
Touche-2020	44.2	29.6	**37.1**	33.4	32.5	**38.8**	33.6	26.4	**37.0**	31.5
CQADupStack	30.0	38.6	37.0	**38.8**	41.3	40.3	**42.1**	40.1	38.1	**40.5**
Quora	78.9	84.6	83.3	**86.4**	84.0	83.6	**86.9**	84.2	82.9	**86.4**
DBPedia	31.8	42.8	43.7	43.7	44.8	**45.0**	46.2	43.1	**45.1**	44.9
SCIDOCS	14.9	16.7	16.8	**17.6**	19.0	18.9	**19.5**	17.0	17.1	**18.0**
FEVER	65.2	78.4	77.6	**79.8**	80.0	79.8	**82.0**	77.9	77.8	**81.0**
Climate-FEVER	16.5	23.1	21.2	**24.0**	**26.2**	24.5	24.8	23.3	20.6	**24.9**
SciFact	67.9	73.1	73.5	**74.1**	76.3	**77.1**	77.0	73.3	73.6	**74.9**
Average	42.5	49.3	49.6	**50.9**	52.3	52.2	**53.6**	48.7	49.7	**51.8**

Table 2: Comparison of ListT5 against **pointwise** ranking models, on BEIR, in NDCG@10. MS MARCO-Top1000 results for 3B models are omitted due to their long inference time. ListT5-base excels in both performance and model size (than MonoT5-3B or RankT5-3B) for datasets in red.

evancy between negative passages is not provided, we use a dual-encoder retrieval model, specifically COCO-DR large (Yu et al., 2022), to retrieve Top-1000 passages and label relevance scores of negatives for the training dataset of MS MARCO. Experiments using GTR (Ni et al., 2022), the same model used to build the training data for Zhuang et al. (2022), are reported at Appendix. B.2. For each query-positive passage pair, we randomly sample 20 groups of $m - 1$ distinct negative passages. We randomly shuffle the positive and negative passages and assign identifiers $\{1, ..., m\}$ to them to form each training data.

We train T5 with two different model sizes: base and 3B. We conduct grid search (App. F) on the subset of BEIR on the learning rate and number of steps in a reasonable range and report the best scores. As a result, we report the T5-base model

	TREC-DL19	TREC-DL20	TREC-COVID	NFC-orpus	Signal-1M (RT)	TREC-NEWS	Robu-st 04	Touche-2020	DBP-edia	Sci-Fact	Avg (In-domain)	Avg (BeIR)
DuoT5-base	71.4	67.4	**80.1**	35.0	31.4	**49.1**	49.6	31.8	**43.9**	69.6	69.4	48.8
ListT5-base ($r = 2$)	**71.8**	**68.1**	78.3	**35.6**	**33.5**	48.5	**52.1**	**33.4**	43.7	**74.1**	**70.0**	**49.9**
RankGPT (GPT3.5)	65.8	62.9	76.7	35.6	32.1	48.9	50.6	**36.2**	44.5	70.4	64.4	49.4
RankVicuna-7b	68.9	66.1	80.5	33.2	**34.2**	46.9	48.9	33.0	44.4	70.8	67.5	49.0
RankZephyr-7b	**73.9**	**70.9**	84.0	36.7	31.8	52.6	54.3	33.8	44.6	74.9	**72.4**	51.6
ListT5-3B ($r = 2$)	71.8	69.1	**84.7**	**37.7**	33.8	**53.2**	**57.8**	33.6	**46.2**	**77.0**	70.5	**53.0**

Table 3: Comparison of ListT5 against **listwise** (RankGPT, RankVicuna, RankZephyr) and **pairwise** (DuoT5) ranking models, in NDCG@10. Initial results are from BM25 Top100. The scores of RankGPT (with GPT-3.5-turbo-0301) are from the RankGPT paper, and the best scores are in bold. DuoT5 is applied on top 50 passages reranked by MonoT5. ListT5 excels the pairwise counterparts as well as the previous listwise ranking models on the selected subset of BEIR.

trained for 20k steps with a learning rate of 1×10^{-4} and T5-3B for 3k steps with a learning rate of 1×10^{-5}, both with linear learning rate scheduler with 10% warmup and an effective batch size of 256 in bfloat16 precision. Maximum input length is set to 230 and 128, and the maximum output length to 7 for T5-base and T5-3B, respectively. With DeepSpeed stage 2 optimization, training T5-base took approximately 8 hours on 4×NVIDIA RTX A6000 GPUs with 48GB, while T5-3B training took 3 hours on 8×NVIDIA A100 GPUs with 40GB.

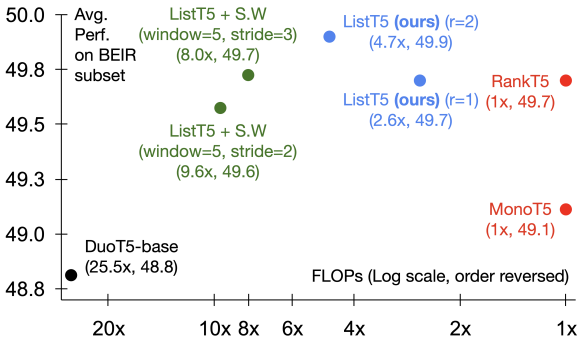

Figure 4: Real-time FLOPs comparison of the models on T5-base, including DuoT5 and the sliding window variants of ListT5. The reported BEIR performance is averaged from a subset of BEIR, same as in Tab. 3.

4.3 Evaluation

We measure the evaluation performance on the official dev set of MS MARCO (passage ranking dataset), TREC-DL19, and TREC-DL20 (Craswell et al., 2021) for in-domain performance, and the BEIR benchmark (Thakur et al., 2021) for zero-shot out-domain performance. BEIR contains 18 diverse sets of domains and tasks. We download the dumped index of the Top-100 and Top-1000 retrieved passages by BM25 from the official Pyserini repository. Reranking results using advanced first-stage retrieval models (e.g., COCO-DR) are reported at App. D.1. We use the evaluation metric of the Normalized Discounted Cumulative Gain@10 (NDCG@10) (Järvelin and Kekäläinen, 2002), the official evaluation metric for BEIR (Thakur et al., 2021). All reported scores are rounded to the nearest tenth. We also report the Mean Reciprocal Rank@10 (MRR@10) scores at App. Tab. 11. For reproducibility, we include the evaluation details including details to run baseline models and links for dataset download at App. G.

4.4 Zero-shot performance and efficiency.

Performance. We measure and compare the performance of ListT5 against *pointwise* ranking models (MonoT5 (Nogueira et al., 2020), RankT5 (Zhuang et al., 2022)) in Tab. 2, and against *pairwise and listwise* ranking models (DuoT5 (Pradeep et al., 2021), RankVicuna (Pradeep et al., 2023a), RankZephyr (Pradeep et al., 2023b)) in Tab. 3. ListT5 ($r = 2$) achieves an average of +1.3 point gain on NDCG@10 than RankT5 for reranking on BM25 Top-100. Also, the gain from listwise reranking improves even further when we rerank from BM25 Top-100 to Top-1000. While pointwise models show small performance difference from BM25 Top-100, ListT5 improves from 50.9 to 51.8, additional 0.9 % gain from BM25 Top-100. Additionally, ListT5 also excels on 3B model variants, outperforming RankVicuna-7b and RankZephyr-7b on the selected subset of BEIR. We additionally provide qualitative analysis at App. I.

Efficiency in FLOPs. In addition to comparing the

	TREC-COVID							FiQA						
	Accuracy when positive passage is at index # :						Aggrement ratio (↑)	Accuracy when positive passage is at index #:						Aggrement ratio (↑)
	1	2	3	4	5	Std. (↓)		1	2	3	4	5	Std. (↓)	
GPT-3.5	81.6	63.3	75.5	67.3	61.2	7.7	55.1	88.3	68.1	78.7	65.9	75.8	8.0	62.1
GPT-4	95.9	83.7	73.5	77.6	71.4	8.8	69.4	94.6	90.5	84.4	86.8	84.8	3.9	82.8
DuoT5	91.3	76.0	-	-	-	7.6	79.6	89.9	76.9	-	-	-	6.5	78.1
ListT5	93.9	87.8	83.7	85.7	81.6	**4.2**	**83.7**	85.3	85.6	82.2	83.3	82.6	**1.4**	**90.4**

Table 4: Robustness to the position of the positive passage in the input, on TREC-COVID and FiQA. GPT-3.5, GPT-4, DuoT5, and ListT5 stands for GPT-3.5-turbo-1106, GPT-4-0613, DuoT5-base, and ListT5-base, respectively. Using the FiD structure effectively mitigates the problem of the positional bias of positive passages, showing lowest standard deviation and highest agreement ratio.

computational complexity in Sec. 3.2, we also conduct real-time measurements of floating point operations (FLOPs) using the FlopsProfiler of the DeepSpeed library[3]. We measure the FLOPs needed to rerank Top-10 passages from BM25-Top100 candidate passages for TREC-DL19 (43 queries, maximum input length of 256) on T5-base, run the profiler, and report the relative FLOPs when we set the FLOPs of MonoT5 (pointwise model) as 1[4]. We also report the performance and FLOPs of the sliding window variants of ListT5, with details at App. J. The results in Fig. 4 shows that ListT5 (r=1) and (r=2) are comparable with MonoT5 and RankT5 (pointwise ranking models), and much more efficient than DuoT5 (pairwise ranking models). Also, tournament sort uses output caching to lower down FLOPs than the sliding window variants, with better performance, on the real-time FLOPs calculation (App. J). Additionally, we can *parallize* computation of the same level (e.g., leaf node) within query for tournament sort, while the sliding window approach should be computed sequentially. Even better, the ListT5 model is *flexible* - according to their needs, users can have the choice to select between the trade-off of inference speed (r=1) and performance (r=2), an option not available for other models.

4.5 Robustness to Positional Bias

Background. One of the biggest problems of lengthy input of zero-shot listwise rerankers is the lost in the middle problem (Liu et al., 2023). Recent studies show that LLMs exhibit strong positional bias to the information in the first and last parts of the input, and often fail to comprehend relevant information in the middle of a long input. A recent work also pointed out that this positional bias problem is also prevalent in applying LLMs to listwise reranking (Tang et al., 2023). The main problem arises from the different assignment of positional encoding. We believe using the FiD architecture effectively addresses this issue, as each passage is processed by the encoder with identical positional encoding. Consequently, the decoder *cannot* exploit positional bias. To validate our statement, we conduct a comparative study to answer the following questions:

(1) How sensitive is the accuracy of each model to the position (location) of the positive passage?

(2) How often does each model *consistently* point to the same passage under position changes?

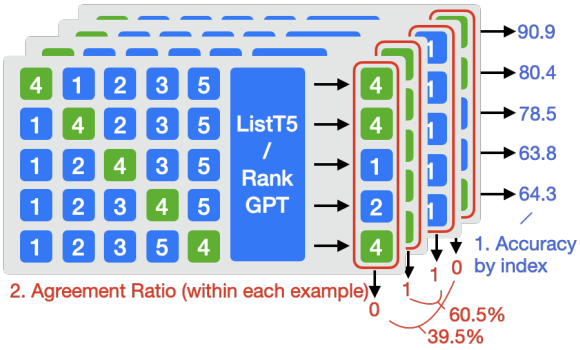

Figure 5: Measuring the robustness to positional bias by shuffling the index of the relevant passage. 4 is the gold (ground truth) relevant passage in the figure.

Baseline Models. We experiment with DuoT5 as representative for pairwise methods, and GPT3.5-turbo-1106 and GPT-4-0613 from OpenAI (OpenAI, 2022), the main model used for RankGPT (Sun et al., 2023b), as representative for listwise reranking methods. Note that we only

[3] https://deepspeed.readthedocs.io/en/latest/flops-profiler.html

[4] The exact value of FLOPs for MonoT5 in this setup was 229,732,539,806,400.

consider the top-1 prediction of each model, so the choie of ListT5 (r=1) and (r=2) does not affect the metric. Detailed explanation about setups with example prompts are in Appendix H.

Experiment and Data Setup. We use the BM25 top 100 retrieved results from the FiQA-2018 and TREC-COVID test subset. For listwise ranking models, we randomly sample negative passages per every positive passage in every query to make groups of [1 positive, 4 negatives].[5] Since pairwise models only take 2 passages for input, we additionally split the original group into 4 groups of [1 positive, 1 negative] for DuoT5. Similar with Tang et al. (2023), we measure the accuracy and agreement ratio after swapping the order of positive passages for each group (Fig. 5). The results in Tab. 4 show that (1) our model suffers less from the positive index change with (2) a higher agreement ratio. Since each passage is distinguished by identifiers instead of positions, ListT5 effectively overcomes the positional bias from long inputs by separately encoding each query-passage pair at the encoder and aggregating the output at the decoder. Still, the decoder can utilize and integrate the encoded information of all passages and effectively perform listwise reranking, achieving a better zero-shot performance than LLMs (Tab. 3) or non-listwise counterparts (Tab. 2).

4.6 Impact of initial ordering

All listwise reranking methods, including the tournament sort and the sliding window approach, depends on the initial ordering returned by the first-stage retriever (e.g., BM25). This may leave a room for ordering sensitivity. To investigate this, we experiment on DL19, DL20, and the selected subset of BEIR, to measure the robustness of initial ordering. For each dataset, we shuffle the initial ordering of candidate passages on 3 different seeds (seed 0,1,2) and report the average score. The results from Table 5 show that (1) ordering sensitivity is indeed prevalent in previous listwise reranking models, and that (2) using ListT5 with FiD greatly reduces the sensitivity in general (in line with our findings in Section. 4.5), and (3) When using ListT5, tournament sort is in general more robust to ordering change (-0.4 drop in average performance), compared to sliding windows (-1.2 drop). This means that ListT5 with tournament sort can perform in a robust way even in conditions where the initial ordering is not very trustable, which is mostly the case for zero-shot retrieval tasks.

[5]For TREC-COVID, one positive passage were sampled from each query since it has an average of 493 positive passages per each query.

Initial ordering	DL19	DL20	TREC-COVID	TREC-NEWS	Touche -2020	Avg.
ListT5-base (tournament sort, r=2)						
No shuffle	71.8	68.1	78.3	48.5	33.4	60.0
Shuffle	71.2	68.1	77.2	48.9	32.8	59.6
Perf. drop						**-0.4**
ListT5-base (sliding windows, stride=3, iter=4)						
No shuffle	71.8	67.7	77.5	50.0	33.1	60.0
Shuffle	69.5	65.5	77.7	49.2	32.1	58.8
Perf. drop						**-1.2**
RankVicuna-7b (sliding windows)						
No shuffle	68.9	66.1	80.5	46.9	33.0	59.1
Shuffle	67.1	64.6	79.2	45.3	30.8	57.4
Perf. drop						**-1.7**
RankGPT-3.5 (sliding windows)						
No shuffle	68.4	64.9	72.6	46.5	38.2	58.1
Shuffle	62.5	57.0	66.1	38.3	22.8	49.3
Perf. drop						**-8.8**

Table 5: NDCG@10 results before and after randomly shuffling the initial top-100 ordering of BM25. Evaluation results for RankGPT-3.5 are run 3 times and averaged to compensate for the unstability of the API output. While RankGPT-3.5 suffers heavily after shuffling (-8.8 drop in performance), ListT5 with tournament sort is relatively robust to the initial order shuffling.

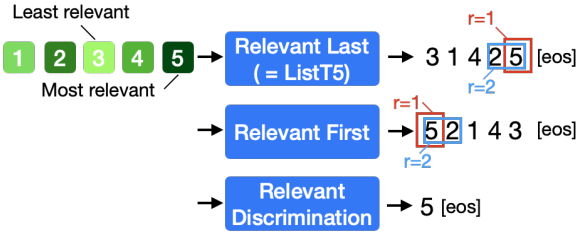

Figure 6: Variants of the output format of listwise ranking models while fixing the input.

4.7 Ablation Study

To break down the role of each feature in our method, we perform experiments with different variants of our model (Fig. 6) in Tab. 6. Additional ablation experiments, such as varying m, are in App. B.1.

r = 1 vs r = 2. Conducting inference by ($r = 2$) improves the average BEIR performance by 0.3, compared with that of ($r = 1$). Since this gain is also applicable to the Relevant First variants, which is the model trained to generate relevant index first,

Dataset	Relevant Discrimination	Relevant First		Relevant Last (ListT5)	
		$(r = 1)$	$(r = 2)$	$(r = 1)$	$(r = 2)$
In-domain					
MS MARCO	40.3	40.8	**40.9**	40.7	40.7
TREC-DL19	**72.5**	69.6	70.8	71.2	71.8
TREC-DL20	67.3	67.0	66.8	67.3	**68.1**
Avg (in-domain)	60.0	59.1	59.5	59.7	**60.2**
Out-domain (BEIR)					
TREC-COVID	74.0	74.9	75.9	76.7	**78.3**
NFCorpus	34.8	35.5	35.6	35.5	**35.6**
BioASQ	55.8	56.6	56.6	**57.2**	56.4
NQ	51.1	52.7	52.9	52.0	**53.1**
HotpotQA	70.9	72.5	72.6	72.1	**72.6**
FiQA-2018	38.1	39.3	39.0	39.5	**39.6**
Signal-1M (RT)	32.9	31.8	31.7	33.3	**33.5**
TREC-NEWS	43.9	46.6	47.3	47.9	**48.5**
Robust04	49.8	**52.3**	**52.3**	52.0	52.1
Arguana	26.1	32.8	34.6	**49.7**	48.9
Touche-2020	**34.2**	31.5	31.3	**34.2**	33.4
CQADupStack	**38.8**	38.3	38.4	38.4	**38.8**
Quora	81.9	84.4	84.8	86.1	**86.4**
DBPedia	42.4	43.4	43.6	**43.9**	43.7
SCIDOCS	16.3	17.3	17.3	17.2	**17.6**
FEVER	77.6	77.4	77.7	77.8	**79.8**
Climate-FEVER	20.7	22.8	23.0	22.8	**24.0**
SciFact	73.0	74.1	**74.2**	74.1	74.1
Avg (BEIR)	47.9	49.1	49.4	50.6	**50.9**

Table 6: NDCG@10 results at in-domain and BEIR on varying the output format and generation order of passages. Generating from the least relevant passages shows better average performance on BEIR.

we hypothesize that $r = 2$ serves as giving a second chance for the model to view passages with different candidates to make a decision with different viewpoints, where there are confusing passages (hard negatives), resulting in better ability to rank.

Discrimination vs Sequential Sorting. Given the same setup and dataset, we also train the model to output only relevant index, annotated as *Relevant Discrimination*, similar to Hofstätter et al. (2022). However, the Relevant Discrimination model shows the lowest performance amongst all model variants (47.9). We hypothesize that training the model to distinguish and order between negatives provides additional information for the model. While tasks that only generate positive indexes only learn the decision boundary between positive and negative contexts, learning to order negatives provides more informative signals and more accurate ranking results, since the model additionally learns to calibrate between any kind of passages.

Relevant First vs Relevant Last (ListT5). We also compare models by changing the ordering of the output. Compared to ListT5, Relevant first shows a performance drop of 1-2 points in average on BEIR. As explained at Sec. 3.1, we conclude that generating in reversed order is important, as it effectively eliminates false negatives and acts as a reasoning chain.

5 Conclusion

We proposed ListT5, a FiD-based listwise reranking model that jointly considers multiple passages as input and generates an ordered list of passages, in increasing order of relevance. ListT5 outperforms others in BEIR, with a remarkable +1.3 gain of average NDCG@10 performance. ListT5 is efficient, leveraging tournament sort to its advantage. Finally, we show that ListT5 overcomes the lost-in-the-middle problem in LLMs, and are more robust regardless of the initial ordering of the passage.

6 Limitations

We believe that the efficiency of our models can be greatly optimized using simple ideas. For example, if we can implement early stopping at sequential decoding (assuming that the model outputs permutation of 5), we can reduce the decoding step by 80% for ListT5 (r=1) and 60% For ListT5 (r=2). There could also be other options, such as designing a more efficient way that uses the tournament sort with fewer number of forward passes, efficient way of encoder output caching, and extension to other langauges or other architectures.

We believe there are still space for optimization of ListT5, for example trying out m other than 5 or 10, exploring with different learning rate or batch size on training, and so on. However, we were only able to experiment a few due to resource and time limitations.

Our results are mainly reported with BM25 Top-100, BM25 Top-1000, and COCO-DR Top-100 as first-stage retrieval modules. Also, our main results are based on the T5-base model and only a few were experimented with 3B models, due to the resource constraints. We hope to evaluate on extended setups for future work.

7 Acknowledgements

We greatly appreciate the members from the EXAONE lab from LG AI research for providing GPU resources to conduct experiments along with the helpful feedbacks. We would also like to thank the members from the SNU Language & Data Intelligence lab for giving constructive feedbacks to improve the paper. We also thank the anonymous reviewers from ARR December and ARR February for pointing out important discussion points and additional experiments that help support our claim. Lastly, we would like to thank Jinwoo Kim from School of Computing, KAIST for helpful discussions.

References

Qingyao Ai, Xuanhui Wang, Sebastian Bruch, Nadav Golbandi, Michael Bendersky, and Marc Najork. 2019. Learning groupwise multivariate scoring functions using deep neural networks. In *Proceedings of the 2019 ACM SIGIR international conference on theory of information retrieval*, pages 85–92.

Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2018. Ms marco: A human generated machine reading comprehension dataset.

Nick Craswell, Bhaskar Mitra, Emine Yilmaz, and Daniel Campos. 2021. Overview of the trec 2020 deep learning track.

Fernando Ferraretto, Thiago Laitz, Roberto Lotufo, and Rodrigo Nogueira. 2023. Exaranker: Explanation-augmented neural ranker.

Sebastian Hofstätter, Jiecao Chen, Karthik Raman, and Hamed Zamani. 2022. Fid-light: Efficient and effective retrieval-augmented text generation.

Gautier Izacard and Edouard Grave. 2021. Leveraging passage retrieval with generative models for open domain question answering.

Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Kushal Lakhotia, Bhargavi Paranjape, Asish Ghoshal, Wen tau Yih, Yashar Mehdad, and Srinivasan Iyer. 2020. Fid-ex: Improving sequence-to-sequence models for extractive rationale generation.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts.

Xueguang Ma, Xinyu Zhang, Ronak Pradeep, and Jimmy Lin. 2023. Zero-shot listwise document reranking with a large language model.

Keith McLuckie and Angus Barber. 1986. *Tournament Sort*, pages 68–86. Macmillan Education UK, London.

Ryan Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. 2019. Relational pooling for graph representations. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4663–4673. PMLR.

Jianmo Ni, Chen Qu, Jing Lu, Zhuyun Dai, Gustavo Hernandez Abrego, Ji Ma, Vincent Zhao, Yi Luan, Keith Hall, Ming-Wei Chang, and Yinfei Yang. 2022. Large dual encoders are generalizable retrievers. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9844–9855, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Rodrigo Nogueira, Zhiying Jiang, Ronak Pradeep, and Jimmy Lin. 2020. Document ranking with a pre-trained sequence-to-sequence model. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 708–718, Online. Association for Computational Linguistics.

OpenAI. 2022. Introducing chatgpt. https://openai.com/blog/chatgpt.

Ronak Pradeep, Rodrigo Nogueira, and Jimmy Lin. 2021. The expando-mono-duo design pattern for text ranking with pretrained sequence-to-sequence models.

Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. 2023a. Rankvicuna: Zero-shot listwise document reranking with open-source large language models.

Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. 2023b. Rankzephyr: Effective and robust zero-shot listwise reranking is a breeze!

Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, Xuanhui Wang, and Michael Bendersky. 2023. Large language models are effective text rankers with pairwise ranking prompting.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks.

Wenbo Ren, Jia Liu, and Ness B. Shroff. 2018. Pac ranking from pairwise and listwise queries: Lower bounds and upper bounds.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389.

Guilherme Rosa, Luiz Bonifacio, Vitor Jeronymo, Hugo Abonizio, Marzieh Fadaee, Roberto Lotufo, and Rodrigo Nogueira. 2022. In defense of cross-encoders for zero-shot retrieval.

Weiwei Sun, Zheng Chen, Xinyu Ma, Lingyong Yan, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023a. Instruction distillation makes large language models efficient zero-shot rankers.

Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023b. Is chatgpt good at search? investigating large language models as re-ranking agents.

Raphael Tang, Xinyu Zhang, Xueguang Ma, Jimmy Lin, and Ferhan Ture. 2023. Found in the middle: Permutation self-consistency improves listwise ranking in large language models.

Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. Beir: A heterogenous benchmark for zero-shot evaluation of information retrieval models.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2024. Large search model: Redefining search stack in the era of llms.

Ruicheng Xian, Honglei Zhuang, Zhen Qin, Hamed Zamani, Jing Lu, Ji Ma, Kai Hui, Han Zhao, Xuanhui Wang, and Michael Bendersky. 2023. Learning list-level domain-invariant representations for ranking.

Dmitry Yarotsky. 2018. Universal approximations of invariant maps by neural networks.

Yue Yu, Chenyan Xiong, Si Sun, Chao Zhang, and Arnold Overwijk. 2022. Coco-dr: Combating distribution shifts in zero-shot dense retrieval with contrastive and distributionally robust learning.

Honglei Zhuang, Zhen Qin, Rolf Jagerman, Kai Hui, Ji Ma, Jing Lu, Jianmo Ni, Xuanhui Wang, and Michael Bendersky. 2022. Rankt5: Fine-tuning t5 for text ranking with ranking losses.

	m = 5 (= ListT5)				m = 10	
	1/5 (r=1)	2/5 (r=2)	3/5 (r=3)	4/5 (r=4)	1/10 (r=1)	4/10 (r=4)
MS MARCO	**40.7**	**40.7**	-	-	40.5	**40.7**
+ Top-1000	44.7	44.9	-	-	44.6	**45.0**
TREC-DL19	71.2	**71.8**	-	-	70.1	70.5
TREC-DL20	67.3	**68.1**	-	-	66.9	67.2
TREC-COVID	76.7	**78.3**	78.1	**78.3**	76.2	77.9
NFCorpus	35.5	35.6	35.3	35.6	36.2	**36.6**
BioASQ	**57.2**	56.4	55.9	55.8	55.4	56.4
NQ	52.0	**53.1**	52.2	52.2	51.5	52.5
HotpotQA	72.1	**72.6**	71.6	71.6	71.4	71.9
FiQA-2018	39.5	**39.6**	**39.6**	39.5	39.0	38.9
Signal-1M (RT)	33.3	**33.5**	33.3	33.2	31.7	32.0
TREC-NEWS	47.9	48.5	**49.2**	48.6	47.1	47.8
Robust04	52.0	52.1	51.6	51.7	52.2	**53.1**
Arguana	49.7	48.9	49.4	**49.9**	38.6	46.6
Touche-2020	**34.2**	33.4	33.8	**34.2**	32.4	32.7
CQADupStack	38.4	**38.8**	38.5	38.6	38.2	28.8
Quora	86.1	86.4	86.0	86.1	85.5	**86.8**
DBPedia	**43.9**	43.7	43.3	43.1	42.7	43.6
SCIDOCS	17.2	17.6	17.4	17.5	17.2	**18.0**
FEVER	77.8	**79.8**	78.6	78.4	76.7	79.1
Climate-FEVER	22.8	**24.0**	23.4	23.3	22.7	23.9
SciFact	74.1	74.1	73.8	74.3	73.4	**74.2**
Avg(In-domain)	56.0	**56.4**	-	-	55.5	55.9
Avg(BeIR)	50.6	**50.9**	50.6	50.7	49.3	50.0

Table 7: NDCG@10 performance results for in-domain & BEIR with ListT5-base, on differing m. (Sec.B.1.) We compare evaluation results that are trained on seeing 5 / 10 passages at once. Like the original ListT5, both models generate relevant indexes at the last.

Model	RankT5	ListT5			
Training data	GTR	COCO-DR	GTR		
Learning Rate	1.00E-04	1.00E-04	1.00E-04	1.00E-04	1.00E-05
Steps	-	20k	20k	10k	30k
TREC-COVID	77.7	78.3	77.3	78.6	77.9
NFCorpus	35.1	35.6	35.4	36.2	35.9
BioASQ	58.2	56.4	54.9	55.1	56.8
NQ	53.2	53.1	52.7	52.8	53.2
HotpotQA	72.8	72.6	72.1	71.9	72.1
FiQA-2018	39.2	39.6	39.1	39.9	39.4
Signal-1M (RT)	30.8	33.5	34.1	32.9	30.9
TREC-NEWS	45.4	48.5	47.6	48.0	48.3
Robust04	54.3	52.1	52.9	52.7	53.6
Arguana	35.5	48.9	43.3	43.6	43.7
Touche-2020	37.1	33.4	31.5	32.7	32.5
CQADupStack	37.0	38.8	38.6	38.5	38.8
Quora	83.3	86.4	86.0	83.9	84.4
DBPedia	43.7	43.7	43.8	43.6	44.5
SCIDOCS	16.8	17.6	17.1	17.1	17.6
FEVER	77.6	79.8	78.9	79.4	79.7
Climate-FEVER	21.2	24.0	24.0	24.8	24.6
SciFact	73.5	74.1	74.0	73.1	74.4
Avg.	49.6	**50.9**	50.2	50.3	**50.5**

Table 8: Comparison of ListT5 models trained with GTR and COCO-DR. For ListT5, the reported scores are evaluated using the (r=2) variant. (Sec. B.2)

A Related Work - FiD in a Similar Setup

Hofstätter et al. (2022) assigned index numbers to each query and passage pair, directing an FiD model to generate relevant passage indexes—an idea initially proposed by Lakhotia et al. (2020). While their approach is similar to ours, it diverges in that they only generate a single positive index, not the sorted list of all indices.

B Design Choice of ListT5

B.1 Impact of the number of passages m ListT5 sees at once

This aligns with Sec. 4.7 from the main paper. In addition to varying the output format, we try to increase the number of passages m the model sees at once, with variants on r as well (See Sec. 3.1 for basic notation). The results from Tab. 7 show that setting m as 10 is not as effective as setting m as 5. We conjecture that there is a trade-off between the (1) benefits from seeing more negatives at once, and (2) the complexity of the ordering task itself. If we increase m, the model can get more information, but at the same time, the task becomes very hard, and the boundary of negative order becomes very weak (and not very informative). Therefore, in the case of T5-base models, we find that 5 performed better than 10 passages. Compared to the original FiD for QA (Izacard and Grave, 2021) where they use large m (=100), ListT5 works best with smaller m. We conjecture the reason for this is that sorting task has more complexity than the original task that only use the input as knowledge source, leading to increased effectiveness when m is smaller. Where the use case of the original FiD are to find relevant information among inputs, ListT5 needs to order between different inputs, a task that needs to analyze the relative relevancy between *all* given inputs. However, we believe the optimal number can be improved (or can be different) for models with different architectures or with larger parametric size.

B.2 Impact of the negative selection model

In our main experiment, we use COCO-DR to select and align negatives from MS MARCO, when constructing the training data. To analyze the impact of using COCO-DR as negatives, we also experiment with training ListT5 using negatives sampled from GTR,[6] since RankT5 (Zhuang et al., 2022) used GTR to retrieve top 1000 passage from each query and conduct random sampling to select

[6] https://huggingface.co/sentence-transformers/gtr-t5-xl

negative passages. (Note that RankT5 samples 35 negative passages per one query from the top 1000 retrieval results, while ListT5 only sees 4 negatives at once.) The results are at Tab. 8. First, we train the T5-base model with the same hyperparameter setup for those used to train the original LISTT5-COCO-DR model. [7] Then, we conduct hyperparameter search on the learning rate and number of steps for the GTR model, and found that training the model with learning rate of 1e-05 for 30k steps performs best (average 50.5). The results indicate that while LISTT5 trained with different negatives can impact performance, the performance gap is not significant, and the ListT5 model based on both COCO-DR and GTR still exhibit better performance than RankT5.

C Details about Our Inference Framework

This aligns with Sec. 3.2 from the main paper.

C.1 Background: Tournament Sort

Tournament sort is a variation of heap sort, inspired by the concept from sports tournaments, where its objective is to identify not just the best player, but also the k-best, with the minimum number of games. While naive selection sort takes $\mathcal{O}(n)$ operations to iteratively select the best element, *tournament sort* leverage its tournament tree structure for efficient sorting, resulting in needing only $\mathcal{O}(\log n)$ operations after the initial building of the tree in $\mathcal{O}(n)$

C.2 Modification to Accept LISTT5 (r = 2)

Fig. 7 illustrates and compare the inference scenario for LISTT5 (r = 1) and LISTT5 (r = 2). Since the (r=2) variant outputs *2* different passage indices, the total number of candidate passages we need to compute is multipled by 2 for the next level of a tree. For example, if we use LISTT5 (r = 1) with m = 5 at the initial iteration, after computation of the bottom-level, we would get a total of 20 (100 / 5) filtered candidates which will go to the next round for comparison. In contrast, if we use LISTT5 (r = 2) with m = 5, we would get a total of 40 (100/5 $*$ 2) candidates to compute at the next round. However, it is important to note that we only use the inference method of (r = 1) for the root computation, since we always outputs the top-1 candidates for every iteration.

[7] With learning rate of 1e-04, linear learning rate scheduler with 10% warmup, effective batch size 256, maximum input length 230, and maximum output length of 7, and train the T5-base model for 20k steps.

	Random replace index number:							
Dataset	21 (orig.)	20	19	18	17	16	15	6
signal	33.5	33.3	33.2	33.1	33.3	33.3	33.3	33.5
trec-covid	78.3	78.2	78.2	78.4	78.1	78.1	78.3	78.3
news	48.5	48.4	48.4	48.4	48.3	48.4	48.4	48.4
scifact	74.1	74.0	74.0	74.0	74.0	74.1	74.2	74.0
nfcorpus	35.6	35.7	35.7	35.7	35.7	35.7	35.7	35.8
fiqa	39.6	39.6	39.7	39.7	39.6	39.6	39.6	39.7
bioasq	56.4	56.6	56.5	56.5	56.4	56.5	56.5	56.5
touche	33.4	33.7	33.4	33.5	33.6	33.8	33.8	33.6
dbpedia	43.7	43.6	43.8	43.7	43.8	43.7	43.7	43.8
robust04	52.1	52.1	52.0	52.1	52.0	52.1	52.1	51.9
scidocs	17.6	17.6	17.6	17.5	17.6	17.5	17.6	17.5
arguana	48.9	49.0	48.8	48.8	48.9	49.0	48.9	48.9
climate-fever	24.0	24.0	24.0	24.0	24.0	24.0	24.0	24.0
dl19	71.8	71.9	71.8	71.7	71.7	71.8	71.7	71.8
dl20	68.1	68.2	68.1	68.1	68.1	68.1	68.3	68.5
Average	**48.4**	**48.4**	**48.3**	**48.3**	**48.3**	**48.4**	**48.4**	**48.4**

Table 9: NDCG@10 results on differing the index of random assignment, with explanations on Section C.3.

C.3 Details on Random Replacement

As explained as the 3rd and 4th modification at the main paper Sec. 3.2, we explain in detail the random assignment process here. Unlike the original tournament sort where they mark the final selected values to infinity, it is not an option for us since we cannot define a passage that has infinity scores of relevance. Therefore, we choose to randomly fill out the place, by the following rules;

Random assignment of top-1 selected passage After one iteration, one index is selected as most relevant out of n candidate passages. Then, we add that index into the global exclude pool and remove the index from the consideration pool. After that, using the initial ordering of the passage by the first-stage retrieval, we add $+21$ to the original selected index. (For example, if the selected index was 8, we replace that passage with 29. We just set it to a larger value than m=5 to avoid meeting duplicates, but it can be anything) If the index is already in the global exclude pool *or* have duplicates inside the input window m, we consider the next passage by adding 1 until the conditions are met.

Random assignment of dummy values to fill m The basic unit of LISTT5 always accepts exactly m passages as the input. However, there are cases when the leftover passages are not a multiple of m. For example, as we proceed to the upper level on the tournament tree, there will be a total of 100 $\mapsto$ 20 $\mapsto$ 4 passages left if we rank 100 passages using

ListT5 (r=1) with m=5. In this case, we would need one more dummy passage to make 4 candidate passages to 5. For reproducibility, we select a dummy passage in a deterministic way by considering the first passage from the initial candidate passage pool. If that passage is already selected as top-k indices during the previous round *or* we have duplicates inside the input window m, we proceed to the next index until the conditions are met. To show that the assignment of the random index is not statistically significant, we have conducted additional experiments on the subset of BEIR with differing the random replace index number. The results are presented at Table. 9. From the table, we can see that the output is robust to the assignment of random replace index number.

Edge Cases. There are some cases, especially for the case of NFCorpus, that the initial retrieval module doesn't give exactly n passages as initial candidate passages. Sometimes, they give n that is smaller than m (e.g., 1,2,3,4). For this case, we exceptionally allow duplicates within the input window m, run one basic unit of ListT5 with (r=n) (We remove duplicate passage indexes so that the output ordering reduces to n). That is, we only run the inference once, and use the output ordering as it is. Also, if k is bigger than n-m, we inevitably meet the case where the leftover (unselected) passage number becomes smaller than m. (For example, ranking the 96th candidate passage for n=100 with m=5). Similar to the previous case, we also allow duplicates for this case and run the basic unit once to order the leftover indices.

C.4 Example Scenario using Output Caching

For example, if we consider reranking top 10 passages from 100 candidate passages (n = 100, k = 10), without output caching results in 250 forward passes in total; since we would need ((100/5) + (20/5) + 1) = 25 forward passes for each iteration, the total will be 25 * 10 = 250. With output caching, after the initial iteration, we only need to do one computation for each level of a tree, resulting in (1+1+1) = 3 forward passes. Summing up results in 25 + 3*9 = 54 forward passes, almost reducing **80%** of the total number of inferences.

	COCO-DR Large (Init.)	MonoT5	RankT5	ListT5 (r=1)	ListT5 (r=2)
MSMARCO Top-1000 (in-domain)	41.9	43.1	46.2	46.1	**46.3**
TREC-COVID	80.8	**83.5**	**83.5**	83.2	**83.5**
NFCorpus	35.5	35.6	35.5	**36.2**	**36.2**
NQ	54.3	57.9	59.6	59.7	**60.0**
HotpotQA	63.3	68.7	**71.1**	70.3	70.9
FiQA-2018	32.3	41.2	41.3	**41.7**	**41.7**
Arguana	46.9	33.0	34.8	49.0	**49.3**
Touche-2020	21.6	25.7	**35.7**	29.1	29.6
CQADupStack	37.3	40.5	38.7	40.7	**40.9**
Quora	**87.3**	84.0	83.0	86.2	86.3
DBPedia	40.7	44.4	**46.1**	45.6	45.4
SCIDOCS	17.3	17.5	17.5	17.7	**18.3**
FEVER	74.9	78.9	79.7	79.8	**81.4**
Climate-FEVER	23.1	24.2	22.9	23.9	**24.9**
SciFact	71.9	73.5	73.6	**74.4**	74.3
Avg. BEIR	49.1	50.6	51.6	52.7	**53.1**

Table 10: NDCG@10 results for reranking on top of COCO-DR large Top-100 first-stage retrieval results (Sec. D.1).

D Additional Results.

D.1 Results on COCO-DR as first-stage retrieval model.

We also conduct reranking using the Top-100 results of COCO-DR-large (Yu et al., 2022), and compare reranked results with MonoT5 (Nogueira et al., 2020), RankT5 (Zhuang et al., 2022), and ListT5. For this experiment, we could only utilize the evaluation subset that is uploaded (fully open) in the BEIR repository.[8] For simplicity, the input token length is fixed to 512 for first-stage retrieval with COCO-DR. From the results at Tab 10, ListT5 achieves an average of +1.5 point gain for reranking on COCO-DR Top-100, additionally showing its applicability and effectiveness on both lexical-based (BM25) and neural-based first-stage retrieval modules.

E MRR scores

Table 11 reports the corresponding MRR scores for BEIR evaluation on the main table.

F Hyperparameter Optimization Methodology

We conduct a structured grid search strategy on a selected subset of the BEIR dataset. The selection

[8]Thus, results for BioASQ, Signal-1M, TREC-NEWS, Robust04 are not computed.

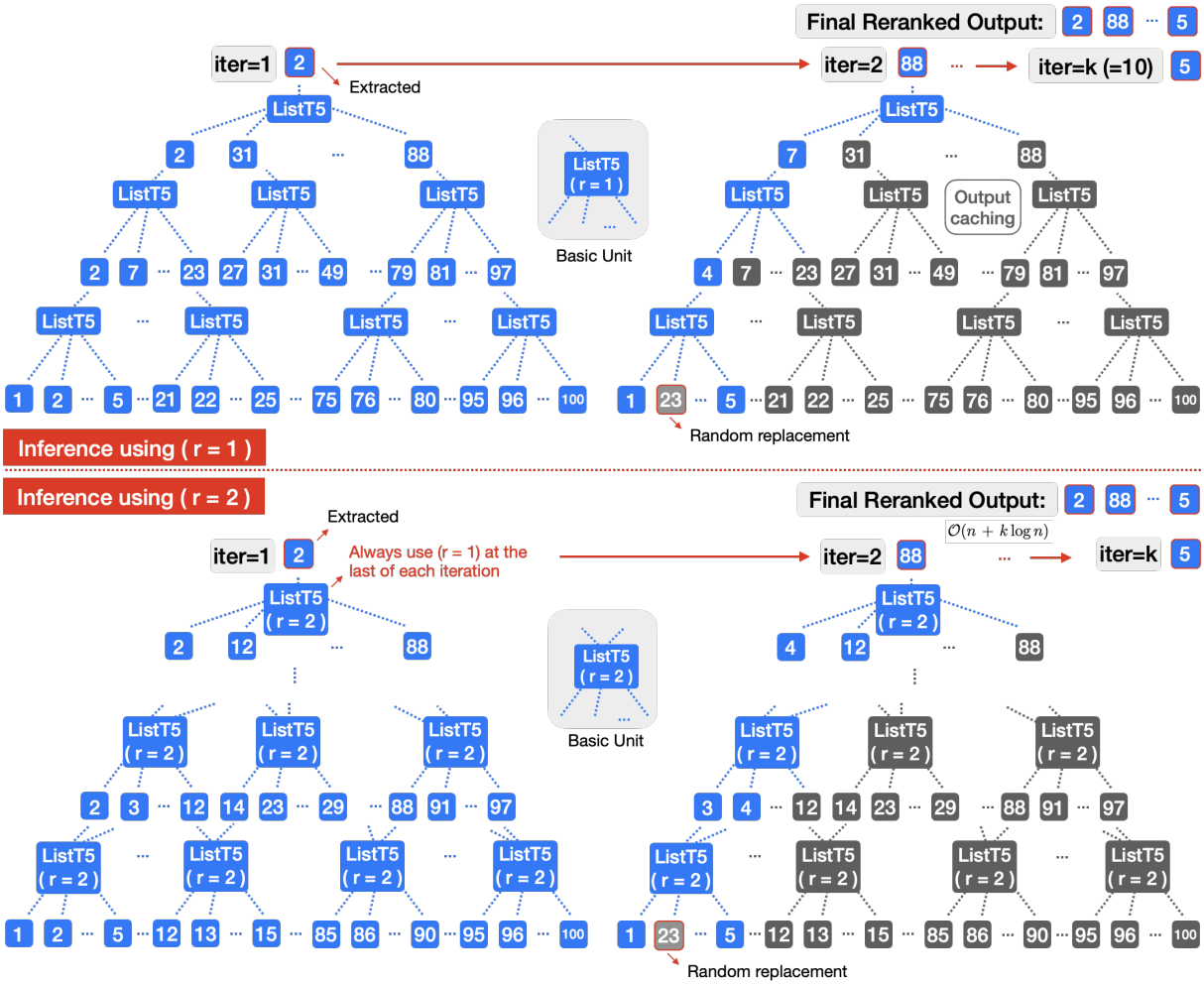

Figure 7: Illustration of our inference scenario. We show and compare the process with (r=1) and (r=2).

of hyperparameters for investigation was informed by prior research and best practices within the domain, specifically referencing the work by (Zhuang et al., 2022), which demonstrated effective use of a learning rate of 1×10^{-4}. To explore the potential for further optimization, we extended the search to include lower learning rates down to 1×10^{-5}, considering the sensitivity of large models to learning rate adjustments.

We employed a linear learning rate scheduler with a 10% warmup over 10 epochs, a strategy for stabilizing training in the initial phases. The grid search was designed with a step size range from 10,000 to 30,000, increasing in increments of 10,000 steps.

For our 3 billion parameter models (3B models), due to their substantial computational requirements, we conducted a more granular evaluation. We assessed performance at every 1,000 steps between 1,000 and 10,000 steps.

This structured approach ensured that our hyperparameter selection was not arbitrary but based on a reasoned strategy aiming to balance computational efficiency with empirical performance. The specific choices of "250k steps" at a "1e-4" learning rate and "3k steps" at a "1e-5" learning rate were outcomes of this optimization process, identified as configurations that provided the best performance metrics within the constraints of our experimental setup.

G Details on Evaluation Dataset

G.1 Baseline Models.

We download and use the officially released checkpoints for the baseline model with the huggingface identifier of `castorini/monot5-base-msmarco-10k`, `monot5-3b-msmarco-10k` for MonoT5 (Nogueira et al., 2020), `castorini/duot5-base-msmarco` for DuoT5 (Pradeep et al., 2021),

First-stage retrieval by:		BM25 Top100					COCO-DR Large Top100				
		Baselines			Ours		Baselines			Ours	
Task (Domain)	Dataset	BM25 (Initial)	MonoT5	RankT5	ListT5 (r = 1)	ListT5 (r = 2)	COCO-DR (Initial)	MonoT5	RankT5	ListT5 (r = 1)	ListT5 (r = 2)
Passage Retrieval	MS MARCO	18.0	34.9	**35.8**	35.6	35.8	36.0	38.2	**40.5**	40.3	40.2
	+ top1000	18.0	37.3	38.6	38.5	**38.8**	36.0	37.2	**40.3**	40.1	40.2
	TREC-DL19	82.3	98.3	100.0	98.5	98.8					
	TREC-DL20	82.4	93.8	95.4	**95.8**	95.6					
In-domain average		61.0	66.1	**67.4**	67.1	67.2	36.0	37.7	**40.4**	40.2	40.2
Bio-Medical Information Retrieval	TREC-COVID	85.3	**95.4**	93.0	93.8	95.0	96.7	96.7	93.8	**97.0**	95.5
	NFCorpus	52.4	58.0	56.3	57.5	**58.5**	56.0	55.9	57.1	58.1	**58.4**
	BioASQ	59.4	64.9	**67.8**	67.6	67.2					
Question Answering (QA)	NQ	26.3	48.2	**49.8**	48.2	49.5	49.2	52.6	54.7	54.6	**55.1**
	HotpotQA	80.3	85.9	88.3	88.2	**88.7**	81.9	84.7	87.9	87.6	**88.5**
	FiQA-2018	29.6	47.7	48.5	**48.7**	48.5	39.1	49.2	**49.8**	49.6	**49.8**
Tweet Retrieval	Signal-1M (RT)	**57.4**	52.6	55.0	55.6	55.4					
News Retrieval	TREC-NEWS	73.9	74.8	75.8	77.9	**79.2**					
	Robust04	67.3	81.5	**82.0**	79.6	80.0					
Argument Retrieval	Arguana	32.8	27.1	27.7	**40.7**	39.6	39.2	26.8	28.6	40.5	**40.8**
	Touche-2020	**74.4**	51.3	62.9	58.5	54.9	45.4	47.0	**61.9**	51.1	51.2
Duplicate Q. Retrieval	CQADupStack	29.6	38.5	36.8	38.3	**38.7**	36.9	40.2	38.4	40.7	**40.8**
	Quora	77.9	83.1	81.2	85.1	**85.2**	**86.4**	82.0	80.5	84.9	84.7
Entity Retrieval	DBPedia	58.2	73.4	74.4	75.7	**76.3**	74.2	73.4	77.2	76.8	**77.8**
Citation Pred.	SCIDOCS	26.0	29.5	29.9	30.9	**31.2**	30.9	30.2	30.6	31.1	**32.0**
Fact Checking	FEVER	62.4	78.4	77.6	77.6	**80.2**	74.2	79.0	80.1	80.3	**82.1**
	Climate-FEVER	22.0	31.9	29.1	31.3	**33.7**	32.9	33.0	31.3	33.0	**34.8**
	SciFact	64.6	70.3	70.6	71.8	**71.9**	68.4	70.9	70.4	71.8	**71.8**
BeIR average		54.4	60.7	61.5	62.6	**63.0**	58.0	58.7	60.2	61.2	**61.7**

Table 11: Reporting Mean Reciprocal Rank (MRR)@10 scores for initial ranking of BM25 top-100 and COCO-DR large top-100. (Sec. E)

`castorini/rank_vicuna_7b_v1` for RankVicuna (Pradeep et al., 2023a), and `castorini/rank_zephyr_7b_v1_full` for RankZephyr (Pradeep et al., 2023b). For RankGPT(Sun et al., 2023b) using GPT-3.5 or GPT-4, we evaluate them using the OpenAI (OpenAI, 2022) API. For RankT5, we download the released T5X checkpoint from the official repository[9] and convert it into a PyTorch checkpoint using the HuggingFace's conversion script[10]. We evaluate DuoT5 on the same setup to the paper's final experiments - we rerank 100 passages using MonoT5, select top 50 passages to be run on DuoT5, and aggregate relevancy scores of individual documents by the SYM-SUM method.

We run the official RankLLM repository[11] to evaluate RankVicuna and RankZephyr on BEIR.

G.2 Links, Maximum Input Length

Since the average length of query and passage differs greatly for each BEIR subset, we assign the appropriate sequence length for each BEIR dataset and evaluate all models in the same setup. We download the dumped index of the top 100 retrieved passages by BM25 from the official Pyserini repository[12], and download the inital query, corpus, and qrels from the BEIR repository.[13]

We referenced the average query + passage length from the official BEIR paper. We selected the smallest maximum length from [256, 512, and 1024], but is still bigger than the (sum of average query + passage length) multiplied by two(for better coverage). One exception is the trec-news, which added up to be bigger than 1024, but we just capped it to be 1024. The *exact* alias with the maximum input length is listed below;

'msmarco': 256, 'dl19': 256, 'dl20': 256, 'trec-covid': 512, 'nfcorpus': 512, 'bioasq': 512, 'nq': 256, 'hotpotqa': 256, 'fiqa': 512, 'signal': 256, 'news': 1024, 'robust04': 1024, 'arguana': 1024, 'touche': 1024, 'cqadupstack': 512, 'quora': 256, 'dbpedia-entity': 256, 'scidocs': 512, 'fever': 256, 'climate-fever': 256, 'msmarco_top1000': 256,

[9] https://github.com/google-research/google-research/tree/master/rankt5

[10] convert_t5x_checkpoint_to_pytorch.py

[11] https://github.com/castorini/rank_llm

[12] https://github.com/castorini/pyserini

[13] https://public.ukp.informatik.tu-darmstadt.de/thakur/BEIR/datasets/

'scifact': 512

G.3 Other Replication Details

Output validation module Upon experiments, we do not use constrained decoding or validation module to ensure the generation of valid permutations of 5. Each digital number is represented as simple integers (The decoded output looks something like "1 2 5 4 3" or "2 3 1 5 4"), with an exception catching module such that if the parsing fail, we go back to the original ordering. Even without any validation module, since LISTT5 is fine-tuned to output valid indexes for 20000 steps, if the input follows the correct format, the model almost always outputs valid index. Nevertheless, we think that adding a constrained decoding module would be beneficial.

Evaluation We run the same evaluation setup with the BEIR library to do evaluation. For example, we append title with a space along with the text(passage) part and treat as full passage. For evaluating reranked results for both BM25 Top100 and COCO-DR Top100 candidate passages, we remove duplicates before evaluation[14] For CQADupStack, we follow the same procedure and aggregated all sub-domains ranging from android to wordpress to make 13,145 test queries, and report the average performance.

H Details about the Experiment Measuring Positional Bias.

H.1 Evaluation process.

We briefly explain here about the evaluation process for the FiQA dataset. We defined the positional bias inspired from Liu et al., [9], where they measure the answer accuracy with respect to the index change of the relevant passage.

1. We break down the original dataset into pairs of one query and one positive passage associated with the query. For each query-positive pair, we randomly sample 4 negative passages from the BM25 top100 results and additionally pair them to make one sample. Preprocessing them makes 818 distinct samples.

2. For every sample, we make 5 variants of the input text, so that the first one assigns index 1 to the positive passage, and the last one assigns index 5 to the positive passage. Note that only the index of the positive passage is changed, and we keep the order of other passages (negatives) as the same.

3. we forward the inputs to each model, and collect the output that corresponds to the most relevant index. To answer (1), we measure the accuracy with respect to the index of positive passage. To answer (2), we analyze the ratio of samples where the model points to the same passage regardless of positive index change.

We discard queries that doesn't have positive indexes in the bm25 top100 dataset, or those that don't have 4 distinct negative contexts. For reproducibility, all randomly sampled datasets used for experiments are conducted with fixed seed. (seed=0) Standard deviation are calculated using the numpy.std function.

H.2 Prompt example.

To enforce a scenario where the model takes lengthy inputs at once, we apply the instruction format of permutation generation (text), and use the same prompt described in the RankVicuna (Pradeep et al., 2023a) paper. The prompts we use to give inputs to GPT-3.5-turbo-0301 are the following:

```
I will provide you with 5 passages,
each indicated by numerical identifier [].
Rank the passages based on their
relevance to the search query: {query}.

[1] {passage_1}
[2] {passage_2}
[3] {passage_3}
[4] {passage_4}
[5] {passage_5}

Search Query: {query}

Rank the 5 passages above based on their
relevance to the search query.
All the passages should be included and
listed using identifiers, in descending
order of relevance. The output format
should be [] > [], e.g., [4] > [2].
Only respond with the ranking
results, do not say any word or explain.
```

Given the above input, we collect the output of the LLM. Following the setup from Sun et al. (2023b), we use the ChatCompletion api by the openai library, with temperature of 0. Using the openai api, running the whole process cost about $35. To run ths GPT4 experiment on FiQA, it additionally cost 195.8$ (total of 6523910 input / 1298 output tokens). After the run, we found that 3 out of 818 outputs had malformed outputs, in a format such as: "[No relevant passage found] > ... [1]", rather than in a correct format such as: "[3] > [1] > [5] > [2]

[14] For example, quora had 1 (out of 10000) duplicates, so the NDCG@10 for BM25 may differ from the BEIR paper.

> [4]". We discard those 3 malformatted instances with incorrect output formats into consideration and compared only with the rest. (That is, we also discard those instances for the FiD-T5 variant) To give an exactly same scenario to both models, we truncate each passages up to 512 tokens (based on the T5 tokenizer), since that is exactly what our fid model sees. After truncation, we also validated that all of the input sizes are below the range of 4096 (as the maximum input window size of gpt-3.5-turbo-0301 is 4096)

H.3 Detailed analysis about pairwise methods

We find that pairwise models also exhibit positional bias, having a tendency to label the passage that comes at the front as positive. Theoretically, we can remove the positional sensitivity by evaluating all possible permutations of input passages (Murphy et al., 2019; Yarotsky, 2018). DuoT5 and other pairwise ranking methods (Qin et al., 2023) already include swapping orders and aggregate scores from both orderings. To measure the positional bias of pairwise models, in this experiment, we removed the averaging from swapping orders in DuoT5. However, the number of forward passes needed to mitigate positional bias problems in this way grows in a factorial scale, making it impossible and impractical to be applied to listwise methods. (We need 5! = 120 number of forward passes in order to remove the positional bias of 5 instances) In contrast, we effectively mitigate the positional bias problem, without any additional forward passes, with the Fusion-in-Decoder architecture.

H.4 Experiments on the consistency of LLMs.

We ran the positional bias experiment (on GPT3.5) on TREC-COVID for 3 times to investigate the following: given the same input, does LLMs generate the same output multiple times? [15] The results from api calls at Tab. 13 were not always exactly the same, but the difference was negligible, not changing the main claims. For the case of LISTT5, it is deterministic and thus fully replicable. On running the same experiments on FiQA twice, we validated that the output files for LISTT5 are exactly the same.

[15]We were only able to run the inconsistency experiment on GPT3.5 due to the high inference cost of GPT-4.

I Qualitative Analysis

Tab. 12 reports a sample output selected from the Arguana Dataset, which seeks the counter-argument most opposed to the query argument .[16] Considering the NDCG@10 reranking performance for MonoT5-base, RankT5-base, and LISTT5-base is 34.4, 35.5, and 48.9, respectively, LISTT5 excels on Arguana than other pointwise baseline models with a large margin. From inspection, we conclude that the listwise reranking nature of LISTT5 helps to discern and select which features are important in determining relevancy in this dataset, with respect to other passages.

We hypothesize that listwise comparison makes the model to favor counterarguments - (Different point of view, but the discussion topic is the same) over similar topics. For example at Tab. 12, RankT5 places passages that talks about accountability and criminal law in a moral aspect as the most relevant passage, which is quite similar to the original query (since it includes words such as 'Prosecuting') but not exactly the same about prosecuting offenders. Also, the passage that is scored highest from MonoT5 talks about ICC's prosecution policy, but the original query is not related to ICC. In contrast, since perhaps LISTT5 is able to see different passages, it correctly determines the GT passage as top-1, giving more weight on talking about the same topic (even though the aspect can be different).

J Measuring efficiency - Sliding window v.s. Tournament Sort.

We briefly describe the difference of tournament sort and the sliding window approach in Figure. 8. We conduct additional experiments to compare the efficiency of tournament sort and the sliding window.

J.1 Implementation Details.

We rerank top-10 passages from BM25 top100 candidate passages, for 43 queries of TREC-DL19. The latency and FLOPs for tournament sort variants are computed using our LISTT5 evaluation code, and the efficiency for the sliding window approach (without the FiD architecture) is computed using the official repository for RankVicuna

[16]The query is always included in the corpus, so for qualitative analysis, we excluded the passage that is exactly the same with the original query.

	GT Rank / GT Score	1st Rank / 1st Score	Passage
ListT5	1	1	Most often, prosecutions that occur are not just with only the **losing side** being prosecuted for their crimes. The Nuremburg trials prosecuted Nazi's for offences they committed, but none of the Allied forces were ever brought for trial(...)
MonoT5	78/ -0.32	1 / -0.13	As the ICC intentionally limits its prosecutions to group leaders, many of those who actually commit atrocities need have no fear of prosecution By prosecuting only those leaders deemed 'most responsible' for the crimes in question, the ICC is effectively allowing lower-ranked perpetrators to commit crimes with impunity. (...)
RankT5	25/ -5.86	1 / -4.58	Accountability It is a fundamental principle of morality that individuals should be held responsible for their crimes – that is the reason why we, as societies, have criminal law. Prosecuting people – holding them responsible for their crimes – is a moral imperative. We all wish to live in a society where everyone is equally accountable when they commit crime as one in which not everyone is held to account is fundamentally unjust (...)
Query			(...) Prosecuting offenders is the **only way to get a just outcome** when there have been horrific crimes committed. At a most principled level, those who commit a crime ought to be held accountable for their actions even if they are powerful or it damages the chances of peace because the powerful must be shown not to be above the law (...)
Relevant Passage			Most often, prosecutions that occur are not just with only the **losing side** being prosecuted for their crimes. The Nuremburg trials prosecuted Nazi's for offences they committed, but none of the Allied forces were ever brought for trial(...)

Table 12: Example of ListT5, MonoT5, RankT5 retrieval result on Arguana dataset. Relevant topics are highlighted with red, non-relevant topics in blue, and opinions are **bolded**. (Sec. I)

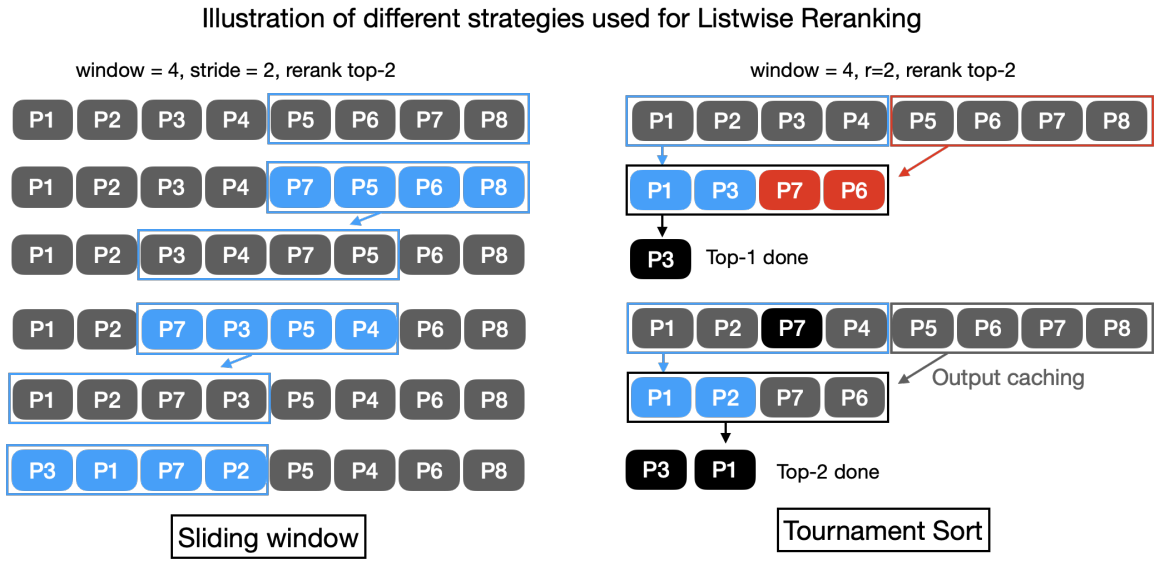

Figure 8: Overview comparison of sliding window v.s. Tournament sort for listwise reranking. (Sec. J)

	GPT-3.5-turbo-1106 Trial 1	Trial 2	Trial 3	Avg.	ListT5 -base
(1) Accuracy when the gold passage is at index #:					
1	81.6	79.6	81.6	**81.0**	**93.9**
2	63.3	63.3	61.2	**62.6**	**87.8**
3	75.5	75.5	75.5	**75.5**	**83.7**
4	67.3	63.3	67.3	**66.0**	**85.7**
5	61.2	63.3	65.3	**63.3**	**81.6**
std	7.68	7.1	7.4	**7.4**	**4.2**
(2) Agreement ratio (%) within index change of positive					
points to same passage	55.1	55.1	55.1	**55.1**	**83.7**
other	44.9	44.9	44.9	44.9	16.3

Table 13: Measuring the LLM consistency on TREC-COVID. (Sec. H.4)

and RankZephyr.[17] The FiD variant of the sliding window approach are computed using our ListT5 evaluation code. We append the deepspeed FlopsProfiler for FLOPs measurement, which is the same as the ones that was appended for ListT5. For a fair comparison, we also measure the FLOPs of the sliding approach on the T5-3b model[18] and the T5-base FiD architecture using the ListT5 code. Example commands used are as follows:

```
CUDA_VISIBLE_DEVICES=0 python
./src/rank_llm/scripts/run_rank_llm.py
--model_path=castorini/lmsys/
fastchat-t5-3b-v1.0
--top_k_candidates=100
--dataset=dl19 --retrieval_method=bm25
```

[17] https://github.com/castorini/rank_llm

[18] we replace the model code from castorini/rank_vicuna_7b_v1 to lmsys/fastchat-t5-3b-v1.0 and measure FLOPs.

Sorting method	# required forward passes to rerank top1	# required forward passes to rerank top10
sliding window, stride=1	1 + $\lceil$(100-5)/1$\rceil$ = 96	96 × $\lceil$10/4$\rceil$ = 288 -> 280
sliding window, stride=2	1 + $\lceil$(100-5)/2$\rceil$ = 49	49 × $\lceil$10/3$\rceil$ = 196 -> 191
sliding window, stride=3	1 + $\lceil$(100-5)/3$\rceil$ = 33	33 × $\lceil$10/2$\rceil$ = 165 -> 162
sliding window, stride=4	1 + $\lceil$(100-5)/4$\rceil$ = **25**	25 × $\lceil$10/1$\rceil$ = 250 -> 248
tournament sort, r=1	(100/5) + (20/5) + 1 = **25**	25 + 9 x (1+1+1) = **52**
tournament sort, r=2	(100/5) + (40/5) + 2 + 1 = 31	31 + 9 × (1+1+1+1) = 67

Table 14: Number of forward passes to rerank top-k candidates from 100 candidate passages per one query, where window size w=5. In the case of reranking top-10 passages, tournament sort requires much more fewer number of forward passes. (Sec. J.2)

Idx	Base Model	Sorting method	Name	FLOPs to rerank:	
				Top-1	Top-10
0	T5-base	pointwise	MonoT5	1x	1x
1	T5-base	tournament	ListT5(r=1)	1.3x	2.6x
2	T5-base	tournament	ListT5(r=2)	1.8x	4.7x
3	T5-base	sliding w.(s=2)	T5(FiD)	2.5x	9.8x
4	T5-base	sliding w.(s=3)	T5(FiD)	1.7x	12.3x
5	T5-3b	tournament	ListT5(r=1)	17.6x	36.3x
6	T5-3b	tournament	ListT5(r=2)	24.6x	66.0x
7	T5-3b	sliding w.(s=2)	T5(FiD)	38.5x	154x
8	T5-3b	sliding w.(s=2)	T5(no FiD)	53.8x	215.1x
9	T5-3b	sliding w.(s=3)	T5(FiD)	25.6x	128x
10	T5-3b	sliding w.(s=3)	T5(no FiD)	35.1x	175.6x

Table 15: FLOPs (In a multiple of FLOPs of MonoT5-base) on the choice of architecture and method, on TREC-DL19. For the sliding window approach, we would need a total of 4 multiple passes for stride = 3 and 5 passes for stride = 2 (Explained at Tab. 14) to rerank Top-10 candidates. (Sec. J.3)

	Rerank Top-10 (NDCG@10)				Rerank Top-1 (NDCG@1)			
Sorting Method	T.S.		S.W		T.S.		S.W.	
Hyperparam.	r=1	r=2	s=2 (iter=5)	s=3 (iter=4)	r=1	r=2	s=2 (iter=1)	s=3 (iter=1)
FLOPS(DL19)	**1x**	1.8x	3.7x	3.1x	**1x**	1.4x	1.96x	1.32x
DL19	71.2	71.8	71.5	71.8	81.0	79.1	81.0	78.7
DL20	67.3	68.1	67.3	67.7	77.8	77.8	79.0	79.6
In-domain avg.	69.3	**70.0**	69.4	69.8	79.4	78.5	**80.0**	79.2
TREC-COVID	76.7	78.3	78.9	77.5	88.0	91.0	88.0	86.0
NFCorpus	35.5	35.6	35.3	35.5	47.8	49.2	48.6	48.6
BioASQ	57.2	56.4	54.5	54.9	59.2	58.4	55.8	57.2
NQ	52.0	53.1	52.7	52.8	36.0	37.6	36.4	36.6
HotpotQA	72.1	72.6	71.2	71.6	83.3	84.1	83.1	83.1
FiQA-2018	39.5	39.6	39.7	39.8	41.2	40.7	41.4	41.5
Signal-1M (RT)	33.3	33.5	32.4	33.2	43.3	41.8	42.3	41.8
TREC-NEWS	47.9	48.5	49.8	50.0	53.2	54.1	52.3	52.9
Robust04	52.0	52.1	51.3	51.7	65.1	66.3	67.1	65.9
Arguana	49.7	48.9	47.7	47.8	25.8	23.9	23.3	22.7
Touche-2020	34.2	33.4	32.7	33.1	34.7	31.6	36.7	36.7
CQADupStack	38.4	38.8	38.9	38.8	31.6	31.9	32.1	32.0
Quora	86.1	86.4	86.3	86.2	77.8	77.8	78.1	77.7
DBPedia	43.9	43.7	42.6	43.2	55.5	56.5	55.1	56.6
SCIDOCS	17.2	17.6	17.9	17.7	21.9	22.0	22.8	21.4
FEVER	77.8	79.8	79.3	79.3	69.4	72.4	70.2	70.4
Climate-FEVER	22.8	24.0	23.8	23.7	20.2	23.3	20.4	21.0
SciFact	74.1	74.1	73.6	73.5	65.0	65.3	65.3	65.7
BEIR avg.	50.6	**50.9**	50.5	50.6	51.1	**51.6**	51.1	51.0

Table 16: Comparison of FLOPs and performance on ListT5 with the **S.W** (Sliding Window) approach and **T.S** (Tournament Sort) with different hyperparameters. ListT5 (r=2) performs the best, with lower FLOPs than the sliding window variants on both setup of reranking top-10 and top-1 candidates. (Sec. J.4)

```
--prompt_mode=rank_GPT --
context_size=4096 --variable_passages
 --window_size 5 --step_size [2 or 3]
```

J.2 Number of Required Forward Passes.

We calculate the number of required forward passes needed for variants of tournament sort and the sliding window approach at Table 14. Given a window size of 5, tournament sort is much more efficient to rerank top-10 passages, requiring fewer number of forward passes. This is because, unlike tournament sort with output caching, the sliding window approach requires re-evaluation over the entire input sequence, depending on the window size, stride, and the number of top-k passages to rerank. Detailed explanation of how we calculated the numbers from the table is below:

- After one pass of a sliding window of size 5 and stride of 4, we discard 4 passages and only carry (5-4)=1 previous passage to the next step as we move the window. Therefore, we would only be able to correctly order top-1 passages, since the top-2 passage cannot be moved.
- Therefore, to rank top 10 candidates, we have to iterate through at least $\lceil$10 / (5-4)$\rceil$ = 10 times, which would result in a total of about 250 forward passes.
- For ranking top-6 with a sliding window of size 5 and stride 4, some forward pass can be saved in the 6th slide because top-5 have already been ordered in the first 5 slides. Therefore, a corrected precise calculation (noted as -> in the table) gives 248.
- The same applies to methods of stride=2 to 4.

- In contrast, tournament sort uses output caching, and after the initial computation (25 for r=1 and 31 for r=2), we only need to compute one path from leaf to root, which only costs only one additional forwards for each level of the tournament tree, which is 3 for (r=1) and 4 for (r=2).

- Therefore, by using tournament sort, we can efficiently reduce the number of forward passes needed to rank top 10 candidates.

J.3 FLOPs comparision

In this section, we compare in detail on the choice of sorting method and architecture at Tab. 15.

T5 (FiD) v.s. T5 (no FiD). ListT5 uses the FiD architecture to effectively mitigate the positional bias problem and handle long inputs efficiently. Comparing with idx 7 v.s 8 (38.5 vs 53.8) and 9 vs 10 (25.6 vs 35.1) at Tab. 15, we can see that using FiD results in lower number of total FLOPs.

Tournament sort v.s. Sliding window on T5 (FiD). By comparing idx 5 and 6 with respect to idx 7 and 9, we conclude that the FLOPs to rerank Top-10 candidates are much lower for both (r = 1) and (r = 2) variants of tournament sort (36.3x and 66.0x), compared with the FiD variant of sliding window, for both (stride = 2) and (stride = 3) (154x and 128x). It also holds the same for models built on top of T5-base, by comparing idx (1,2) with respect to (3,4).

J.4 Performance Comparison.

Comparison of FLOPs and performance on ListT5 with the S.W(Sliding Window) approach and T.S(Tournament Sort) with different hyperparameters. ListT5 (r=2) performs the best, with lower FLOPs than the sliding window variants on both setup of reranking top-10 and top-1 candidates.

K Additional Experiments on Tournament Sort.

K.1 Cases where m > k

In the main paper, we have discussed that the sliding window approach can be much more efficient when $m \ll k$. However, even when $m > k$, e.g., m = 20 and k = 10, the number of forward passes needed to get top-10 rankings for tournament sort can less than or equal to the sliding window variants, with simple modifications. For example, the number of iterations to rank top-10 with m=20 and r = 10 with ListT5 requires 9 forward passes if we take into account all top-r results from one iteration and take into account unwanted wins. Tournament sort with r=10 can correctly rank global top-10 candidates in one iteration, as illustrated at Fig. 9. This is the same amount of iterations needed with the sliding window approach with window size of 20 and stride of 10. We also compute the number of forward passes with different value of s, or r in Table. 17.

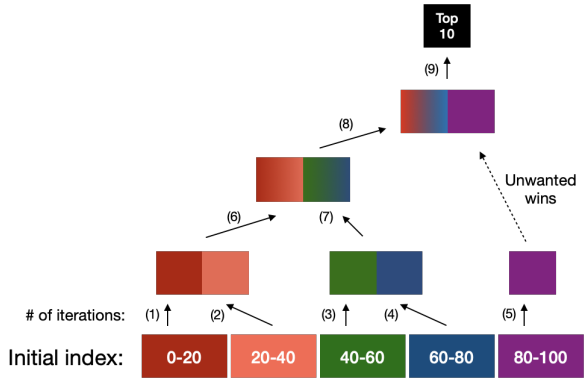

Figure 9: Illustration for the tournament sort to rank top-10 passages among 100 candidates for r=10 and m=20. It takes 9 forward passes with unwanted wins. (Sec. K.1)

	# of stride == value of r		
	10	5	1
Sliding w.	9 x 1 iter. = **9**	17 x 2 iter. = 34	81 x 10 iter. = 810
Tournament S.	9 x 1 iter. = **9**	7 x 2 iter. = **14**	6 x 10 iter. = **60**

Table 17: Comparison of the number of forward passes needed to rank top-10 passages, when m = 20, and n = 100. The number is written in the format of {number of forward pass for each iteration} x {number of iterations to assure global ranking}. (Sec. K.1)

K.2 Tournament Sort with LLMs.

We have also analyzed the performance of RankGPT (Sun et al., 2023b), listwise reranking with GPT3.5, with tournament sort. The performance difference of tournament sort with respect to the sliding window approach for w=20, s=10 were not significant, while the number of required forward passes to rank top-10 passages were the same (9) for both variants.

Method	dl19	dl20	trec-covid	news	touche
sliding	68.4 ± 0.4	64.9 ± 1.1	72.6 ± 1.4	46.5 ± 1.0	**38.2 ± 0.5**
tournament	67.4 ± 0.9	65.8 ± 0.6	**76.4 ± 0.4**	45.5 ± 1.0	33.1 ± 1.7

Table 18: NDCG@10 on the selected subset of BEIR, on RankGPT-3.5 with different sorting methods. For fair comparison, we used w = 20, s = 10 for the sliding approach, and m = 20, r = 10 for the tournament sort. To compensate for the instability of APIs, all results are run for 3 times. Except for trec-covid and touche, differences are statistically non-significant ($p > 0.1$). (Sec. K.2)